## Isabelle/HOL — Higher-Order Logic

May 23, 2024

### Contents

1. Loading the code generator and related modules 33

2. The basis of Higher-Order Logic 34
   
   2.1. Primitive logic .......................... 35
   
   2.1.1. Core syntax .......................... 35
   2.1.2. Defined connectives and quantifiers ........ 36
   2.1.3. Additional concrete syntax ................ 36
   2.1.4. Axioms and basic definitions .......... 38

2.2. Fundamental rules ......................... 39
   
   2.2.1. Equality ............................. 39
   2.2.2. Congruence rules for application ........ 39
   2.2.3. Equality of booleans – iff ............. 40
   2.2.4. True (1) ............................. 40
   2.2.5. Universal quantifier (1) ................ 40
   2.2.6. False ................................ 40
   2.2.7. Negation .............................. 41
   2.2.8. Implication ........................... 41
   2.2.9. Disjunction (1) ........................ 42
   2.2.10. Derivation of $iff$ ................ 42
   2.2.11. True (2) ............................. 42
   2.2.12. Universal quantifier (2) ............. 43
   2.2.13. Existential quantifier ............... 43
   2.2.14. Conjunction .......................... 43
   2.2.15. Disjunction (2) ........................ 43
   2.2.16. Classical logic ........................ 44
   2.2.17. Unique existence ..................... 44
   2.2.18. Classical intro rules for disjunction and existential quantifiers 45
   2.2.19. Intuitionistic Reasoning ................ 46
   2.2.20. Atomizing meta-level connectives ........ 47
   2.2.21. Atomizing elimination rules ............ 48
2.3 Package setup ................................................. 48
  2.3.1 Sledgehammer setup ..................................... 49
  2.3.2 Classical Reasoner setup ................................. 49
  2.3.3 THE: definite description operator ....................... 51
  2.3.4 Simplifier ................................................ 52
  2.3.5 Generic cases and induction .............................. 63
  2.3.6 Coherent logic .......................................... 66
  2.3.7 Reorienting equalities .................................. 66
  2.4 Other simple lemmas and lemma duplicates .................... 67
  2.5 Basic ML bindings ......................................... 68

3 NO-MATCH simproc .............................................. 70
  3.1 Code generator setup ....................................... 71
    3.1.1 Generic code generator preprocessor setup .......... 71
    3.1.2 Equality ............................................. 72
    3.1.3 Generic code generator foundation .................... 72
    3.1.4 Generic code generator target languages ............. 74
    3.1.5 Evaluation and normalization by evaluation ........... 76
  3.2 Counterexample Search Units ............................... 76
    3.2.1 Quickcheck ........................................... 76
    3.2.2 Nitpick setup ......................................... 76
  3.3 Preprocessing for the predicate compiler ................... 76
  3.4 Legacy tactics and ML bindings ............................. 77

4 Abstract orderings ............................................ 77
  4.1 Abstract ordering .......................................... 77
  4.2 Syntactic orders .......................................... 81
  4.3 Quasi orders .............................................. 81
  4.4 Partial orders ............................................ 83
  4.5 Linear (total) orders ..................................... 85
  4.6 Reasoning tools setup ..................................... 87
  4.7 Bounded quantifiers ....................................... 91
  4.8 Transitivity reasoning ..................................... 93
  4.9 min and max – fundamental ................................ 99
  4.10 (Unique) top and bottom elements ......................... 99
  4.11 Dense orders ............................................. 101
  4.12 Wellorders ............................................... 103
  4.13 Order on bool ............................................ 104
  4.14 Order on - ⇒ - ........................................... 105
  4.15 Order on unary and binary predicates ..................... 107
  4.16 Name duplicates .......................................... 108
# Groups, also combined with orderings

5.1 Dynamic facts ........................................ 109
5.2 Abstract structures .................................. 109
5.3 Generic operations .................................. 112
5.4 Semigroups and Monoids ............................. 113
5.5 Groups .................................................. 117
5.6 (Partially) Ordered Groups .......................... 121
5.7 Support for reasoning about signs ................. 123
5.8 Canonically ordered monoids ......................... 136
5.9 Tools setup ............................................ 138

# Abstract lattices

6.1 Abstract semilattice .................................. 139
6.2 Syntactic infimum and supremum operations .... 142
6.3 Concrete lattices ..................................... 142
   6.3.1 Intro and elim rules ........................... 143
   6.3.2 Equational laws ............................... 144
   6.3.3 Strict order ..................................... 147
6.4 Distributive lattices ................................ 147
6.5 Bounded lattices ..................................... 148
6.6 min/max as special case of lattice ................. 150
6.7 Uniqueness of inf and sup .......................... 151
6.8 Lattice on \(-\Rightarrow-\) ............................. 152

# Boolean Algebras

7.1 Abstract boolean algebra ............................ 153
   7.1.1 Complement .................................... 154
   7.1.2 Conjunction .................................... 155
   7.1.3 Disjunction .................................... 156
   7.1.4 De Morgan’s Laws .............................. 156
7.2 Symmetric Difference ................................. 157
7.3 Type classes .......................................... 159
7.4 Lattice on bool ...................................... 161
7.5 Lattice on unary and binary predicates .......... 162
7.6 Simproc setup .......................................

# Set theory for higher-order logic

8.1 Sets as predicates .................................... 165
8.2 Subsets and bounded quantifiers .................... 167
8.3 Basic operations ..................................... 173
   8.3.1 Subsets ....................................... 173
   8.3.2 Equality ....................................... 174
   8.3.3 The empty set .................................. 175
   8.3.4 The universal set – UNIV ...................... 175
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.3.5</td>
<td>The Powerset operator – Pow</td>
<td>176</td>
</tr>
<tr>
<td>8.3.6</td>
<td>Set complement</td>
<td>177</td>
</tr>
<tr>
<td>8.3.7</td>
<td>Binary intersection</td>
<td>177</td>
</tr>
<tr>
<td>8.3.8</td>
<td>Binary union</td>
<td>177</td>
</tr>
<tr>
<td>8.3.9</td>
<td>Set difference</td>
<td>178</td>
</tr>
<tr>
<td>8.3.10</td>
<td>Augmenting a set – insert</td>
<td>179</td>
</tr>
<tr>
<td>8.3.11</td>
<td>Singletons, using insert</td>
<td>180</td>
</tr>
<tr>
<td>8.3.12</td>
<td>Image of a set under a function</td>
<td>181</td>
</tr>
<tr>
<td>8.3.13</td>
<td>Some rules with if</td>
<td>184</td>
</tr>
<tr>
<td>8.4</td>
<td>Further operations and lemmas</td>
<td>185</td>
</tr>
<tr>
<td>8.4.1</td>
<td>The “proper subset” relation</td>
<td>185</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Derived rules involving subsets</td>
<td>186</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Equalities involving union, intersection, inclusion, etc.</td>
<td>187</td>
</tr>
<tr>
<td>8.4.4</td>
<td>Monotonicity of various operations</td>
<td>197</td>
</tr>
<tr>
<td>8.4.5</td>
<td>Inverse image of a function</td>
<td>198</td>
</tr>
<tr>
<td>8.4.6</td>
<td>Singleton sets</td>
<td>200</td>
</tr>
<tr>
<td>8.4.7</td>
<td>Getting the contents of a singleton set</td>
<td>200</td>
</tr>
<tr>
<td>8.4.8</td>
<td>Monad operation</td>
<td>200</td>
</tr>
<tr>
<td>8.4.9</td>
<td>Operations for execution</td>
<td>201</td>
</tr>
<tr>
<td>9</td>
<td>HOL type definitions</td>
<td>205</td>
</tr>
<tr>
<td>10</td>
<td>Notions about functions</td>
<td>207</td>
</tr>
<tr>
<td>10.1</td>
<td>The Identity Function id</td>
<td>208</td>
</tr>
<tr>
<td>10.2</td>
<td>The Composition Operator f o g</td>
<td>208</td>
</tr>
<tr>
<td>10.3</td>
<td>The Forward Composition Operator fcomp</td>
<td>209</td>
</tr>
<tr>
<td>10.4</td>
<td>Mapping functions</td>
<td>210</td>
</tr>
<tr>
<td>10.5</td>
<td>Injectivity and Bijectivity</td>
<td>210</td>
</tr>
<tr>
<td>10.5.1</td>
<td>Inj/surj/bij of Algebraic Operations</td>
<td>222</td>
</tr>
<tr>
<td>10.6</td>
<td>Function Updating</td>
<td>225</td>
</tr>
<tr>
<td>10.7</td>
<td>override-on</td>
<td>226</td>
</tr>
<tr>
<td>10.8</td>
<td>Inversion of injective functions</td>
<td>226</td>
</tr>
<tr>
<td>10.9</td>
<td>Monotonicity</td>
<td>228</td>
</tr>
<tr>
<td>10.9.1</td>
<td>Specializations For ord Type Class And More</td>
<td>229</td>
</tr>
<tr>
<td>10.9.2</td>
<td>Least value operator</td>
<td>235</td>
</tr>
<tr>
<td>10.10</td>
<td>Setup</td>
<td>236</td>
</tr>
<tr>
<td>10.10.1</td>
<td>Proof tools</td>
<td>236</td>
</tr>
<tr>
<td>10.10.2</td>
<td>Functorial structure of types</td>
<td>236</td>
</tr>
<tr>
<td>11</td>
<td>Complete lattices</td>
<td>237</td>
</tr>
<tr>
<td>11.1</td>
<td>Syntactic infimum and supremum operations</td>
<td>237</td>
</tr>
<tr>
<td>11.2</td>
<td>Abstract complete lattices</td>
<td>238</td>
</tr>
<tr>
<td>11.3</td>
<td>Complete lattice on bool</td>
<td>249</td>
</tr>
<tr>
<td>11.4</td>
<td>Complete lattice on ( \Rightarrow )</td>
<td>250</td>
</tr>
</tbody>
</table>
11.5 Complete lattice on unary and binary predicates 251
11.6 Complete lattice on - set 252
11.6.1 Inter 253
11.6.2 Intersections of families 254
11.6.3 Union 256
11.6.4 Unions of families 257
11.6.5 Distributive laws 260
11.7 Injections and bijections 261
11.7.1 Complement 263
11.7.2 Miniscoping and maxiscoping 263

12 Wrapping Existing Freely Generated Type’s Constructors 265

13 Knaster-Tarski Fixpoint Theorem and inductive definitions 266
13.1 Least fixed points 266
13.2 General induction rules for least fixed points 267
13.3 Greatest fixed points 268
13.4 Coinduction rules for greatest fixed points 269
13.5 Even Stronger Coinduction Rule, by Martin Coen 271
13.6 Rules for fixed point calculus 272
13.7 Inductive predicates and sets 274
13.8 The Schroeder-Bernstein Theorem 274
13.9 Inductive datatypes and primitive recursion 276

14 Cartesian products 277
14.1 bool is a datatype 277
14.2 The unit type 278
14.3 The product type 280
14.3.1 Type definition 280
14.3.2 Tuple syntax 282
14.3.3 Code generator setup 284
14.3.4 Fundamental operations and properties 284
14.3.5 Derived operations 291
14.4 Simproc for rewriting a set comprehension into a pointfree expression 302
14.5 Lemmas about disjointness 302
14.6 Inductively defined sets 303
14.7 Legacy theorem bindings and duplicates 303

15 The Disjoint Sum of Two Types 304
15.1 Construction of the sum type and its basic abstract operations 304
15.2 Projections 307
15.3 The Disjoint Sum of Sets 308
16 Rings
  16.1 Semirings and rings ........................................ 309
  16.2 Abstract divisibility ........................................ 311
  16.3 Towards integral domains .................................... 317
  16.4 (Partial) Division ........................................... 322
  16.5 Quotient and remainder in integral domains ............... 345
  16.6 Interlude: basic tool support for algebraic and arithmetic calculations ........................................ 346
  16.7 Ordered semirings and rings .................................. 347
  16.8 Dioids .......................................................... 366

17 Natural numbers
  17.1 Type ind ....................................................... 366
  17.2 Type nat ........................................................ 366
  17.3 Arithmetic operators .......................................... 370
    17.3.1 Addition .................................................. 372
    17.3.2 Difference ............................................... 373
    17.3.3 Multiplication .......................................... 373
  17.4 Orders on nat .................................................. 374
    17.4.1 Operation definition .................................... 374
    17.4.2 Introduction properties ................................ 376
    17.4.3 Elimination properties .................................. 376
    17.4.4 Inductive (?) properties ................................ 377
    17.4.5 Monotonicity of Addition ................................ 381
    17.4.6 min and max ............................................. 383
    17.4.7 Additional theorems about (≤) ........................... 385
    17.4.8 More results about difference ........................... 391
    17.4.9 Monotonicity of multiplication .......................... 393
  17.5 Natural operation of natural numbers on functions ........ 395
  17.6 Kleene iteration ............................................... 398
  17.7 Embedding of the naturals into any semiring-1: of-nat .... 400
  17.8 The set of natural numbers .................................... 404
  17.9 Further arithmetic facts concerning the natural numbers ... 405
    17.9.1 Greatest operator ........................................ 413
    17.10 Monotonicity of funpow .................................... 414
    17.11 The divides relation on nat ................................ 414
    17.12 Aliasses ..................................................... 416
    17.13 Size of a datatype value .................................... 417
    17.14 Code module namespace ..................................... 418

18 Fields
  18.1 Division rings ................................................. 418
  18.2 Fields .......................................................... 425
  18.3 Ordered fields ................................................ 429
19 Relations – as sets of pairs, and binary predicates

19.1 Fundamental ............................................. 446
19.1.1 Relations as sets of pairs .......................... 446
19.1.2 Conversions between set and predicate relations .... 446
19.2 Properties of relations ................................. 448
19.2.1 Reflexivity ............................................ 448
19.2.2 Irreflexivity ........................................... 451
19.2.3 Asymmetry ............................................ 452
19.2.4 Symmetry .............................................. 454
19.2.5 Antisymmetry ........................................ 456
19.2.6 Transitivity ........................................... 458
19.2.7 Totality ................................................ 461
19.2.8 Single valued relations ............................. 463
19.3 Relation operations ................................. 464
19.3.1 The identity relation ............................... 464
19.3.2 Diagonal: identity over a set ....................... 464
19.3.3 Composition ......................................... 465
19.3.4 Converse .............................................. 467
19.3.5 Domain, range and field ......................... 471
19.3.6 Image of a set under a relation .................. 474
19.3.7 Inverse image ....................................... 476
19.3.8 Powerset .............................................. 476

20 Finite sets

20.1 Predicate for finite sets ............................. 477
20.1.1 Choice principles .................................. 478
20.1.2 Finite sets are the images of initial segments of natural numbers .......................... 478
20.2 Finiteness and common set operations ............... 480
20.3 Further induction rules on finite sets ............... 489
20.4 Class finite ............................................ 491
20.5 A basic fold functional for finite sets ............... 492
20.5.1 From fold-graph to fold ......................... 494
20.5.2 Liftings to comp-fun-commute-on etc. ............ 500
20.5.3 UNIV as carrier set ............................... 501
20.5.4 Expressing set operations via fold ............... 503
20.5.5 Expressing relation operations via fold ........... 507
20.6 Locales as mini-packages for fold operations ....... 509
20.6.1 The natural case ................................. 509
20.6.2 With idempotency ............................... 510
20.6.3 UNIV as the carrier set ......................... 510
20.7 Finite cardinality .................................... 511
20.7.1 Cardinality of image ............................... 524
20.7.2 Pigeonhole Principles ............................. 526
20.7.3 Cardinality of sums ........................................... 527
20.8 Minimal and maximal elements of finite sets ................. 527
  20.8.1 Finite orders ............................................. 534
  20.8.2 Relating injectivity and surjectivity .................... 534
20.9 Infinite Sets .................................................. 536
20.10 The finite powerset operator .................................. 538

21 Reflexive and Transitive closure of a relation ............... 539
  21.1 Reflexive closure ............................................. 540
  21.2 Reflexive-transitive closure ................................. 541
  21.3 Transitive closure ........................................... 547
  21.4 Symmetric closure ............................................ 556
  21.5 The power operation on relations ............................ 557
  21.6 Bounded transitive closure .................................. 567
  21.7 Acyclic relations ............................................. 568
  21.8 Setup of transitivity reasoner ................................ 569

22 Well-founded Recursion ......................................... 571
  22.1 Basic Definitions ............................................. 571
  22.2 Equivalence of Definitions ................................... 571
  22.3 Induction Principles ......................................... 572
  22.4 Introduction Rules .......................................... 573
  22.5 Ordering Properties ......................................... 573
  22.6 Basic Results ................................................ 575
    22.6.1 Minimal-element characterization of well-foundedness 576
    22.6.2 Antimonotonicity ....................................... 577
    22.6.3 Well-foundedness of transitive closure ................. 578
    22.6.4 Well-foundedness of image ............................... 580
  22.7 Well-Foundedness Results for Unions ......................... 582
  22.8 Well-Foundedness of Composition ............................. 584
  22.9 Acyclic relations ............................................ 586
    22.9.1 Wellfoundedness of finite acyclic relations ........... 586
  22.10 nat is well-founded ........................................ 587
  22.11 Accessible Part ............................................. 588
  22.12 Tools for building wellfounded relations .................. 591
    22.12.1 Conversion to a known well-founded relation ......... 591
    22.12.2 Measure functions into nat ............................. 592
    22.12.3 Lexicographic combinations ............................. 593
    22.12.4 Bounded increase must terminate ....................... 597
  22.13 Code Generation Setup ..................................... 597

23 Well-Founded Recursion Combinator ......................... 598
  23.0.1 Well-founded recursion via genuine fixpoints .......... 599
  23.1 Wellfoundedness of same-fst ................................. 599
### 24 Orders as Relations

24.1 Orders on a set ............................................. 600
24.2 Orders on the field ........................................ 602
24.3 Relations given by a predicate and the field .......... 603
24.4 Orders on a type ........................................... 604
24.5 Order-like relations ....................................... 604
   24.5.1 Auxiliaries .......................................... 604
   24.5.2 The upper and lower bounds operators ........... 605
24.6 Variations on Well-Founded Relations ................. 608
   24.6.1 Characterizations of well-foundedness ............ 609
   24.6.2 Characterizations of well-foundedness .......... 611

### 25 Hilbert's Epsilon-Operator and the Axiom of Choice

25.1 Hilbert's epsilon ........................................... 613
25.2 Hilbert's Epsilon-operator ................................ 613
25.3 Axiom of Choice, Proved Using the Description Operator .. 614
25.4 Function Inverse .......................................... 615
25.5 Other Consequences of Hilbert’s Epsilon ............... 626
25.6 An aside: bounded accessible part .................... 627
25.7 More on injections, bijections, and inverses .......... 629
25.8 Specification package – Hilbertized version .......... 632
25.9 Complete Distributive Lattices – Properties depending on Hilbert Choice .................................. 632

### 26 Zorn’s Lemma and the Well-ordering Theorem

26.1 Zorn’s Lemma for the Subset Relation ................. 638
   26.1.1 Results that do not require an order ............. 638
   26.1.2 Hausdorff’s Maximum Principle .................... 644
   26.1.3 Results for the proper subset relation .......... 645
   26.1.4 Zorn’s lemma ........................................ 645
26.2 Zorn’s Lemma for Partial Orders ....................... 646
26.3 Other variants of Zorn’s Lemma ......................... 648
26.4 The Well Ordering Theorem ............................. 651

### 27 Well-Order Relations as Needed by Bounded Natural Functions

27.1 Auxiliaries ................................................. 657
27.2 Well-founded induction and recursion adapted to non-strict well-order relations ............................... 658
27.3 The notions of maximum, minimum, supremum, successor and order filter .................................. 658
   27.3.1 Properties of max2 .................................. 659
   27.3.2 Existence and uniqueness for isMinim and well-definedness of minim ............................... 660
27.3.3 Properties of minim ................................. 661
27.3.4 Properties of successor .............................. 662
27.3.5 Properties of order filters ......................... 663
27.3.6 Other properties .......................... 665

28 Well-Order Embeddings as Needed by Bounded Natural Functors 667
28.1 Auxiliaries ......................................... 668
28.2 (Well-order) embeddings, strict embeddings, isomorphisms and order-compatible functions ................................. 668
28.3 Given any two well-orders, one can be embedded in the other 677
28.4 Uniqueness of embeddings ............................... 683
28.5 More properties of embeddings, strict embeddings and isomorphisms .................................. 685

29 Constructions on Wellorders as Needed by Bounded Natural Functors 691
29.1 Restriction to a set .................................... 691
29.2 Order filters versus restrictions and embeddings ................................. 693
29.3 The strict inclusion on proper ofilters is well-founded ................. 697
29.4 Ordering the well-orders by existence of embeddings ................. 697
29.5 \(<\omega \) is well-founded .................................. 707
29.6 Copy via direct images .................................. 708
29.7 Bounded square ......................................... 712

30 Cardinal-Order Relations as Needed by Bounded Natural Functors 724
30.1 Cardinal orders ....................................... 725
30.2 Cardinal of a set ...................................... 726
30.3 Cardinals versus set operations on arbitrary sets .......... 731
30.4 Cardinals versus set operations involving infinite sets ........ 739
30.5 The cardinal \(\omega \) and the finite cardinals ............ 746
  30.5.1 First as well-orders ................................. 746
  30.5.2 Then as cardinals .................................. 747
30.6 The successor of a cardinal .............................. 748
30.7 Regular cardinals ...................................... 755
30.8 Others ................................................. 757
30.9 Regular vs. stable cardinals .................................. 759

31 Cardinal Arithmetic as Needed by Bounded Natural Functors 764
31.1 Zero ..................................................... 764
31.2 (In)finite cardinals ..................................... 765
31.3 Binary sum ............................................. 766
31.4 One ....................................................... 768
31.5 Two ....................................................... 769
31.6 Family sum ........................................... 769
31.7 Product ............................................... 769
31.8 Exponentiation ..................................... 773

32 Function Definition Base .................................. 781

33 Definition of Bounded Natural Functors .................... 781

34 Composition of Bounded Natural Functors ................... 787

35 Registration of Basic Types as Bounded Natural Functors .. 792

36 Shared Fixpoint Operations on Bounded Natural Functors .. 798

37 Least Fixpoint (Datatype) Operation on Bounded Natural Functors ........................................ 804

38 Equivalence Relations in Higher-Order Set Theory ............ 808
  38.1 Equivalence relations – set version ....................... 808
  38.2 Equivalence classes ................................... 809
  38.3 Quotients .............................................. 810
  38.4 Refinement of one equivalence relation WRT another .... 811
  38.5 Defining unary operations upon equivalence classes .... 812
  38.6 Defining binary operations upon equivalence classes ... 813
  38.7 Quotients and finiteness ................................ 815
  38.8 Projection ............................................. 815
  38.9 Equivalence relations – predicate version ............... 816
  38.10 Equivalence closure ................................... 818

39 MESON Proof Method ........................................ 823
  39.1 Negation Normal Form .................................. 823
  39.2 Pulling out the existential quantifiers .................... 823
  39.3 Lemmas for Forward Proof ............................... 824
  39.4 Clausification helper ................................... 825
  39.5 Skolemization helpers .................................. 826
  39.6 Meson package ........................................ 826

40 Automatic Theorem Provers (ATPs) .......................... 826
  40.1 ATP problems and proofs ............................... 827
  40.2 Higher-order reasoning helpers ......................... 827
  40.3 Basic connection between ATPs and HOL ................. 829
41 Metis Proof Method 829
   41.1 Literal selection and lambda-lifting helpers . . . . . . 830
   41.2 Metis package . . . . . . . . . . . . . . . . . . . . . . . 830

42 Generic theorem transfer using relations 831
   42.1 Relator for function space . . . . . . . . . . . . . . . . 831
   42.2 Transfer method . . . . . . . . . . . . . . . . . . . . . . 831
   42.3 Predicates on relations, i.e. “class constraints” . . . . . 832
   42.4 Properties of relators . . . . . . . . . . . . . . . . . . . 837
   42.5 Transfer rules . . . . . . . . . . . . . . . . . . . . . . . 839
   42.6 of-bool and of-nat . . . . . . . . . . . . . . . . . . . . . 844

43 Lifting package 845
   43.1 Function map . . . . . . . . . . . . . . . . . . . . . . . . 845
   43.2 Quotient Predicate . . . . . . . . . . . . . . . . . . . . . 845
   43.3 Quotient composition . . . . . . . . . . . . . . . . . . . . 849
   43.4 Respects predicate . . . . . . . . . . . . . . . . . . . . . . 849
   43.5 Domains . . . . . . . . . . . . . . . . . . . . . . . . . . . . 854
   43.6 ML setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 856

44 Definition of Quotient Types 856
   44.1 Quotient Predicate . . . . . . . . . . . . . . . . . . . . . . 857
   44.2 lemmas for regularisation of ball and bex . . . . . . . . . 861
   44.3 Bounded abstraction . . . . . . . . . . . . . . . . . . . . . 863
   44.4 Bex1-rel quantifier . . . . . . . . . . . . . . . . . . . . . . 864
   44.5 Various respects and preserve lemmas . . . . . . . . . . . 865
   44.6 Quotient composition . . . . . . . . . . . . . . . . . . . . 868
   44.7 Quotient3 to Quotient . . . . . . . . . . . . . . . . . . . . 869
   44.8 ML setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 870
   44.9 Methods / Interface . . . . . . . . . . . . . . . . . . . . . 871

45 Lifting of BNFs 872

46 Binary Numerals 875
   46.1 The num type . . . . . . . . . . . . . . . . . . . . . . . . . 875
   46.2 Numeral operations . . . . . . . . . . . . . . . . . . . . . 877
   46.3 Binary numerals . . . . . . . . . . . . . . . . . . . . . . . . 880
   46.4 Class-specific numeral rules . . . . . . . . . . . . . . . . . 882
      46.4.1 Structures with addition: class numeral . . . . . . . . 882
      46.4.2 Structures with negation: class neg-numeral . . . . . . 882
      46.4.3 Structures with multiplication: class semiring-numeral 886
      46.4.4 Structures with a zero: class semiring-1 . . . . . . . . 886
      46.4.5 Equality: class semiring-char-0 . . . . . . . . . . . . 887
      46.4.6 Comparisons: class linordered-nonzero-semiring . . . 888
46.4.7 Multiplication and negation: class ring-I 890
46.4.8 Equality using iszero for rings with non-zero characteristic 890
46.4.9 Equality and negation: class ring-char-0 891
46.4.10 Structures with negation and order: class linordered-idom 893
46.4.11 Natural numbers 895
46.5 Particular lemmas concerning 2::'a 899
46.6 Numeral equations as default simplification rules 900
46.6.1 Special Simplification for Constants 900
46.6.2 Optional Simplification Rules Involving Constants 902
46.7 Setting up simprocs 902
46.7.1 Simplification of arithmetic operations on integer constants 903
46.7.2 Simplification of arithmetic when nested to the right 904
46.8 Code module namespace 905
46.9 Printing of evaluated natural numbers as numerals 905
46.10 More on auxiliary conversion 905

47 Exponentiation 906
47.1 Powers for Arbitrary Monoids 906
47.2 Exponentiation on ordered types 915
47.3 Miscellaneous rules 924
47.4 Exponentiation for the Natural Numbers 925
47.4.1 Cardinality of the Powerset 927
47.5 Code generator tweak 928

48 Big sum and product over finite (non-empty) sets 928
48.1 Generic monoid operation over a set 928
48.1.1 Standard sum or product indexed by a finite set 928
48.1.2 HOL Light variant: sum/product indexed by the non-neutral subset 939
48.2 Generalized summation over a set 941
48.2.1 Properties in more restricted classes of structures 942
48.2.2 Cardinality as special case of sum 951
48.2.3 Cardinality of products 956
48.3 Generalized product over a set 957
48.3.1 Properties in more restricted classes of structures 958

49 Chain-complete partial orders and their fixpoints 964
49.1 Chains 964
49.2 Chain-complete partial orders 965
49.3 Transfinite iteration of a function 965
49.4 Fixpoint combinator 967
49.5 Fixpoint induction 968
# Datatype option

50.0.1 Operations .......................... 973
50.1 Transfer rules for the Transfer package ........................................... 978
   50.1.1 Interaction with finite sets .................................................. 978
   50.1.2 Code generator setup ......................................................... 978

# Partial Function Definitions

51.1 Axiomatic setup ..................................................... 980
51.2 Flat interpretation: tailrec and option .................................... 984

# Reconstructing external resolution proofs for propositional logic

52 Reconstructing external resolution proofs for propositional logic 989

# Function Definitions and Termination Proofs

53 Function Definitions and Termination Proofs 990
   53.1 Definitions with default value .............................................. 990
   53.2 Measure functions ............................................................ 992
   53.3 Congruence rules .............................................................. 992
   53.4 Simp rules for termination proofs ....................................... 993
   53.5 Decomposition ............................................................... 993
   53.6 Reduction pairs ............................................................. 993
   53.7 Concrete orders for SCNP termination proofs .......................... 994
   53.8 Yet more induction principles on the natural numbers .............. 996
   53.9 Tool setup ................................................................. 996

# The Integers as Equivalence Classes over Pairs of Natural Numbers

54 The Integers as Equivalence Classes over Pairs of Natural Numbers 997
   54.1 Definition of integers as a quotient type .............................. 997
   54.2 Integers form a commutative ring ......................................... 997
   54.3 Integers are totally ordered ............................................... 998
   54.4 Ordering properties of arithmetic operations .......................... 999
   54.5 Embedding of the Integers into any ring-1: of-int ................. 1001
   54.6 Magnitude of an Integer, as a Natural Number: nat .............. 1007
   54.7 Lemmas about the Function of-nat and Orderings .................. 1011
   54.8 Cases and induction ....................................................... 1012
      54.8.1 Binary comparisons .................................................. 1014
      54.8.2 Comparisons, for Ordered Rings .................................. 1014
   54.9 The Set of Integers ....................................................... 1015
   54.10 sum and prod .............................................................. 1019
   54.11 Setting up simplification procedures ................................. 1019
   54.12 More Inequality Reasoning .................................................. 1020
   54.13 The functions nat and int .................................................. 1021
   54.14 Induction principles for int ............................................. 1023
   54.15 Intermediate value theorems .............................................. 1026
   54.16 Products and 1, by T. M. Rasmussen ................................. 1026
54.17 The divides relation .................................................. 1027
54.18 Powers with integer exponents ....................................... 1032
54.19 Finiteness of intervals ................................................ 1039
54.20 Configuration of the code generator ................................. 1040
54.21 Duplicates .............................................................. 1043

55 Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets 1043
55.1 Generic lattice operations over a set .................................. 1043
  55.1.1 Without neutral element .......................................... 1043
  55.1.2 With neutral element .............................................. 1047
55.2 Lattice operations on finite sets ...................................... 1049
55.3 Infimum and Supremum over non-empty sets ......................... 1050
55.4 Minimum and Maximum over non-empty sets ......................... 1053
55.5 Arg Min ............................................................... 1061
55.6 Arg Max ............................................................... 1064

56 Division in euclidean (semi)rings 1065
56.1 Euclidean (semi)rings with explicit division and remainder .... 1065
56.2 Euclidean (semi)rings with cancel rules .............................. 1069
56.3 Uniquely determined division ......................................... 1078
56.4 Division on nat ........................................................ 1083
56.5 Division on int ........................................................ 1099
  56.5.1 Basic instantiation ................................................. 1099
  56.5.2 Algebraic foundations ............................................. 1100
  56.5.3 Basic conversions ................................................ 1102
  56.5.4 Euclidean division ................................................. 1103
  56.5.5 Trivial reduction steps .......................................... 1105
  56.5.6 More uniqueness rules .......................................... 1107
  56.5.7 Laws for unary minus .......................................... 1107
  56.5.8 Borders ........................................................... 1108
  56.5.9 Splitting Rules for div and mod ................................. 1109
  56.5.10 Algebraic rewrites .............................................. 1110
  56.5.11 Distributive laws for conversions ............................... 1112
  56.5.12 Monotonicity in the First Argument (Dividend) ............... 1112
  56.5.13 Monotonicity in the Second Argument (Divisor) ............... 1113
  56.5.14 Quotients of Signs ............................................. 1114
  56.5.15 Further properties ............................................. 1117
  56.5.16 Computing div and mod by shifting ............................ 1118
56.6 Code generation ...................................................... 1121
57 Parity in rings and semirings

57.1 Ring structures with parity and even/odd predicates

57.2 Instance for nat

57.3 Parity and powers

57.4 Instance for int

57.5 Special case: euclidean rings structurally containing the natural numbers

57.6 Generic symbolic computations

57.6.1 Computation by simplification

57.7 Computing congruences modulo \(2^q\)

58 Combination and Cancellation Simprocs for Numeral Expressions

59 Semiring normalization

60 Groebner bases

60.1 Groebner Bases

61 Set intervals

61.1 Various equivalences

61.2 Logical Equivalences for Set Inclusion and Equality

61.3 Two-sided intervals

61.3.1 Emptiness, singletons, subset

61.4 Infinite intervals

61.4.1 Intersection

61.5 Intervals of natural numbers

61.5.1 The Constant lessThan

61.5.2 The Constant greaterThan

61.5.3 The Constant atLeast

61.5.4 The Constant atMost

61.5.5 The Constant atLeastLessThan

61.5.6 The Constant atLeastAtMost

61.5.7 Intervals of nats with Suc

61.5.8 Intervals and numerals

61.5.9 Image

61.5.10 Finiteness

61.5.11 Proving Inclusions and Equalities between Unions

61.5.12 Cardinality

61.6 Intervals of integers

61.6.1 Finiteness

61.6.2 Cardinality

61.7 Lemmas useful with the summation operator sum

61.7.1 Disjoint Unions
61.7.2 Disjoint Intersections ........................................ 1197
61.7.3 Some Differences ........................................... 1197
61.7.4 Some Subset Conditions ................................. 1198
61.8 Generic big monoid operation over intervals ............. 1198
61.9 Summation indexed over intervals ......................... 1201
  61.9.1 Shifting bounds ........................................ 1207
  61.9.2 Telescoping sums ..................................... 1208
  61.9.3 The formula for geometric sums ....................... 1208
  61.9.4 Geometric progressions ............................... 1211
  61.9.5 The formulae for arithmetic sums ..................... 1212
  61.9.6 Division remainder .................................... 1214
61.10 Products indexed over intervals .......................... 1214
  61.10.1 Telescoping products .................................. 1215
61.11 Efficient folding over intervals ........................... 1216

62 Decision Procedure for Presburger Arithmetic .......... 1218
  62.1 The $-\infty$ and $+\infty$ Properties ....................... 1218
  62.2 The A and B sets ......................................... 1220
  62.3 Cooper’s Theorem $-\infty$ and $+\infty$ Version ........ 1224
    62.3.1 First some trivial facts about periodic sets or predicates 1224
    62.3.2 The $-\infty$ Version ................................ 1224
    62.3.3 The $+\infty$ Version ................................ 1226
  62.4 Nice facts about division by $4\div'\!a$ ............... 1230
  62.5 Try0 .................................................... 1230

63 Bindings to Satisfiability Modulo Theories (SMT) solvers
  based on SMT-LIB 2 ............................................ 1230
  63.1 A skolemization tactic and proof method .............. 1230
  63.2 Triggers for quantifier instantiation ................. 1231
  63.3 Higher-order encoding .................................. 1231
  63.4 Normalization ............................................ 1232
  63.5 Integer division and modulo for Z3 .................... 1233
  63.6 Extra theorems for veriT reconstruction .............. 1233
  63.7 Setup .................................................... 1243
  63.8 Configuration ............................................ 1243
  63.9 General configuration options ...................... 1243
  63.10 Certificates ........................................... 1244
  63.11 Tracing ................................................ 1245
  63.12 Schematic rules for Z3 proof reconstruction .......... 1245

64 Sledgehammer: Isabelle–ATP Linkup ....................... 1248
65 Setup for Lifting/Transfer for the set type

65.1 Relator and predicator properties ........................................ 1248
65.2 Quotient theorem for the Lifting package .............................. 1250
65.3 Transfer rules for the Transfer package ............................... 1251
   65.3.1 Unconditional transfer rules ....................................... 1251
   65.3.2 Rules requiring bi-unique, bi-total or right-total rela-
   tions .............................................................. 1252

66 The datatype of finite lists .................................................. 1256

66.1 Basic list processing functions .......................................... 1257
   66.1.1 List comprehension ................................................ 1264
   66.1.2 [] and (\#) ...................................................... 1272
   66.1.3 length .................................................................. 1273
   66.1.4 @ – append ................................................................ 1275
   66.1.5 map ..................................................................... 1278
   66.1.6 rev ...................................................................... 1280
   66.1.7 set ...................................................................... 1282
   66.1.8 concat ................................................................... 1285
   66.1.9 filter ..................................................................... 1287
   66.1.10 List partitioning ....................................................... 1290
   66.1.11 (!) ..................................................................... 1291
   66.1.12 list-update ............................................................. 1294
   66.1.13 last and butlast ......................................................... 1296
   66.1.14 take and drop ........................................................... 1298
   66.1.15 takeWhile and dropWhile .......................................... 1305
   66.1.16 zip ..................................................................... 1309
   66.1.17 list-all2 ................................................................. 1315
   66.1.18 List.product and product-lists .................................... 1318
   66.1.19 fold with natural argument order .................................. 1319
   66.1.20 Fold variants: foldr and foldl ..................................... 1322
   66.1.21 upt .................................................................... 1323
   66.1.22 upto: interval-list on int .......................................... 1326
   66.1.23 successively ............................................................. 1328
   66.1.24 distinct and remdups and remdups-adj ......................... 1329

66.2 distinct-adj ...................................................................... 1343
   66.2.1 insert ..................................................................... 1344
   66.2.2 List.union ............................................................... 1344
   66.2.3 find ..................................................................... 1345
   66.2.4 count-list ............................................................... 1345
   66.2.5 List.extract ............................................................. 1346
   66.2.6 removeI ................................................................. 1347
   66.2.7 removeAll ............................................................... 1348
   66.2.8 replicate ................................................................. 1349
   66.2.9 enumerate ............................................................... 1353
66.2.10 rotate1 and rotate .......................... 1354
66.2.11 nths — a generalization of (!) to sets .......... 1358
66.2.12 subseqs and List.n-lists .......................... 1360
66.2.13 splice ............................................. 1361
66.2.14 shuffles ............................................. 1361
66.2.15 Transpose ............................................. 1364
66.2.16 min and arg-min ................................. 1366
66.2.17 (In)finiteness ....................................... 1367
66.3 Sorting .................................................. 1370
  66.3.1 sorted-wrt ........................................ 1370
  66.3.2 sorted ............................................. 1372
  66.3.3 Sorting functions .................................. 1376
  66.3.4 transpose on sorted lists .......................... 1381
  66.3.5 sorted-key-list-of-set ............................. 1384
  66.3.6 lists: the list-forming operator over sets ...... 1389
  66.3.7 Inductive definition for membership ............. 1391
  66.3.8 Lists as Cartesian products ........................ 1391
66.4 Relations on Lists .................................... 1392
  66.4.1 Length Lexicographic Ordering ..................... 1392
  66.4.2 Lexicographic Ordering ............................. 1396
  66.4.3 Lexicographic combination of measure functions .......... 1405
  66.4.4 Lifting Relations to Lists: one element ........ 1405
  66.4.5 Lifting Relations to Lists: all elements .......... 1408
66.5 Size function ........................................... 1411
66.6 Monad operation ....................................... 1412
66.7 Code generation ......................................... 1412
  66.7.1 Counterparts for set-related operations .......... 1413
  66.7.2 Optimizing by rewriting ........................... 1417
  66.7.3 Pretty lists ....................................... 1419
  66.7.4 Use convenient predefined operations ............ 1420
  66.7.5 Implementation of sets by lists ................... 1421
66.8 Setup for Lifting/Transfer ................................ 1423
  66.8.1 Transfer rules for the Transfer package ........ 1423

67 Sum and product over lists .............................. 1429
  67.1 List summation ....................................... 1430
  67.2 Horner sums ......................................... 1436
  67.3 Further facts about List.n-lists ..................... 1439
  67.4 Tools setup ......................................... 1440
  67.5 List product ......................................... 1440
68 Bit operations in suitable algebraic structures ........................................ 1441
  68.1 Abstract bit structures ................................................................. 1441
  68.2 Bit operations .................................................................................. 1452
  68.3 Common algebraic structure ............................................................. 1473
  68.4 Instance int ....................................................................................... 1476
  68.5 Instance nat ....................................................................................... 1491
  68.6 Symbolic computations on numeral expressions ................................. 1497
  68.7 Symbolic computations for code generation ....................................... 1507
  68.8 More properties ................................................................................ 1511
  68.9 Bit concatenation ............................................................................. 1512
  68.10 Taking bits with sign propagation .................................................... 1514
  68.11 Key ideas of bit operations ............................................................... 1520
  68.12 Lemma duplicates and other ............................................................ 1521

69 Numeric types for code generation onto target language numerals only ...... 1526
  69.1 Type of target language integers ......................................................... 1526
  69.2 Code theorems for target language integers ........................................ 1535
  69.3 Serializer setup for target language integers ....................................... 1542
  69.4 Type of target language naturals .......................................................... 1544
  69.5 Inductive representation of target language naturals ............................ 1551
  69.6 Code refinement for target language naturals ...................................... 1551

70 A HOL random engine ............................................................................ 1553
  70.1 Auxiliary functions ............................................................................ 1554
  70.2 Random seeds .................................................................................... 1554
  70.3 Base selectors ..................................................................................... 1554
  70.4 ML interface ....................................................................................... 1556

71 Maps ....................................................................................................... 1557
  71.1 empty ............................................................................................... 1559
  71.2 map-upd ......................................................................................... 1559
  71.3 map-of .......................................................................................... 1560
  71.4 map-option related ........................................................................... 1563
  71.5 map-comp related ............................................................................ 1563
  71.6 ++ .................................................................................................. 1564
  71.7 restrict-map ..................................................................................... 1565
  71.8 map-upds ....................................................................................... 1566
  71.9 dom ............................................................................................... 1568
  71.10 ran ............................................................................................... 1571
  71.11 graph ............................................................................................ 1572
  71.12 map-le .......................................................................................... 1573
  71.13 Various ........................................................................................... 1575
72 Finite types as explicit enumerations

72.1 Class \textit{enum} ........................................ 1577
72.2 Implementations using \textit{enum} .......................... 1578
  72.2.1 Unbounded operations and quantifiers .................. 1578
  72.2.2 An executable choice operator .......................... 1579
  72.2.3 Equality and order on functions ......................... 1579
  72.2.4 Operations on relations ............................... 1580
  72.2.5 Bounded accessible part ............................... 1580
72.3 Default instances for \textit{enum} ............................ 1582
72.4 Small finite types ........................................ 1587
72.5 Closing up ............................................... 1601

73 Character and string types ..................................... 1601

73.1 Strings as list of bytes .................................... 1601
  73.1.1 Bytes as datatype ...................................... 1602
73.2 Strings as dedicated type for target language code generation 1609
  73.2.1 Logical specification ................................... 1609
  73.2.2 Syntactic representation ............................... 1611
  73.2.3 Operations .............................................. 1612
  73.2.4 Executable conversions ................................. 1614
  73.2.5 Technical code generation setup ......................... 1615
  73.2.6 Code generation utility ............................... 1618
  73.2.7 Finally .................................................. 1619

74 Reflecting Pure types into HOL ................................ 1619

75 Predicates as enumerations .................................... 1621

75.1 The type of predicate enumerations (a monad) ............... 1621
75.2 Emptiness check and definite choice ........................ 1625
75.3 Derived operations ......................................... 1627
75.4 Implementation .............................................. 1629

76 Lazy sequences ................................................ 1636

76.1 Type of lazy sequences ..................................... 1636
76.2 Code setup ................................................... 1640
76.3 Generator Sequences ......................................... 1640
  76.3.1 General lazy sequence operation ......................... 1640
  76.3.2 Small lazy typeclasses ................................ 1640
76.4 With Hit Bound Value ........................................ 1642

77 Depth-Limited Sequences with failure element ................. 1643

77.1 Depth-Limited Sequence ..................................... 1643
77.2 Positive Depth-Limited Sequence ............................ 1645
77.3 Negative Depth-Limited Sequence ........................... 1645
78 Term evaluation using the generic code generator 1647
78.1 Term representation ................................. 1648
78.1.1 Terms and class \texttt{term-of} .................... 1648
78.1.2 Syntax .......................................... 1648
78.2 Tools setup and evaluation ......................... 1649
78.3 Dedicated \texttt{term-of} instances ................ 1649
78.4 Generic reification .................................. 1650
78.5 Diagnostic ......................................... 1650

79 A simple counterexample generator performing random testing 1650
79.1 Catching Match exceptions ......................... 1651
79.2 The \texttt{random} class ............................. 1651
79.3 Fundamental and numeric types .................... 1651
79.4 Complex generators .................................. 1654
79.5 Deriving random generators for datatypes .......... 1656
79.6 Code setup ......................................... 1656

80 The Random-Predicate Monad 1656

81 Various kind of sequences inside the random monad 1658

82 A simple counterexample generator performing exhaustive testing 1662
82.1 Basic operations for exhaustive generators ........ 1662
82.2 Exhaustive generator type classes .................. 1662
82.2.1 A smarter enumeration scheme for functions over finite datatypes ......................... 1668
82.3 Bounded universal quantifiers ....................... 1675
82.4 Fast exhaustive combinators ....................... 1675
82.5 Continuation passing style functions as plus monad .... 1675
82.6 Defining generators for any first-order data type .... 1677
82.7 Defining generators for abstract types ............ 1677

83 A compiler for predicates defined by introduction rules 1678
83.1 Set membership as a generator predicate .......... 1679

84 Counterexample generator performing narrowing-based testing 1680
84.1 Counterexample generator .......................... 1680
84.1.1 Code generation setup ........................... 1680
84.1.2 Narrowing’s deep representation of types and terms . 1681
84.1.3 From narrowing’s deep representation of terms to HOL. Code-Evaluation's terms ................................. 1681
84.1.4 Auxiliary functions for Narrowing .......................... 1682
84.1.5 Narrowing’s basic operations ............................... 1682
84.1.6 Narrowing generator type class ............................ 1683
84.1.7 class is-testable ........................................... 1683
84.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way ............... 1684
84.1.9 Setting up the counterexample generator ................. 1684
84.2 Narrowing for sets ........................................... 1685
84.3 Narrowing for integers ....................................... 1685
84.4 The find-unused-assms command .............................. 1687
84.5 Closing up ..................................................... 1687

85 Program extraction for HOL 1688
85.1 Setup ......................................................... 1688
85.2 Type of extracted program .................................. 1688
85.3 Realizability ................................................ 1690
85.4 Computational content of basic inference rules ........... 1691

86 Extensible records with structural subtyping 1696
86.1 Introduction ................................................ 1696
86.2 Operators and lemmas for types isomorphic to tuples .... 1697
86.3 Logical infrastructure for records .......................... 1698
86.4 Concrete record syntax ..................................... 1704
86.5 Record package .............................................. 1705

87 Greatest common divisor and least common multiple 1705
87.1 Abstract bounded quasi semilattices as common foundation . 1705
87.2 Abstract GCD and LCM ...................................... 1708
87.3 An aside: GCD and LCM on finite sets for incomplete gcd rings ...................................................... 1726
87.4 Coprimality ................................................... 1729
87.5 GCD and LCM for multiplicative normalisation functions .. 1739
87.6 GCD and LCM on nat and int ................................ 1740
87.7 Bezout’s theorem ............................................. 1749
87.8 LCM properties on nat and int ................................ 1754
87.9 The complete divisibility lattice on nat and int .......... 1755
87.9.1 Setwise GCD and LCM for integers ....................... 1758
87.10 GCD and LCM on integer ................................... 1760
87.11 Characteristic of a semiring ................................ 1763

88 Nitpick: Yet Another Counterexample Generator for Isabelle/HOL 1767
### 89 Greatest Fixpoint (Codatatype) Operation on Bounded Natural Functors

89.1 Equivalence relations, quotients, and Hilbert’s choice

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>89.1 Equivalence relations, quotients, and Hilbert’s choice</td>
<td>1777</td>
</tr>
</tbody>
</table>

### 90 Filters on predicates

90.1 Filters

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>90.1 Filters</td>
<td>1779</td>
</tr>
<tr>
<td>90.1.1 Eventually</td>
<td>1779</td>
</tr>
<tr>
<td>90.2 Frequently as dual to eventually</td>
<td>1782</td>
</tr>
<tr>
<td>90.2.1 Finer-than relation</td>
<td>1785</td>
</tr>
<tr>
<td>90.2.2 Map function for filters</td>
<td>1791</td>
</tr>
<tr>
<td>90.2.3 Contravariant map function for filters</td>
<td>1792</td>
</tr>
<tr>
<td>90.2.4 Standard filters</td>
<td>1795</td>
</tr>
<tr>
<td>90.2.5 Order filters</td>
<td>1796</td>
</tr>
<tr>
<td>90.3 Sequentially</td>
<td>1798</td>
</tr>
<tr>
<td>90.4 Increasing finite subsets</td>
<td>1799</td>
</tr>
<tr>
<td>90.5 The cofinite filter</td>
<td>1801</td>
</tr>
<tr>
<td>90.5.1 Product of filters</td>
<td>1802</td>
</tr>
<tr>
<td>90.6 Limits</td>
<td>1807</td>
</tr>
<tr>
<td>90.7 Limits to at-top and at-bot</td>
<td>1810</td>
</tr>
<tr>
<td>90.8 Setup 'a filter for lifting and transfer</td>
<td>1813</td>
</tr>
</tbody>
</table>

### 91 Conditionally-complete Lattices

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>91.1 Preorders</td>
<td>1825</td>
</tr>
<tr>
<td>91.2 Lattices</td>
<td>1828</td>
</tr>
<tr>
<td>91.3 Conditionally complete lattices</td>
<td>1829</td>
</tr>
<tr>
<td>91.4 Complete lattices</td>
<td>1834</td>
</tr>
<tr>
<td>91.5 Instances</td>
<td>1836</td>
</tr>
</tbody>
</table>

### 92 Factorial Function, Rising Factorials

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>92.1 Factorial Function</td>
<td>1845</td>
</tr>
<tr>
<td>92.2 Pochhammer’s symbol: generalized rising factorial</td>
<td>1849</td>
</tr>
<tr>
<td>92.3 Misc</td>
<td>1854</td>
</tr>
</tbody>
</table>

### 93 Binomial Coefficients, Binomial Theorem, Inclusion-exclusion Principle

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>93.1 Binomial coefficients</td>
<td>1854</td>
</tr>
<tr>
<td>93.2 The binomial theorem (courtesy of Tobias Nipkow):</td>
<td>1858</td>
</tr>
<tr>
<td>93.3 Generalized binomial coefficients</td>
<td>1862</td>
</tr>
<tr>
<td>93.4 Summation on the upper index</td>
<td>1872</td>
</tr>
<tr>
<td>93.5 More on Binomial Coefficients</td>
<td>1879</td>
</tr>
<tr>
<td>93.6 Inclusion-exclusion principle</td>
<td>1882</td>
</tr>
<tr>
<td>93.7 Versions for unrestrictedly additive functions</td>
<td>1884</td>
</tr>
<tr>
<td>93.8 General &quot;Moebius inversion&quot; inclusion-exclusion principle</td>
<td>1886</td>
</tr>
<tr>
<td>93.9 Executable code</td>
<td>1889</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>94</td>
<td>Misc lemmas on division, to be sorted out finally</td>
</tr>
<tr>
<td>95</td>
<td>Main HOL</td>
</tr>
<tr>
<td>95.1</td>
<td>Namespace cleanup</td>
</tr>
<tr>
<td>95.2</td>
<td>Syntax cleanup</td>
</tr>
<tr>
<td>95.3</td>
<td>Lattice syntax</td>
</tr>
<tr>
<td>96</td>
<td>Archimedean Fields, Floor and Ceiling Functions</td>
</tr>
<tr>
<td>96.1</td>
<td>Class of Archimedean fields</td>
</tr>
<tr>
<td>96.2</td>
<td>Existence and uniqueness of floor function</td>
</tr>
<tr>
<td>96.3</td>
<td>Floor function</td>
</tr>
<tr>
<td>96.4</td>
<td>Ceiling function</td>
</tr>
<tr>
<td>96.4.1</td>
<td>Ceiling with numerals</td>
</tr>
<tr>
<td>96.4.2</td>
<td>Addition and subtraction of integers</td>
</tr>
<tr>
<td>96.5</td>
<td>Negation</td>
</tr>
<tr>
<td>96.6</td>
<td>Natural numbers</td>
</tr>
<tr>
<td>96.7</td>
<td>Frac Function</td>
</tr>
<tr>
<td>96.8</td>
<td>Fractional part arithmetic</td>
</tr>
<tr>
<td>96.9</td>
<td>Rounding to the nearest integer</td>
</tr>
<tr>
<td>97</td>
<td>Rational numbers</td>
</tr>
<tr>
<td>97.1</td>
<td>Rational numbers as quotient</td>
</tr>
<tr>
<td>97.1.1</td>
<td>Construction of the type of rational numbers</td>
</tr>
<tr>
<td>97.1.2</td>
<td>Representation and basic operations</td>
</tr>
<tr>
<td>97.1.3</td>
<td>Function normalize</td>
</tr>
<tr>
<td>97.1.4</td>
<td>Various</td>
</tr>
<tr>
<td>97.1.5</td>
<td>The ordered field of rational numbers</td>
</tr>
<tr>
<td>97.1.6</td>
<td>Rationals are an Archimedean field</td>
</tr>
<tr>
<td>97.2</td>
<td>Linear arithmetic setup</td>
</tr>
<tr>
<td>97.3</td>
<td>Embedding from Rationals to other Fields</td>
</tr>
<tr>
<td>97.4</td>
<td>The Set of Rational Numbers</td>
</tr>
<tr>
<td>97.5</td>
<td>Implementation of rational numbers as pairs of integers</td>
</tr>
<tr>
<td>97.6</td>
<td>Setup for Nitpick</td>
</tr>
<tr>
<td>97.7</td>
<td>Float syntax</td>
</tr>
<tr>
<td>97.8</td>
<td>Hiding implementation details</td>
</tr>
<tr>
<td>98</td>
<td>Development of the Reals using Cauchy Sequences</td>
</tr>
<tr>
<td>98.1</td>
<td>Preliminary lemmas</td>
</tr>
<tr>
<td>98.2</td>
<td>Sequences that converge to zero</td>
</tr>
<tr>
<td>98.3</td>
<td>Cauchy sequences</td>
</tr>
<tr>
<td>98.4</td>
<td>Equivalence relation on Cauchy sequences</td>
</tr>
<tr>
<td>98.5</td>
<td>The field of real numbers</td>
</tr>
<tr>
<td>98.6</td>
<td>Positive reals</td>
</tr>
<tr>
<td>98.7</td>
<td>Completeness</td>
</tr>
</tbody>
</table>
98.8 Supremum of a set of reals  ........................................ 1955
98.9 Hiding implementation details  .................................... 1956
98.10 Embedding numbers into the Reals  .............................. 1956
98.11 Embedding the Naturals into the Reals ......................... 1958
98.12 The Archimedean Property of the Reals ....................... 1959
98.13 Rationals  .......................................................... 1960
98.15 Numerals and Arithmetic  ......................................... 1963
98.16 Simple rules combining \( x + y \) and \( \theta \) .................... 1963
98.17 Lemmas about powers  ............................................. 1963
98.18 Density of the Reals  ............................................... 1964
98.19 Archimedean properties and useful consequences ............ 1964
98.20 Floor and Ceiling Functions from the Reals to the Integers 1966
98.21 Exponentiation with floor  ........................................ 1970
98.22 Implementation of rational real numbers ...................... 1970
98.23 Setup for Nitpick  .................................................. 1973
98.24 Setup for SMT  ...................................................... 1973
98.25 Setup for Argo  ...................................................... 1974

99 Topological Spaces  ........................................................................ 1974
99.1 Topological space  ....................................................... 1975
99.2 Hausdorff and other separation properties  ....................... 1978
99.3 Generators for topologies  ............................................ 1979
99.4 Order topologies  ........................................................ 1980
99.5 Setup some topologies  ................................................ 1982
  99.5.1 Boolean is an order topology  .................................... 1982
  99.5.2 Topological filters  .............................................. 1984
  99.5.3 Tendsto  ............................................................. 1990
  99.5.4 Rules about \( \text{Lim} \) .......................................... 1997
99.6 Limits on sequences  ..................................................... 1999
99.7 Monotone sequences and subsequences  .......................... 2000
  99.7.1 Definition of subsequence. ...................................... 2000
  99.7.2 Subsequence (alternative definition, (e.g. Hoskins)) .... 2002
  99.7.3 Increasing and Decreasing Series  ............................. 2007
99.8 First countable topologies  ........................................... 2007
99.9 Function limit at a point  ............................................ 2011
  99.9.1 Relation of \( \text{LIM} \) and \( \text{LIMSEQ} \) .................... 2013
99.10 Continuity  ............................................................. 2016
  99.10.1 Continuity on a set  ............................................ 2016
  99.10.2 Continuity at a point  ......................................... 2022
  99.10.3 Open-cover compactness  ..................................... 2027
99.11 Finite intersection property  ......................................... 2028
99.12 Connectedness  ........................................................ 2032
100 Linear Continuum Topologies 2038
100.1 Intermediate Value Theorem .................................. 2041
100.2 Uniform spaces ................................................. 2046
  100.2.1 Totally bounded sets ....................................... 2047
  100.2.2 Cauchy filter ............................................... 2048
  100.2.3 Uniformly continuous functions ........................... 2050

101 Product Topology 2050
  101.1 Product is a topological space ............................... 2050
         101.1.1 Continuity of operations ............................ 2053
         101.1.2 Connectedness of products .......................... 2055
         101.1.3 Separation axioms .................................... 2056
  101.2 A generic notion of the convex, affine, conic hull, or closed
       "hull". .......................................................... 2058

102 Modules 2059
  102.1 Locale for additive functions ............................... 2060

103 Subspace 2062

104 Span: subspace generated by a set 2063

105 Dependent and independent sets 2067

106 Representation of a vector on a specific basis 2070

107 Vector Spaces 2080

108 Vector Spaces and Algebras over the Reals 2116
  108.1 Real vector spaces ........................................... 2116
  108.2 Embedding of the Reals into any real-algebra-1: of-real ... 2121
  108.3 The Set of Real Numbers .................................... 2124
  108.4 Ordered real vector spaces .................................. 2126
  108.5 Real normed vector spaces ................................... 2131
  108.6 Metric spaces ................................................. 2139
  108.7 Class instances for real numbers ............................ 2143
  108.8 Extra type constraints ...................................... 2144
  108.9 Sign function ................................................. 2145
  108.10 Bounded Linear and Bilinear Operators ...................... 2146
  108.11 Filters and Limits on Metric Space ......................... 2153
         108.11.1 Limits of Sequences .................................. 2156
         108.11.2 Limits of Functions ................................ 2157
  108.12 Complete metric spaces .................................... 2158
  108.13 Cauchy sequences ............................................ 2158
  108.14 Cauchy Sequences are Convergent ........................... 2164
The set of real numbers is a complete metric space.

109 Limits on Real Vector Spaces

109.1 Filter going to infinity norm
109.1.1 Boundedness
109.1.2 Bounded Sequences
109.1.3 A Few More Equivalence Theorems for Boundedness
109.1.4 Upper Bounds and Lubs of Bounded Sequences
109.1.5 Polynomial function extremal theorem, from HOL Light

109.2 Convergence to Zero
109.2.1 Distance and norms

109.3 Topological Monoid
109.3.1 Topological group
109.3.2 Linear operators and multiplication
109.3.3 Inverse and division

109.4 Relate \( at \), \( at-left \) and \( at-right \)

109.5 Floor and Ceiling

109.6 Limits of Sequences

109.7 Convergence on sequences

109.8 More about \( filterlim \) (thanks to Wenda Li)

109.9 Power Sequences

109.10 Limits of Functions

109.11 Continuity

109.12 Uniform Continuity

109.13 Nested Intervals and Bisection – Needed for Compactness

109.14 Boundedness of continuous functions

110 Infinite Series

110.1 Definition of infinite summability
110.2 Infinite summability on topological monoids
110.3 Infinite summability on ordered, topological monoids
110.4 Infinite summability on topological monoids
110.5 Infinite summability on real normed vector spaces
110.6 Infinite summability on real normed algebras
110.7 Infinite summability on real normed fields
110.8 Telescoping
110.9 Infinite summability on Banach spaces
110.10 The Ratio Test
110.11 Cauchy Product Formula
110.12 Series on \( reals \)
11 Comprehensive Complex Theory

2543
1 Loading the code generator and related modules

theory Code-Generator
imports Pure
keywords
  print-codeproc code-thms code-deps :: diag and
  export-code code-identifier code-printing code-reserved
  code-monad code-reflect :: thy-decl and
  checking and
datatypes functions module-name file file-prefix
  constant type-constructor type-class class-relation class-instance code-module
  :: quasi-command

begin

ML-file ⟨~/src/Tools/cache-io.ML⟩
ML-file ⟨~/src/Tools/Code/code-preproc.ML⟩
ML-file ⟨~/src/Tools/Code/code-symbol.ML⟩
ML-file ⟨~/src/Tools/Code/code-thingol.ML⟩
ML-file ⟨~/src/Tools/Code/code-simp.ML⟩
ML-file ⟨~/src/Tools/Code/code-printer.ML⟩
ML-file ⟨~/src/Tools/Code/code-target.ML⟩
ML-file ⟨~/src/Tools/Code/code-name.ML⟩
ML-file ⟨~/src/Tools/Code/code-scala.ML⟩

code-datatype TYPE(′a::{})

definition holds :: prop where
  holds ≡ ((λx::prop. x) ≡ (λx. x))

lemma holds: PROP holds
  by (unfold holds-def) (rule reflexive)

code-datatype holds

lemma implies-code [code]:
  (PROP holds ⇒ PROP P) ≡ PROP P
  (PROP P ⇒ PROP holds) ≡ PROP holds

proof –
  show (PROP holds ⇒ PROP P) ≡ PROP P
  proof
    assume PROP holds ⇒ PROP P
    then show PROP P using holds .
  next
    assume PROP P
    then show PROP P .
  qed
next
show \((PROP \, P \implies \, PROP\, holds) \equiv \, PROP\, holds\)
by rule (rule holds)+

qed

ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{code-runtime}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{nbe}\.ML\rangle\)

hide-const (open) holds

end

2 The basis of Higher-Order Logic

theory \(HOL\)
imports Pure Tools.Code-Generator
keywords
  try solve-direct quickcheck print-coercions print-claset
  print-induct-rules :: diag and
  quickcheck-params :: thy-decl
abbrevs \(\forall < = \exists \leq_1\)
begin

ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{misc-legacy}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{try}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{quickcheck}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{solve-direct}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{IsaPlanner}/\text{zipper}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{IsaPlanner}/\text{isand}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{rw-inst}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Provers}/\text{hypsubst}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Provers}/\text{splitter}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Provers}/\text{classical}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Provers}/\text{blast}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Provers}/\text{clasimp}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{eqsubst}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Provers}/\text{quantifier1}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{atomize-elim}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{cong-tac}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{intuitionistic}\.ML\rangle\) setup \(\langle\text{Intuitionistic.method-setup binding (iprover)}\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{project-rule}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{subtyping}\.ML\rangle\)
ML-file \(\langle\sim\sim/\text{src}/\text{Tools}/\text{case-product}\.ML\rangle\)

ML \(\langle\text{Plugin-Name.declare-setup binding (extraction)}\rangle\)

ML \(\langle\text{Plugin-Name.declare-setup binding (quickcheck-random)}\rangle\);
THEORY “HOL”

Plugin-Name. declare-setup binding {quickcheck-exhaustive};
Plugin-Name. declare-setup binding {quickcheck-bounded-forall};
Plugin-Name. declare-setup binding {quickcheck-full-exhaustive};
Plugin-Name. declare-setup binding {quickcheck-narrowing};

ML (
Plugin-Name. define-setup binding {quickcheck}
  [plugin {quickcheck-exhaustive},
   plugin {quickcheck-random},
   plugin {quickcheck-bounded-forall},
   plugin {quickcheck-full-exhaustive},
   plugin {quickcheck-narrowing}]
)

2.1 Primitive logic

The definition of the logic is based on Mike Gordon’s technical report [2] that describes the first implementation of HOL. However, there are a number of differences. In particular, we start with the definite description operator and introduce Hilbert’s ε operator only much later. Moreover, axiom $(P → Q) → (Q → P) → (P = Q)$ is derived from the other axioms. The fact that this axiom is derivable was first noticed by Bruno Barras (for Mike Gordon’s line of HOL systems) and later independently by Alexander Maletzky (for Isabelle/HOL).

2.1.1 Core syntax

setup Axclass.class-axiomatization (binding {type}, []);
default-sort type
setup Object-Logic.add-base-sort sort {type}"

setup Proofterm.set-preproc (Proof-Rewrite-Rules.standard-preproc []))

axiomatization where fun-arity: OFCLASS(‘a ⇒ ‘b, type-class)
instance fun :: (type, type) type by (rule fun-arity)

axiomatization where itself-arity: OFCLASS(‘a itself, type-class)
instance itself :: (type) type by (rule itself-arity)

typedcl bool

judgment Trueprop :: bool ⇒ prop ((-) 5)

axiomatization implies :: [bool, bool] ⇒ bool (infixr → 25)
  and eq :: ['a, 'a] ⇒ bool
  and The :: ('a ⇒ bool) ⇒ 'a

notation (input)
\( \text{eq (infixl} = 50) \)
\( \text{notation (output)} \)
\( \text{eq (infix} = 50) \)

The input syntax for eq is more permissive than the output syntax because of the large amount of material that relies on infixl.

2.1.2 Defined connectives and quantifiers

definition True :: bool
  where True \( \equiv (\lambda x::\text{bool}. \; x) = (\lambda x. \; x)) \)

definition All :: (\( 'a \Rightarrow \text{bool} \)) \Rightarrow \text{bool} \; (\text{binder} \; \forall \; 10)
  where All P \( \equiv (\forall Q. \; (\forall x. \; P x \; \rightarrow Q) \; \rightarrow Q) \)

definition Ex :: (\( 'a \Rightarrow \text{bool} \)) \Rightarrow \text{bool} \; (\text{binder} \; \exists \; 10)
  where Ex P \( \equiv \forall Q. \; (\forall x. \; P x \; \rightarrow Q) \; \rightarrow Q \)

definition False :: bool
  where False \( \equiv (\forall P. \; P) \)

definition Not :: bool \Rightarrow bool \; (\neg - [40] 40)
  where not-def: \( \neg P \; \equiv P \; \rightarrow \text{False} \)

definition conj :: [\text{bool}, \text{bool}] \Rightarrow bool \; (\text{infixr} \; \land 35)
  where and-def: \( \land P \; \equiv (\forall R. \; (P \; \rightarrow Q) \; \rightarrow R) \)

definition disj :: [\text{bool}, \text{bool}] \Rightarrow bool \; (\text{infixr} \; \lor 30)
  where or-def: \( \lor P \; \equiv (\forall R. \; (P \; \rightarrow R) \; \rightarrow (Q \; \rightarrow R)) \)

definition Uniq :: (\( 'a \Rightarrow \text{bool} \)) \Rightarrow \text{bool}
  where Uniq P \( \equiv (\forall x \; y. \; P x \; \rightarrow P y \; \rightarrow y = x) \)

definition Ex1 :: (\( 'a \Rightarrow \text{bool} \)) \Rightarrow \text{bool}
  where Ex1 P \( \equiv (\exists x. \; P x \; \land (\forall y. \; P y \; \rightarrow y = x) \)

2.1.3 Additional concrete syntax

syntax (ASCII) -Uniq :: pttrn \Rightarrow bool \Rightarrow bool \; ((4? < -/-) [0, 10] 10)
syntax -Uniq :: pttrn \Rightarrow bool \Rightarrow bool \; ((23 \leq 1 -/-) [0, 10] 10)
translations \( \exists \leq 1. \; x. \; P \; \Rightarrow \text{CONST} \; \text{Uniq} \; (\lambda x. \; P) \)

print-translation :
[Syntax-Trans.preserve-binder-abs-tr' const-syntax Uniq] syntax-const (-Uniq)]
\( \rightarrow \) to avoid eta-contraction of body

syntax (ASCII)
-Ex1 :: pttrn \Rightarrow bool \Rightarrow bool \; ((3EX! -/-) [0, 10] 10)
THEOREY "HOL"

**syntax** (input)
-Ex1 :: pttrn ⇒ bool ⇒ bool ((∃?!/-) [0, 10] 10)
syntax -Ex1 :: pttrn ⇒ bool ⇒ bool ((∃?!/−−) [0, 10] 10)
translations ‹∃!x. P ⇌ CONST Ex1 (λx. P)›

data-translation :
[Syntax-Trans.preserve-binder-abs-tr′ const-syntax ‹Ex1› syntax-const ‹−Ex1›]

syntax
-Not-Ex :: idts ⇒ bool ⇒ bool ((∃?!/-) [0, 10] 10)
-Not-Ex1 :: pttrn ⇒ bool ⇒ bool ((∃?!/−−) [0, 10] 10)
translations
¬x. P ⇌ ¬(∃x. P)
¬!x. P ⇌ ¬(∃!x. P)

abbreviation not-equal :: [(a, 'a)] ⇒ bool (infix ≠ 50)
where x ≠ y ⇌ ¬(x = y)

notation (ASCII)
Not (∼ [40] 40) and
conj (infixr & 35) and
disj (infixr | 30) and
implies (infixr −−→ 25) and
not-equal (infix ∼= 50)

abbreviation (iff)
iff :: [bool, bool] ⇒ bool (infixr ↔ 25)
where A ↔ B ⇌ A = B

syntax -The :: [pttrn, bool] ⇒ 'a ((3THE ./ −) [0, 10] 10)
translations THE x. P ⇌ CONST The (λx. P)
data-translation :
[(const-syntax ‹The›, fn −−→ fn [Abs abs] −−→
  let val (x, t) = Syntax-Trans.atomic-abs-tr′ abs
  in Syntax.const syntax-const ‹−The› $ x $ t end)]

− To avoid eta-contraction of body

nonterminal letbinds and letbind
syntax
-let bind :: [pttrn, 'a] ⇒ letbind ((2− =/ −) 10)
  :: letbind ⇒ letbinds (-)
-let binds :: [letbind, letbinds] ⇒ letbinds ((::/ −)
-let Let :: [letbinds, 'a] ⇒ 'a ((let (·)/ in (·)) [0, 10] 10)

nonterminal case-syn and cases-syn
syntax
THEORY “HOL”

- case-syntax :: ['a, cases-syn] ⇒ 'b ((case - of/-) 10)
- case1 :: ['a, 'b] ⇒ case-syn ((2→/-) 10)
  :: case-syn ⇒ cases-syn (-)
- case2 :: [case-syn, cases-syn] ⇒ cases-syn (-/ |)

syntax (ASCII)
- case1 :: ['a, 'b] ⇒ case-syn ((2→>=/-) 10)

notation (ASCII)
  All (binder ALL 10) and
  Ex (binder EX 10)

notation (input)
  All (binder ! 10) and
  Ex (binder ? 10)

2.1.4 Axioms and basic definitions

axiomatization where
  refl: t = (t::'a) and
  subst: s = t ⇒ P s ⇒ P t and
  ext: (∀x:'a. f x ::'b = g x) ⇒ (λx. f x) = (λx. g x)
  — Extensionality is built into the meta-logic, and this rule expresses a related
  property. It is an eta-expanded version of the traditional rule, and similar to the
  ABS rule of HOL and

    the-eq-trivial: (THE x. x = a) = (a::'a)

axiomatization where
  impI: (P ⇒ Q) ⇒ P ⇒ Q and
  mp: [P ⇒ Q; P] ⇒ Q and

  True-or-False: (P = True) ∨ (P = False)

definition If :: bool ⇒ 'a ⇒ 'a ⇒ 'a (((if (-)/ then (-)/ else (-)) [0, 0, 10] 10)
  where If P x y ≡ (THE z::'a. (P = True ⇒ z = x) ∧ (P = False ⇒ z = y))

definition Let :: 'a ⇒ ('a ⇒ 'b) ⇒ 'b
  where Let s f ≡ f s

translations
  - Let (-binds b bs) e ≡ - Let b (- Let bs e)
  let x = a in e ≡ CONST Let a (λx. e)

axiomatization undefined :: 'a

class default = fixes default :: 'a
2.2 Fundamental rules

2.2.1 Equality

lemma sym: \( s = t \implies t = s \)
by (erule subst) (rule refl)

lemma ssubst: \( t = s \implies P s \implies P t \)
by (erule sym) (erule subst)

lemma trans: \[ r = s; s = t \] \( \implies r = t \)
by (erule subst)

lemma trans-sym [Pure.elim?]: \( r = s \implies t = s \implies r = t \)
by (rule trans [OF - sym])

lemma meta-eq-to-obj-eq:
assumes \( A \equiv B \)
shows \( A = B \)
unfolding assms by (rule refl)

Useful with erule for proving equalities from known equalities.

lemma box-equals: \[ [a = b; a = c; b = d] \implies c = d \]
by (iprover intro: sym trans)

For calculational reasoning:

lemma forw-subst: \( a = b \implies P b \implies P a \)
by (rule ssubst)

lemma back-subst: \( P a \implies a = b \implies P b \)
by (rule subst)

2.2.2 Congruence rules for application

Similar to AP-THM in Gordon’s HOL.

lemma fun-cong: \( (f :: 'a \Rightarrow 'b) = g \Rightarrow f x = g x \)
by (iprover intro: refl elim: subst)

Similar to AP-TERM in Gordon’s HOL and FOL’s subst-context.

lemma arg-cong: \( x = y \Rightarrow f x = f y \)
by (iprover intro: refl elim: subst)

lemma arg-cong2: \[ [a = b; c = d] \Rightarrow f a c = f b d \]
by (iprover intro: refl elim: subst)

lemma cong: \( f = g; (x:'a) = y \) \( \Rightarrow f x = g y \)
by (iprover intro: refl elim: subst)

ML \{fun cong_tac ctxt = Cong-Tac.cong_tac ctxt @{thm cong}\}
2.2.3 Equality of booleans – iff

**lemma iffD2**: \[P = Q; Q\] \(\Rightarrow\) \(P\)

by (erule ssubst)

**lemma rev-iffD2**: \[Q; P = Q\] \(\Rightarrow\) \(P\)

by (erule iffD2)

**lemma iffD1**: \(Q = P \Rightarrow Q \Rightarrow P\)

by (drule sym) (rule iffD2)

**lemma rev-iffD1**: \(Q \Rightarrow Q = P \Rightarrow P\)

by (drule sym) (rule rev-iffD2)

**lemma iffE**:  
assumes major: \(P = Q\)  
and minor: \([P \rightarrow Q; Q \rightarrow P] \Rightarrow R\)

shows \(R\)

by (iprover intro: minor impI major \[THEN\] iffD2 \[THEN\] iffD1)

2.2.4 True (1)

**lemma TrueI**: \(True\)

unfolding True-def by (rule refl)

**lemma eqTrueE**: \(P = True \Rightarrow P\)

by (erule iffD2) (rule TrueI)

2.2.5 Universal quantifier (1)

**lemma spec**: \(\forall x::'a. P x \Rightarrow P x\)

unfolding All-def by (iprover intro: eqTrueE fun-cong)

**lemma allE**:  
assumes major: \(\forall x. P x\) and minor: \(P x \Rightarrow R\)

shows \(R\)

by (iprover intro: minor major \[THEN\] spec)

**lemma all-dupE**:  
assumes major: \(\forall x. P x\) and minor: \([P x; \forall x. P x] \Rightarrow R\)

shows \(R\)

by (iprover intro: minor major major \[THEN\] spec)

2.2.6 False

Depends upon spec; it is impossible to do propositional logic before quantifiers!

**lemma FalseE**: \(False \Rightarrow P\)

unfolding False-def by (erule spec)
lemma False-neq-True: False = True ⇒ P
by (erule eqTrueE [THEN FalseE])

2.2.7 Negation

lemma notI:
  assumes P ⇒ False
  shows ¬ P
  unfolding not-def by (iprover intro: impI assms)

lemma False-not-True: False ≠ True
by (iprover intro: notI elim: False-neq-True)

lemma True-not-False: True ≠ False
by (iprover intro: notI dest: sym elim: False-neq-True)

lemma notE: [¬ P; P] ⇒ R
unfolding not-def
by (iprover intro: mp [THEN FalseE])

2.2.8 Implication

lemma impE:
  assumes P ⇒ Q Q ⇒ R
  shows R
  by (iprover intro: assms mp)

Reduces Q to P ⇒ Q, allowing substitution in P.

lemma rev-mp: [P; P ⇒ Q] ⇒ Q
by (rule mp)

lemma contrapos-nn:
  assumes major: ¬ Q
  and minor: P ⇒ Q
  shows ¬ P
  by (iprover intro: notI minor major [THEN notE])

Not used at all, but we already have the other 3 combinations.

lemma contrapos-pn:
  assumes major: Q
  and minor: P ⇒ ¬ Q
  shows ¬ P
  by (iprover intro: notI minor major notE)

lemma not-sym: t ≠ s ⇒ s ≠ t
by (erule contrapos-nn) (erule sym)

lemma eq-neq-eq-imp-neq: [x = a; a ≠ b; b = y] ⇒ x ≠ y
by (erule subst, erule ss subst, assumption)
The theory "HOL"

2.2.9 Disjunction (1)

Lemma `disjE`:

- Assumes `major: P ∨ Q`
- and `minorP: P ⇒ R`
- and `minorQ: Q ⇒ R`
- Shows `R`

by `(iprover intro: minorP minorQ impI`
  `major [unfolded or-def, THEN spec, THEN mp, THEN mp])`

2.2.10 Derivation of `iffI`

In an intuitionistic version of HOL `iffI` needs to be an axiom.

Lemma `iffI`:

- Assumes `P =⇒ Q` and `Q =⇒ P`
- Shows `P = Q`

Proof (rule `disjE` [OF `True-or-False` [of `P`]])

Assume `1: P = True`

Note `Q = assms(1)` [OF `eqTrueE` [OF `this`]]

From `1` show `?thesis`

Proof (rule `ssubst`)

From `True-or-False` [of `Q`] show `True = Q`

Proof (rule `disjE`)

Assume `Q = True`

Thus `?thesis` by (rule `sym`)

Next

Assume `Q = False`

With `Q` have `False` by (rule `rev-iffD1`)

Thus `?thesis` by (rule `FalseE`)

Qed

Qed

Next

Assume `2: P = False`

Thus `?thesis`

Proof (rule `ssubst`)

From `True-or-False` [of `Q`] show `False = Q`

Proof (rule `disjE`)

Assume `Q = True`

From `2 assms(2)` [OF `eqTrueE` [OF `this`]] have `False` by (rule `iffD1`)

Thus `?thesis` by (rule `FalseE`)

Next

Assume `Q = False`

Thus `?thesis` by (rule `sym`)

Qed

Qed

2.2.11 True (2)

Lemma `eqTrueI`: `P =⇒ P = True`
2.2.12 Universal quantifier (2)

lemma allI:
  assumes $\forall x::'a. P\ x$
  shows $\forall x. P\ x$
  unfolding All-def by (iprover intro: ext eqTrueI assms)

2.2.13 Existential quantifier

lemma exI: $P\ x \Rightarrow \exists x::'a. P\ x$
  unfolding Ex-def by (iprover intro: allI allE impI mp)

lemma exE:
  assumes major: $\exists x::'a. P\ x$
    and minor: $\forall x. P\ x \Rightarrow Q$
  shows Q
  by (rule major [unfolded Ex-def, THEN spec, THEN mp]) (iprover intro: impI [THEN allI] minor)

2.2.14 Conjunction

lemma conjI: $[P; Q] \Rightarrow P \land Q$
  unfolding and-def by (iprover intro: impl [THEN allI] mp)

lemma conjunct1: $[P \land Q] \Rightarrow P$
  unfolding and-def by (iprover intro: impl dest: spec mp)

lemma conjunct2: $[P \land Q] \Rightarrow Q$
  unfolding and-def by (iprover intro: impl dest: spec mp)

lemma conjE:
  assumes major: $P \land Q$
    and minor: $[P; Q] \Rightarrow R$
  shows R
  proof (rule minor)
    show P by (rule major [THEN conjunct1])
    show Q by (rule major [THEN conjunct2])
  qed

lemma context-conjI:
  assumes P P $\Rightarrow Q$
  shows $P \land Q$
  by (iprover intro: conjI assms)

2.2.15 Disjunction (2)

lemma disjI1: $P \Rightarrow P \lor Q$
  unfolding or-def by (iprover intro: allI impI mp)
lemma \( \text{disjI2} \): \( Q \implies P \lor Q \)
unfolding \( \text{or-def} \) by (\text{iprover intro: allI impI mp})

2.2.16 Classical logic

lemma \( \text{classical} \):
 assumes \( \neg P \implies P \)
 shows \( P \)
proof (rule \text{True-or-False [THEN disjE]})
 show \( P \) if \( P = \text{True} \)
 using that by (\text{iprover intro: eqTrueE})
 show \( P \) if \( P = \text{False} \)
proof (intro notI assms)
 assume \( P \)
 with that show \( \text{False} \)
 by (\text{iprover elim: subst})
qed
qed

lemmas \( \text{ccontr} = \text{FalseE [THEN classical]} \)

notE with premises exchanged; it discharges \( \neg R \) so that it can be used to make elimination rules.

lemma \( \text{rev-notE} \):
 assumes \( \text{premp: } P \)
 and \( \text{premnot: } \neg R \implies \neg P \)
 shows \( R \)
 by (\text{iprover intro: ccontr notE [OF premnot premp]})

Double negation law.

lemma \( \text{notnotD} \): \( \neg P \implies P \)
 by (\text{iprover intro: ccontr notE })

lemma \( \text{contrapos-pp} \):
 assumes \( \text{p1: } Q \)
 and \( \text{p2: } \neg P \implies \neg Q \)
 shows \( P \)
 by (\text{iprover intro: classical p1 p2 notE})

2.2.17 Unique existence

lemma \( \text{Uniq-I [intro?]} \):
 assumes \( x y. [P x; P y] \implies y = x \)
 shows \( \text{Uniq } P \)
unfolding \( \text{Uniq-def} \) by (\text{iprover intro: assms allI impI})

lemma \( \text{Uniq-D [dest?]} \): \( [\text{Uniq } P; P a; P b] \implies a = b \)
unfolding \( \text{Uniq-def} \) by (\text{iprover dest: spec mp})
lemma ex1I:
  assumes P a \land x. P x \implies x = a
  shows \exists !x. P x
  unfolding Ex1-def by (iprover intro: assms exI conjI allI impI)

Sometimes easier to use: the premises have no shared variables. Safe!

lemma ex-ex1I:
  assumes ex-prem: \exists x. P x
  and eq: \land x y. [P x; P y] \implies x = y
  shows \exists !x. P x
  by (iprover intro: ex-prem [THEN exE] ex1I eq)

lemma ex1E:
  assumes major: \exists !x. P x and minor: \land x. [P x; \forall y. P y \implies y = x] \implies R
  shows R
  proof (rule major [unfolded Ex1-def, THEN exE])
    show \land x. P x \land (\forall y. P y \implies y = x) \implies R
      by (iprover intro: minor elim: conjE)
  qed

lemma ex1-implies-ex: \exists !x. P x \implies \exists !x. P x
  by (iprover intro: exI elim: ex1E)

2.2.18 Classical intro rules for disjunction and existential quantifiers

lemma disjCI:
  assumes \neg Q \implies P
  shows P \lor Q
  by (rule classical) (iprover intro: assms disjI1 disjI2 notI elim: notE)

lemma excluded-middle: \neg P \lor P
  by (iprover intro: disjCI)

Case distinction as a natural deduction rule. Note that \neg P is the second
  case, not the first.

lemma case-split [case-names True False]:
  assumes P \implies Q \land P \implies Q
  shows Q
  using excluded-middle [of P]
    by (iprover intro: assms elim: disjE)

Classical implies (\implies) elimination.

lemma impCE:
  assumes major: P \implies Q
  and minor: \neg P \implies R Q \implies R
  shows R
  using excluded-middle [of P]
This version of \( \rightarrow \) elimination works on \( Q \) before \( P \). It works best for those cases in which \( P \) holds "almost everywhere". Can’t install as default: would break old proofs.

**lemma** \( \text{impCE}' \):
- **assumes** major: \( P \rightarrow Q \)
- and minor: \( Q \Rightarrow R \land P \Rightarrow R \)
- **shows** \( R \)
- **using** assms by (elim \( \text{impCE} \))

Classical \( \iff \rightarrow \) elimination.

**lemma** \( \text{iffCE} \):
- **assumes** major: \( P = Q \)
- and minor: \( [P; Q] \Rightarrow R [\neg P; \neg Q] \Rightarrow R \)
- **shows** \( R \)
- by (rule major \( \text{THEN} \ \text{iffE} \)) (iprover intro: minor elim: impCE notE)

**lemma** \( \text{exCI} \):
- **assumes** \( \forall x. \neg P x \Rightarrow P a \)
- **shows** \( \exists x. P x \)
- **by** (rule \( \text{ccontr} \)) (iprover intro: assms \( \text{exI} \) \( \text{allI} \) \( \text{notI} \) \( \text{notE} \) [of \( \exists x. P x \)])

### 2.2.19 Intuitionistic Reasoning

**lemma** \( \text{impE}' \):
- **assumes** \( I: P \rightarrow Q \)
- and \( 2: Q \Rightarrow R \)
- and \( 3: P \rightarrow Q \Rightarrow P \)
- **shows** \( R \)
- **proof** –
  - from 3 and 1 have \( P \).
  - with 1 have \( Q \) by (rule \( \text{impE} \))
  - with 2 show \( R \).
  - qed

**lemma** \( \text{allE}' \):
- **assumes** \( I: \forall x. P x \)
- and \( 2: P x \Rightarrow \forall x. P x \Rightarrow Q \)
- **shows** \( Q \)
- **proof** –
  - from 1 have \( P x \) by (rule \( \text{spec} \))
  - from this and 1 show \( Q \) by (rule 2)
  - qed

**lemma** \( \text{notE}' \):
- **assumes** \( I: \neg P \)
- and \( 2: \neg P \Rightarrow P \)
- **shows** \( R \)
proof –
  from 2 and 1 have P.
  with 1 show R by (rule notE)
qed

lemma TrueE: True \implies P \implies P.
lemma notFalseE: \neg False \implies P \implies P.

lemmas [Pure.elim!] = disjE iffE FalseE conjE exE TrueE notFalseE
  and [Pure.intro!] = iffI conjI implI TrueI notI allI refl
  and [Pure.clim 2] = allE notE’ impE’
  and [Pure.intro] = exI disjI2 disjI1

2.2.20 Atomizing meta-level connectives

axiomatization where
  eq-reflection: x = y \implies x \equiv y — admissible axiom

lemma atomize-all [atomize]: (\forall x. P x) \equiv Trueprop (\forall x. P x)
proof
  assume \forall x. P x
  then show \forall x. P x ..
next
  assume \forall x. P x
  then show \forall x. P x by (rule allE)
qed

lemma atomize-imp [atomize]: (A \implies B) \equiv Trueprop (A \rightarrow B)
proof
  assume r: A \implies B
  show A \implies B by (rule impI) (rule r)
next
  assume A \implies B and A
  then show B by (rule mp)
qed

lemma atomize-not: (A \implies False) \equiv Trueprop (\neg A)
proof
  assume r: A \implies False
  show \neg A by (rule notI) (rule r)
next
  assume \neg A and A
  then show False by (rule notE)
qed
lemma atomize-eq [atomize, code]: \( (x = y) \equiv \text{Trueprop} \ (x = y) \)
proof
  assume \( x = y \)
  show \( x = y \) by (unfold \( x = y \)) (rule refl)
next
  assume \( x = y \)
  then show \( x = y \) by (rule eq-reflection)
qed

lemma atomize-conj [atomize]: \( (A &&& B) \equiv \text{Trueprop} \ (A \wedge B) \)
proof
  assume conj: \( A &&& B \)
  show \( A \wedge B \)
  proof (rule conjI)
    from conj show \( A \) by (rule conjunctionD1)
    from conj show \( B \) by (rule conjunctionD2)
  qed
next
  assume conj: \( A \wedge B \)
  show \( A &&& B \)
  proof
    from conj show \( A \)
    from conj show \( B \)
  qed
qed

lemmas [symmetric, rulify] = atomize-all atomize-imp
and [symmetric, defn] = atomize-all atomize-imp atomize-eq

2.2.21 Atomizing elimination rules

lemma atomize-exL[atomize-elim]: \( (\forall x. \ P \ x \Rightarrow Q) \equiv ((\exists x. \ P \ x) \Rightarrow Q) \)
  by (rule equal-intr-rule) iprover+

lemma atomize-conjL[atomize-elim]: \( (A \Rightarrow B \Rightarrow C) \equiv (A \wedge B \Rightarrow C) \)
  by (rule equal-intr-rule) iprover+

lemma atomize-disjL[atomize-elim]: \( ((A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C) \equiv ((A \vee B \Rightarrow C) \Rightarrow C) \)
  by (rule equal-intr-rule) iprover+

lemma atomize-elimL[atomize-elim]: \( (\forall B. \ (A \Rightarrow B) \Rightarrow B) \equiv \text{Trueprop} \ A \)

2.3 Package setup

ML-file ⟨Tools/hologic.ML⟩
ML-file ⟨Tools/rewrite-hol-proof.ML⟩

setup ⟨Proofterm.set-preproc (Proof-Rewrite-Rules.standard-preproc Rewrite-HOL-Proof.rews)⟩
2.3.1 Sledgehammer setup

Theorems blacklisted to Sledgehammer. These theorems typically produce clauses that are prolific (match too many equality or membership literals) and relate to seldom-used facts. Some duplicate other rules.

**named-theorems** no-atp theorems that should be filtered out by Sledgehammer

2.3.2 Classical Reasoner setup

```ml
lemma imp-elem: P → Q → (¬ R → P) → (Q → R) → R
  by (rule classical) iprover

lemma swap: ¬ P → (¬ R → P) → R
  by (rule classical) iprover

lemma thin-refl: [x = x; PROP W] → PROP W.
```

ML

```ml
structure Hypsubst = Hypsubst

(val dest-eq = HOLogic.dest_eq
  val dest-Trueprop = HOLogic.dest_Trueprop
  val dest-imp = HOLogic.dest_imp
  val eq-reflection = @{thm eq-reflection}
  val rev-eq-reflection = @{thm meta-eq-to-obj-eq}
  val imp-intr = @{thm impI}
  val imp-elim = @{thm impE}
  val swap = @{thm swap}
  val subst = @{thm subst}
  val sym = @{thm sym}
  val thin-refl = @{thm thin-refl};
);
open Hypsubst;

structure Classical = Classical

(val imp-elem = @{thm imp-elem}
  val not-elem = @{thm notE}
  val swap = @{thm swap}
  val classical = @{thm classical}
  val sizef = Drule.size-of-thm
  val hyp-subst-tacs = [Hypsubst.hyp-subst-tac]
);

structure Basic-Classical: BASIC-CLASSICAL = Classical;
open Basic-Classical;
```

setup

```ml
(*prevent substitution on bool*)
```
let
fun non-bool-eq (const-name HOL.eq, Type (_, [T, -])) = T <> typ bool
| non-bool-eq - = false;
fun hyp-subst-tac' ctxt =
  SUBGOAL (fn (goal, i) =>
    if Term.exists-Const non-bool-eq goal
    then Hypsubst.hyp-subst-tac ctxt i
    else no-tac);
in
  Context-Rules.addSWrapper (fn ctxt => fn tac => hyp-subst-tac' ctxt ORELSE' tac)
  end

declare iffI [intro!]
and notI [intro!]
and impI [intro!]
and disjCI [intro!]
and conjI [intro!]
and TrueI [intro!]
and refl [intro!]

declare iffCE [elim!]
and FalseE [elim!]
and impCE [elim!]
and disjE [elim!]
and conjE [elim!]

declare ex-ex1I [intro!]
and allI [intro!]
and exI [intro]

declare exE [elim!]
  allE [elim]

ML "val HOL-cs = claset-of context"

lemma contrapos-np: \¬ Q \implies (\¬ P \implies Q) \implies P
  by (erule swap)

declare ex-ex1I [rule del, intro! 2]
  and exII [intro]

declare ext [intro]

lemmas [intro?] = ext
  and [elim?] = ex1-implies-ex

Better than ex1E for classical reasoner: needs no quantifier duplication!
lemma alt-ex1E [elim!]:
assumes major: \( \exists x. P x \)
and minor: \( \forall x. [P x; \forall y y'. P y \land P y' \rightarrow y = y'] \rightarrow R \)
shows \( R \)

proof (rule ex1E [OF major minor])
show \( \forall y y'. P y \land P y' \rightarrow y = y' \) if \( P x \) and \( \forall y. P y \rightarrow y = x \) for \( x \)
using \( \langle P x \rangle \) \& \( \langle \langle P x \rangle \rangle \) by fast

qed assumption

And again using Uniq

lemma alt-ex1E':
assumes \( \exists! x. P x /\forall x. [P x; \exists \leq 1 x. P x] \rightarrow R \)
shows \( R \)
using assms unfolding Uniq-def by fast

lemma ex1-iff-ex-Uniq: \( (\exists! x. P x) \leftrightarrow (\exists x. P x) \land (\exists \leq 1 x. P x) \)
unfolding Uniq-def by fast

ML
structure Blast = Blast
|
structure Classical = Classical
val Trueprop-const = dest-Const Const (Trueprop)
val equality-name = const-name (HOL.eq)
val not-name = const-name (Not)
val notE = @{thm notE}
val ccontr = @{thm ccontr}
val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac

val blast-tac = Blast.blast-tac;

2.3.3 THE: definite description operator

lemma the-equality [intro]:
assumes \( P a \)
and \( \forall x. P x \rightarrow x = a \)
shows \( (\text{THE } x. P x) = a \)
by (blast intro: assms trans [OF arg-cong [where \( f=\text{The} \) the-eq-trivial]])

lemma theI:
assumes \( P a \)
and \( \forall x. P x \rightarrow x = a \)
shows \( P (\text{THE } x. P x) \)
by (iprover intro: assms the-equality [THEN ssubst])

lemma theI': \( \exists! x. P x \rightarrow P (\text{THE } x. P x) \)
by (blast intro: theI)
Easier to apply than the\textit{I}: only one occurrence of $P$.

\textbf{lemma the12:}
- assumes $P \ a \ \land \ x. \ P \ x \ \Longrightarrow \ x = a \ \land \ x. \ P \ x \ \Longrightarrow \ Q \ x$
- shows $Q \ (\text{THE} \ x. \ P \ x)$
  by (iprover intro: assms the1)

\textbf{lemma the1I2:}
- assumes $\exists! x. \ P \ x \ \land \ x. \ P \ x \ \Longrightarrow \ Q \ x$
- shows $Q \ (\text{THE} \ x. \ P \ x)$
  by (iprover intro: assms(2) theI2[where $P=\text{P}$ and $Q=\text{Q}$] ex1E[OF assms(1)]) elim: allE impE)

\textbf{lemma the1-equality [elim?]:} $[\exists! x. \ P \ x; \ P \ a] \ \Longrightarrow \ (\text{THE} \ x. \ P \ x) = a$
  by blast

\textbf{lemma the1-equality':} $[\exists \leq 1 x. \ P \ x; \ P \ a] \ \Longrightarrow \ (\text{THE} \ x. \ P \ x) = a$
  unfolding Uniq-def by blast

\textbf{lemma the-sym-eq-trivial:} $(\text{THE} \ y. \ x = y) = x$
  by blast

\subsection*{2.3.4 Simpler}

\textbf{lemma eta-contract-eq:} $(\lambda s. \ f \ s) = f \ldots$

\textbf{lemma subst-all:}
- $\langle \langle \forall \ x. \ x = a \ \Longrightarrow \ PROP \ P \ x \rangle \ \equiv \ PROP \ P \ a \rangle$
- $\langle \langle \forall \ x. \ a = x \ \Longrightarrow \ PROP \ P \ x \rangle \ \equiv \ PROP \ P \ a \rangle$

proof –
- show $\langle \langle \forall \ x. \ x = a \ \Longrightarrow \ PROP \ P \ x \rangle \ \equiv \ PROP \ P \ a \rangle$
  proof (rule equal-intr-rule)
    - assume $\ast$: $\langle \forall \ x. \ x = a \ \Longrightarrow \ PROP \ P \ x \rangle$
    - show $\langle PROP \ P \ a \rangle$
      by (rule $\ast$) (rule refl)
  next
    fix $x$
    assume $\langle PROP \ P \ a \rangle$ and $\langle x = a \rangle$
    from $\langle x = a \rangle$ have $\langle x \equiv a \rangle$
      by (rule eq-reflection)
    with $\langle PROP \ P \ a \rangle$ show $\langle PROP \ P \ x \rangle$
      by simp
  qed

show $\langle \langle \forall \ x. \ a = x \ \Longrightarrow \ PROP \ P \ x \rangle \ \equiv \ PROP \ P \ a \rangle$
proof (rule equal-intr-rule)
  - assume $\ast$: $\langle \forall \ x. \ a = x \ \Longrightarrow \ PROP \ P \ x \rangle$
  - show $\langle PROP \ P \ a \rangle$
    by (rule $\ast$) (rule refl)
next
  fix $x$
assume \( \langle \text{PROP } P \alpha \rangle \) and \( \langle \alpha = x \rangle \)
from \( \langle a = x \rangle \) have \( \langle a \equiv x \rangle \)
by (rule eq-reflection)
with \( \langle \text{PROP } P \alpha \rangle \) show \( \langle \text{PROP } P x \rangle \)
by simp

qed

qed

lemma simp-thms:
shows not-not: \( (\neg \neg P) = P \)
and Not-eq-iff: \( ((\neg P) = (\neg Q)) = (P = Q) \)
and
\( (P \neq Q) = (P = (\neg Q)) \)
\( (P \lor \neg P) = \text{True} \) \( (\neg P \lor P) = \text{True} \)
\( (x = x) = \text{True} \)
and not-True-eq-False [code]: \( (\neg \text{True}) = \text{False} \)
and not-False-eq-True [code]: \( (\neg \text{False}) = \text{True} \)
and
\( (\neg P) \neq P \) \( P \neq (\neg P) \)
\( (\text{True} = P) = P \)
and eq-True: \( (P = \text{True}) = P \)
and \( (\text{False} = P) = (\neg P) \)
and eq-False: \( (P = \text{False}) = (\neg P) \)
and
\( (\text{True} \longrightarrow P) = P \) \( (\text{False} \longrightarrow P) = \text{True} \)
\( (P \longrightarrow \text{True}) = \text{True} \) \( (P \longrightarrow P) = \text{True} \)
\( (P \longrightarrow \neg P) = (\neg P) \) \( (P \longrightarrow P) = (\neg P) \)
\( (P \wedge \text{True}) = P \) \( (P \wedge P) = P \)
\( (P \wedge \text{False}) = \text{False} \) \( (\text{False} \wedge P) = \text{False} \)
\( (P \wedge P) = P \) \( (P \wedge (P \wedge Q)) = (P \wedge Q) \)
\( (P \wedge \neg P) = \text{False} \) \( (\neg P \wedge P) = \text{False} \)
\( (P \lor \text{True}) = \text{True} \) \( (\text{True} \lor P) = \text{True} \)
\( (P \lor \text{False}) = P \) \( (P \lor P) = P \)
\( (P \lor (P \lor Q)) = (P \lor Q) \) and
\( (\forall x. \ P) = P \) \( (\exists x. \ P) = P \)
\( (\exists x. \ t = x) \equiv (\exists x. \ t = x) = x \)
and
\( \forall x. \ t = x \wedge P \)
\( \forall x. \ t = x \wedge P \)
\( \forall x. \ t = x \rightarrow P \)
\( \forall x. \ t = x \rightarrow P \)
\( \forall x. \ t \neq x \equiv \text{False} \)
by (blast, blast, blast, blast, blast, iprove+)

lemma disj-absorb: \( A \lor A \leftrightarrow A \)
by blast

lemma disj-left-absorb: \( A \lor (A \lor B) \leftrightarrow A \lor B \)
by blast
These two are specialized, but \textit{imp-disj-not1} is useful in \textit{Auth/Yahalom}.

\textbf{lemma} \textit{imp-disj-not1}: \( (P \rightarrow Q \rightarrow R) \iff (\neg Q \rightarrow P \rightarrow R) \) \textbf{by blast}

\textbf{lemma} \textit{imp-disj-not2}: \( (P \rightarrow Q \rightarrow R) \iff (\neg R \rightarrow P \rightarrow Q) \) \textbf{by blast}

\textbf{lemma} \textit{imp-disj1}: \( (P \rightarrow Q \rightarrow R) \iff (P \rightarrow Q \rightarrow Q') \) \textbf{by blast}

\textbf{lemma} \textit{imp-disj2}: \( (Q \rightarrow (P \rightarrow R)) \iff (P \rightarrow Q \rightarrow R) \) \textbf{by blast}

\textbf{lemma} \textit{imp-cong}: \( (P = P') \implies (P' \implies (Q = Q')) \implies ((P \rightarrow Q) \iff (P' \rightarrow Q')) \) \textbf{by impover}
theory "HOL"

lemma de-Morgan-disj: \( \neg (P \lor Q) \iff \neg P \land \neg Q \) by iprover
lemma de-Morgan-conj: \( \neg (P \land Q) \iff \neg P \lor \neg Q \) by blast
lemma not-imp: \( \neg (P \rightarrow Q) \iff P \land \neg Q \) by blast
lemma not-iff: \( P \neq Q \iff (P \iff \neg Q) \) by blast
lemma disj-not1: \( P \lor \neg Q \iff (P \rightarrow Q) \) by blast
lemma disj-not2: \( P \lor \neg Q \iff (Q \rightarrow P) \) by blast
lemma imp-conv-disj: \( P \rightarrow Q \iff (\neg P) \lor Q \) by blast
lemma disj-imp: \( P \lor Q \iff \neg P \rightarrow Q \) by blast

lemma iff-conv-conj-imp: \( P \iff Q \iff (P \iff Q) \land (Q \iff P) \) by iprover

lemma cases-simp: \( (P \rightarrow Q) \land (\neg P \rightarrow Q) \iff Q \)
— Avoids duplication of subgoals after if-split, when the true and false
— cases boil down to the same thing.
  by blast
lemma not-all: \( \neg (\forall x. P x) \iff (\exists x. \neg P x) \) by blast
lemma imp-all: \( (\forall x. P x) \rightarrow Q \iff (\exists x. P x \rightarrow Q) \) by blast
lemma not-ex: \( \neg (\exists x. P x) \iff (\forall x. \neg P x) \) by iprover
lemma imp-ex: \( (\exists x. P x) \rightarrow Q \iff (\forall x. P x \rightarrow Q) \) by iprover
lemma all-not-ex: \( (\forall x. P x) \iff \neg (\exists x. \neg P x) \) by blast

declare All-def [no-atp]

lemma ex-disj-distrib: \( (\exists x. P x \lor Q x) \iff (\exists x. P x) \lor (\exists x. Q x) \) by iprover
lemma all-conj-distrib: \( (\forall x. P x \land Q x) \iff (\forall x. P x) \land (\forall x. Q x) \) by iprover
lemma all-imp-conj-distrib: \( (\forall x. P x \rightarrow Q x \land R x) \iff (\forall x. P x \rightarrow Q x) \land (\forall x. P x \rightarrow R x) \)
  by iprover

The \( \land \) congruence rule: not included by default! May slow rewrite proofs
down by as much as 50%
lemma conj-cong: \( P = P' \Longrightarrow (P' \Longrightarrow Q = Q') \Longrightarrow (P \land Q) = (P' \land Q') \)
  by iprover
lemma rev-conj-cong: \( Q = Q' \Longrightarrow (Q' \Longrightarrow P = P') \Longrightarrow (P \land Q) = (P' \land Q') \)
  by iprover

The \( \mid \) congruence rule: not included by default!
lemma disj-cong: \( P = P' \Longrightarrow (\neg P' \Longrightarrow Q = Q') \Longrightarrow (P \lor Q) = (P' \lor Q') \)
  by blast

if-then-else rules
lemma if-True [code]: \( (\text{if True then } x \text{ else } y) = x \)
  unfolding If-def by blast
lemma if-False [code]: \( (\text{if False then } x \text{ else } y) = y \)
unfolding If-def by blast

lemma if-P: P \implies (if P then x else y) = x
  unfolding If-def by blast

lemma if-not-P: \neg P \implies (if P then x else y) = y
  unfolding If-def by blast

lemma if-split: P (if Q then x else y) = ((Q \implies P x) \land (\neg Q \implies P y))
proof (rule case-split [of Q])
  show ?thesis if Q
    using that by (simplesubst if-P) blast+
  show ?thesis if \neg Q
    using that by (simplesubst if-not-P) blast+
qed

lemma if-split-asm: P (if Q then x else y) = (\neg ((Q \land \neg P x) \lor (\neg Q \land \neg P y)))
  by (simplesubst if-split) blast

lemmas if-splits [no-atp] = if-split if-split-asm

lemma if-cancel: (if c then x else x) = x
  by (simplesubst if-split) blast

lemma if-eq-cancel: (if x = y then y else x) = x
  by (simplesubst if-split) blast

lemma if-bool-eq-conj: (if P then Q else R) = ((P \implies Q) \land (\neg P \implies R))
  — This form is useful for expanding if-s on the RIGHT of the \implies symbol.
  by (rule if-split)

lemma if-bool-eq-disj: (if P then Q else R) = ((P \land Q) \lor (\neg P \land R))
  — And this form is useful for expanding if-s on the LEFT.
  by (simplesubst if-split) blast

lemma Eq-TrueI: P \implies P \equiv True
  unfolding atomize-eq by iprover
lemma Eq-FalseI: \neg P \implies P \equiv False
  unfolding atomize-eq by iprover

let rules for simproc

lemma Let-folded: f x \equiv g x \implies Let x f \equiv Let x g
  by (unfold Let-def)

lemma Let-unfold: f x \equiv g \implies Let x f \equiv g
  by (unfold Let-def)

The following copy of the implication operator is useful for fine-tuning congruence rules. It instructs the simplifier to simplify its premise.

definition simp-implies :: prop \Rightarrow prop \Rightarrow prop (infixr =simp=> 1)
where simp-implies \equiv (\rightarrow)

**lemma** simp-impliesI:
assumes PQ: (PROP P \rightarrow PROP Q)
shows PROP P =simp=> PROP Q
unfolding simp-implies-def
by (iprover intro: PQ)

**lemma** simp-impliesE:
assumes PQ: PROP P =simp=> PROP Q
and P: PROP P
and QR: PROP Q \rightarrow PROP R
shows PROP R
by (iprover intro: QR P PQ [unfolded simp-implies-def])

**lemma** simp-implies-cong:
assumes PP':PROP P \equiv PROP P'
and P'Q': PROP P' \rightarrow (PROP Q \equiv PROP Q')
shows (PROP P =simp=> PROP Q) \equiv (PROP P' =simp=> PROP Q')
unfolding simp-implies-def
proof (rule equal-intr-rule)
assume PQ: PROP P \rightarrow PROP Q
and P': PROP P'
from PP'[symmetric] and P' have PROP P
  by (rule equal-elim-rule1)
then have PROP Q by (rule PQ)
with P'Q'[OF P'] show PROP Q' by (rule equal-elim-rule1)
next
assume P'Q': PROP P' \rightarrow PROP Q'
and P: PROP P
from PP' and P have P': PROP P' by (rule equal-elim-rule1)
then have PROP Q' by (rule P'Q')
with P'Q'[OF P', symmetric] show PROP Q
  by (rule equal-elim-rule1)
qed

**lemma** uncurry:
assumes P \rightarrow Q \rightarrow R
shows P \land Q \rightarrow R
using assms by blast

**lemma** iff-allI:
assumes \(\forall x. P \, x = Q \, x\)
shows (\forall x. P \, x) = (\forall x. Q \, x)
using assms by blast

**lemma** iff-exI:
assumes \(\exists x. P \, x = Q \, x\)
shows (\exists x. P \, x) = (\exists x. Q \, x)
using assms by blast

lemma all-comm: (∀ x y. P x y) = (∀ y x. P x y)
  by blast

lemma ex-comm: (∃ x y. P x y) = (∃ y x. P x y)
  by blast

ML-file ‹Tools/simpdata.ML›
ML ‹open Simpdata›

setup ‹
  map-theory-simpset (put-simpset HOL-basic-ss) #>
  Simplifier.method-setup Splitter.split-modifiers
 ›

simproc-setup defined-Ex (∃ x. P x) = ‹K Quantifier1.rearrange-Ex›
simproc-setup defined-All (∀ x. P x) = ‹K Quantifier1.rearrange-All›
simproc-setup defined-all (⋀ x. PROP P x) = ‹K Quantifier1.rearrange-all›

Simproc for proving (y = x) ≡ False from premise ¬ (x = y):

simproc-setup neq (x = y) = ‹
  let
    val neq-to-EQ-False = @{thm not-sym} RS @{thm Eq-FalseI};
    fun is-neq eq lhs rhs thm =
      (case Thm.prop-of thm of
        - $ (Not ($ (eq $ l $ r'))) => Not = HOLogic.Not andalso eq $ eq andalso
           r' aconv lhs andalso l' aconv rhs
        | - => false);
    fun proc ss ct =
      (case Thm.term-of ct of
        eq $ lhs $ rhs =>
          (case find-first (is-neq eq lhs rhs) (Simplifier.prems-of ss) of
            SOME thm => SOME (thm RS neq-to-EQ-False)
            | NONE => NONE)
        | - => NONE);
  in K proc end
 ›

simproc-setup let-simp (Let x f) = ‹
  let
    fun count-loose (Bound i) k = if i >= k then 1 else 0
    | count-loose (s $ t) k = count-loose s k + count-loose t k
    | count-loose (Abs (-, -, t)) k = count-loose t (k + 1)
    | count-loose - - = 0;
    fun is-trivial-let (Const (const-name‹Let›, -) $ x $ t) =
      (case t of
        Abs (-, -, t') => count-loose t' 0 <= 1
      | - => false)
  in K proc end
 ›
in

K (fn ctxt => fn ct =>
    if is-trivial-let (Thm.term-of ct)
    then SOME @{thm Let-def} (*no or one occurrence of bound variable*)
    else
        let (*Norbert Schirmer's case*)
            val t = Thm.term-of ct;
            val (t', ctxt') = yield-singleton (Variable.import-terms false) t ctxt;
        in Option.map (hd o Variable.export ctxt' ctxt o single)
            (case t' of Const (const-name ‹Let›, -) $ x $ f => (* x and f are already in normal form *)
                if is-Free x orelse is-Bound x orelse is-Const x
                then SOME @{thm Let-unfold}
                else let
                    val n = case f of (Abs (x, -, -)) => x | - => x;
                    val cx = Thm.cterm-of ctxt x;
                    val xT = Thm.typ-of-cterm cx;
                    val cf = Thm.cterm-of ctxt f;
                    val fx-g = Simplifier.rewrite ctxt (Thm.apply cf cx);
                    val (- $ - $ g) = Thm.prop-of fx-g;
                    val g' = abstract-over (x, g);
                    val abs-g' = Abs (n, xT, g');
                in
                    if g aconv g' then
                        let
                            val rl = infer-instantiate ctxt
                                [[(f, 0), cf], ((x, 0), cx)] @{thm Let-unfold};
                            in SOME (rl OF [fx-g]) end
                        else if (Envir.beta-eta-contract f) aconv (Envir.beta-eta-contract abs-g')
                            then NONE (*avoid identity conversion*)
                        else let
                            val g'x = abs-g' $ x;
                            val g-g'x = Thm.symmetric (Thm.beta-conversion false (Thm.cterm-of ctxt g'x));
                        in
                            val rl =
                                infer-instantiate ctxt
                                    [[(f, 0), Thm.cterm-of ctxt f],
                                        ((x, 0), cx),
                                        ((g, 0), Thm.cterm-of ctxt abs-g')];
                            in SOME (rl OF [Thm.transitive fx-g g-g'x]) end
                        end
                    end
                end
            end
        end
    end
| - => true);
lemma True-implies-equals: (True ⇒ PROP P) ≡ PROP P
proof
  assume True ⇒ PROP P
  from this [OF TrueI] show PROP P .
next
  assume PROP P
  then show PROP P .
qed

lemma implies-True-equals: (PROP P ⇒ True) ≡ Trueprop True
  by standard (intro TrueI)

lemma False-implies-equals: (False ⇒ P) ≡ Trueprop True
  by standard simp-all

lemma implies-False-swap:
  (False =⇒ PROP P =⇒ PROP Q) ≡ (PROP P =⇒ False =⇒ PROP Q)
  by (rule swap-prems-eq)

ML

fun eliminate-false-implies ct =
  let
    val (prems, concl) = Logic.strip-horn (Thm.term_of ct)
    fun go n =
      if n > 1 then
        Conv.rewr-conv @{thm Pure.swap-prems-eq}
        then-conv Conv.arg-conv (go (n - 1))
        then-conv Conv.rewr-conv @{thm HOL.implies-True-equals}
      else
        Conv.rewr-conv @{thm HOL.False-implies-equals}
    in
      case concl of
        Const (@{const-name HOL.Trueprop}, -) $ - => SOME (go (length prems))
        _ => NONE
    end
  end

simproc-setup eliminate-false-implies (False ⇒ PROP P) = {K (K eliminate-false-implies)}

lemma ex-simps:
  λ P Q. (∃ x. P x ∧ Q) = (∃ x. P x) ∧ Q
  λ P Q. (∃ x. P ∧ Q x) = (P ∧ (∃ x. Q x))
  λ P Q. (∃ x. P x ∨ Q) = (∃ x. P x) ∨ Q
  λ P Q. (∃ x. P ∨ Q x) = (P ∨ (∃ x. Q x))
\( \forall P Q. (\exists x. P x \rightarrow Q) = ((\forall x. P x) \rightarrow Q) \)
\( \forall P Q. (\exists x. P \rightarrow Q x) = (P \rightarrow (\exists x. Q x)) \)

— Miniscoping: pushing in existential quantifiers.

by (iprover | blast)+

lemma all-simps:
\( \forall P Q. (\forall x. P x \land Q) = ((\forall x. P x) \land Q) \)
\( \forall P Q. (\forall x. P \land Q x) = (P \land (\forall x. Q x)) \)
\( \forall P Q. (\forall x. P x \lor Q) = ((\forall x. P x) \lor Q) \)
\( \forall P Q. (\forall x. P \lor Q x) = (P \lor (\forall x. Q x)) \)
\( \forall P Q. (\forall x. P x \rightarrow Q) = ((\exists x. P x) \rightarrow Q) \)
\( \forall P Q. (\forall x. P \rightarrow Q x) = (P \rightarrow (\forall x. Q x)) \)

— Miniscoping: pushing in universal quantifiers.

by (iprover | blast)+

lemmas [simp] =
triv-forall-equality — prunes params
True-implies-equals implies-True-equals — prune True in asms
False-implies-equals — prune False in asms
if-True
if-False
if-cancel
if-eq-cancel
imp-disjL — In general it seems wrong to add distributive laws by default: they might cause exponential blow-up. But imp-disjL has been in for a while and cannot be removed without affecting existing proofs. Moreover, rewriting by \((P \lor Q \rightarrow R) = ((P \rightarrow R) \land (Q \rightarrow R))\) might be justified on the grounds that it allows simplification of \(R\) in the two cases.
conj-assoc
disj-assoc
de-Morgan-conj
de-Morgan-disj
imp-disj1
imp-disj2
not-imp
disj-not1
not-all
not-ex
cases-simp
the-eq-trivial
the-sym-eq-trivial
ex-simps
all-simps
simp-thms
subst-all

lemmas [cong] = imp-cong simp-implies-cong
lemmas [split] = if-split
Simplifies $x$ assuming $c$ and $y$ assuming $\neg c$.

**lemma if-cong:**
- **assumes** $b = c$
- and $c \Rightarrow x = u$
- and $\neg c \Rightarrow y = v$
- **shows** $(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$
- **using** `assms by simp`

Prevents simplification of $x$ and $y$: faster and allows the execution of functional programs.

**lemma if-weak-cong [cong]:**
- **assumes** $b = c$
- **shows** $(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } x \text{ else } y)$
- **using** `assms by (rule arg-cong)`

Prevents simplification of $t$: much faster

**lemma let-weak-cong:**
- **assumes** $a = b$
- **shows** $(\text{let } x = a \text{ in } t x) = (\text{let } x = b \text{ in } t x)$
- **using** `assms by (rule arg-cong)`

To tidy up the result of a simproc. Only the RHS will be simplified.

**lemma eq-cong2:**
- **assumes** $u = u'$
- **shows** $(t \equiv u) \equiv (t \equiv u')$
- **using** `assms by simp`

**lemma if-distrib:** $f \ (\text{if } c \text{ then } x \text{ else } y) = (\text{if } c \text{ then } f x \text{ else } f y)$
- **by** `simp`

**lemma if-distribR:** $(\text{if } b \text{ then } f \text{ else } g) \ x = (\text{if } b \text{ then } f \text{ else } g \ x)$
- **by** `simp`

**lemma all-if-distrib:** $(\forall x. \text{if } x = a \text{ then } P x \text{ else } Q x) \iff P a \land (\forall x. \ x \neq a \rightarrow Q x)$
- **by** `auto`

**lemma ex-if-distrib:** $(\exists x. \text{if } x = a \text{ then } P x \text{ else } Q x) \iff P a \lor (\exists x. \ x \neq a \land Q x)$
- **by** `auto`

**lemma if-if-eq-conj:** $(\text{if } P \text{ then } P \text{ if } Q \text{ then } x \text{ else } y \text{ else } y) = (\text{if } P \land Q \text{ then } x \text{ else } y)$
- **by** `simp`

As a simplification rule, it replaces all function equalities by first-order equalities.
lemma fun-eq-iff: \( f = g \iff (\forall x. f x = g x) \)
by auto

2.3.5 Generic cases and induction

Rule projections:

ML ‹
structure Project-Rule = Project-Rule
{
  val conjunct1 = @{thm conjunct1}
  val conjunct2 = @{thm conjunct2}
  val mp = @{thm mp}
};
›

countext begin

qualified definition induct-forall P \equiv \forall x. P x
qualified definition induct-implies A B \equiv A \to B
qualified definition induct-equal x y \equiv x = y
qualified definition induct-conj A B \equiv A \land B
qualified definition induct-true \equiv True
qualified definition induct-false \equiv False

lemma induct-forall-eq: \((\forall x. P x) \equiv \text{Trueprop (induct-forall (\\lambda x. P x))}\)
by (unfold atomize-all induct-forall-def)

lemma induct-implies-eq: \((A \to B) \equiv \text{Trueprop (induct-implies A B)}\)
by (unfold atomize-imp induct-implies-def)

lemma induct-equal-eq: \((x \equiv y) \equiv \text{Trueprop (induct-equal x y)}\)
by (unfold atomize-eq induct-equal-def)

lemma induct-conj-eq: \((A \land B) \equiv \text{Trueprop (induct-conj A B)}\)
by (unfold atomize-conj induct-conj-def)

lemmas induct-atomize' = induct-forall-eq induct-implies-eq induct-conj-eq
lemmas induct-atomize = induct-atomize' induct-equal-eq
lemmas induct-rulify' [symmetric] = induct-atomize'
lemmas induct-rulify [symmetric] = induct-atomize
lemmas induct-rulify-fallback =
  induct-forall-def induct-implies-def induct-equal-def induct-conj-def
  induct-true-def induct-false-def

lemma induct-forall-conj: induct-forall (\\lambda x. induct-conj (A x) (B x)) =
  induct-conj (induct-forall A) (induct-forall B)
by (unfold induct-forall-def induct-conj-def) iprove
lemma induct-implies-conj: induct-implies C (induct-conj A B) =
    induct-conj (induct-implies C A) (induct-implies C B)
by (unfold induct-implies-def induct-conj-def) iprover

lemma induct-conj-curry: (induct-conj A B \implies PROP C) \equiv (A \implies B \implies PROP C)
proof
    assume r: induct-conj A B \implies PROP C
    assume ab: A B
    show PROP C by (rule r) (simp add: induct-conj-def ab)
next
    assume r: A \implies B \implies PROP C
    assume ab: induct-conj A B
    show PROP C by (rule r) (simp-all add: ab [unfolded induct-conj-def])
qed

lemmas induct-conj = induct-forall-conj induct-implies-conj induct-conj-curry

lemma induct-trueI: induct-true
    by (simp add: induct-true-def)

Method setup.

ML-file \(\sim\)/src/Tools/induct.ML
ML (in structure Induct = Induct

val cases-default = @{thm case-split}
val atomize = @{thms induct-atomize}
val rulify = @{thms induct-rulify'}
val rulify-fallback = @{thms induct-rulify-fallback}
val equal-def = @{thm induct-equal-def}

fun dest-def (Const (const-name 'induct-equal', _) $ t $ u) = SOME (t, u)
| dest-def = NONE

fun trivial-tac ctxt = match-tac ctxt @{thms induct-trueI}
|
)

ML-file \(\sim\)/src/Tools/induction.ML

simproc-setup passive swap-induct-false (induct-false \implies PROP P \implies PROP Q) =
\langle fn - => fn - => fn ct =>>
    (case Thm.term_of ct of
      - $ (P as $ Const 'induct-false') $ (- $ Q $ -) =>
        if P <> Q then SOME Drule.swap-prems-eq else NONE
      | - => NONE);

simproc-setup passive induct-equal-conj-curry (induct-conj P Q \implies PROP R) =
THEORY "HOL"

\begin{verbatim}
<fn -=> fn -=> fn ct =>
  (case Thm.term-of ct of
   - $ (- $ P) $ - =>
     let
       fun is-conj Const-〈induct-conj for P Q; = is-conj P andalso is-conj Q
       | is-conj Const-〈induct-equal - for - => = true
       | is-conj Const-〈induct-true> = true
       | is-conj Const-〈induct-false> = true
       in if is-conj P then SOME @{thm induct-conj-curry} else NONE end
     in
     \end{verbatim}

\begin{verbatim}
declaration ( K (Induct.map-simpset (fn ss => ss
    addsimprocs [simp proc (swap-induct-false), simp proc (induct-equal-conj-curry)]
    |> Simplifier.set-mksimps (fn ctxt =>
        Simpdata.mksimps Simpdata.mksimps-pairs ctxt #>
        map (rewrite-rule ctxt (map Thm.symmetric @{thms induct-rulify-fallback})))))))
\end{verbatim}

Pre-simplification of induction and cases rules

\begin{verbatim}
lemma [induct-simp]: (∀x. induct-equal x t → PROP P x) ≡ PROP P t
  unfolding induct-equal-def
proof
  assume r: (∀x. x = t → PROP P x
  show PROP P t by (rule r ![OF refl])
next
  fix x
  assume PROP P t x = t
  then show PROP P x by simp
qed

lemma [induct-simp]: (∀x. induct-equal t x → PROP P x) ≡ PROP P t
  unfolding induct-equal-def
proof
  assume r: (∀x. t = x → PROP P x
  show PROP P t by (rule r ![OF refl])
next
  fix x
  assume PROP P t t = x
  then show PROP P x by simp
qed

lemma [induct-simp]: (induct-false → PROP P) ≡ Trueprop induct-true
  unfolding induct-false-def induct-true-def
by (iprover intro: equal-intr-rule)

lemma [induct-simp]: (induct-true → PROP P) ≡ PROP P
  unfolding induct-true-def
\end{verbatim}
proof
  assume True ⇒ PROP P
  then show PROP P using TrueI.
next
  assume PROP P
  then show PROP P.
qed

lemma [induct-simp]: (PROP P ⇒ induct-true) ≡ Trueprop induct-true
  unfolding induct-true-def
  by (iprover intro: equal-intr-rule)

lemma [induct-simp]: (∀x::'a::{}. induct-true) ≡ Trueprop induct-true
  unfolding induct-true-def
  by (iprover intro: equal-intr-rule)

lemma [induct-simp]: induct-implies induct-true P ≡ P
  by (simp add: induct-implies-def induct-true-def)

lemma [induct-simp]: x = x ↔ True
  by (rule simp-thms)
end

ML-file (~~/src/Tools/induct-tacs.ML)

2.3.6 Coherent logic

ML-file (~~/src/Tools/coherent.ML)
ML <
structure Coherent = Coherent
{
  val atomize-elimL = @{thm atomize-elimL};
  val atomize-exL = @{thm atomize-exL};
  val atomize-conjL = @{thm atomize-conjL};
  val atomize-disjL = @{thm atomize-disjL};
  val operator-names = [const-name HOL.disj, const-name HOL.conj, const-name Ex];
};
>

2.3.7 Reorienting equalities

ML <
signature REORIENT-PROC =
  sig
    val add : (term ⇒ bool) ⇒ theory ⇒ theory
    val proc : Simplifier.proc
  end;

structure Reorient-Proc : REORIENT-PROC =
struct
structure Data = Theory-Data
(type T = ((term => bool) * stamp) list;
val empty = [];
fun merge data : T = Library.merge (eq-snd (op =)) data;
fun add m = Data.map (cons (m, stamp ()));
fun matches thy t = exists (fn (m, -) => m t) (Data.get thy);

val meta-reorient = @{thm eq-commute [THEN eq-reflection]};

fun proc ctxt ct =
let
  val thy = Proof-Context.theory-of ctxt;
in
  case Thm.term-of ct of
    (s t u) => if matches thy u then NONE else SOME meta-reorient
    | _ => NONE
  end;
end;

2.4 Other simple lemmas and lemma duplicates

lemma eq-iff-swap: (x = y <-> P) --- (y = x <-> P)
by blast

lemma all-cong1: (\forall x. P x = P' x) --- (\forall x. P x) = (\forall x. P' x)
by auto

lemma ex-cong1: (\exists x. P x = P' x) --- (\exists x. P x) = (\exists x. P' x)
by auto

lemma all-cong: (\forall x. Q x --- P x = P' x) --- (\forall x. Q x --- P x) = (\forall x. Q x --- P' x)
by auto

lemma ex-cong: (\exists x. Q x --- P x = P' x) --- (\exists x. Q x \land P x) = (\exists x. Q x \land P' x)
by auto

lemma ex1-eq [iff]: \exists! x. x = t \exists! x. t = x
by blast

lemma choice-eq: (\forall x. \exists! y. P x y) = (\exists! f. \forall x. P x (f x)) (is ?lhs = ?rhs)
proof (intro iffI allI)
  assume L: ?lhs
  then have $\forall x. P x (\text{THE } y. P x y)$
  by (best intro: theI)

show \( \forall x. P x (f x) \) and \( \forall x. P x (y x) \implies y = f \)
by (blast elim: ex1E)

show \( \exists! y. P x y \)
proof (rule ex1I)
  show \( P x f x \)
    using \( f \) by blast

qed
qed

lemmas eq-sym-conv = eq-commute

lemma nnf-simps:
  \((\neg (P \land Q)) = (\neg P \lor \neg Q)\)
  \((\neg (P \lor Q)) = (\neg P \land \neg Q)\)
  \((P \implies Q) = (\neg P \lor Q)\)
  \((P = Q) = ((P \land Q) \lor (\neg P \land \neg Q))\)
  \((\neg (P = Q)) = ((P \land \neg Q) \lor (\neg P \land Q))\)
  \((\neg \neg P) = P\)
by blast+

2.5 Basic ML bindings

ML (∗
val FalseE = @{thm FalseE}
val Let-def = @{thm Let-def}
val TrueI = @{thm TrueI}
val allE = @{thm allE}
val allI = @{thm allI}
val all-dupE = @{thm all-dupE}
val arg-cong = @{thm arg-cong}
val box-equals = @{thm box-equals}
val ccontr = @{thm ccontr}
val classical = @{thm classical}
val conjE = @{thm conjE}
val conjI = @{thm conjI}
val conjunct1 = @{thm conjunct1}∗)
locale cnf

begin

lemma clause2raw-notE: [P; ¬P] ⇒ False by auto
lemma clause2raw-not-disj: [¬P; ¬Q] ⇒ ¬(P ∨ Q) by auto
lemma clause2raw-not-not: P ⇒ ¬¬P by auto

lemma iff-refl: (P::bool) = P by auto
lemma iff-trans: P = Q ⇒ R by auto
lemma conj-cong: P = P'; Q = Q' ⇒ (P ∧ Q) = (P' ∧ Q') by auto
lemma disj-cong: P = P'; Q = Q' ⇒ (P ∨ Q) = (P' ∨ Q') by auto
THEORY “HOL”

```
lemma make-nnf-imp: \( \parallel (P \land Q) \parallel = P_\parallel; Q = Q_\parallel \parallel \rightarrow (P \rightarrow Q) = (P_\parallel \lor Q_\parallel) \) by auto

lemma make-nnf-iff: \( \parallel P = P_\parallel; (\neg P) = NP; Q = Q_\parallel; (\neg Q) = NQ \parallel \rightarrow (P \leftrightarrow Q) = ((P_\parallel \lor NQ) \land (NP \lor Q_\parallel)) \) by auto

lemma make-nnf-not-false: \( (\neg False) = True \) by auto

lemma make-nnf-not-true: \( (\neg True) = False \) by auto

lemma make-nnf-not-disj: \( \parallel (\neg P) = P_\parallel; (\neg Q) = Q_\parallel \parallel \rightarrow (\neg(P \lor Q)) = (P_\parallel \lor Q_\parallel) \) by auto

lemma make-nnf-not-imp: \( \parallel P = P_\parallel; (\neg Q) = Q_\parallel \parallel \rightarrow (\neg(P \rightarrow Q)) = ((P_\parallel \lor Q_\parallel) \land (NP \lor NQ)) \) by auto

lemma make-nnf-not-iff: \( \parallel P = P_\parallel; (\neg P) = NP; Q = Q_\parallel; (\neg Q) = NQ \parallel \rightarrow (\neg(P = Q)) = ((P_\parallel \lor Q_\parallel) \land (NP \lor NQ)) \) by auto

lemma simp-TF-conj-True-l: \( \parallel P = True; Q = Q_\parallel \parallel \rightarrow (P \land Q) = Q_\parallel \) by auto

lemma simp-TF-conj-True-r: \( \parallel P = P_\parallel; Q = True \parallel \rightarrow (P \land Q) = P_\parallel \) by auto

lemma simp-TF-conj-False-l: \( P = False \rightarrow (P \land Q) = False \) by auto

lemma simp-TF-conj-False-r: \( Q = False \rightarrow (P \land Q) = False \) by auto

lemma simp-TF-disj-True-l: \( P = True \rightarrow (P \lor Q) = True \) by auto

lemma simp-TF-disj-True-r: \( Q = True \rightarrow (P \lor Q) = True \) by auto

lemma simp-TF-disj-False-l: \( P = False \rightarrow (P \lor Q) = Q_\parallel \) by auto

lemma simp-TF-disj-False-r: \( Q = False \rightarrow (P \lor Q) = P_\parallel \) by auto

lemma make-cnfx-disj-ex-l: \( (\exists (x::bool). P x) \lor Q = (\exists x. P x \lor Q) \) by auto

lemma make-cnfx-disj-ex-r: \( P \lor (\exists (x::bool). Q x) = (\exists x. P \lor Q x) \) by auto

lemma make-cnfx-nelim: \( (P \lor Q) = (\exists x. (P \lor x) \land (Q \lor \neg x)) \) by auto

lemma make-cnfx-ex-cong: \( (\forall (x::bool). P x = Q x) \rightarrow (\exists x. P x = (\exists x. Q x)) \) by auto

lemma weakening-thm: \( \parallel P; Q \parallel \rightarrow Q \) by auto

lemma cnf_tac-eq-imp: \( \parallel P = Q; P \parallel \rightarrow Q \) by auto

end

ML-file <Tools/cnf.ML>

3 No-Match simproc

The simplification procedure can be used to avoid simplification of terms of a certain form.
definition NO-MATCH :: 'a ⇒ 'b ⇒ bool
  where NO-MATCH pat val ≡ True

lemma NO-MATCH-cong[cong]: NO-MATCH pat val = NO-MATCH pat val
  by (rule refl)

declare [[coercion-args NO-MATCH −−]]

simproc-setup NO-MATCH (NO-MATCH pat val) = K (fn ctxt => fn ct =>
  let
    val thy = Proof-Context.theory-of ctxt
    val dest-binop = Term.dest-comb #> apfst (Term.dest-comb #> snd)
    val m = Pattern.matches thy (dest-binop (Thm.term-of ct))
  in if m then NONE else SOME @{thm NO-MATCH-def} end)

This setup ensures that a rewrite rule of the form NO-MATCH pat val ⇒ t is only applied, if the pattern pat does not match the value val.

Tagging a premise of a simp rule with ASSUMPTION forces the simplifier not to simplify the argument and to solve it by an assumption.

definition ASSUMPTION :: bool ⇒ bool
  where ASSUMPTION A ≡ A

lemma ASSUMPTION-cong[cong]: ASSUMPTION A = ASSUMPTION A
  by (rule refl)

lemma ASSUMPTION-I: A ⇒ ASSUMPTION A
  by (simp add: ASSUMPTION-def)

lemma ASSUMPTION-D: ASSUMPTION A ⇒ A
  by (simp add: ASSUMPTION-def)

setup <
  let
    val asm-sol = mk-solver ASSUMPTION (fn ctxt =>
    resolve-tac ctxt [@{thm ASSUMPTION-I}] THEN'
    resolve-tac ctxt (Simplifier.prems-of ctxt))
  in
    map-theory-simpset (fn ctxt => Simplifier.addSolver (ctxt,asm-sol))
  end
>

3.1 Code generator setup

3.1.1 Generic code generator preprocessor setup

lemma conj-left-cong: P ↔ Q ⇒ P ∧ R ↔ Q ∧ R
  by (fact arg-cong)
lemma disj-left-cong: \( P \longleftrightarrow Q \implies P \lor R \longleftrightarrow Q \lor R \)
by (fact ary-cong)

setup {
  Code-Preproc.map-pre (put-simpset HOL-basic-ss) #>
  Code-Preproc.map-post (put-simpset HOL-basic-ss) #>
  Code-Simp.map-ss (put-simpset HOL-basic-ss #>
  Simplifier.add-cong @{thm conj-left-cong} #>
  Simplifier.add-cong @{thm disj-left-cong})
}

3.1.2 Equality

class equal =
  fixes equal :: 'a ⇒ 'a ⇒ bool
  assumes equal-eq: equal x y ←→ x = y
begin

lemma equal: equal = (=)
  by (rule ext equal-eq)+

lemma equal-refl: equal x x ←→ True
  unfolding equal by (rule iffI TrueI refl)+

lemma eq-equal: (=) ≡ equal
  by (rule eq-reflection) (rule ext, rule ext, rule sym, rule equal-eq)

end

declare eq-equal [symmetric, code-post]
declare eq-equal [code]

simproc-setup passive equal (HOL.eq) =
  (fn - => fn - => fn ct =>
   (case Thm.term_of ct of
    Const (-, Type (type-name fun, [Type _, _])) => SOME @{thm eq-equal}
    | - => NONE)
   )

setup (Code-Preproc.map-pre (fn ctxt => ctxt addsimprocs [simproc (equal)]))

3.1.3 Generic code generator foundation

Datatype bool

code_datatype True False

lemma [code]:
  shows False ∧ P ←→ False
  and True ∧ P ←→ P
  and P ∧ False ←→ False
and $P \land \text{True} \leftrightarrow P$
by simp-all

lemma [code]:
shows $\neg P \lor P \leftrightarrow P$
and $\text{True} \lor P \leftrightarrow \text{True}$
and $P \lor \text{False} \leftrightarrow P$
and $P \lor \text{True} \leftrightarrow \text{True}$
by simp-all

lemma [code]:
shows $(\text{False} \rightarrow P) \leftrightarrow \text{True}$
and $(\text{True} \rightarrow P) \leftrightarrow P$
and $(P \rightarrow \text{False}) \leftrightarrow \neg P$
and $(P \rightarrow \text{True}) \leftrightarrow \text{True}$
by simp-all

More about prop

lemma [code nbe]:
shows $(\text{True} \Rightarrow PROP	ext{ Q}) \equiv PROP	ext{ Q}$
and $(PROP	ext{ Q} \Rightarrow \text{True}) \equiv \text{Trueprop True}$
and $(P \Rightarrow R) \equiv \text{Trueprop (P} \rightarrow R)$
by (auto intro!: equal-intr-rule)

lemma $\text{Trueprop-code [code]}$: $\text{Trueprop True} \equiv \text{Code-Generator.holds}$
by (auto intro!: equal-intr-rule holds)

declare $\text{Trueprop-code [symmetric, code-post]}$

Equality

declare simp-thms(6) [code nbe]

instantiation $\text{itself :: (type) equal}$
begin

definition $\text{equal-itself :: 'a itself \Rightarrow 'a itself \Rightarrow bool}$
where $\text{equal-itself x y} \leftrightarrow x = y$

instance
by standard (fact $\text{equal-itself-def}$)
end

lemma $\text{equal-itself-code [code]}$: $\text{equal TYPE('a) TYPE('a} \leftrightarrow \text{True}$
by (simp add: equal)

setup `(Sign.add-const-constraint (const-name equal), SOME typ ('a::type \Rightarrow 'a \Rightarrow bool))`
lemma equal-alias-cert: OFCLASS(\', equal-class) \equiv ((\(=): \', a \Rightarrow bool) \equiv equal)
(is ?ofclass \equiv ?equal)

proof
  assume PROP ?ofclass
  show PROP ?equal
  by (tactic \texttt{\{ALLGOALS (resolve-tac context \texttt{Thm.unconstrainT}@\{thm eq-equal\})\}})
    (fact \texttt{\{PROP ?ofclass\}})
next
  assume PROP ?equal
  show PROP ?ofclass
  proof
    qed
  qed

setup \texttt{\{Sign.add-const-constraint (const-name \texttt{\{equal\}, SOME typ \{a::equal \Rightarrow \', a \Rightarrow bool\}})\}}

setup \texttt{\{Nbe.add-const-alias @\{thm equal-alias-cert\}\}}

Cases

lemma Let-case-cert:
  assumes CASE \equiv (\lambda x. Let x f)
  shows CASE x \equiv f x
  using assms by simp-all

setup \texttt{\{Code.declare-case-global @\{thm Let-case-cert\} \#>\}}
  \texttt{\{Code.declare-undefined-global const-name \{undefined\}\}}

declare \texttt{[[code abort: undefined]]}

\subsection{Generic code generator target languages}

\begin{verbatim}

3.1.4  Generic code generator target languages

\texttt{type bool}

\texttt{code-printing}
  \texttt{type-constructor bool \rightarrow}
  \texttt{(SML) \texttt{bool} and (OCaml) \texttt{bool} and (Haskell) \texttt{Bool} and (Scala) \texttt{Boolean}}
  | \texttt{constant True \rightarrow}
    \texttt{(SML) true and (OCaml) true and (Haskell) True and (Scala) true}
  | \texttt{constant False \rightarrow}
    \texttt{(SML) false and (OCaml) false and (Haskell) False and (Scala) false}

\texttt{code-reserved SML}
  \texttt{bool true false}

\texttt{code-reserved OCaml}
  \texttt{bool}

\end{verbatim}


code-reserved Scala

Boolean

code-printing
constant Not →
   (SML) not and (OCaml) not and (Haskell) not and (Scala) ? -
code-printing
constant HOL.conj →
   (SML) infixl 1 andalso and (OCaml) infixl 3 && and (Haskell) infixr 3 &&
   and (Scala) infixl 3 &&
code-printing
constant HOL.disj →
   (SML) infixl 0 orelse and (OCaml) infixl 2 || and (Haskell) infixl 2 || and
   (Scala) infixl 1 ||
code-printing
constant HOL.implies →
   (SML) !(if (-)/ then (-)/ else true)
   and (OCaml) !(if (-)/ then (-)/ else true)
   and (Haskell) !(if (-)/ then (-)/ else True)
   and (Scala) !((-) match { / case true => (-)/ case false => true/ })
code-printing
constant If →
   (SML) !(if (-)/ then (-)/ else (-))
   and (OCaml) !(if (-)/ then (-)/ else (-))
   and (Haskell) !(if (-)/ then (-)/ else (-))
   and (Scala) !((-) match { / case true => (-)/ case false => (-)/ })

code-reserved SML
not

code-reserved OCaml
not

code-identifier

code-module Pure →
   (SML) HOL and (OCaml) HOL and (Haskell) HOL and (Scala) HOL

Using built-in Haskell equality.
code-printing
type-class equal → (Haskell) Eq
constant HOL.equal → (Haskell) infix 4 ==
code-printing
code-printing
constant undefined →
   (SML) !(raise/ Fail/ undefined)
   and (OCaml) failwith/ undefined
   and (Haskell) error/ undefined
   and (Scala) !sys.error(undefined)
3.1.5 Evaluation and normalization by evaluation

\[\text{method-setup } \text{eval} = \langle \]
\[\quad \text{let }
\[\quad \quad \text{fun eval-tac ctxt} =
\[\quad \quad \quad \text{let val conv} = \text{Code/Runtime.dynamic-holds-conv}
\[\quad \quad \quad \text{in }
\[\quad \quad \quad \quad \text{CONVERSION (Conv.params-conv '1 (Conv.concl-conv '1 o conv) ctxt)}
\[\quad \quad \quad \text{THEN'}
\[\quad \quad \quad \quad \text{resolve-tac ctxt [TrueI]}
\[\quad \quad \quad \text{end}
\[\quad \quad \text{in }
\[\quad \quad \quad \text{Scan.succeed (SIMPLE-METHOD' o eval-tac)}
\[\quad \quad \text{end}
\[\rangle \text{ solve goal by evaluation.} \]

\[\text{method-setup } \text{normalization} = \langle \]
\[\quad \text{Scan.succeed (fn ctxt =>)
\[\quad \quad \text{SIMPLE-METHOD'}
\[\quad \quad \quad \text{(CHANGED-PROP o}
\[\quad \quad \quad \quad \text{(CONVERSION (Nbe.dynamic-conv ctxt)}
\[\quad \quad \quad \quad \text{THEN-ALL-NEW (TRY o resolve-tac ctxt [TrueI]))))}
\[\rangle \text{ solve goal by normalization.} \]

3.2 Counterexample Search Units

3.2.1 Quickcheck

\[\text{quickcheck-params [size = 5, iterations = 50]} \]

3.2.2 Nitpick setup

\[\text{named-theorems nitpick-unfold alternative definitions of constants as needed by Nitpick} \]
\[\quad \text{and nitpick-simp equational specification of constants as needed by Nitpick} \]
\[\quad \text{and nitpick-psimp partial equational specification of constants as needed by Nitpick} \]
\[\quad \text{and nitpick-choice-spec choice specification of constants as needed by Nitpick} \]
\[\text{declare if-bool-eq-conj [nitpick-unfold, no-atp]} \]
\[\quad \text{and if-bool-eq-disj [no-atp]} \]

3.3 Preprocessing for the predicate compiler

\[\text{named-theorems code-pred-def alternative definitions of constants for the Predicate Compiler} \]
\[\quad \text{and code-pred-inline inlining definitions for the Predicate Compiler} \]
\[\quad \text{and code-pred-simp simplification rules for the optimisations in the Predicate Compiler} \]
3.4 Legacy tactics and ML bindings

ML

(* combination of (spec RS spec RS ...(j times) ... spec RS mp) *)
local
  fun wrong-prem (Const (const-name 'All', -) $ Abs (_, _, t)) = wrong-prem t
  | wrong-prem (Bound _) = true
  | wrong-prem _ = false;
  val filter-right = filter (not o wrong-prem o HOLogic.dest_Trueprop o hd o Thm.prems_of);
  fun smp i = funpow i (fn m => filter-right ([spec RS RL m]) [mp]);
  in
    fun smp-tac ctxt j = EVERY'[dresolve-tac ctxt (smp j), assume-tac ctxt];
  end;

local
  val nnf-ss = simpset_of (put-simpset HOL-basic-ss context addsimps @{context addsimps nnn-simps});
  in
    fun nnf-conv ctxt = Simplifier.rewrite (put-simpset nnf-ss ctxt);
  end

hide-const (open) eq equal
end

4 Abstract orderings

theory Orderings
imports HOL
keywords print-orders :: diag
begin

4.1 Abstract ordering
locale partial-preordering =
  fixes less-eq :: 'a ⇒ 'a ⇒ bool (infix ≤) 50
  assumes refl: 'a ≤ 'a — not iff: makes problems due to multiple (dual) interpretations
    and trans: 'a ≤ 'b → 'b ≤ 'c → 'a ≤ 'c
locale preordering = partial-preordering +
  fixes less :: 'a ⇒ 'a ⇒ bool (infix <) 50
  assumes strict-iff-not: 'a < 'b ↔ 'a ≤ 'b ∧ ¬ 'b ≤ 'a
begin
lemma strict-implies-order:
\begin{document}

\section*{Theory "Orderings"}

\begin{lstlisting}
\begin{enumerate}
\item \[ a \prec b \implies a \preceq b \]
  by (simp add: strict-iff-not)
\end{enumerate}

\textbf{lemma \texttt{irrefl}}: \textit{--- not iff}: makes problems due to multiple (dual) interpretations
\begin{enumerate}
\item \[ \neg a \prec a \]
  by (simp add: strict-iff-not)
\end{enumerate}

\textbf{lemma \texttt{asym}}:
\begin{enumerate}
\item \[ a \prec b \implies b \prec a \implies \text{False} \]
  by (auto simp add: strict-iff-not)
\end{enumerate}

\textbf{lemma \texttt{strict-trans1}}:
\begin{enumerate}
\item \[ a \preceq b \implies b \prec c \implies a \prec c \]
  by (auto simp add: strict-iff-not intro: trans)
\end{enumerate}

\textbf{lemma \texttt{strict-trans2}}:
\begin{enumerate}
\item \[ a \prec b \implies b \preceq c \implies a \prec c \]
  by (auto simp add: strict-iff-not intro: trans)
\end{enumerate}

\textbf{lemma \texttt{strict-trans}}:
\begin{enumerate}
\item \[ a \prec b \implies b \prec c \implies a \prec c \]
  by (auto intro: strict-trans1 strict-implies-order)
\end{enumerate}

end

\textbf{lemma \texttt{preordering-strictI}}: \textit{--- Alternative introduction rule with bias towards strict order}
\begin{enumerate}
\item \texttt{fixes less-eq (infix \(\leq\)) 50}
  \texttt{and less (infix \(<\)) 50}
\item \texttt{assumes less-eq-less: \(\forall a\ b.\ a \leq b \iff a < b \lor a = b\)}
  \texttt{assumes asym: \(\forall a\ b.\ a < b \implies \neg b < a\)}
  \texttt{assumes irrefl: \(\forall a.\ \neg a < a\)}
  \texttt{assumes trans: \(\forall a\ b\ c.\ a < b \implies b < c \implies a < c\)}
\item \texttt{shows preordering (\(\leq\)) (\(<\))}
\end{enumerate}

\textbf{proof}
\begin{enumerate}
\item \texttt{fix a b}
  \texttt{show \(a < b \iff a \leq b \land \neg b \leq a\)}
  by (auto simp add: less-eq-less asym irrefl)
\item \texttt{fix a}
  \texttt{show \(a \leq a\)}
  by (auto simp add: less-eq-less)
\item \texttt{fix a b c}
  \texttt{assume \(a \leq b\) and \(b \leq c\) then show \(a \leq c\)}
  by (auto simp add: less-eq-less intro: trans)
\end{enumerate}

qed

\textbf{lemma \texttt{preordering-dualI}}:
\end{document}

fixes less-eq (infix ⟨≤⟩ 50)
  and less (infix ⟨<⟩ 50)
assumes ‹preordering (λa b. b ≤ a) (λa b. b < a)›
shows ‹preordering (≤) (⟨⟩)›
proof
  from assms interpret preordering ‹λa b. b ≤ a› ‹λa b. b < a›.
  show ‹thesis›
    by standard (auto simp: strict-iff-not refl intro: trans)
qed

locale ordering = partial-preordering +
fixes less :: ‹′a ‚⇒ ′a ‚⇒ bool› (infix ‹⟨⟩› 50)
assumes strict-iff-order: ‹a < b ←→ a ≤ b ∧ a ≠ b›
assumes antisym: ‹a ≤ b ⇒ b ≤ a ⇒ a = b›
begin

sublocale preordering ‹(≤)› ‹(⟨⟩)›
proof
  show ‹a < b ←→ a ≤ b ∧ ¬ b ≤ a› for a b
    by (auto simp add: strict-iff-order intro: antisym)
qed

lemma strict-implies-not-eq:
  ‹a < b ⇒ a ≠ b›.
  by (simp add: strict-iff-order)
lemma not-eq-order-implies-strict:
  ‹a ≠ b ⇒ a ≤ b ⇒ a < b›
  by (simp add: strict-iff-order)
lemma order-iff-strict:
  ‹a ≤ b ←→ a < b ∨ a = b›
  by (auto simp add: strict-iff-order refl)
lemma eq-iff: ‹a = b ←→ a ≤ b ∧ b ≤ a›
  by (auto simp add: refl intro: antisym)
end

lemma ordering-strictI: — Alternative introduction rule with bias towards strict order
  fixes less-eq (infix ⟨≤⟩ 50)
  and less (infix ⟨<⟩ 50)
assumes less-eq-less: ‹∀a b. a ≤ b ←→ a < b ∨ a = b›
  assumes asym: ‹∀a b. a < b ⇒ ¬ b < a›
  assumes irrefl: ‹∀a. ¬ a < a›
  assumes trans: ‹∀a b c. a < b ⇒ b < c ⇒ a < c›
  shows ‹ordering (≤) (⟨⟩)›,
proof
fix a b
  show \( a < b \iff a \leq b \land a \neq b \)
    by (auto simp add: less-eq-less asym irrefl)
next
fix a
  show \( a \leq a \)
    by (auto simp add: less-eq)
next
fix a b c
  assume \( a \leq b \) and \( b \leq c \) then show \( a \leq c \)
    by (auto simp add: less-eq-less intro: trans)
next
fix a b
  assume \( a \leq b \) and \( b \leq a \) then show \( a = b \)
    by (auto simp add: less-eq-less asym)
qed

lemma ordering-dualI:
  fixes less-eq (infix \( \leq \)) and less (infix \( < \))
  assumes \( \text{ordering} (\lambda a b. b \leq a) (\lambda a b. b < a) \)
  shows \( \text{ordering} (\leq) (<) \)
proof –
  from assms interpret ordering \( \lambda a b. b \leq a \) \( \lambda a b. b < a \) .
  show \( \text{thesis} \)
    by standard (auto simp: strict-iff-order refl intro: antisym trans)
qed

locale ordering-top = ordering +
  fixes top :: \( \top \)
  assumes extremum [simp]: \( a \leq \top \)
begin

lemma extremum-uniqueI:
  \( \top \leq a \implies a = \top \)
  by (rule antisym) auto

lemma extremum-unique:
  \( \top \leq a \iff a = \top \)
  by (auto intro: antisym)

lemma extremum-strict [simp]:
  \( \top < a \)
  using extremum [of a] by (auto simp add: order-iff-strict intro: asym irrefl)

lemma not-eq-extremum:
  \( a \neq \top \iff a < \top \)
  by (auto simp add: order-iff-strict intro: not-eq-order-implies-strict extremum)
4.2 Syntactic orders

class ord =
  fixes less-eq :: 'a ⇒ 'a ⇒ bool
  and less :: 'a ⇒ 'a ⇒ bool
begin

notation
  less-eq ('(≤')) and
  less-eq (l/- ≤ -) [51, 51] 50) and
  less ('('<')) and
  less (l/- < -) [51, 51] 50)

abbreviation (input)
  greater-eq (infix ≥ 50)
  where x ≥ y ≡ y ≤ x

abbreviation (input)
  greater (infix > 50)
  where x > y ≡ y < x

notation (ASCII)
  less-eq ('(<=')) and
  less-eq (l/- <= -) [51, 51] 50)

notation (input)
  greater-eq (infix >= 50)
end

4.3 Quasi orders

class preorder = ord +
  assumes less-le-not-le: x < y ↔ x ≤ y ∧ ¬ (y ≤ x)
  and order-refl [iff]: x ≤ x
  and order-trans: x ≤ y ⇒ y ≤ z ⇒ x ≤ z
begin

sublocale order: preordering less-eq less + dual-order: preordering greater-eq greater
proof –
  interpret preordering less-eq less
  by standard (auto intro: order-trans simp add: less-le-not-le)
  show (preordering less-eq less)
    by (fact preordering-axioms)
  then show (preordering greater-eq greater)
    by (rule preordering-dualI)
qed
Reflexivity.

**Lemma** \( \text{eq-refl} \): \( x = y \implies x \leq y \)

— This form is useful with the classical reasoner.

*by (erule ssubst) (rule order-refl)*

**Lemma** \( \text{less-irrefl [iff]} \): \( \neg x < x \)

*by (simp add: less-le-not-le)*

**Lemma** \( \text{less-imp-le} \): \( x < y \implies x \leq y \)

*by (simp add: less-le-not-le)*

Asymmetry.

**Lemma** \( \text{less-not-sym} \): \( x < y \implies \neg (y < x) \)

*by (simp add: less-le-not-le)*

**Lemma** \( \text{less-asym} \): \( x < y \implies (\neg P \implies y < x) \implies P \)

*by (drule less-not-sym, erule contrapos-np) simp*

Transitivity.

**Lemma** \( \text{less-trans} \): \( x < y \implies y < z \implies x < z \)

*by (auto simp add: less-le-not-le intro: order-trans)*

**Lemma** \( \text{le-less-trans} \): \( x \leq y \implies y < z \implies x < z \)

*by (auto simp add: less-le-not-le intro: order-trans)*

**Lemma** \( \text{less-le-trans} \): \( x < y \implies y \leq z \implies x < z \)

*by (auto simp add: less-le-not-le intro: order-trans)*

Useful for simplification, but too risky to include by default.

**Lemma** \( \text{less-imp-not-less} \): \( x < y \implies (\neg y < x) \iff \text{True} \)

*by (blast elim: less-asym)*

**Lemma** \( \text{less-imp-triv} \): \( x < y \implies (y < x \implies P) \iff \text{True} \)

*by (blast elim: less-asym)*

Transitivity rules for calculational reasoning

**Lemma** \( \text{less-asym} \cdot \): \( a < b \implies b < a \implies P \)

*by (rule less-asym)*

Dual order

**Lemma** \( \text{dual-preorder} \):

\(<\text{class.preorder} \ (\geq) \ (\geq)\)\

*by standard (auto simp add: less-le-not-le intro: order-trans)*

end

**Lemma** \( \text{preordering-preorderI} \):

\(<\text{class.preorder} \ (\leq) \ (\leq)\) \text{ if } \langle \text{preordering} \ (\leq) \ (\leq)\)
for less-eq (infix \( \leq \) 50) and less (infix \( \langle \) 50)
proof -
  from that interpret preorder \( (\leq) \ (\langle) \).
  show thesis
    by standard (auto simp add: strict-iff-not refl intro: trans)
qed

4.4 Partial orders

class order = preorder +
  assumes order-antisym: \( x \leq y \Rightarrow y \leq x \Rightarrow x = y \)
begin

lemma less-le: \( x < y \leftrightarrow x \leq y \land x \neq y \)
  by (auto simp add: less-le-not-le intro: order-antisym)

sublocale order: ordering less-eq less + dual-order: ordering greater-eq greater
proof -
  interpret ordering less-eq less
    by standard (auto intro: order-antisym order-trans simp add: less-le)
  show ordering less-eq less
    by (fact ordering-axioms)
  then show ordering greater-eq greater
    by (rule ordering-dualI)
qed

Reflexivity.

lemma le-less: \( x \leq y \leftrightarrow x < y \lor x = y \)
  — NOT suitable for iff, since it can cause PROOF FAILED.
  by (fact order.order-iff-strict)

lemma le-imp-less-or-eq: \( x \leq y \Rightarrow x < y \lor x = y \)
  by (simp add: less-le)

Useful for simplification, but too risky to include by default.

lemma less-imp-not-eq: \( x < y \Rightarrow (x = y) \leftrightarrow False \)
  by auto

lemma less-imp-not-eq2: \( x < y \Rightarrow (y = x) \leftrightarrow False \)
  by auto

Transitivity rules for calculational reasoning

lemma neg-le-trans: \( a \neq b \Rightarrow a \leq b \Rightarrow a < b \)
  by (fact order.not-eq-order-implies-strict)

lemma le-neg-trans: \( a \leq b \Rightarrow a \neq b \Rightarrow a < b \)
  by (rule order.not-eq-order-implies-strict)

Asymmetry.
THEORY "Orderings"

lemma order-eq-iff: \( x = y \iff x \leq y \land y \leq x \)
  by (fact order.eq-iff)

lemma antisym-conv: \( y \leq x \implies x \leq y \iff x = y \)
  by (simp add: order.eq-iff)

lemma less-imp-neq: \( x < y \implies x \neq y \)
  by (fact order.strict-implies-not-eq)

lemma antisym-conv1: \( \neg x < y \implies x \leq y \iff x = y \)
  by (simp add: local.le-less)

lemma antisym-conv2: \( x \leq y \implies \neg x < y \iff x = y \)
  by (simp add: local.less-le)

lemma leD: \( y \leq x \implies \neg x < y \)
  by (auto simp: less-le order.antisym)

Least value operator
definition (in ord)
  Least :: \( 'a \Rightarrow bool \Rightarrow 'a \) (binder \( \text{LEAST} \) 10) where
  \( \text{Least } P = (\text{THE } x. \ P x \land (\forall y. P y \implies x \leq y)) \)

lemma Least-equality:
  assumes \( P x \) and \( \forall y. P y \implies x \leq y \)
  shows \( \text{Least } P = x \)
  unfolding Least-def by (rule the-equality)
  (blast intro: assms order.antisym)+

lemma Least12-order:
  assumes \( P x \) and \( \forall y. P y \implies x \leq y \)
  and \( \forall x. P x \implies \forall y. P y \implies x \leq y \implies Q x \)
  shows \( Q (\text{Least } P) \)
  unfolding Least-def by (rule theI2)
  (blast intro: assms order.antisym)+

lemma Least-ex1:
  assumes \( \exists! x. P x \land (\forall y. P y \implies x \leq y) \)
  shows \( \text{Least1}: P (\text{Least } P) \) and \( \text{Least1-le}: P z \implies \text{Least } P \leq z \)
  using theI[OF assms]
  unfolding Least-def
  by auto

Greatest value operator
definition Greatest :: \( 'a \Rightarrow bool \Rightarrow 'a \) (binder \( \text{GREATEST} \) 10) where
  \( \text{Greatest } P = (\text{THE } x. \ P x \land (\forall y. P y \implies x \geq y)) \)
theory "Orderings"

lemma GreatestI2-order:
\[ \[ P x; \forall y. P y \Rightarrow x \geq y; \forall x. [ P x; \forall y. P y \Rightarrow x \geq y ] \Rightarrow Q x ] \Rightarrow Q (\text{Greatest } P) \]

unfolding Greatest-def
by (rule theI2) (blast intro: order.antisym)+

lemma Greatest-equality:
\[ [ P x; \forall y. P y \Rightarrow x \geq y ] \Rightarrow \text{Greatest } P = x \]

unfolding Greatest-def
by (rule the-equality) (blast intro: order.antisym)+

end

lemma ordering-orderI:
fixes less-eq (infix \leq 50)
and less (infix < 50)
assumes ordering less-eq less
shows class.order less-eq less

proof -
from assms interpret ordering less-eq less .
show \?thesis
  by standard (auto intro: antisym trans simp add: refl strict-iff-order)
qed

lemma order-strictI:
fixes less (infix < 50)
and less-eq (infix \leq 50)
assumes \( a \leq b \iff a < b \lor a = b \)
assumes \( \forall a. b. a < b \Rightarrow \neg b < a \)
assumes \( \forall a. \neg a < a \)
assumes \( \forall a b c. a < b \Rightarrow b < c \Rightarrow a < c \)
shows class.order less-eq less
by (rule ordering-orderI) (rule ordering-strictI, (fact assms)+)

context order
begin

Dual order

lemma dual-order:
class.order (\geq) (>)
using dual-order.ordering-axioms by (rule ordering-orderI)

end

4.5 Linear (total) orders

class linorder = order +
assumes linear: \( x \leq y \lor y \leq x \)
begin

lemma less-linear: \( x < y \lor x = y \lor y < x \)
unfolding less-le using less-le linear by blast

lemma le-less-linear: \( x \leq y \lor y < x \)
by (simp add: le-less less-linear)

lemma le-cases [case-names le ge]:
\[
(x \leq y \Longrightarrow P) \Longrightarrow (y \leq x \Longrightarrow P) \Longrightarrow P
\]
using linear by blast

lemma (in linorder) le-cases3:
\[
[[ x \leq y ; y \leq z ] \Longrightarrow P ; [ y \leq x ; x \leq z ] \Longrightarrow P ; [ x \leq z ; z \leq y ] \Longrightarrow P ; [ z \leq x ; x \leq y ] \Longrightarrow P ] \Longrightarrow P
\]
by (blast intro: le-cases)

lemma linorder-cases [case-names less equal greater]:
\[
(x < y \Longrightarrow P) \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow (y < x \Longrightarrow P) \Longrightarrow P
\]
using less-linear by blast

lemma linorder-wlog[case-names le sym]:
\[
(\forall a b. a \leq b \Longrightarrow P a b) \Longrightarrow (\forall a b. P b a \Longrightarrow P a b)
\]
by (cases rule: le-cases[of a b]) blast+

lemma not-less: \( \neg x < y \leftrightarrow y \leq x \)
unfolding less-le
using linear by (blast intro: order.antisym)

lemma not-less-iff-gr-or-eq: \( \neg(x < y) \longleftrightarrow (x > y \lor x = y) \)
by (auto simp add: not-less le-less)

lemma not-le: \( \neg x \leq y \leftrightarrow y < x \)
unfolding less-le
using linear by (blast intro: order.antisym)

lemma neq-iff: \( x \neq y \longleftrightarrow x < y \lor y < x \)
by (cut-tac x = x and y = y in less-linear, auto)

lemma neqE: \( x \neq y \Longrightarrow (x < y \Longrightarrow R) \Longrightarrow (y < x \Longrightarrow R) \Longrightarrow R \)
by (simp add: neq-iff) blast

lemma antisym-conv3: \( \neg y < x \Longrightarrow \neg x < y \longleftrightarrow x = y \)
by (blast intro: order.antisym dest: not-less [THEN iffD1])

lemma leI: \( \neg x < y \Longrightarrow y \leq x \)
unfolding not-less .
lemma not-le-imp-less: \( \neg y \leq x \implies x < y \)
unfolding not-le.

lemma linorder-less-wlog[case-names less refl sym]:
\[
\forall a b. a < b \implies P a b; \ \forall a. P a a; \ \forall a b. P b a \implies P a b
\]
using antisym-conv3 by blast

Dual order

lemma dual-linorder:
\[
\text{class.linorder} (\geq) (>) \\
\text{by (rule class.linorder.intro, rule dual-order) (unfold-locales, rule linear)}
\]
end

Alternative introduction rule with bias towards strict order

lemma linorder-strictI:
\[
\text{fixes less-eq (infix \leq 50)} \\
\text{and less (infix < 50)} \\
\text{assumes class.order less-eq less} \\
\text{assumes trichotomy: } \forall a b. a < b \lor a = b \lor b < a \\
\text{shows class.linorder less-eq less}
\]
proof
\[
\text{interpret order less-eq less} \\
\text{by (fact class.order less-eq less)}
\]
show ?thesis
disable

ML-file (~/src/Provers/order-procedure.ML)

ML-file (~/src/Provers/order-tac.ML)

ML (structure Logic-Signature : LOGIC-SIGNATURE = struct
val mk-Trueprop = HOLogic.mk-Trueprop
val dest-Trueprop = HOLogic.dest-Trueprop
val Trueprop-conv = HOLogic.Trueprop-conv
val Not = HOLogic.Not
val conj = HOLogic.conj
val disj = HOLogic.disj

val notI = @{thm notI}
val ccontr = @{thm ccontr}
val conjI = @{thm conjI}
val conjE = @{thm conjE}
val disjE = @{thm disjE}

val not-not-conv = Conv.rewr-conv @{thm eq-reflection[OF not-not]}
val de-Morgan-conj-conv = Conv.rewr-conv @{thm eq-reflection[OF de-Morgan-conj]}
val de-Morgan-disj-conv = Conv.rewr-conv @{thm eq-reflection[OF de-Morgan-disj]}
val conj-disj-distribL-conv = Conv.rewr-conv @{thm eq-reflection[OF conj-disj-distribL]}
val conj-disj-distribR-conv = Conv.rewr-conv @{thm eq-reflection[OF conj-disj-distribR]}
end

structure HOL-Base-Order-Tac = Base-Order-Tac(
  structure Logic-Sig = Logic-Signature;
  (* Exclude types with specialised solvers. *)
  val excluded-types = [HOLogic.natT, HOLogic.intT, HOLogic.realT]
)

structure HOL-Order-Tac = Order-Tac(structure Base-Tac = HOL-Base-Order-Tac)

fun print-orders ctxt0 =
  let
    val ctxt = Config.put show-sorts true ctxt0
    val orders = HOL-Order-Tac.Data.get (Context.Proof ctxt)
    fun pretty-term t = Pretty.block
      [Pretty.quote (Syntax.pretty-term ctxt t), Pretty.brk 1,
       Pretty.str ::, Pretty.brk 1,
       Pretty.quote (Syntax.pretty-typ ctxt (type-of t)), Pretty.brk 1]
    fun pretty-order (kinds = kinds, ops = ops, ...) =
      Pretty.block ([Pretty.str (Pretty.make-string kind),
                      Pretty.str ::, Pretty.brk 1]
                     @ map pretty-term ops)

    in
      Pretty.writeln (Pretty.big-list order structures: (map pretty-order orders))
    end

val order = Outer-Syntax.command (command-keyword (Scan.succeed (fn ctxt => SIMPLE-METHOD' (HOL-Order-Tac.tac [] ctxt))))

The method order allows one to use the order tactic located in Provers/order_tac.ML in a standalone fashion.

The tactic rearranges the goal to prove False, then retrieves order literals of partial and linear orders (i.e. \( x = y \), \( x \leq y \), \( x < y \), and their negated versions) from the premises and finally tries to derive a contradiction. Its
main use case is as a solver to simp (see below), where it e.g. solves premises of conditional rewrite rules.

The tactic has two configuration attributes that control its behaviour:

- **order-trace** toggles tracing for the solver.
- **order-split-limit** limits the number of order literals of the form \( \neg x < y \) that are passed to the tactic. This is helpful since these literals lead to case splitting and thus exponential runtime. This only applies to partial orders.

We setup the solver for HOL with the structure **HOL_Order_Tac** here but the prover is agnostic to the object logic. It is possible to register orders with the prover using the functions **HOL_Order_Tac.declare_order** and **HOL_Order_Tac.declare_linorder**, which we do below for the type classes **order** and **linorder**. If possible, one should instantiate these type classes instead of registering new orders with the solver. One can also interpret the type class locales **order** and **linorder**. An example can be seen in **Library/Sublist.thy**, which contains e.g. the prefix order on lists.

The diagnostic command **print-orders** shows all orders known to the tactic in the current context.

**Theories to set up transitivity reasoner of partial and linear orders.**

**context** order
**begin**

**lemma** nless-le: \( (\neg a < b) \iff (\neg a \leq b) \lor a = b \)
**using** local.dual-order.order-iff-strict by blast

**local-setup**

\[
\begin{align*}
\text{HOL-Order-Tac.declare-order} \{ \\
\text{ops} &= \{eq = \oplus \{ \text{term } \langle=\rangle : \to \text{ bool} \}, le = \oplus \{ \text{term } \langle\leq\rangle \}, lt = \oplus \{ \text{term } \langle<\rangle \} \}, \\
\text{thms} &= \{ \text{trans} = \oplus \{ \text{thm order-trans} \}, \text{refl} = \oplus \{ \text{thm order-refl} \}, \text{eqD1} = \oplus \{ \text{thm eq-refl} \}, \\
& \quad \text{eqD2} = \oplus \{ \text{thm eq-refl}[OF sym] \}, \text{antisym} = \oplus \{ \text{thm order-antisym} \}, \text{contr} \\
& \quad = \oplus \{ \text{thm notE} \} \}, \\
\text{conv-thms} &= \{ \text{less-le} = \oplus \{ \text{thm eq-reflection}[OF less-le] \}, \\
& \quad \text{nless-le} = \oplus \{ \text{thm eq-reflection}[OF nless-le] \} \}
\end{align*}
\]

**end**

**context** linorder
begin

lemma nle-le: \((\neg a \leq b) \iff b \leq a \land b \neq a\)
using not-le less-le by simp

local-setup

HOL-Order-Tac.declare-linorder {
ops = \{eq = \@{\text{term } (=) :: 'a \Rightarrow 'a \Rightarrow bool}, le = \@{\text{term } (\leq)}, lt = \@{\text{term } (\lt)}\},

thms = \{trans = \@{thm order-trans}, refl = \@{thm order-refl}, eqD1 = \@{thm eq-refl},
eqD2 = \@{thm eq-refl[OF sym]}, antisym = \@{thm order-antisym}, contr
= \@{thm notE}\},
cone-thms = \{less-le = \@{thm eq-reflection[OF less-le]},
nless-le = \@{thm eq-reflection[OF not-less]},
nle-le = \@{thm eq-reflection[OF nle-le]}\}
}

end

setup

map-theory-simpset (fn ctxt0 => ctxt0 addsolver

mk-solver partial and linear orders (fn ctxt => HOL-Order-Tac.tac (Simplifier.prems-of ctxt) ctxt))

ML

local

fun prp t thm = Thm.prop-of thm = t; (* FIXME proper aconv! ? *)
in

fun antisym-le-simproc ctxt ct =
(case Thm.term-of ct of
(le as Const (\_, T)) $ r $ s =>
(let
val prems = Simplifier.prems-of ctxt;
val less = Const (const-name \"less\", T);
val t = HOLogic.mk_Trueprop(le $ s $ r);
in
(case find-first (prp t) prems of
NONE =>
let val t = HOLogic.mk_Trueprop(HOLogic.Not $ (less $ r $ s)) in
(case find-first (prp t) prems of
NONE => NONE
| SOME thm => SOME(mk-meta-eq(thm RS \@{thm antisym-conv1}))))
end
| SOME thm => SOME (mk-meta-eq (thm RS \@{thm order-class.antisym-conv})))
end handle THM - => NONE)
fun antisym-less-simproc ctxt ct =
  (case Thm.term_of ct of
   NotC $ ((less as Const(\., T)) $ r $ s) =>
   (let
     val prems = Simplifier.prems_of ctxt;
     val le = Const (const-name \(less-eq\), T);
     val t = HOLogic.mk_Trueprop(le $ r $ s);
   in
     (case find-first (prp t) prems of
      NONE =>
      let val t = HOLogic.mk_Trueprop (NotC $ (less $ s $ r)) in
        (case find-first (prp t) prems of
         NONE => NONE
         | SOME thm => SOME (mk-meta-eq (thm RS @\{thm linorder-class.antisym_conv3\})))
      end
      | SOME thm => SOME (mk-meta-eq (thm RS @\{thm antisym_conv2\})))
    end)
   end handle THM - => NONE)
 | - => NONE);
THEORY “Orderings”

-All-greater :: [idt, 'a, bool] => bool \ ((\forall \cdot \cdot \cdot \cdot) \ [0, 0, 10] \ 10)
-Ex-greater :: [idt, 'a, bool] => bool \ ((\exists \cdot \cdot \cdot \cdot) \ [0, 0, 10] \ 10)
-All-greater-eq :: [idt, 'a, bool] => bool \ ((\forall \cdot \cdot \cdot \cdot) \ [0, 0, 10] \ 10)
-Ex-greater-eq :: [idt, 'a, bool] => bool \ ((\exists \cdot \cdot \cdot \cdot) \ [0, 0, 10] \ 10)

-All-neq :: [idt, 'a, bool] => bool \ ((\forall \cdot \cdot \cdot \cdot) \ [0, 0, 10] \ 10)
-Ex-neq :: [idt, 'a, bool] => bool \ ((\exists \cdot \cdot \cdot \cdot) \ [0, 0, 10] \ 10)


translations
\forall x<y. P \rightarrow \forall x. x < y \rightarrow P
\exists x<y. P \rightarrow \exists x. x < y \land P
\forall x\leq y. P \rightarrow \forall x. x \leq y \rightarrow P
\exists x\leq y. P \rightarrow \exists x. x \leq y \land P
\forall x>y. P \rightarrow \forall x. x > y \rightarrow P
\exists x>y. P \rightarrow \exists x. x > y \land P
\forall x\geq y. P \rightarrow \forall x. x \geq y \rightarrow P
\exists x\geq y. P \rightarrow \exists x. x \geq y \land P
\forall x\neq y. P \rightarrow \forall x. x \neq y \rightarrow P
\exists x\neq y. P \rightarrow \exists x. x \neq y \land P

print-translation :
let
val All-binder = Mixfix.binder-name const-syntax (All);
val Ex-binder = Mixfix.binder-name const-syntax (Ex);
val impl = const-syntax (HOL.implies);
val conj = const-syntax (HOL.conj);
val less = const-syntax (less);
val less-eq = const-syntax (less-eq);

val trans =
[(All-binder, impl, less),
 (syntax-const (-All-less), syntax-const (-All-greater))],
[(All-binder, impl, less-eq),
 (syntax-const (-All-less-eq), syntax-const (-All-greater-eq))],
[(Ex-binder, conj, less),
 (syntax-const (-Ex-less), syntax-const (-Ex-greater))],
[(Ex-binder, conj, less-eq),
 (syntax-const (-Ex-less-eq), syntax-const (-Ex-greater-eq))];

fun matches-bound v t =
(case t of
4.8 Transitivity reasoning

context ord
begin

lemma ord-le-eq-trans: a ≤ b ⇒ b = c ⇒ a ≤ c
  by (rule subst)

lemma ord-eq-le-trans: a = b ⇒ b ≤ c ⇒ a ≤ c
  by (rule ssubst)

lemma ord-less-eq-trans: a < b ⇒ b = c ⇒ a < c
  by (rule subst)

lemma ord-eq-less-trans: a = b ⇒ b < c ⇒ a < c
  by (rule ssubst)

end

lemma order-less-subst2: (a::'a::order) < b ⇒ f b < (c::'c::order) ⇒
  (!!!x y. x < y ⇒ f x < f y) ⇒ f a < c

proof –
  assume r: !!!x y. x < y ⇒ f x < f y
  assume a < b hence f a < f b by (rule r)
  also assume f b < c
  finally (less-trans) show ?thesis .
qed

lemma order-less-subst1: (a::'a::order) < b ⇒ (b::'b::order) < c ⇒
  (!!!x y. x < y ⇒ f x < f y) ⇒ a < f c

lemma ord-eq-eq-trans: a = b ⇒ b = c ⇒ a = c
  by (rule ssubst)

end
proof

assume $r : \forall x y. x < y \implies f x < f y$
assume $a < f b$
also assume $b < c$ hence $f b < f c$ by (rule r)
finally (less-trans) show thesis.
qed

lemma order-le-less-subst2: ($a :: 'a::order) <= b \implies f b <= (c :: 'c::order) <=
(!!x y. x <= y \implies f x <= f y) <= a < c
proof

assume $r : !!x y. x <= y \implies f x <= f y$
assume $a <= f b$
also assume $b <= c$ hence $f b <= f c$ by (rule r)
finally (le-less-trans) show thesis.
qed

lemma order-le-less-subst1: ($a :: 'a::order) <= f b <= (b :: 'b::order) <= c <=
(!!x y. x < y \implies f x < f y) <= a < f c
proof

assume $r : !!x y. x < y \implies f x < f y$
assume $a < f b$
also assume $b < c$ hence $f b < f c$ by (rule r)
finally (le-less-trans) show thesis.
qed

lemma order-less-le-subst2: ($a :: 'a::order) < b <= f b <= (c :: 'c::order) <=
(!!x y. x < y <= f x <= f y) <= a < c
proof

assume $r : !!x y. x < y <= f x <= f y$
assume $a < f b$
also assume $b < c$ hence $f b < f c$ by (rule r)
finally (less-le-trans) show thesis.
qed

lemma order-less-le-subst1: ($a :: 'a::order) < f b <= (b :: 'b::order) <= c <=
(!!x y. x < y <= f x <= f y) <= a < f c
proof

assume $r : !!x y. x < y <= f x <= f y$
assume $a < f b$
also assume $b < c$ hence $f b < f c$ by (rule r)
finally (less-le-trans) show thesis.
qed

lemma order-subst1: ($a :: 'a::order) <= f b <= (b :: 'b::order) <= c <=
(!!x y. x <= y <= f x <= f y) <= a <= f c
proof

assume $r : !!x y. x <= y <= f x <= f y$
assume $a <= f b$
also assume $b <= c$ hence $f b <= f c$ by (rule r)
finally (order-trans) show ?thesis.

qed

lemma order-subst2: \((a::'a::order) \leq b \Longrightarrow f b \leq (c::'c::order) \Longrightarrow (\forall x y. x \leq y \Longrightarrow f x \leq f y) \Longrightarrow f a \leq c\)

proof
  assume r: \((\forall x y. x \leq y \Longrightarrow f x \leq f y)\)
  assume a \leq b hence f a \leq f b by (rule r)
  also assume f b \leq c
  finally (order-trans) show ?thesis.

qed

lemma ord-le-eq-subst: \(a \leq b \Longrightarrow f b = c \Longrightarrow (\forall x y. x \leq y \Longrightarrow f x \leq f y) \Longrightarrow f a \leq c\)

proof
  assume r: \((\forall x y. x \leq y \Longrightarrow f x \leq f y)\)
  assume a \leq b hence f a \leq f b by (rule r)
  also assume f b = c

qed

lemma ord-eq-le-subst: \(a = f b \Longrightarrow b \leq c \Longrightarrow (\forall x y. x < y \Longrightarrow f x < f y) \Longrightarrow a < f c\)

proof
  assume r: \((\forall x y. x < y \Longrightarrow f x < f y)\)
  assume a = f b
  also assume b \leq c hence f b \leq f c by (rule r)

qed

lemma ord-less-eq-subst: \(a < b \Longrightarrow f b = c \Longrightarrow (\forall x y. x < y \Longrightarrow f x < f y) \Longrightarrow f a < c\)

proof
  assume r: \((\forall x y. x < y \Longrightarrow f x < f y)\)
  assume a < b hence f a < f b by (rule r)
  also assume f b = c
  finally (ord-less-eq-trans) show ?thesis.

qed

lemma ord-eq-less-subst: \(a = f b \Longrightarrow b < c \Longrightarrow (\forall x y. x < y \Longrightarrow f x < f y) \Longrightarrow a < f c\)

proof
  assume r: \((\forall x y. x < y \Longrightarrow f x < f y)\)
  assume a = f b
  also assume b < c hence f b < f c by (rule r)
  finally (ord-eq-less-trans) show ?thesis.

qed

Note that this list of rules is in reverse order of priorities.
lemmas \( [\text{trans}] = \)

\begin{align*}
\text{order-less-subst2} \\
\text{order-less-subst1} \\
\text{order-le-less-subst2} \\
\text{order-le-less-subst1} \\
\text{order-less-le-subst2} \\
\text{order-less-le-subst1} \\
\text{order-subst2} \\
\text{order-subst1} \\
\text{ord-le-eq-subst} \\
\text{ord-eq-le-subst} \\
\text{ord-less-eq-subst} \\
\text{ord-eq-less-subst} \\
\text{forw-subst} \\
\text{back-subst} \\
\text{rev-mp} \\
\text{mp}
\end{align*}

lemmas (in order) \( [\text{trans}] = \)

\begin{align*}
\text{neg-le-trans} \\
\text{le-neg-trans}
\end{align*}

lemmas (in preorder) \( [\text{trans}] = \)

\begin{align*}
\text{less-trans} \\
\text{less-asym'} \\
\text{le-less-trans} \\
\text{less-le-trans} \\
\text{order-trans}
\end{align*}

lemmas (in order) \( [\text{trans}] = \)

\begin{align*}
\text{order.antisym}
\end{align*}

lemmas (in ord) \( [\text{trans}] = \)

\begin{align*}
\text{ord-le-eq-trans} \\
\text{ord-eq-le-trans} \\
\text{ord-less-eq-trans} \\
\text{ord-eq-less-trans}
\end{align*}

lemmas \( [\text{trans}] = \)

\begin{align*}
\text{trans}
\end{align*}

lemmas order-trans-rules =

\begin{align*}
\text{order-less-subst2} \\
\text{order-less-subst1} \\
\text{order-le-less-subst2} \\
\text{order-le-less-subst1} \\
\text{order-less-le-subst2} \\
\text{order-less-le-subst1} \\
\text{order-subst2}
\end{align*}
These support proving chains of decreasing inequalities $a \geq b \geq c \ldots$ in Isar proofs.

**lemma xt1 [no-atp]:**

- $a = b \implies b > c \implies a > c$
- $a > b \implies b = c \implies a > c$
- $a = b \implies b \geq c \implies a \geq c$
- $a \geq b \implies b = c \implies a \geq c$
- $(x::'a::order) \geq y \implies y \geq x \implies x = y$
- $(x::'a::order) \geq y \implies y \geq z \implies z \geq y$
- $(x::'a::order) > y \implies y \geq z \implies x > z$
- $(a::'a::order) \geq b \implies b > a \implies P$
- $(x::'a::order) > y \implies y \geq z \implies x > z$
- $(a::'a::order) \geq b \implies a \neq b \implies a > b$
- $(a::'a::order) \neq b \implies a \geq b \implies a > b$
- $a = f b \implies b > c \implies (\forall x. x > y \implies f x > f y) \implies a > f c$
- $a > b \implies f b = c \implies (\forall x. x > y \implies f x > f y) \implies f a > c$
- $a = f b \implies b \geq c \implies (\forall x. x \geq y \implies f x \geq f y) \implies a \geq f c$
- $a \geq b \implies f b = c \implies (\forall x. x \geq y \implies f x \geq f y) \implies f a \geq c$

**by auto**

**lemma xt2 [no-atp]:**

- **assumes** $(a::'a::order) \geq f b$
- **and** $b \geq c$
- **and** $\forall x. x \geq y \implies f x \geq f y$
- **shows** $a \geq f c$
using assms by force

lemma xt3 [no-atp]:
assumes (a::'a::order) ≥ b
  and (f b::'b::order) ≥ c
  and ∀x y. x ≥ y ⇒ f x ≥ f y
shows f a ≥ c
using assms by force

lemma xt4 [no-atp]:
assumes (a::'a::order) > f b
  and (b::'b::order) ≥ c
  and ∀x y. x ≥ y ⇒ f x ≥ f y
shows a > f c
using assms by force

lemma xt5 [no-atp]:
assumes (a::'a::order) > b
  and (f b::'b::order) ≥ c
  and ∀x y. x > y ⇒ f x > f y
shows f a > c
using assms by force

lemma xt6 [no-atp]:
assumes (a::'a::order) ≥ f b
  and b > c
  and ∀x y. x > y ⇒ f x > f y
shows a > f c
using assms by force

lemma xt7 [no-atp]:
assumes (a::'a::order) ≥ b
  and (f b::'b::order) > c
  and ∀x y. x ≥ y ⇒ f x ≥ f y
shows f a > c
using assms by force

lemma xt8 [no-atp]:
assumes (a::'a::order) > f b
  and (b::'b::order) > c
  and ∀x y. x > y ⇒ f x > f y
shows a > f c
using assms by force

lemma xt9 [no-atp]:
assumes (a::'a::order) > b
  and (f b::'b::order) > c
  and ∀x y. x > y ⇒ f x > f y
shows f a > c
4.9  min and max – fundamental

**definition (in ord) min :: 'a => 'a => 'a where**

\[
\text{min } a \ b = (\text{if } a \leq b \text{ then } a \text{ else } b)
\]

**definition (in ord) max :: 'a => 'a => 'a where**

\[
\text{max } a \ b = (\text{if } a \leq b \text{ then } b \text{ else } a)
\]

**lemma min-absorb1: x \leq y \implies \text{min } x \ y = x**

by (simp add: min-def)

**lemma max-absorb2: x \leq y \implies \text{max } x \ y = y**

by (simp add: max-def)

**lemma min-absorb2: (y :: 'a::order) \leq x \implies \text{min } x \ y = y**

by (simp add: min-def)

**lemma max-absorb1: (y :: 'a::order) \leq x \implies \text{max } x \ y = x**

by (simp add: max-def)

**lemma max-min-same [simp]:**

\[
\text{fixes } x \ y :: 'a :: linorder
\]

**shows** \[
\text{max } x \ (\text{min } x \ y) = x \text{ max } (\text{min } x \ y) \ y = y \text{ max } y
\]

by (auto simp add: max-def min-def)

4.10  (Unique) top and bottom elements

**class bot =**

**fixes bot :: 'a (⊥)**

**class order-bot = order + bot +**

**assumes bot-least: \bot \leq a**

begin

**sublocale bot: ordering-top greater-eq greater bot**

by standard (fact bot-least)

**lemma le-bot:**

\[a \leq ⊥ \implies a = ⊥\]

by (fact bot.extremum-uniqueI)

**lemma bot-unique:**

\[a \leq ⊥ \iff a = ⊥\]

by (fact bot.extremum-unique)
THEORY “Orderings”

lemma not-less-bot:
\[ \neg a < \bot \]
by (fact bot.extremum-strict)

lemma bot-less:
\[ a \not= \bot \iff \bot < a \]
by (fact bot.not-eq-extremum)

lemma max-bot[simp]: max bot \( x = x \)
by (simp add: max-def bot-unique)

lemma max-bot2[simp]: max \( x \) bot = \( x \)
by (simp add: max-def bot-unique)

lemma min-bot[simp]: min bot \( x = bot \)
by (simp add: min-def bot-unique)

lemma min-bot2[simp]: min \( x \) bot = bot
by (simp add: min-def bot-unique)

end

class top =
fixes top :: 'a (\top)

class order-top = order + top +
assumes top-greatest: \( a \leq \top \)
begin

sublocale top: ordering-top less-eq less top
by standard (fact top-greatest)

lemma top-le:
\[ \top \leq a \implies a = \top \]
by (fact top.extremum-uniqueI)

lemma top-unique:
\[ \top \leq a \iff a = \top \]
by (fact top.extremum-unique)

lemma not-top-less:
\[ \neg \top < a \]
by (fact top.extremum-strict)

lemma less-top:
\[ a \not= \top \iff a < \top \]
by (fact top.not-eq-extremum)

lemma max-top[simp]: max top \( x = top \)
by (simp add: max-def top-unique)

lemma max-top2 [simp]: max x top = top
by (simp add: max-def top-unique)

lemma min-top [simp]: min top x = x
by (simp add: min-def top-unique)

lemma min-top2 [simp]: min x top = x
by (simp add: min-def top-unique)

end

4.11 Dense orders

class dense-order = order +
  assumes dense: x < y ⇒ (∃z. x < z ∧ z < y)

class dense-linorder = linorder + dense-order
begin

lemma dense-le:
  fixes y z :: 'a
  assumes ∃x. x < y ⇒ x ≤ z
  shows y ≤ z
proof (rule dense-le)
  assume ¬ ?thesis
  hence z < y by simp
  from dense [OF this] obtain x where x < y and z < x by safe
  moreover have x ≤ z using assms [OF x < y]
  ultimately show False by auto
qed

lemma dense-le-bounded:
  fixes x y z :: 'a
  assumes x < y
  assumes *: ∃w. [x < w ; w < y] ⇒ w ≤ z
  shows y ≤ z
proof (rule dense-le)
  fix w assume w < y
  from dense [OF x < y] obtain u where x < u u < y by safe
  from linear [of u w] show w ≤ z
  proof (rule disjE)
    assume u ≤ w
    from less-le-trans [OF x < u (u ≤ w) (w < y)] show w ≤ z by (rule *)
  next
next
assume \( w \leq u \)
from \( (w \leq u) \ast [OF \ (x < w \cdot u < y)] \)
show \( w \leq z \) by (rule order-trans)
qed

lemma dense-ge:
fixes \( y \ z :: 'a \)
assumes \( \forall x. \ z < x \Rightarrow y \leq x \)
shows \( y \leq z \)
proof (rule ccontr)
assume \( \neg \ ?thesis \)
hence \( z < y \) by simp
from dense \([OF this]\)
obtain \( x \) where \( x < y \) and \( z < x \) by safe
moreover have \( y \leq x \) using assms \([OF \ (z < x)]\).
ultimately show \( False \) by auto
qed

lemma dense-ge-bounded:
fixes \( x \ y \ z :: 'a \)
assumes \( z < x \)
assumes \( \ast : \ \forall w. \ [z < w \cdot w < x] \Rightarrow y \leq w \)
shows \( y \leq z \)
proof (rule dense-ge)
fix \( \ w \) assume \( \ z < w \)
from dense \([OF \ (z < x)]\) obtain \( u \) where \( z < u u < x \) by safe
from linear \([of \ u w]\)
show \( y \leq w \)
proof (rule disjE)
assume \( \ w \leq u \)
from \( \z < w \) le-less-trans \([OF \ (w \leq u \cdot u < x)]\)
show \( y \leq w \) by (rule \( \ast \))
next
assume \( \ u \leq w \)
from \( \ast : [OF \ (z < u \cdot u < x) \cdot u \leq w] \)
show \( y \leq w \) by (rule order-trans)
qed
qed

definition dense-ge-bound:
fixes \( y \ z :: 'a \)
assumes \( \forall x. \ z < x \Rightarrow y \leq x \)
shows \( y \leq z \)
proof (rule ccontr)
assume \( \neg \ ?thesis \)
hence \( z < y \) by simp
from dense \([OF this]\)
obtain \( x \) where \( x < y \) and \( z < x \) by safe
moreover have \( y \leq x \) using assms \([OF \ (z < x)]\).
ultimately show \( False \) by auto
qed

end

class no-top = order +
assumes gt-ex: \( \exists y. \ x < y \)

class no-bot = order +
assumes lt-ex: \( \exists y. \ y < x \)

class unbounded-dense-linorder = dense-linorder + no-top + no-bot
4.12 Wellorders

class wellorder = linorder +
  assumes less-induct [case-names less]: \( (\forall x. (\forall y. y < x \Rightarrow P y) \Rightarrow P x) \Rightarrow P a \)
begin

lemma wellorder-Least-lemma:
  fixes k :: 'a
  assumes P k
  shows LeastI: \( P (LEAST x. P x) \) and Least-le: \( (LEAST x. P x) \leq k \)
proof
  have \( P (LEAST x. P x) \land (LEAST x. P x) \leq k \)
  using assms proof (induct k rule: less-induct)
    case (less x) then have \( P x \) by simp
    show \( ?case \) proof (rule classical)
      assume assm: \( \neg (P (LEAST a. P a) \land (LEAST a. P a) \leq x) \)
      have \( \forall y. P y \Rightarrow x \leq y \)
      proof (rule classical)
        fix y
        assume \( P y \) and \( \neg x \leq y \)
        with less have \( P (LEAST a. P a) \) and \( (LEAST a. P a) \leq y \)
        by (auto simp add: not-le)
        with assm have \( x < (LEAST a. P a) \) and \( (LEAST a. P a) \leq y \)
        by auto
        then show \( x \leq y \) by auto
      qed
      with \( P x \) have Least: \( (LEAST a. P a) = x \)
      by (rule Least-equality)
      with \( P x \) show \( ?thesis \) by simp
    qed
    qed
    then show \( P (LEAST x. P x) \) and \( (LEAST x. P x) \leq k \) by auto
  qed

— The following 3 lemmas are due to Brian Huffman

lemma LeastI-ex: \( \exists x. P x \Rightarrow P (Least P) \)
  by (erule exE) (erule LeastI)

lemma LeastI2:
  \( P a \Rightarrow (\forall x. P x \Rightarrow Q x) \Rightarrow Q (Least P) \)
  by (blast intro: LeastI)

lemma LeastI2-ex:
  \( \exists a. P a \Rightarrow (\forall x. P x \Rightarrow Q x) \Rightarrow Q (Least P) \)
  by (blast intro: LeastI-ex)

lemma LeastI2-wellorder:
  assumes \( P a \)
  and \( \forall a. P a; \forall b. P b \Rightarrow a \leq b \) \( \Rightarrow Q a \)
shows \( Q \) (Least \( P \))

proof (rule LeastI2-order)
  show \( P \) (Least \( P \)) using \( \langle P \, a \rangle \) by (rule LeastI)
next
  fix \( y \) assume \( P \, y \) thus Least \( P \leq y \) by (rule Least-le)
next
  fix \( x \) assume \( P \, x \) \( \forall y. \, P \, y \longrightarrow x \leq y \) thus \( Q \, x \) by (rule assms(2))
qed

lemma LeastI2-wellorder-ex:
  assumes \( \exists \, x. \, P \, x \)
  and \( \forall a. \, \langle P \, a; \, \forall b. \, P \, b \longrightarrow a \leq b \rangle \Longrightarrow Q \, a \)
  shows \( Q \) (Least \( P \))
using assms by clarify (blast intro: LeastI2-wellorder)

lemma not-less-Least: \( k \prec (\text{LEAST } \, P \, x) \Longrightarrow \neg P \, k \)
apply (simp add: not-le [symmetric])
apply (erule contrapos-nn)
apply (erule Least-le)
done

lemma exists-least-iff: \( \exists \, n. \, P \, n \) \( \iff \) \( \exists \, n. \, P \, n \land (\forall m < n. \, \neg P \, m) \) (is \( \text{?lhs} \) \( \iff \) \( \text{?rhs} \))
proof
  assume \( \text{?rhs} \) thus \( \text{?lhs} \) by blast
next
  assume \( H \): \( \text{?lhs} \) then obtain \( n \) where \( n: \, P \, n \) by blast
  let \( ?x = \text{Least} \, P \)
  { fix \( m \) assume \( m: \, m < ?x \)
    from not-less-Least[OF \( m \)] have \( \neg P \, m \). }
  with LeastI-ex[OF \( H \)] show \( \text{?rhs} \) by blast
qed

end

4.13 Order on bool

instantiation bool :: \{order-bot, order-top, linorder\}
begin

definition le-bool-def [simp]: \( P \leq Q \) \( \iff \) \( P \longrightarrow Q \)
definition [simp]: \( P::\text{bool} \) \( < \) \( Q \) \( \iff \) \( \neg P \land Q \)
definition [simp]: \( \bot \) \( \iff \) \( \text{False} \)
**THEORY “Orderings”**

**definition**

\[ \text{simp}: \top \leftrightarrow \text{True} \]

**instance proof**

**qed auto**

**end**

**lemma le-boolI**: \((P \Rightarrow Q) \Rightarrow P \leq Q\)

**by simp**

**lemma le-boolI’**: \(P \Rightarrow Q \Rightarrow P \leq Q\)

**by simp**

**lemma le-boolE**: \(P \leq Q \Rightarrow P \Rightarrow (Q \Rightarrow R) \Rightarrow R\)

**by simp**

**lemma le-boolD**: \(P \leq Q \Rightarrow P \rightarrow Q\)

**by simp**

**lemma bot-boolE**: \(\bot \Rightarrow P\)

**by simp**

**lemma top-boolI**: \(\top\)

**by simp**

**lemma [code]:**

\(\text{False} \leq b \leftrightarrow \text{True}\)

\(\text{True} \leq b \leftrightarrow b\)

\(\text{False} < b \leftrightarrow b\)

\(\text{True} < b \leftrightarrow \text{False}\)

**by simp-all**

**4.14 Order on - ⇒ -**

**instantiation** fun :: (type, ord) ord

**begin**

**definition**

\(\text{le-fun-def}: f \leq g \leftrightarrow (\forall x. f x \leq g x)\)

**definition**

\((f::'a ⇒ 'b) < g \leftrightarrow f \leq g \land \neg (g \leq f)\)

**instance ..**

**end**

**instance** fun :: (type, preorder) preorder **proof**
instance fun :: (type, order) order proof
qed (auto simp add: le-fun-def intro: order-trans order.antisym)

instantiation fun :: (type, bot) bot begin

definition ⊥ = (λx. ⊥)

instance ..
end

instantiation fun :: (type, order-bot) order-bot begin

lemma bot-apply [simp, code]:
⊥ x = ⊥
by (simp add: bot-fun-def)

instance proof
qed (simp add: le-fun-def)
end

instantiation fun :: (type, top) top begin

definition [no-atp]: ⊤ = (λx. ⊤)

instance ..
end

instantiation fun :: (type, order-top) order-top begin

lemma top-apply [simp, code]:
⊤ x = ⊤
by (simp add: top-fun-def)

instance proof
qed (simp add: le-fun-def)
end
lemma le-funI: \( (\forall x. f x \leq g x) \Rightarrow f \leq g \)
  unfolding le-fun-def by simp

lemma le-funE: \( f \leq g \Rightarrow (f x \leq g x \Rightarrow P) \Rightarrow P \)
  unfolding le-fun-def by simp

lemma le-funD: \( f \leq g \Rightarrow f x \leq g x \)
  by (rule le-funE)

4.15 Order on unary and binary predicates

lemma predicate1I:
  assumes PQ: \( \forall x. P x \Rightarrow Q x \)
  shows P \( \leq Q \)
  apply (rule le-funI)
  apply (rule le-boolI)
  apply (rule PQ)
  apply assumption
  done

lemma predicate1D:
  P \( \leq Q \Rightarrow P x \Rightarrow Q x \)
  apply (erule le-funE)
  apply (erule le-boolE)
  apply assumption+
  done

lemma rev-predicate1D:
  P x \( \Rightarrow P \leq Q \Rightarrow Q x \)
  by (rule predicate1ID)

lemma predicate2I:
  assumes PQ: \( \forall x y. P x y \Rightarrow Q x y \)
  shows P \( \leq Q \)
  apply (rule le-funI+)
  apply (rule le-boolI+)
  apply (rule PQ)
  apply assumption
  done

lemma predicate2D:
  P \( \leq Q \Rightarrow P x y \Rightarrow Q x y \)
  apply (erule le-funE+)
  apply (erule le-boolE+)
  apply assumption+
  done

lemma rev-predicate2D:
THEORY "Orderings"

\[ P \times y \Rightarrow P \leq Q \Rightarrow Q \times y \]
by (rule predicate2D)

lemma \textit{bot1E} [no-atp]: \( \bot \times x \Rightarrow P \)
by (simp add: bot-fun-def)

lemma \textit{bot2E}: \( \bot \times y \Rightarrow P \)
by (simp add: bot-fun-def)

lemma \textit{top1I}: \( \top \times x \)
by (simp add: top-fun-def)

lemma \textit{top2I}: \( \top \times y \)
by (simp add: top-fun-def)

4.16 Name duplicates

lemmas \textit{antisym} = \textit{order.antisym}

lemmas \textit{eq-iff} = \textit{order.eq-iff}

lemmas \textit{order-eq-refl} = \textit{preorder-class.eq-refl}

lemmas \textit{order-less-irrefl} = \textit{preorder-class.less-irrefl}

lemmas \textit{order-less-imp-le} = \textit{preorder-class.less-imp-le}

lemmas \textit{order-less-not-sym} = \textit{preorder-class.less-not-sym}

lemmas \textit{order-less-asym} = \textit{preorder-class.less-asym}

lemmas \textit{order-less-trans} = \textit{preorder-class.less-trans}

lemmas \textit{order-le-less-trans} = \textit{preorder-class.le-less-trans}

lemmas \textit{order-less-imp-not-less} = \textit{preorder-class.less-imp-not-less}

lemmas \textit{order-less-imp-triv} = \textit{preorder-class.less-imp-triv}

lemmas \textit{order-less-asym'} = \textit{preorder-class.less-asym'}

lemmas \textit{order-less-le} = \textit{order-class.less-le}

lemmas \textit{order-le-less} = \textit{order-class.le-less}

lemmas \textit{order-le-imp-less-or-eq} = \textit{order-class.le-imp-less-or-eq}

lemmas \textit{order-less-imp-not-eq} = \textit{order-class.less-imp-not-eq}

lemmas \textit{order-less-imp-not-eq2} = \textit{order-class.less-imp-not-eq2}

lemmas \textit{order-neq-le-trans} = \textit{order-class.neq-le-trans}

lemmas \textit{order-le-neq-trans} = \textit{order-class.le-neq-trans}

lemmas \textit{order-eq-iff} = \textit{order-class.order-eq-iff}

lemmas \textit{order-antisym-conv} = \textit{order-class.antisym-conv}

lemmas \textit{linorder-linear} = \textit{linorder-class.linear}

lemmas \textit{linorder-less-linear} = \textit{linorder-class.less-linear}

lemmas \textit{linorder-le-less-linear} = \textit{linorder-class.le-less-linear}

lemmas \textit{linorder-le-cases} = \textit{linorder-class.le-cases}

lemmas \textit{linorder-not-less} = \textit{linorder-class.not-less}

lemmas \textit{linorder-not-le} = \textit{linorder-class.not-le}

lemmas \textit{linorder-neq-iff} = \textit{linorder-class.neq-iff}
lemmas linorder-neqE = linorder-class.neqE

end

5 Groups, also combined with orderings

theory Groups
  imports Orderings
begin

5.1 Dynamic facts

named-theorems ac-simps associativity and commutativity simplification rules
and algebra-simps algebra simplification rules for rings
and algebra-split-simps algebra simplification rules for rings, with potential goal splitting
and field-simps algebra simplification rules for fields
and field-split-simps algebra simplification rules for fields, with potential goal splitting

The rewrites accumulated in algebra-simps deal with the classical algebraic structures of groups, rings and family. They simplify terms by multiplying everything out (in case of a ring) and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides group and ring equalities but also helps with inequalities.

Of course it also works for fields, but it knows nothing about multiplicative inverses or division. This is catered for by field-simps.

Facts in field-simps multiply with denominators in (in)equations if they can be proved to be non-zero (for equations) or positive/negative (for inequalities). Can be too aggressive and is therefore separate from the more benign algebra-simps.

Collections algebra-split-simps and field-split-simps correspond to algebra-simps and field-simps but contain more aggressive rules that may lead to goal splitting.

5.2 Abstract structures

These locales provide basic structures for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

locale semigroup =
fixes f :: 'a ⇒ 'a ⇒ 'a (infixl * 70)
assumes assoc [ac-simps]: a * b * c = a * (b * c)

locale abel-semigroup = semigroup +
assumes commute [ac-simps]: \( a \cdot b = b \cdot a \)

begin

lemma left-commute [ac-simps]: \( b \cdot (a \cdot c) = a \cdot (b \cdot c) \)

proof

  have \((b \cdot a) \cdot c = (a \cdot b) \cdot c\)
  by (simp only: commute)

  then show \(\text{thesis}\)
  by (simp only: assoc)

qed

end

locale monoid = semigroup +
  fixes \(z::'a (1)\)

assumes left-neutral [simp]: \(1 \cdot a = a\)

assumes right-neutral [simp]: \(a \cdot 1 = a\)

locale comm-monoid = abel-semigroup +
  fixes \(z::'a (1)\)

assumes comm-neutral: \(a \cdot 1 = a\)

begin

locale group = semigroup +
  fixes \(z::'a (1)\)
  fixes inverse :: 'a ⇒ 'a

assumes group-left-neutral: \(1 \cdot a = a\)

assumes left-inverse [simp]: \(\text{inverse } a \cdot a = 1\)

begin

lemma left-cancel: \(a \cdot b = a \cdot c \iff b = c\)

proof

  assume \(a \cdot b = a \cdot c\)

  then have \(\text{inverse } a \cdot (a \cdot b) = \text{inverse } a \cdot (a \cdot c)\) by simp

  then have \((\text{inverse } a \cdot a) \cdot b = (\text{inverse } a \cdot a) \cdot c\)
  by (simp only: assoc)

  then show \(b = c\) by (simp add: group-left-neutral)

qed simp

sublocale monoid

proof

  fix \(a\)

  have \(\text{inverse } a \cdot a = 1\) by simp

  then have \(\text{inverse } a \cdot (a \cdot 1) = \text{inverse } a \cdot a\)
by (simp add: group-left-neutral assoc [symmetric])
with left-cancel show a * 1 = a
  by (simp only; left-cancel)
qed (fact group-left-neutral)

lemma inverse-unique: 
  assumes a * b = 1
  shows inverse a = b
proof – 
  from assms have inverse a * (a * b) = inverse a
    by simp
  then show ?thesis
    by (simp add: assoc [symmetric])
qed

lemma inverse-neutral [simp]: inverse 1 = 1
  by (rule inverse-unique) simp

lemma inverse-inverse [simp]: inverse (inverse a) = a
  by (rule inverse-unique) simp

lemma right-inverse [simp]: a * inverse a = 1
proof – 
  have a * inverse a = inverse (inverse a) * inverse a
    by simp
  also have . . . = 1
    by (rule left-inverse)
  then show ?thesis by simp
qed

lemma inverse-distrib-swap: inverse (a * b) = inverse b * inverse a
proof (rule inverse-unique) 
  have a * b * (inverse b * inverse a) =
    a * (b * inverse b) * inverse a
    by (simp only; assoc)
  also have . . . = 1
    by simp
  finally show a * b * (inverse b * inverse a) = 1 .
qed

lemma right-cancel: b * a = c * a ⟷ b = c
proof
  assume b * a = c * a
  then have b * a * inverse a = c * a * inverse a
    by simp
  then show b = c
    by (simp add: assoc)
qed simp
5.3 Generic operations

class zero =
  fixes zero :: 'a (0)

class one =
  fixes one :: 'a (1)

hide-const (open) zero one

lemma Let-0 [simp]: Let 0 f = f 0
  unfolding Let-def ..

lemma Let-1 [simp]: Let 1 f = f 1
  unfolding Let-def ..

setup (Reorient-Proc.add
  (fn Const(const-name Groups.zero, _) => true
  | Const(const-name Groups.one, _) => true
  | _ => false)
)

simproc-setup reorient-zero (0 = x) = (K Reorient-Proc.proc)
simproc-setup reorient-one (1 = x) = (K Reorient-Proc.proc)

typed-print-translation (let
  fun tr' c = (c, fn ctxt => fn T => fn ts =>
    if null ts andalso Printer.type-emphasis ctxt T then
      Syntax.const syntax-const (\-\-constrain) $ Syntax.const c $
      Syntax-Phases.term-of-typ ctxt T
    else raise Match);
  in map tr' [const-syntax Groups.one, const-syntax Groups.zero] end
) — show types that are presumably too general

class plus =
  fixes plus :: 'a ⇒ 'a ⇒ 'a (infixl + 65)

class minus =
  fixes minus :: 'a ⇒ 'a ⇒ 'a (infixl - 65)

class uminus =
  fixes uminus :: 'a ⇒ 'a (\- \- [81] 80)

class times =
  fixes times :: 'a ⇒ 'a ⇒ 'a (infixl * 70)
5.4 Semigroups and Monoids

class semigroup-add = plus +
  assumes add-assoc: \((a + b) + c = a + (b + c)\)
begin

sublocale add: semigroup plus
  by standard (fact add-assoc)

declare add.assoc [algebra-simps, algebra-split-simps, field-simps, field-split-simps]

end

hide-fact add-assoc

class ab-semigroup-add = semigroup-add +
  assumes add-commute: \(a + b = b + a\)
begin

sublocale add: abel-semigroup plus
  by standard (fact add-commute)

declare add.commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]
add.left-commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]
lemmas add-ac = add.assoc add.commute add.left-commute

end

hide-fact add-commute

lemmas add-ac = add.assoc add.commute add.left-commute

class semigroup-mult = times +
  assumes mult-assoc: \((a * b) * c = a * (b * c)\)
begin

sublocale mult: semigroup times
  by standard (fact mult-assoc)

declare mult.assoc [algebra-simps, algebra-split-simps, field-simps, field-split-simps]

end

hide-fact mult-assoc

class ab-semigroup-mult = semigroup-mult +
  assumes mult-commute: \(a * b = b * a\)
begin

**THEORY “Groups”**

sublocale mult: abel-semigroup times
  by standard (fact mult-commute)

declare mult.commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]
  mult.left-commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]

lemmas mult-ac = mult.assoc mult.commute mult.left-commute
end

hide-fact mult-commute

lemmas mult-ac = mult.assoc mult.commute mult.left-commute

class monoid-add = zero + semigroup-add +
  assumes add-0-left: 0 + a = a
  and add-0-right: a + 0 = a
begin

subclass add: monoid plus 0
  by standard (fact add-0-left add-0-right)+
end

lemma zero-reorient: 0 = x ←→ x = 0
  by (fact eq-commute)

class comm-monoid-add = zero + ab-semigroup-add +
  assumes add-0: 0 + a = a
begin

subclass monoid-add
  by standard (simp-all add: add-0 add.commute [of - 0])

sublocale add: comm-monoid plus 0
  by standard (simp add: ac-simps)
end

class monoid-mult = one + semigroup-mult +
  assumes mult-1-left: 1 * a = a
  and mult-1-right: a * 1 = a
begin

subclass mult: monoid times 1
  by standard (fact mult-1-left mult-1-right)+
end
lemma one-reorient: 1 = x \iff x = 1
  by (fact eq-commute)

class comm-monoid-mult = one + ab-semigroup-mult +
  assumes mult-1: 1 * a = a
begin

subclass monoid-mult
  by standard (simp-all add: mult-1 mult.commute [of - 1])

sublocale mult: comm-monoid times 1
  by standard (simp add: ac-simps)
end

class cancel-semigroup-add = semigroup-add +
  assumes add-left-imp-eq: a + b = a + c \implies b = c
  assumes add-right-imp-eq: b + a = c + a \implies b = c
begin

lemma add-left-cancel [simp]: a + b = a + c \iff b = c
  by (blast dest: add-left-imp-eq)

lemma add-right-cancel [simp]: b + a = c + a \iff b = c
  by (blast dest: add-right-imp-eq)
end

class cancel-ab-semigroup-add = ab-semigroup-add + minus +
  assumes add-diff-cancel-left" [simp]: (a + b) - a = b
  assumes diff-diff-add [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
    a - b - c = a - (b + c)
begin

lemma add-diff-cancel-right" [simp]: (a + b) - b = a
  using add-diff-cancel-left" [of b a] by (simp add: ac-simps)

subclass cancel-semigroup-add
proof
  fix a b c :: 'a
  assume a + b = a + c
  then have a + b - a = a + c - a
    by simp
  then show b = c
    by simp
next
  fix a b c :: 'a
  assume b + a = c + a
  then have b + a - a = c + a - a

by simp
then show \( b = c \)
  by simp
qed

lemma add-diff-cancel-left [simp]: \((c + a) - (c + b) = a - b\)
unfolding diff-diff-add [symmetric] by simp

lemma add-diff-cancel-right [simp]: \((a + c) - (b + c) = a - b\)
using add-diff-cancel-left [symmetric] by (simp add: ac-simps)

lemma diff-right-commute: \(a - c - b = a - b - c\)
  by (simp add: diff-diff-add add.commute)

end

class cancel-comm-monoid-add = cancel-ab-semigroup-add + comm-monoid-add
begin

lemma diff-zero [simp]: \(a - 0 = a\)
  using add-diff-cancel-right' [of a 0] by simp

lemma diff-cancel [simp]: \(a - a = 0\)
proof
  have \((a + 0) - (a + 0) = 0\)
    by (simp only: add-diff-cancel-left diff-zero)
  then show \(\text{thesis}\) by simp
qed

lemma add-implies-diff:
  assumes \(c + b = a\)
  shows \(c = a - b\)
proof
  from assms have \((b + c) - (b + 0) = a - b\)
    by (simp add: add.commute)
  then show \(c = a - b\) by simp
qed

lemma add-cancel-right-right [simp]: \(a = a + b \iff b = 0\)
  (is \(\text{?P} \iff \text{?Q}\))
p
proof
  assume \(\text{?Q}\)
  then show \(\text{?P}\) by simp
next
  assume \(\text{?P}\)
  then have \(a - a = a + b - a\) by simp
  then show \(\text{?Q}\) by simp
qed
lemma add-cancel-right-left [simp]: \( a = b + a \iff b = 0 \)
using add-cancel-right-right [of a b] by (simp add: ac-simps)

lemma add-cancel-left-right [simp]: \( a + b = a \iff b = 0 \)
by (auto dest: sym)

lemma add-cancel-left-left [simp]: \( b + a = a \iff b = 0 \)
by (auto dest: sym)

end

class comm-monoid-diff = cancel-comm-monoid-add +
  assumes zero-diff [simp]: \( 0 - a = 0 \)
begin

lemma diff-add-zero [simp]: \( a - (a + b) = 0 \)
proof
  have \( a - (a + b) = (a + 0) - (a + b) \)
    by simp
  also have \( \ldots = 0 \)
    by (simp only: add-diff-cancel-left zero-diff)
  finally show ?thesis .
qed

derive

end

5.5 Groups

class group-add = minus + uminus + monoid-add +
  assumes left-minus: \(- a + a = 0 \)
  assumes add-uminus-conv-diff [simp]: \( a + (- b) = a - b \)
begin

lemma diff-conv-add-uminus: \( a - b = a + (- b) \)
  by simp

sublocale add: group plus 0 uminus
  by standard (simp-all add: left-minus)

lemma minus-unique: \( a + b = 0 \implies - a = b \)
  by (fact add.inverse-unique)

lemma minus-zero: \(- 0 = 0 \)
  by (fact add.inverse-neutral)

lemma minus-minus: \(- (- a) = a \)
  by (fact add.inverse-inverse)

lemma right-minus: \( a + - a = 0 \)
by (fact add.right-inverse)

lemma diff-self [simp]: \( a - a = 0 \)
using right-minus [of a] by simp

subclass cancel-semigroup-add
by standard (simp-all add: add.left-cancel add.right-cancel)

lemma minus-add-cancel [simp]: \(- a + (a + b) = b\)
by (simp add: add.assoc [symmetric])

lemma add-minus-cancel [simp]: \( a + (- a + b) = b \)
by (simp add: add.assoc [symmetric])

lemma diff-add-cancel [simp]: \( a - b + b = a \)
by (simp only: diff-conv-add-uminus add assoc)

lemma add-diff-cancel [simp]: \( a + b - b = a \)
by (simp only: diff-conv-add-uminus add assoc)

lemma minus-add: \(- (a + b) = - b + - a\)
by (fact add.inverse-distrib-swap)

lemma right-minus-eq [simp]: \( a - b = 0 \iff a = b \)
proof
  assume \( a - b = 0 \)
  have \( a = (a - b) + b \) by (simp add: add.assoc)
  also have \( \ldots = b \) using \( a - b = 0 \) by simp
  finally show \( a = b \).
next
  assume \( a = b \)
  then show \( a - b = 0 \) by simp
qed

lemma eq-iff-diff-eq-0: \( a = b \iff a - b = 0 \)
by (fact right-minus-eq [symmetric])

lemma diff-0 [simp]: \( 0 - a = - a \)
by (simp only: diff-conv-add-uminus add-0-left)

lemma diff-0-right [simp]: \( a - 0 = a \)
by (simp only: diff-conv-add-uminus minus-zero add-0-right)

lemma diff-minus-eq-add [simp]: \( a - b = a + b \)
by (simp only: diff-conv-add-uminus minus-minus)

lemma neg-equal-iff-equal [simp]: \( - a = - b \iff a = b \)
proof
  assume \(- a = - b\)

then have \((- a) = - (- b)\) by simp
then show \(a = b\) by simp
next
  assume \(a = b\)
  then show \(- a = - b\) by simp
qed

lemma neg-equal-0-iff-equal [simp]: \((- a) = 0 \iff a = 0\)
  by (subst neg-equal-iff-equal [symmetric]) simp

lemma neg-0-equal-iff-equal [simp]: \(0 = - a \iff 0 = a\)
  by (subst neg-equal-iff-equal [symmetric]) simp

The next two equations can make the simplifier loop!

lemma equation-minus-iff: \(a = - b \iff b = - a\)
  proof -
    have \((- - a) = - b \iff - a = b\)
      by (rule neg-equal-iff-equal)
    then show \(?thesis\)
      by (simp add: eq-commute)
  qed

lemma minus-equation-iff: \(- a = b \iff - b = a\)
  proof -
    have \(- a = - (- b) \iff a = -b\)
      by (rule neg-equal-iff-equal)
    then show \(?thesis\)
      by (simp add: eq-commute)
  qed

lemma eq-neg-iff-add-eq-0: \(a = - b \iff a + b = 0\)
  proof
    assume \(a = - b\)
    then show \(a + b = 0\) by simp
  next
    assume \(a + b = 0\)
    moreover have \(a + (b + - b) = (a + b) + - b\)
      by (simp only: add.assoc)
    ultimately show \(a = - b\)
      by simp
  qed

lemma add-eq-0-iff2: \(a + b = 0 \iff a = - b\)
  by (fact eq-neg-iff-add-eq-0 [symmetric])

lemma neg-eq-iff-add-eq-0: \(- a = b \iff a + b = 0\)
  by (auto simp add: add-eq-0-iff2)

lemma add-eq-0-iff: \(a + b = 0 \iff b = - a\)
by (auto simp add: neg-eq-iff-add-eq-0 [symmetric])

lemma minus-diff-eq [simp]: \(- (a - b) = b - a\)
  by (simp only: neg-eq-iff-add-eq-0 diff-conv-add-uminus add.assoc minus-add-cancel)

lemma add-diff-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
  \(a + (b - c) = (a + b) - c\)
  by (simp only: diff-conv-add-uminus add.assoc)

lemma diff-add-diff-swap: \(a - (b + c) = a - c - b\)
  by (simp only: diff-conv-add-uminus add.assoc minus-add)

lemma diff-eq-eq-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
  \(a - b = c \iff a = c + b\)
  by auto

lemma eq-diff-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
  \(a = c - b \iff a + b = c\)
  by auto

lemma diff-diff-eq2 [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
  \(a - (b - c) = (a + c) - b\)
  by (simp only: diff-conv-add-uminus add.assoc) simp

lemma diff-eq-diff-eq: \(a - b = c - d \implies a = b \iff c = d\)
  by (simp only: eq-iff-diff-eq-0 [of a b] eq-iff-diff-eq-0 [of c d])

end

class ab-group-add = minus + uminus + comm-monoid-add +
  assumes ab-left-minus: \(- a + a = 0\)
  assumes ab-diff-conv-add-uminus: \(a - b = a + (- b)\)
begin

subclass group-add
  by standard (simp-all add: ab-left-minus ab-diff-conv-add-uminus)

subclass cancel-comm-monoid-add
proof
  fix a b c :: 'a
  have \(b + a - a = b\)
    by simp
  then show \(a + b - a = b\)
    by (simp add: ac-simps)
  show \(a - b - c = a - (b + c)\)
    by (simp add: algebra-simps)
qed
THEORY “Groups” 121

lemma uminus-add-conv-diff [simp]: \(- a + b = b - a\)
  by (simp add: add.commute)

lemma minus-add-distrib [simp]: \(- (a + b) = -a + -b\)
  by (simp add: algebra-simps)

lemma diff-add-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
  \((a - b) + c = (a + c) - b\)
  by (simp add: algebra-simps)

lemma minus-diff-commute:
  \(-b - a = -a - b\)
  by (simp only: diff-conv-add-uminus add.commute)

end

5.6 (Partially) Ordered Groups

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society, 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press, 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer

class ordered-ab-semigroup-add = order + ab-semigroup-add +
  assumes add-left-mono: \(a \leq b \implies c + a \leq c + b\)
begin

lemma add-right-mono: \(a \leq b \implies a + c \leq b + c\)
  by (simp add: add.commute [of \(-c\)] add-left-mono)

non-strict, in both arguments

lemma add-mono: \(a \leq b \implies c \leq d \implies a + c \leq b + d\)
  by (simp add: add.commute add-left-mono add-right-mono [THEN order-trans])

end

Strict monotonicity in both arguments

class strict-ordered-ab-semigroup-add = ordered-ab-semigroup-add +
  assumes add-strict-mono: \(a < b \implies c < d \implies a + c < b + d\)
class ordered-cancel-ab-semigroup-add = 
  ordered-ab-semigroup-add + cancel-ab-semigroup-add
begin

lemma add-strict-left-mono: a < b \implies c + a < c + b
by (auto simp add: less_le add-left-mono)

lemma add-strict-right-mono: a < b \implies a + c < b + c
by (simp add: add.commute [of - c] add-strict-left-mono)

subclass strict-ordered-ab-semigroup-add
proof
  show \(\forall a b c d. [a < b; c < d] \implies a + c < b + d\)
  by (iprover intro: add-strict-left-mono add-strict-right-mono less-trans)
qed

lemma add-less-le-mono: a < b \implies c \leq d \implies a + c < b + d
by (iprover intro: add-left-mono add-strict-right-mono less-le-trans)

lemma add-le-less-mono: a \leq b \implies c < d \implies a + c < b + d
by (iprover intro: add-strict-left-mono add-right-mono less-le-trans)
end

class ordered-ab-semigroup-add-imp-le = ordered-cancel-ab-semigroup-add +
  assumes add-le-imp-le-left: c + a \leq c + b \implies a \leq b
begin

lemma add-less-imp-less-left:
  assumes less: c + a < c + b
  shows a < b
proof
  from less have le: c + a \leq c + b
  by (simp add: order-le-less)
  have a \leq b
  using add-le-imp-le-left [OF le] .
  moreover have a \neq b
  proof (rule ccontr)
    assume \neg \thesis
    then have a = b by simp
    then have c + a = c + b by simp
    with less show False by simp
  qed
  ultimately show a < b
  by (simp add: order-le-less)
  qed

lemma add-less-imp-less-right: a + c < b + c \implies a < b
by (rule add-less-imp-less-left [of c]) (simp add: add.commute)
lemma add-less-cancel-left [simp]: \( c + a < c + b \iff a < b \)
  by (blast intro: add-less-imp-less-left add-strict-left-mono)

lemma add-less-cancel-right [simp]: \( a + c < b + c \iff a < b \)
  by (blast intro: add-less-imp-less-right add-strict-right-mono)

lemma add-le-cancel-left [simp]: \( c + a \leq c + b \iff a \leq b \)
  by (auto simp: dest: add-le-imp-le-left add-left-mono)

lemma add-le-cancel-right [simp]: \( a + c \leq b + c \iff a \leq b \)
  by (simp add: add.commute[of a c] add.commute[of b c])

lemma add-le-imp-le-right: \( a + c \leq b + c \implies a \leq b \)
  by simp

lemma max-add-distrib-left: \( \max x y + z = \max (x + z) (y + z) \)
  unfolding max-def by auto

lemma min-add-distrib-left: \( \min x y + z = \min (x + z) (y + z) \)
  unfolding min-def by auto

lemma max-add-distrib-right: \( x + \max y z = \max (x + y) (x + z) \)
  unfolding max-def by auto

lemma min-add-distrib-right: \( x + \min y z = \min (x + y) (x + z) \)
  unfolding min-def by auto

end

5.7 Support for reasoning about signs

class ordered-comm-monoid-add = comm-monoid-add + ordered-ab-semigroup-add
begin

lemma add-nonneg-nonneg [simp]: \( 0 \leq a \implies 0 \leq b \implies 0 \leq a + b \)
  using add-mono[of 0 a 0 b] by simp

lemma add-nonpos-nonpos: \( a \leq 0 \implies b \leq 0 \implies a + b \leq 0 \)
  using add-mono[of a 0 b 0] by simp

lemma add-nonneg-eq-0-iff: \( 0 \leq x \implies 0 \leq y \implies x + y = 0 \iff x = 0 \land y = 0 \)
  using add-left-mono[of 0 y x] add-right-mono[of 0 x y] by auto

lemma add-nonpos-eq-0-iff: \( x \leq 0 \implies y \leq 0 \implies x + y = 0 \iff x = 0 \land y = 0 \)
  using add-left-mono[of y 0 x] add-right-mono[of x 0 y] by auto

lemma add-increasing: \( 0 \leq a \implies b \leq c \implies b \leq a + c \)
  using add-mono[of 0 a b c] by simp
lemma add-increasing2: $0 \leq c \Rightarrow b \leq a \Rightarrow b \leq a + c$
  by (simp add: add-increasing add.commute [of a])

lemma add-decreasing: $a \leq 0 \Rightarrow c \leq b \Rightarrow a + c \leq b$
  using add-mono [of a b c] by simp

lemma add-decreasing2: $c \leq 0 \Rightarrow a \leq b \Rightarrow a + c \leq b$
  using add-mono [of a b c 0] by simp

lemma add-pos-nonneg: $0 < a \Rightarrow 0 \leq b \Rightarrow 0 < a + b$
  using less-le-trans [of a a + b] by (simp add: add-increasing2)

lemma add-pos-pos: $0 < a \Rightarrow 0 < b \Rightarrow 0 < a + b$
  by (intro add-pos-nonneg less-imp-le)

lemma add-nonneg-pos: $0 < a \Rightarrow 0 < b \Rightarrow 0 < a + b$
  using add-pos-nonneg [of b a] by (simp add: add-commute)

lemma add-neg-nonpos: $a < 0 \Rightarrow b \leq 0 \Rightarrow a + b < 0$
  using le-less-trans [of b a + b] by (simp add: add-decreasing2)

lemma add-neg-neg: $a < 0 \Rightarrow b < 0 \Rightarrow a + b < 0$
  by (intro add-neg-nonpos less-imp-le)

lemma add-nonpos-neg: $a < 0 \Rightarrow b < 0 \Rightarrow a + b < 0$
  using add-neg-nonpos [of b a] by (simp add: add-commute)

lemmas add-sign-intros =
  add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonpos
  add-neg-nonpos add-neg-neg add-nonneg-pos add-nonneg-pos add-nonpos-pos
end

class strict-ordered-comm-monoid-add = comm-monoid-add + strict-ordered-ab-semigroup-add
begin

lemma pos-add-strict: $0 < a \Rightarrow b < c \Rightarrow b < a + c$
  using add-strict-mono [of 0 a b c] by simp
end

class ordered-cancel-comm-monoid-add = ordered-comm-monoid-add + cancel-ab-semigroup-add
begin

subclass ordered-cancel-ab-semigroup-add ..
subclass strict-ordered-comm-monoid-add ..

lemma add-strict-increasing: $0 < a \Rightarrow b \leq c \Rightarrow b < a + c$
using add-less-le-mono [of 0 a b c] by simp

lemma add-strict-increasing2: 0 ≤ a ⟹ b < c ⟹ b < a + c
  using add-le-less-mono [of 0 a b c] by simp

end

class ordered-ab-semigroup-monoid-add-imp-le = monoid-add + ordered-ab-semigroup-add-imp-le
  begin
  subclass cancel-comm-monoid-add
    by standard auto
  subclass ordered-cancel-comm-monoid-add
    by standard
  end

class ordered-ab-group-add = ab-group-add + ordered-ab-semigroup-add
  begin
  subclass ordered-cancel-ab-semigroup-add ..
  subclass ordered-ab-semigroup-monoid-add-imp-le
  proof
fix $a$, $b$, $c$ :: 'a
assume $c + a \leq c + b$
then have $(-c) + (c + a) \leq (-c) + (c + b)$
  by (rule add-left-mono)
then have $((-c) + c) + a \leq ((-c) + c) + b$
  by (simp only: add.assoc)
then show $a \leq b$ by simp
qed

lemma max-diff-distrib-left: $\max x y - z = \max (x - z) (y - z)$
using max-add-distrib-left [of $x y z$] by simp

lemma min-diff-distrib-left: $\min x y - z = \min (x - z) (y - z)$
using min-add-distrib-left [of $x y z$] by simp

lemma le-imp-neg-le:
  assumes $a \leq b$
  shows $-b \leq -a$
proof
  from assms have $-a + a \leq -a + b$
    by (rule add-left-mono)
  then have $0 \leq -a + b$
    by simp
  then have $0 + (-b) \leq (-a + b) + (-b)$
    by (rule add-right-mono)
  then show $\text{thesis}$
    by (simp add: algebra-simps)
qed

lemma neg-le-iff-le [simp]: $-b \leq -a \iff a \leq b$
proof
  assume $-b \leq -a$
  then have $(-a) \leq -(-b)$
    by (rule le-imp-neg-le)
  then show $a \leq b$
    by simp
next
  assume $a \leq b$
  then show $-b \leq -a$
    by (rule le-imp-neg-le)
qed

lemma neg-le-0-iff-le [simp]: $-a \leq 0 \iff 0 \leq a$
  by (subst neg-le-iff-le [symmetric]) simp

lemma neg-0-le-iff-le [simp]: $0 \leq -a \iff a \leq 0$
  by (subst neg-le-iff-le [symmetric]) simp

lemma neg-less-iff-less [simp]: $-b < -a \iff a < b$
by (auto simp add: less-le)

lemma neg-less-0-iff-less [simp]: \(- a < 0 \leftrightarrow 0 < a\)
  by (subst neg-less-iff-less [symmetric]) simp

lemma neg-0-less-iff-less [simp]: \(0 < - a \leftrightarrow a < 0\)
  by (subst neg-less-iff-less [symmetric]) simp

The next several equations can make the simplifier loop!

lemma less-minus-iff: \(a < - b \leftrightarrow b < - a\)
proof
  have \(- (\ - a) < - b \leftrightarrow b < - a\)
    by (rule neg-less-iff-less)
  then show \(?thesis\) by simp
qed

lemma minus-less-iff: \(- a < b \leftrightarrow - b < a\)
proof
  have \(- a < - (\ - b) \leftrightarrow - b < a\)
    by (rule neg-less-iff-less)
  then show \(?thesis\) by simp
qed

lemma le-minus-iff: \(a \leq - b \leftrightarrow - b \leq a\)
  by (auto simp: order.order-iff-strict less-minus-iff)

lemma minus-le-iff: \(- a \leq b \leftrightarrow - b \leq a\)
  by (auto simp add: le-less minus-less-iff)

lemma diff-less-0-iff [simp]: \(a - b < 0 \leftrightarrow a < b\)
proof
  have \(a - b < 0 \leftrightarrow a + (\ - b) < b + (\ - b)\)
    by simp
  also have \(\ldots \leftrightarrow a < b\)
    by (simp only: add-less-cancel-right)
  finally show \(?thesis\).
qed

lemmas less-iff-diff-less-0 = diff-less-0-iff-less [symmetric]

lemma diff-less-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
  \(a - b < c \leftrightarrow a < c + b\)
proof (subst less-iff-diff-less-0 [of a])
  show \((a - b < c) = (a - (c + b) < 0)\)
    by (simp add: algebra-simps less-iff-diff-less-0 [of - c])
qed

lemma less-diff-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
  \(a < c - b \leftrightarrow a + b < c\)
proof ( subst less_iff_diff_less_0 [of a + b])
  show \((a < c - b) = (a + b - c < 0)\)
    by (simp add: algebra_simps less_iff_diff_less_0 [of a])
qed

lemma diff_gt_0_iff_gt [simp]: \(a - b > 0 \iff a > b\)
  by (simp add: less_diff_eq)

lemma diff_le_eq [algebra_simps, algebra_split_simps, field_simps, field_split_simps]:
  \(a - b \leq c \iff a \leq c + b\)
  by (auto simp add: le_less)

lemma le_iff_diff_le_0 [symmetric]
  by (simp add: algebra_simps)

lemma diff_ge_0_iff_ge [simp]: \(a - b \geq 0 \iff a \geq b\)
  by (simp add: le_iff_diff_le_0)

lemma diff_eq_diff_less [simp only: less_iff_diff_less_0 [of a b] less_iff_diff_less_0 [of c d]]:
  \(a - b = c - d \implies a < b \iff c < d\)
  by (auto simp add: less_iff_diff_less_0)

lemma diff_eq_diff_less_eq [simp only: le_iff_diff_le_0 [of a b] le_iff_diff_le_0 [of c d]]:
  \(a - b = c - d \implies a \leq b \iff c \leq d\)
  by (auto simp add: le_iff_diff_le_0)

lemma diff_mono: \(a \leq b \implies d \leq c \implies a - c \leq b - d\)
  by (simp add: field_simps)

lemma diff_left_mono: \(b \leq a \implies c - a \leq c - b\)
  by (simp add: field_simps)

lemma diff_right_mono: \(a \leq b \implies a - c \leq b - c\)
  by (simp add: field_simps)

lemma diff_strict_mono: \(a < b \implies d < c \implies a - c < b - d\)
  by (simp add: field_simps)

lemma diff_strict_left_mono: \(b < a \implies c - a < c - b\)
  by (simp add: field_simps)

lemma diff_strict_right_mono: \(a < b \implies a - c < b - c\)
  by (simp add: field_simps)

end
locale group-cancel

begin

lemma add1: (A : 'a::comm-monoid-add) ≡ k + a ⟹ A + b ≡ k + (a + b)
by (simp only: ac-simps)

lemma add2: (B : 'a::comm-monoid-add) ≡ k + b ⟹ a + B ≡ k + (a + b)
by (simp only: ac-simps)

lemma sub1: (A : 'a::ab-group-add) ≡ k + a ⟹ A - b ≡ k + (a - b)
by (simp only: add-diff-eq)

lemma sub2: (B : 'a::ab-group-add) ≡ k + b ⟹ a - B ≡ - k + (a - b)
by (simp only: minus-add-diff-conv-add-uminus ac-simps)

lemma neg1: (A : 'a::ab-group-add) ≡ k + a ⟹ - A ≡ - k + - a
by (simp only: minus-add-distrib)

lemma rule0: (a : 'a::comm-monoid-add) ≡ a + 0
by (simp only: add-0-right)

end

ML-file ‹Tools/group-cancel.ML›

simproc-setup group-cancel-add (a + b : 'a::ab-group-add) =
(\fn phi => fn ss => try Group-Cancel.cancel-add-conv)

simproc-setup group-cancel-diff (a - b : 'a::ab-group-add) =
(\fn phi => fn ss => try Group-Cancel.cancel-diff-conv)

simproc-setup group-cancel-eq (a = (b : 'a::ab-group-add)) =
(\fn phi => fn ss => try Group-Cancel.cancel-eq-conv)

simproc-setup group-cancel-le (a ≤ (b : 'a::ordered-ab-group-add)) =
(\fn phi => fn ss => try Group-Cancel.cancel-le-conv)

simproc-setup group-cancel-less (a < (b : 'a::ordered-ab-group-add)) =
(\fn phi => fn ss => try Group-Cancel.cancel-less-conv)

class linordered-ab-semigroup-add =
linorder + ordered-ab-semigroup-add

class linordered-cancel-ab-semigroup-add =
linorder + ordered-cancel-ab-semigroup-add

begin

subclass linordered-ab-semigroup-add ..
subclass ordered-ab-semigroup-add-imp-le
proof
  fix a b c :: 'a
  assume le1: c + a ≤ c + b
  show a ≤ b
    proof (rule ccontr)
      assume *: ¬ ?thesis
      then have b ≤ a by (simp add: linorder-not-le)
      then have c + b ≤ c + a by (rule add-left-mono)
      then have c + a = c + b
        using le1 by (iprover intro: order.antisym)
      then have a = b
        by simp
      with * show False
        by (simp add: linorder-not-le [symmetric])
    qed
  qed
end

class linordered-ab-group-add = linorder + ordered-ab-group-add
begin

subclass linordered-cancel-ab-semigroup-add ..

lemma equal-neg-zero [simp]: a = − a ↔ a = 0
proof
  assume a = 0
  then show a = − a by simp
next
  assume A: a = − a
  show a = 0
    proof (cases 0 ≤ a)
      case True
      with A have 0 ≤ − a by auto
      with le-minus-iff have a ≤ 0 by simp
      with True show ?thesis by (auto intro: order-trans)
    next
      case False
      then have B: a ≤ 0 by auto
      with A have − a ≤ 0 by auto
      with B show ?thesis by (auto intro: order-trans)
    qed
  qed

lemma neg-equal-zero [simp]: − a = a ↔ a = 0
  by (auto dest: sym)
lemma neg-less-eq-nonneg [simp]: \(-a \leq a \iff 0 \leq a\)
proof
assume *: \(-a \leq a\)
show \(0 \leq a\)
proof (rule classical)
assume \(\neg \thesis\)
then have \(a < 0\) by auto
with * have \(-a < 0\) by (rule le-less-trans)
then show \(\thesis\) by auto
qed
next
assume *: \(0 \leq a\)
then have \(-a \leq 0\) by (simp add: minus-le-iff)
from this * show \(-a \leq a\) by (rule order-trans)
qed

lemma neg-less-pos [simp]: \(-a < a \iff 0 < a\)
by (auto simp add: less-le)

lemma less-eq-neg-nonpos [simp]: \(a \leq -a \iff a \leq 0\)
using neg-less-eq-nonneg [of \(-a\)] by simp

lemma less-neg-neg [simp]: \(a < -a \iff a < 0\)
using neg-less-pos [of \(-a\)] by simp

lemma double-zero [simp]: \(a + a = 0 \iff a = 0\)
proof
assume \(a + a = 0\)
then have \(-a = a\) by (rule minus-unique)
then show \(a = 0\) by (simp only: neg-equal-zero)
next
assume \(a = 0\)
then show \(a + a = 0\) by simp
qed

lemma double-zero-sym [simp]: \(0 = a + a \iff a = 0\)
using double-zero [of \(a\)] by (simp only: eq-commute)

lemma zero-less-double-add-iff-zero-less-single-add [simp]: \(0 < a + a \iff 0 < a\)
proof
assume \(0 < a + a\)
then have \(0 - a < a\) by (simp only: diff-less-eq)
then have \(-a < a\) by simp
then show \(0 < a\) by simp
next
assume \(0 < a\)
with this have \(0 + 0 < a + a\)
by (rule add-strict-mono)
then show \(0 < a + a\) by simp
qed

lemma zero-le-double-add-iff-zero-le-single-add [simp]: 0 ≤ a + a ←→ 0 ≤ a
  by (auto simp add: le_less)

lemma double-add-less-zero-iff-single-add-less-zero [simp]: a + a < 0 ←→ a < 0
proof
  have ¬ a + a < 0 ←→ ¬ a < 0
    by (simp add: not-less)
  then show ?thesis by simp
qed

lemma double-add-le-zero-iff-single-add-le-zero [simp]: a + a ≤ 0 ←→ a ≤ 0
proof
  have ¬ a + a ≤ 0 ←→ ¬ a ≤ 0
    by (simp add: not-le)
  then show ?thesis by simp
qed

lemma minus-max-eq-min: − max x y = min (− x) (− y)
  by (auto simp add: max_def min_def)

lemma minus-min-eq-max: − min x y = max (− x) (− y)
  by (auto simp add: max_def min_def)

end

class abs =
  fixes abs :: 'a ⇒ 'a (|−|)

class sgn =
  fixes sgn :: 'a ⇒ 'a

class ordered-ab-group-add-abs = ordered-ab-group-add + abs +
  assumes abs-ge-zero [simp]: |a| ≥ 0
  and abs-ge-self: a ≤ |a|
  and abs-leI: a ≤ b ⇒ − a ≤ b ⇒ |a| ≤ b
  and abs-minus-cancel [simp]: |− a| = |a|
  and abs-triangle-ineq: |a + b| ≤ |a| + |b|
begin

lemma abs-minus-le-zero: − |a| ≤ 0
  unfolding neg-le-0-iff-le by simp

lemma abs-of-nonneg [simp]:
  assumes nonneg: 0 ≤ a
  shows |a| = a
proof (rule order.antisym)
  show a ≤ |a| by (rule abs-ge-self)

from nonneg le-imp-neg-le have \(-a \leq 0\) by simp
from this nonneg have \(-a \leq a\) by (rule order-trans)
then show \(|a| \leq a\) by (auto intro: abs-leI)
qed

lemma abs-idempotent [simp]: \(||a|| = |a|\)
  by (rule order.antisym) (auto intro: abs-ge-self abs-leI order-trns[of \(-|a|\) 0 [\(|a|\)])

lemma abs-eq-0 [simp]: \(|a| = 0\) \iff \(a = 0\)
proof-
  have \(|a| = 0\) \implies \(a = 0\)
  proof (rule order.antisym)
    assume zero: \(|a| = 0\)
    with abs-ge-self show \(a \leq 0\) by auto
    from zero have \(|-a| = 0\) by simp
    with abs-ge-self \([of \(-a\)]\) have \(-a \leq 0\) by auto
    with neg-le-0-iff-le show \(0 \leq a\) by auto
  qed
  then show \(?thesis\) by auto
qed

lemma abs-zero [simp]: \(|0| = 0\)
  by simp

lemma abs-0-eq [simp]: \(0 = |a|\) \iff \(a = 0\)
proof-
  have \(0 = |a|\) \iff \(|a| = 0\) by (simp only: eq-ac)
  then show \(?thesis\) by simp
qed

lemma abs-le-zero-iff [simp]: \(|a| \leq 0\) \iff \(a = 0\)
proof
  assume \(|a| \leq 0\)
  then have \(|a| = 0\) by (rule order.antisym) simp
  then show \(a = 0\) by simp
next
  assume \(a = 0\)
  then show \(|a| \leq 0\) by simp
qed

lemma abs-le-self-iff [simp]: \(|a| \leq a\) \iff \(\theta \leq a\)
proof-
  have \(\theta \leq |a|\)
    using abs-ge-zero by blast
  then have \(|a| \leq a\) \implies \(\theta \leq a\)
    using order.trans by blast
  then show \(?thesis\)
    using abs-of-nonneg eq-refl by blast
qed

lemma zero-less-abs-iff [simp]: \(0 < |a| \iff a \neq 0\)
  by (simp add: less-le)

lemma abs-not-less-zero [simp]: \(\neg |a| < 0\)
proof
  have \(x \leq y \Longrightarrow \neg y < x\) for \(x\ y\) by auto
  then show ?thesis by simp
qed

lemma abs-ge-minus-self: \(-a \leq |a|\)
proof
  have \(-a \leq |-a|\) by (rule abs-ge-self)
  then show ?thesis by simp
qed

lemma abs-minus-commute: \(|a - b| = |b - a|\)
proof
  have \(|a - b| = |- (a - b)|\)
    by (simp only: abs-minus-cancel)
  also have \(. . . = |b - a|\) by simp
  finally show ?thesis
qed

lemma abs-of-pos: \(0 < a \Longrightarrow |a| = a\)
  by (rule abs-of-nonneg) (rule less-imp-le)

lemma abs-of-nonpos [simp]:
  assumes \(a \leq 0\)
  shows \(|a| = -a\)
proof
  let \(?b = -a\)
  have \(- ?b \leq 0 \Longrightarrow |- ?b| = - (- ?b)\)
    unfolding abs-minus-cancel [of ?b]
    unfolding neg-le-0-iff-le [of ?b]
    unfolding minus-minus by (erule abs-of-nonneg)
  then show ?thesis using assms by auto
qed

lemma abs-of-neg: \(a < 0 \Longrightarrow |a| = -a\)
  by (rule abs-of-nonneg) (rule less-imp-le)

lemma abs-le-D1: \(|a| \leq b \Longrightarrow a \leq b\)
  using abs-ge-self by (blast intro: order-trans)

lemma abs-le-D2: \(|a| \leq b \Longrightarrow -a \leq b\)
  using abs-le-D1 [of \(-a\)] by simp
lemma abs-le-iff: \(|a| \leq b \iff a \leq b \land -a \leq b\)
by (blast intro: abs-leI dest: abs-le-D1 abs-le-D2)

lemma abs-triangle-ineq2: \(|a| - |b| \leq |a - b|\)
proof -
  have \(|a| = |b + (a - b)|\)
  by (simp add: algebra-simps)
  then have \(|a| \leq |b| + |a - b|\)
  by (simp add: abs-triangle-ineq)
  then show \(?thesis\)
  by (simp add: algebra-simps)
qed

lemma abs-triangle-ineq2-sym: \(|a| - |b| \leq |b - a|\)
by (simp only: abs-minus-commute [of b] abs-triangle-ineq2)

lemma abs-triangle-ineq3: \(||a| - |b|| \leq |a - b|\)
by (simp add: abs-le-iff abs-triangle-ineq2 abs-triangle-ineq2-sym)

lemma abs-triangle-ineq4: \(|a - b| \leq |a| + |b|\)
proof -
  have \(|a - b| = |a + (-b)|\)
  by (simp add: algebra-simps)
  also have \(\ldots \leq |a| + |-b|\)
  by (rule abs-triangle-ineq)
  finally show \(?thesis\)
  by simp
qed

lemma abs-diff-triangle-ineq: \(|a + b - (c + d)| \leq |a - c| + |b - d|\)
proof -
  have \(|a + b - (c + d)| = |(a - c) + (b - d)|\)
  by (simp add: algebra-simps)
  also have \(\ldots \leq |a - c| + |b - d|\)
  by (rule abs-triangle-ineq)
  finally show \(?thesis\)
  .
qed

lemma abs-add-abs [simp]: \(||a| + |b|| = |a| + |b|\)
  (is \(?L = ?R\))
proof (rule order_antisym)
  show \(?L \geq ?R\) by (rule abs-ge-self)
  have \(?L \leq ||a|| + ||b||\) by (rule abs-triangle-ineq)
  also have \(\ldots = ?R\) by simp
  finally show \(?L \leq ?R\).
qed

end

lemma dense-engl-I:
fixes $x, a :: \{\text{dense-linorder, ordered-ab-group-add-abs}\}$
assumes $\forall e. \ 0 < e \implies |x| \leq e$
shows $x = 0$
proof (cases $|x| = 0$)
  case False
  then have $|x| > 0$
    by simp
  then obtain $z$ where $0 < z \leq |x|$
    using dense by force
  then show ?thesis
    using assms by (simp flip: not-less)
qed auto

hide-fact (open) ab-diff-conv-add-uminus add-0 mult-1 ab-left-minus
lemmas add-0 = add-0-left
lemmas mult-1 = mult-1-left
lemmas ab-left-minus = left-minus
lemmas diff-diff-eq = diff-diff-add

5.8 Canonically ordered monoids

Canonically ordered monoids are never groups.

class canonically-ordered-monoid-add = comm-monoid-add + order +
assumes le-iff-add: $a \leq b \iff (\exists c. b = a + c)$
begin

lemma zero-le[simp]: $0 \leq x$
  by (auto simp: le-iff-add)

lemma le-zero-eq[simp]: $n \leq 0 \iff n = 0$
  by (auto intro: order.antisym)

lemma not-less-zero[simp]: $\neg n < 0$
  by (auto simp: less-le)

lemma zero-less_iff_neq_zero: $0 < n \iff n \neq 0$
  by (auto simp: less-le)

This theorem is useful with blast

lemma gr-zeroI: $(n = 0 \iff \text{False}) \implies 0 < n$
  by (rule zero-less_iff_neq_zero[THEN iffD2]) iprover

lemma not-gr-zero[simp]: $\neg 0 < n \iff n = 0$
  by (simp add: zero-less_iff_neq_zero)

subclass ordered-comm-monoid-add
proof qed (auto simp: le_iff_add add_ac)
lemma grimplies-not-zero: \( m < n \implies n \neq 0 \)
by auto

lemma add-eq-0-iff-both-eq-0[simp]: \( x + y = 0 \iff x = 0 \land y = 0 \)
by (intro add-nonneg-eq-0-iff zero-le)

lemma zero-eq-add-iff-both-eq-0[simp]: \( 0 = x + y \iff x = 0 \land y = 0 \)
using add-eq-0-iff-both-eq-0[of x y] unfolding eq-commute[of 0].

lemma less-eqE:
assumes \( a \leq b \)
obtains \( c \) where \( b = a + c \)
using assms by (auto simp add: le-iff-add)

lemma lessE:
assumes \( a < b \)
obtains \( c \) where \( b = a + c \) and \( c \neq 0 \)
proof –
from assms have \( a \leq b \) \( a \neq b \)
  by simp-all
from \( a \leq b \) obtain \( c \) where \( b = a + c \)
  by (rule less-eqE)
moreover have \( c \neq 0 \) using \( a \neq b \) \( b = a + c \)
  by auto
ultimately show \( \text{thesis} \)
  by (rule that)
qed

lemmas zero-order = zero-le le-zero-eq not-less-zero zero-less-iff-neq-zero not-gr-zero
— This should be attributed with \([iff]\), but then blast fails in \( Set \).

end

class ordered-cancel-comm-monoid-diff =
canonically-ordered-monoid-add + comm-monoid-diff + ordered-ab-semigroup-add-imp-le
begin

context
fixes a b :: 'a
assumes le: \( a \leq b \)
begin

lemma add-diff-inverse: \( a + (b - a) = b \)
using le by (auto simp add: le-iff-add)

lemma add-diff-assoc: \( c + (b - a) = c + b - a \)
using le by (auto simp add: le-iff-add add.left-commute [of c])

lemma add-diff-assoc2: \( b - a + c = b + c - a \)
using le by (auto simp add: le_iff_add add_assoc)

lemma diff-add-assoc: \( c + b - a = c + (b - a) \)
using le by (simp add: add.commute add-diff-assoc)

lemma diff-addassoc2: \( b + c - a = b - a + c \)
using le by (simp add: add.commute add-diff-assoc)

lemma diff-diff-right: \( c - (b - a) = c + a - b \)
by (simp add: add-diff-inverse add-diff-cancel-left [of a c b - a, symmetric]
add.commute)

lemma diff-add: \( b - a + a = b \)
by (simp add: add.commute add-diff-inverse)

lemma le-add-diff: \( c \leq b + c - a \)
by (auto simp add: add-diff-inverse add.commute)

lemma le-imp-diff-is-add: \( a \leq b \Rightarrow b - a = c \longleftrightarrow b = c + a \)
by (auto simp add: add-diff-inverse add.commute)

lemma le-diff-conv2: \( c \leq b - a \longleftrightarrow c + a \leq b \)
(is \(?P \longleftrightarrow \?Q\))
proof
assume \(?P\)
then have \(c + a \leq b - a + a\)
  by (rule add-right-mono)
then show \(?Q\)
  by (simp add: add-diff-inverse add.commute)
next
assume \(?Q\)
then have \(a + c \leq a + (b - a)\)
  by (simp add: add-diff-inverse add.commute)
then show \(?P\) by simp
qed

end

end

5.9 Tools setup

lemma add-mono-thms-linordered-semiring:
fixes \( i \, j \, k :: 'a::ordered-ab-semigroup-add \)
shows \( i \leq j \land k \leq l \Rightarrow i + k \leq j + l \)
  and \( i = j \land k \leq l \Rightarrow i + k \leq j + l \)
  and \( i \leq j \land k = l \Rightarrow i + k \leq j + l \)
  and \( i = j \land k = l \Rightarrow i + k = j + l \)
by (rule add-mono, clarify+)+
lemma add-mono-thms-linordered-field:
fixes i j k :: 'a::ordered-cancel-ab-semigroup-add
shows i < j ∧ k = l =⇒ i + k < j + l
    and i = j ∧ k < l =⇒ i + k < j + l
    and i < j ∧ k ≤ l =⇒ i + k < j + l
    and i ≤ j ∧ k < l =⇒ i + k < j + l
by (auto intro: add-strict-right-mono add-strict-left-mono
    add-less-le-mono add-le-less-mono add-strict-mono)

code-identifier
code-module Groups → (SML) Arith and (OCaml) Arith and (Haskell) Arith
end

6 Abstract lattices

theory Lattices
imports Groups
begin

6.1 Abstract semilattice

These locales provide a basic structure for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

locale semilattice = abel-semigroup +
  assumes idem [simp]: a * a = a
begin

lemma left-idem [simp]: a * (a * b) = a * b
  by (simp add: assoc [symmetric])

lemma right-idem [simp]: (a * b) * b = a * b
  by (simp add: assoc)
end

locale semilattice-neutr = semilattice + comm-monoid

locale semilattice-order = semilattice +
  fixes less-eq :: 'a ⇒ 'a ⇒ bool (infix ≤ 50)
    and less :: 'a ⇒ 'a ⇒ bool (infix < 50)
  assumes order-iff: a ≤ b =⇒ a = a * b
    and strict-order-iff: a < b =⇒ a = a * b ∧ a ≠ b
begin
lemma orderI: \( a = a * b \implies a \leq b \) 
by (simp add: order-iff)

lemma orderE: 
assumes \( a \leq b \) 
obtains \( a = a * b \) 
using assms by (unfold order-iff)

sublocale ordering less-eq less 
proof 
  show \( a < b \iff a \leq b \land a \neq b \) for a b 
  by (simp add: order-iff strict-order-iff)
next 
  show \( a \leq a \) for a 
  by (simp add: order-iff)
next 
  fix a b 
  assume \( a \leq b \land b \leq a \) 
  then have \( a = a * b \land a * b = b \) 
  by (simp-all add: order-iff commute)
  then show \( a = b \) by simp
next 
  fix a b c 
  assume \( a \leq b \land b \leq c \) 
  then have \( a = a * b \land b * c \) 
  by (simp-all add: order-iff commute)
  then have \( a = (a * b) * c \) 
  by simp
  then have \( a = a * (b * c) \) 
  by (simp add: assoc)
  with \( a = a * b \) [symmetric] have \( a = a * c \) by simp
  then show \( a \leq c \) by (rule orderI)
qed

lemma cobounded1 [simp]: \( a * b \leq a \) 
by (simp add: order-iff commute)

lemma cobounded2 [simp]: \( a * b \leq b \) 
by (simp add: order-iff)

lemma boundedI: 
assumes \( a \leq b \) and \( a \leq c \) 
shows \( a \leq b * c \) 
proof (rule orderI)
  from assms obtain \( a * b = a \) and \( a * c = a \) 
  by (auto elim!: orderE)
  then show \( a = a * (b * c) \) 
  by (simp add: assoc [symmetric])
qed
lemma boundedE:
  assumes \( a \leq b \cdot c \)
  obtains \( a \leq b \) and \( a \leq c \)
  using assms by (blast intro: trans coboundedI1 coboundedI2)

lemma bounded-iff [simp]: \( a \leq b \cdot c \iff a \leq b \land a \leq c \)
  by (blast intro: boundedI elim: boundedE)

lemma strict-boundedE:
  assumes \( a < b \cdot c \)
  obtains \( a < b \) and \( a < c \)
  using assms by (auto simp add: commute strict-iff-order elim: orderE intro: !:)

that

lemma coboundedI1: \( a \leq c \Rightarrow a \cdot b \leq c \)
  by (rule trans) auto

lemma coboundedI2: \( b \leq c \Rightarrow a \cdot b \leq c \)
  by (rule trans) auto

lemma strict-coboundedI1: \( a < c \Rightarrow a \cdot b < c \)
  using irrefl by (auto intro: not-eq-order-implies-strict coboundedI1 strict-implies-order elim: strict-boundedE)

lemma strict-coboundedI2: \( b < c \Rightarrow a \cdot b < c \)
  using strict-coboundedI1 [of b c a] by (simp add: commute)

lemma mono: \( a \leq c \Rightarrow b \leq d \Rightarrow a \cdot b \leq c \cdot d \)
  by (blast intro: boundedI coboundedI1 coboundedI2)

lemma absorb1: \( a \leq b \Rightarrow a \cdot b = a \)
  by (rule antisym) (auto simp: refl)

lemma absorb2: \( b \leq a \Rightarrow a \cdot b = b \)
  by (rule antisym) (auto simp: refl)

lemma absorb3: \( a < b \Rightarrow a \cdot b = a \)
  by (rule absorb1) (rule strict-implies-order)

lemma absorb4: \( b < a \Rightarrow a \cdot b = b \)
  by (rule absorb2) (rule strict-implies-order)

lemma absorb-iff1: \( a \leq b \iff a \cdot b = a \)
  using order-iff by auto

lemma absorb-iff2: \( b \leq a \iff a \cdot b = b \)
  using order-iff by (auto simp add: commute)
locale semilattice-neutr-order = semilattice-neutr + semilattice-order

begin

sublocale ordering-top less-eq less
  by standard (simp add: order-iff)

lemma eq-neutr-iff [simp]: \( a \ast b = 1 \iff a = 1 \land b = 1 \)
  by (simp add: eq-iff)

lemma neutr-eq-iff [simp]: \( 1 = a \ast b \iff a = 1 \land b = 1 \)
  by (simp add: eq-iff)

end

Interpretations for boolean operators

interpretation conj: semilattice-neutr \((\land)\) True
  by standard auto

interpretation disj: semilattice-neutr \((\lor)\) False
  by standard auto

declare conj-assoc [ac-simps del] disj-assoc [ac-simps del] — already simp by default

6.2 Syntactic infimum and supremum operations

class inf =
  fixes inf :: 'a \Rightarrow 'a \Rightarrow 'a (infixl \( \cap \))

class sup =
  fixes sup :: 'a \Rightarrow 'a \Rightarrow 'a (infixl \( \cup \))

6.3 Concrete lattices

class semilattice-inf = order + inf +
  assumes inf-le1 [simp]: \( x \cap y \leq x \)
  and inf-le2 [simp]: \( x \cap y \leq y \)
  and inf-greatest: \( x \leq y \implies x \leq z \implies x \leq y \cap z \)

class semilattice-sup = order + sup +
  assumes sup-ge1 [simp]: \( x \leq x \cup y \)
  and sup-ge2 [simp]: \( y \leq x \cup y \)
  and sup-least: \( y \leq x \implies z \leq x \implies y \cup z \leq x \)
begin

Dual lattice.
lemma dual-semilattice: class.semilattice-inf sup greater-eq greater
  by (rule class.semilattice-inf.intro, rule dual-order)
  (unfold-locales, simp-all add: sup-least)
end

class lattice = semilattice-inf + semilattice-sup

6.3.1 Intro and elim rules

context semilattice-inf
begin

lemma le-infI1: a ≤ x ⇒ a ∩ b ≤ x
  by (rule order-trans) auto

lemma le-infI2: b ≤ x ⇒ a ∩ b ≤ x
  by (rule order-trans) auto

lemma le-infI: x ≤ a ⇒ x ≤ b ⇒ x ≤ a ∩ b
  by (fact inf-greatest)

lemma le-infE: x ≤ a ∩ b ⇒ (x ≤ a ⇒ x ≤ b ⇒ P) ⇒ P
  by (blast intro: order-trans inf-le1 inf-le2)

lemma le-inf-iff: x ≤ y ⊓ z ⇔ x ≤ y ∧ x ≤ z
  by (blast intro: le-infI elim: le-infE)

lemma le-iff-inf: x ≤ y ⇔ x ∩ y = x
  by (auto intro: le-infI1 order.antisym dest: order.eq-iff [THEN iffD1] simp add: le-inf-iff)

lemma inf-mono: a ≤ c ⇒ b ≤ d ⇒ a ∩ b ≤ c ∩ d
  by (fast intro: inf-greatest le-infI1 le-infI2)
end

context semilattice-sup
begin

lemma le-supI1: x ≤ a ⇒ x ≤ a ⊔ b
  by (rule order-trans) auto

lemma le-supI2: x ≤ b ⇒ x ≤ a ⊔ b
  by (rule order-trans) auto

lemma le-supI: a ≤ x ⇒ b ≤ x ⇒ a ⊔ b ≤ x
  by (fact sup-least)
end
THEORY “Lattices”

lemma le-supE: \( a \sqcup b \leq x \implies (a \leq x \implies b \leq x \implies P) \implies P \)
  by (blast intro: order-trans sup-ge1 sup-ge2)

lemma le-sup-iff: \( x \sqcup y \leq z \iff x \leq z \land y \leq z \)
  by (blast intro: le-supI elim: le-supE)

lemma le-if-sup: \( x \sqsubseteq y \iff x \leq z \land y \leq z \)
  by (auto intro: le-supI2 order.antisym dest: order.eq-iff THEN iffD1 simp add: le-sup-iff)

lemma sup-mono: \( a \leq c \implies b \leq d \implies a \sqcup b \leq c \sqcup d \)
  by (fast intro: sup-least le-supI1 le-supI2)

end

6.3.2 Equational laws

context semilattice-inf
begin

sublocale inf: semilattice inf
proof
  fix a b c
  show \( (a \sqcap b) \sqcap c = a \sqcap (b \sqcap c) \)
    by (rule order.antisym (auto intro: le-infI1 le-infI2 simp add: le-inf-iff))
  show \( a \sqcap b = b \sqcap a \)
    by (rule order.antisym (auto simp add: le-inf-iff))
  show \( a \sqcap a = a \)
    by (rule order.antisym (auto simp add: le-inf-iff))
qed

sublocale inf: semilattice-order inf less-eq less
by standard (auto simp add: le-inf-inf less-eq less-le)

lemma inf-assoc: \( (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z) \)
  by (fact inf.assoc)

lemma inf-commute: \( x \sqcap y = (y \sqcap x) \)
  by (fact inf.commute)

lemma inf-left-commute: \( x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z) \)
  by (fact inf.left-commute)

lemma inf-idem: \( x \sqcap x = x \)
  by (fact inf.idem)

lemma inf-left-idem: \( x \sqcap (x \sqcap y) = x \sqcap y \)
  by (fact inf.left-idem)
lemma inf-right-idem: $(x \cap y) \cap y = x \cap y$
  by (fact inf.right-idem)

lemma inf-absorb1: $x \leq y \Rightarrow x \cap y = x$
  by (rule order.antisym) auto

lemma inf-absorb2: $y \leq x \Rightarrow x \cap y = y$
  by (rule order.antisym) auto

lemmas inf-aci = inf-commute inf-assoc inf-left-commute inf-left-idem

end

context semilattice-sup
begin

sublocale sup: semilattice sup
proof
  fix a b c
  show $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
    by (rule order.antisym) (auto intro: le-supI1 le-supI2 simp add: le-sup-iff)
  show $a \sqcup b = b \sqcup a$
    by (rule order.antisym) (auto simp add: le-sup-iff)
  show $a \sqcup a = a$
    by (rule order.antisym) (auto simp add: le-sup-iff)
qed

sublocale sup: semilattice-order sup greater-eq greater
  by standard (auto simp add: le-iff-sup sup.commute less-le)

lemma sup-assoc: $(x \sqsubseteq y) \sqcup z = x \sqsubseteq (y \sqsubseteq z)$
  by (fact sup.assoc)

lemma sup-commute: $(x \sqsubseteq y) = (y \sqsubseteq x)$
  by (fact sup.commute)

lemma sup-left-commute: $x \sqsubseteq (y \sqsubseteq z) = y \sqsubseteq (x \sqsubseteq z)$
  by (fact sup.left-commute)

lemma sup-idem: $x \sqsubseteq x = x$
  by (fact sup.idem)

lemma sup-left-idem [simp]: $x \sqsubseteq (x \sqsubseteq y) = x \sqsubseteq y$
  by (fact sup.left-idem)

lemma sup-absorb1: $y \leq x \Rightarrow x \sqcup y = x$
  by (rule order.antisym) auto

lemma sup-absorb2: $x \leq y \Rightarrow x \sqcup y = y$
by (rule order.antisym) auto

lemmas sup-aci = sup-commute sup-assoc sup-left-commute sup-left-idem

context lattice

begin

lemma dual-lattice: class.lattice sup (≥) (> ) inf
  by (rule class.lattice.intro,
      rule dual-semilattice,
      rule class.semilattice-sup.intro,
      rule dual-order)
  (unfold-locales, auto)

lemma inf-sup-absorb [simp]: x ⊓ (x ⊔ y) = x
  by (blast intro: order.antisym inf-le1 inf-greatest sup-ge1)

lemma sup-inf-absorb [simp]: x ⊔ (x ⊓ y) = x
  by (blast intro: order.antisym sup-ge1 sup-least inf-le1)

lemmas inf-sup-aci = inf-aci sup-aci

lemmas inf-sup-ord = inf-le1 inf-le2 sup-ge1 sup-ge2

Towards distributivity.

lemma distrib-sup-le: x ⊔ (y ⊓ z) ≤ (x ⊔ y) ⊓ (x ⊔ z)
  by (auto intro: le-infI1 le-infI2 le-supI1 le-supI2)

lemma distrib-inf-le: (x ⊓ y) ⊔ (x ⊓ z) ≤ x ⊓ (y ⊓ z)
  by (auto intro: le-infI1 le-infI2 le-supI1 le-supI2)

If you have one of them, you have them all.

lemma distrib-imp1:
  assumes distrib: (∀ x y z. x ⊓ (y ⊓ z) = (x ⊓ y) ⊓ (x ⊓ z))
  shows x ⊔ (y ⊓ z) = (x ⊔ y) ⊓ (x ⊓ z)
  proof -
    have x ⊓ (y ⊓ z) = (x ⊓ (x ⊓ z)) ⊓ (y ⊓ z)
      by simp
    also have ... = x ⊓ (z ⊓ (x ⊔ y))
      by (simp add: distrib inf-commute sup-assoc del: sup-inf-absorb)
    also have ... = ((x ⊔ y) ⊓ x) ⊓ ((x ⊔ y) ⊓ z)
      by (simp add: inf-commute)
    also have ... = (x ⊔ y) ⊓ (x ⊔ z) by (simp add: distrib)
    finally show ?thesis .
  qed

lemma distrib-imp2:
assumes distrib: $\bigwedge x \, y \, z \; x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
shows $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$

proof –
  have $x \sqcap (y \sqcup z) = (x \sqcap (x \sqcup z)) \sqcap (y \sqcup z)$
    by simp
  also have $\ldots = x \sqcap (z \sqcup (x \sqcup y))$
    by (simp add: distrib sup-commute inf-assoc del: inf-sup-absorb)
  also have $\ldots = ((x \sqcap y) \sqcup x) \sqcap ((x \sqcap y) \sqcup z)$
    by (simp add: sup-commute)
  also have $\ldots = (x \sqcap y) \sqcup (x \sqcap z)$ by (simp add:distrib)
  finally show $?thesis$.
qed

6.3.3 Strict order

context semilattice-inf
begin

lemma less-infI1: $a < x \Longrightarrow a \sqcap b < x$
  by (auto simp add: less-le inf-absorb1 intro: le-infI1)

lemma less-infI2: $b < x \Longrightarrow a \sqcap b < x$
  by (auto simp add: less-le inf-absorb2 intro: le-infI2)

end

context semilattice-sup
begin

lemma less-supI1: $x < a \Longrightarrow x < a \sqcup b$
  using dual-semilattice
  by (rule semilattice-inf.less-infI1)

lemma less-supI2: $x < b \Longrightarrow x < a \sqcup b$
  using dual-semilattice
  by (rule semilattice-inf.less-infI2)

end

6.4 Distributive lattices

class distrib-lattice = lattice +
  assumes sup-inf-distrib1: $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

context distrib-lattice
begin

lemma sup-inf-distrib2: $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
by (simp add: sup-commute sup-inf-distrib1)

lemma inf-sup-distrib1: \( x \cap (y \cup z) = (x \cap y) \cup (x \cap z) \)
  by (rule distrib-imp2 [OF sup-inf-distrib1])

lemma inf-sup-distrib2: \( (y \cup z) \cap x = (y \cap x) \cup (z \cap x) \)
  by (simp add: inf-commute inf-sup-distrib1)

lemma dual-distrib-lattice: class.distrib-lattice sup (\( \geq \)) (\( > \)) inf
  by (rule class.distrib-lattice.intro, rule dual-lattice)
  (unfold-locales, fact inf-sup-distrib1)

lemmas sup-inf-distrib = sup-inf-distrib1 sup-inf-distrib2
lemmas inf-sup-distrib = inf-sup-distrib1 inf-sup-distrib2
lemmas distrib = sup-inf-distrib1 sup-inf-distrib2 inf-sup-distrib1 inf-sup-distrib2

end

6.5 Bounded lattices

class bounded-semilattice-inf-top = semilattice-inf + order-top
begin

sublocale inf-top: semilattice-neutr inf top
  + inf-top: semilattice-neutr-order inf top less-eq less
proof
  show \( x \cap \top = x \) for \( x \)
    by (rule inf-absorb1) simp
qed

lemma inf-top-left: \( \top \cap x = x \)
  by (fact inf-top.left-neutral)

lemma inf-top-right: \( x \cap \top = x \)
  by (fact inf-top.right-neutral)

lemma inf-eq-top-iff: \( x \cap y = \top \,\iff\, x = \top \land y = \top \)
  by (fact inf-top.eq-neutr-iff)

lemma top-eq-inf-iff: \( \top = x \cap y \,\iff\, x = \top \land y = \top \)
  by (fact inf-top.neutr-eq-iff)

end

class bounded-semilattice-sup-bot = semilattice-sup + order-bot
begin
sublocale sup-bot: semilattice-neutr sup bot
  + sup-bot: semilattice-neutr-order sup bot greater-eq greater
proof
  show \( x \sqcup \perp = x \) for \( x \)
    by (rule sup-absorb1) simp
qed

lemma sup-bot-left: \( \perp \sqcup x = x \)
  by (fact sup-bot.left-neutral)

lemma sup-bot-right: \( x \sqcup \perp = x \)
  by (fact sup-bot.right-neutral)

lemma sup-eq-bot-iff: \( x \sqcup y = \perp \longleftrightarrow x = \perp \land y = \perp \)
  by (fact sup-bot.eq-neutr-iff)

lemma bot-eq-sup-iff: \( \perp = x \sqcup y \longleftrightarrow x = \perp \land y = \perp \)
  by (fact sup-bot.neutr-eq-iff)
end

class bounded-lattice-bot = lattice + order-bot
begin

subclass bounded-semilattice-sup-bot ..

lemma inf-bot-left [simp]: \( \perp \sqcap x = \perp \)
  by (rule inf-absorb1) simp

lemma inf-bot-right [simp]: \( x \sqcap \perp = \perp \)
  by (rule inf-absorb2) simp
end

class bounded-lattice-top = lattice + order-top
begin

subclass bounded-semilattice-inf-top ..

lemma sup-top-left [simp]: \( \top \sqcup x = \top \)
  by (rule sup-absorb1) simp

lemma sup-top-right [simp]: \( x \sqcup \top = \top \)
  by (rule sup-absorb2) simp
end

class bounded-lattice = lattice + order-bot + order-top
begin
subclass bounded-lattice-bot ..
subclass bounded-lattice-top ..

lemma dual-bounded-lattice: class.bounded-lattice sup greater-eq greater inf ⊤ ⊥ 
  by unfold-locales (auto simp add: less-le-not-le)
end

6.6 min/max as special case of lattice

context linorder
begin

sublocale min: semilattice-order min less-eq less 
  + max: semilattice-order max greater-eq greater 
  by standard (auto simp add: min-def max-def)

declare min.absorb1 [simp] min.absorb2 [simp]
min.absorb3 [simp] min.absorb4 [simp]
max.absorb1 [simp] max.absorb2 [simp]
max.absorb3 [simp] max.absorb4 [simp]

lemma min-le-iff-disj: min x y ≤ z ←→ x ≤ z ∨ y ≤ z 
  unfolding min-def using linear by (auto intro: order-trans)

lemma le-max-iff-disj: z ≤ max x y ←→ z ≤ x ∨ z ≤ y 
  unfolding max-def using linear by (auto intro: order-trans)

lemma min-less-iff-disj: min x y < z ←→ x < z ∨ y < z 
  unfolding min-def le-less using less-linear by (auto intro: less-trans)

lemma less-max-iff-disj: z < max x y ←→ z < x ∨ z < y 
  unfolding max-def le-less using less-linear by (auto intro: less-trans)

lemma min-less-iff-conj [simp]: z < min x y ←→ z < x ∧ z < y 
  unfolding min-def le-less using less-linear by (auto intro: less-trans)

lemma max-less-iff-conj [simp]: max x y < z ←→ x < z ∧ y < z 
  unfolding max-def le-less using less-linear by (auto intro: less-trans)

lemma min-max-distrib1: min (max b c) a = max (min b a) (min c a) 
  by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym)

lemma min-max-distrib2: min a (max b c) = max (min a b) (min a c) 
  by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym)
lemma max-min-distrib1: \( \max (\min b c) a = \min (\max b a) (\max c a) \)
  by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym)

lemma max-min-distrib2: \( \max a (\min b c) = \min (\max a b) (\max a c) \)
  by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym)

lemmas min-max-distribs = min-max-distrib1 min-max-distrib2 max-min-distrib1
max-min-distrib2

lemma split-min [no-atp]: \( P (\min i j) \leftrightarrow (i \leq j \rightarrow P i) \land (\neg i \leq j \rightarrow P j) \)
by (simp add: min-def)

lemma split-max [no-atp]: \( P (\max i j) \leftrightarrow (i \leq j \rightarrow P j) \land (\neg i \leq j \rightarrow P i) \)
by (simp add: max-def)

lemma split-min-lin [no-atp]:
  \( \langle P (\min a b) \leftrightarrow (b = a \rightarrow P a) \land (a < b \rightarrow P b) \rangle \)
by (cases a b rule: linorder-cases) auto

lemma split-max-lin [no-atp]:
  \( \langle P (\max a b) \leftrightarrow (b = a \rightarrow P a) \land (a < b \rightarrow P b) \rangle \)
by (cases a b rule: linorder-cases) auto

end

lemma inf-min: \( \inf = (\min :: 'a::{semilattice-inf,linorder} \Rightarrow 'a \Rightarrow 'a) \)
by (auto intro: antisym simp add: min-def fun-eq-iff)

lemma sup-max: \( \sup = (\max :: 'a::{semilattice-sup,linorder} \Rightarrow 'a \Rightarrow 'a) \)
by (auto intro: antisym simp add: max-def fun-eq-iff)

6.7 Uniqueness of inf and sup

lemma (in semilattice-inf) inf-unique:
  fixes f (infixl \( \triangle \) 70)
  assumes le1: \( \forall x y. x \triangle y \leq x \)
  and le2: \( \forall x y. x \triangle y \leq y \)
  and greatest: \( \forall x z. x \leq y \Rightarrow x \leq z \Rightarrow x \leq y \triangle z \)
  shows \( x \cap y = x \triangle y \)
proof (rule order.antisym)
  show \( x \triangle y \leq x \cap y \)
    by (rule le-infI) (rule le1, rule le2)
  have le1: \( \forall x y z. x \leq y \Rightarrow x \leq z \Rightarrow x \leq y \triangle z \)
    by (blast intro: greatest)
  show \( x \cap y \leq x \triangle y \)
    by (rule le1) simp-all
qed
lemma (in semilattice-sup) sup-unique:
fixes f (infixl \( \triangledown \))
assumes ge1 [simp]: \( \forall x y. x \leq x \triangledown y \)
and ge2: \( \forall x y. y \leq x \triangledown y \)
and least: \( \forall x y z. y \leq x \Longrightarrow z \leq x \Longrightarrow y \triangledown z \leq x \)
shows \( x \sqcup y = x \triangledown y \)
proof (rule order.antisym)
show \( x \sqcup y \leq x \triangledown y \)
by (rule le-supI, rule ge1, rule ge2)
have leI: \( \forall x y z. x \leq z \Longrightarrow y \leq z \Longrightarrow x \triangledown y \leq z \)
by (blast intro: least)
show \( x \triangledown y \leq x \sqcup y \)
by (rule leI) simp-all
qed

6.8 Lattice on - \( \Rightarrow \) -

instantiation fun :: (type, semilattice-sup) semilattice-sup
begin

definition \( f \sqcup g = (\lambda x. f x \sqcup g x) \)

lemma sup-apply [simp, code]: \( (f \sqcup g) x = f x \sqcup g x \)
by (simp add: sup-fun-def)

instance
by standard (simp-all add: le-fun-def)
end

instantiation fun :: (type, semilattice-inf) semilattice-inf
begin

definition \( f \sqcap g = (\lambda x. f x \sqcap g x) \)

lemma inf-apply [simp, code]: \( (f \sqcap g) x = f x \sqcap g x \)
by (simp add: inf-fun-def)

instance by standard (simp-all add: le-fun-def)
end

instance fun :: (type, lattice) lattice ..

instance fun :: (type, distrib-lattice) distrib-lattice
by standard (rule ext, simp add: sup-inf-distrib1)

instance fun :: (type, bounded-lattice) bounded-lattice ..
instantiation fun :: (type, uminus) uminus begin

definition fun-Compl-def: − A = (λx. − A x)

lemma uminus-apply [simp, code]: (− A) x = − (A x)
  by (simp add: fun-Compl-def)

instance ..
end

instantiation fun :: (type, minus) minus begin

definition fun-diff-def: A − B = (λx. A x − B x)

lemma minus-apply [simp, code]: (A − B) x = A x − B x
  by (simp add: fun-diff-def)

instance ..
end

end

7 Boolean Algebras

theory Boolean-Algebras
  imports Lattices
begin

7.1 Abstract boolean algebra

locale abstract-boolean-algebra = conj: abel-semigroup ⟨(⊓)⟩ + disj: abel-semigroup ⟨(⊔)⟩

for conj :: ‚a ⇒ ‚a ⇒ ‚a (infixr ⟨⟨⟩⟩ 70)
  and disj :: ‚a ⇒ ‚a ⇒ ‚a (infixr ⟨⊔⟩ 65) +

fixes compl :: ‚a ⇒ ‚a (% → [81] 80)
  and zero :: ‚a ⟨0⟩
  and one :: ‚a ⟨1⟩

assumes conj-disj-distrib: ⟨x ⊓ (y⊔z)⟩ = (⟨x ⊓ y⟩ ⊔ ⟨x ⊓ z⟩)
  and disj-conj-distrib: ⟨x ⊔ (y ⊓ z)⟩ = (⟨x ⊔ y⟩ ⊓ ⟨x ⊔ z⟩)
  and conj-one-right: ⟨x ⊓ 1 = x⟩
  and disj-zero-right: ⟨x ⊔ 0 = x⟩
  and conj-cancel-right [simp]: ⟨x ⊓ x = 0⟩
  and disj-cancel-right [simp]: ⟨x ⊔ x = 1⟩

begin
sublocale conj: semilattice-neutr $(\cap) \cdot (1)$
proof
  show $x \cap 1 = x$ for $x$
    by (fact conj-one-right)
  show $x \cap x = x$ for $x$
proof
    have $x \cap x = (x \cap x) \cup 0$
      by (simp add: disj-zero-right)
    also have $\ldots = (x \cap x) \cup (x \cap -x)$
      by simp
    also have $\ldots = x \cap (x \cap -x)$
      by (simp only: conj-disj-distrib)
    also have $\ldots = x \cap 1$
      by simp
    also have $\ldots = x$
      by (simp add: conj-one-right)
    finally show $\land thesis$.
  qed
qed

sublocale disj: semilattice-neutr $(\cup) \cdot (0)$
proof
  show $x \cup 0 = x$ for $x$
    by (fact disj-zero-right)
  show $x \cup x = x$ for $x$
proof
    have $x \cup x = (x \cup x) \cap 1$
      by simp
    also have $\ldots = (x \cup x) \cap (x \cap -x)$
      by simp
    also have $\ldots = x \cup (x \cap -x)$
      by (simp only: disj-conj-distrib)
    also have $\ldots = x \cup 0$
      by simp
    also have $\ldots = x$
      by (simp add: disj-zero-right)
    finally show $\land thesis$.
  qed
qed

7.1.1 Complement

lemma complement-unique:
  assumes 1: $a \cap x = 0$
  assumes 2: $a \cup x = 1$
  assumes 3: $a \cap y = 0$
  assumes 4: $a \cup y = 1$
  shows $x = y$
proof

from 1 3 have \((a \sqcap x) \sqcup (x \sqcap y) = (a \sqcap y) \sqcup (x \sqcap y)\)
  by simp

then have \((x \sqcap a) \sqcup (x \sqcap y) = (y \sqcap a) \sqcup (y \sqcap x)\)
  by (simp add: ac-simps)

then have \(x \sqcap (a \sqcup y) = y \sqcap (a \sqcup x)\)
  by (simp add: conj-disj-distrib)

with 2 4 have \(x \sqcap 1 = y \sqcap 1\)
  by simp

then show \(x = y\)
  by simp

qed

lemma compl-unique: \(x \sqcap y = 0 \implies x \sqcup y = 1 \implies -x = y\)
by (rule complement-unique [OF conj-cancel-right disj-cancel-right])

lemma double-compl [simp]: \(-(-x) = x\)
proof (rule compl-unique)
  show \(-x \sqcap x = 0\)
    by (simp only: conj-cancel-right conj.commute)

  show \(-x \sqcup x = 1\)
    by (simp only: disj-cancel-right disj.commute)

qed

lemma compl-eq-compl-iff [simp]:
\(\langle -x = -y \iff x = y \rangle \quad \text{(is } \langle P \iff Q \rangle)\)

proof
  assume \(Q\)
  then show \(P\) by simp

next
  assume \(P\)
  then have \((-(-x)) = -(-y)\)
    by simp

  then show \(Q\)
    by simp

qed

7.1.2 Conjunction

lemma conj-zero-right [simp]: \(x \sqcap 0 = 0\)
using conj.left-idem conj-cancel-right by fastforce

lemma compl-one [simp]: \(-1 = 0\)
by (rule compl-unique [OF conj-zero-right disj-zero-right])

lemma conj-zero-left [simp]: \(0 \sqcap x = 0\)
by (subst conj.commute) (rule conj-zero-right)

lemma conj-cancel-left [simp]: \(-x \sqcap x = 0\)
by (subst conj.commute) (rule conj-cancel-right)

lemma conj-disj-distrib2: (y ⊔ z) ∩ x = (y ∩ x) ∪ (z ∩ x)
  by (simp only: conj.commute conj-disj-distrib)

lemmas conj-disj-distribs = conj-disj-distrib conj-disj-distrib2

### 7.1.3 Disjunction

context
begin

interpretation dual: abstract-boolean-algebra ‹⊔› ‹⊓›
compl ‹1› ‹0›

apply standard
  apply (rule disj-conj-distrib)
  apply (rule conj-disj-distrib)
  apply simp-all
done

lemma disj-one-right [simp]: x ⊔ 1 = 1
  by (fact dual.conj-zero-right)

lemma compl-zero [simp]: −0 = 1
  by (fact dual.compl-one)

lemma disj-one-left [simp]: 1 ⊔ x = 1
  by (fact dual.conj-zero-left)

lemma disj-cancel-left [simp]: −x ⊔ x = 1
  by (fact dual.conj-cancel-left)

lemma disj-conj-distrib2: (y ∩ z) ⊔ x = (y ⊔ x) ∩ (z ⊔ x)
  by (fact dual.conj-disj-distrib2)

lemmas disj-conj-distribs = disj-conj-distrib disj-conj-distrib2

end

### 7.1.4 De Morgan’s Laws

lemma de-Morgan-conj [simp]: −(x ∩ y) = −x ⊔ −y
proof (rule compl-unique)
  have (x ∩ y) ∩ (−x ⊔ −y) = ((x ∩ y) ∩ −x) ∪ ((x ∩ y) ∩ −y)
    by (rule conj-disj-distrib)
  also have ... = (y ∩ (x ∩ −x)) ∪ (x ∩ (y ∩ −y))
    by (simp only: ac-simps)
  finally show (x ∩ y) ∩ (−x ⊔ −y) = 0
    by (simp only: conj-cancel-right conj-zero-right disj-zero-right)
next
  have (x ∩ y) ∪ (−x ⊔ −y) = (x ∪ (−x ⊔ −y)) ∩ (y ∪ (−x ⊔ −y))
by (rule disj-conj-distrib2)
also have . . . = (− y ⊔ (x ⊔ − x)) ∩ (− x ∪ (y ∪ − y))
by (simp only: ac-simps)
finally show (x ∩ y) ⊔ (− x ∪ y) = 1
by (simp only: disj-cancel-right disj-one-right conj-one-right)
qed

context
begin
interpretation
dual:
abstract-boolean-algebra ‹(⊔)› ‹(⊓)›
compl〈1› ‹0›
apply standard
apply (rule disj-conj-distrib)
apply (rule conj-disj-distrib)
apply simp-all
done

lemma de-Morgan-disj [simp]: − (x ⊔ y) = − x ⊓ − y
by (fact dual.de-Morgan-conj)
end

7.2 Symmetric Difference

locale abstract-boolean-algebra-sym-diff =
fixes xor :: ‹(⊖)› (infixr ‹⊖› 65)
assumes xor-def: ‹x ⊖ y = (x ⊓ − y) ⊔ (− x ⊓ y)›
begin
sublocale xor: comm-monoid xor ‹0›
proof
fix x y z :: ′a
let ?t = (x ∩ y ∩ z) ∪ (x ∩ − y ∩ z) ∪ (− x ∩ y ∩ − z) ∪ (− x ∩ − y ∩ z)
have ?t ∪ (z ∩ x ∩ − x) ∪ (z ∩ y ∩ − y) = ?t ∪ (x ∩ y ∩ − y) ∪ (x ∩ z ∩ − z)
by (simp only: conj-cancel-right conj-zero-right)
then show (x ⊖ y) ⊖ z = x ⊖ (y ⊖ z)
by (simp only: xor-def de-Morgan-disj de-Morgan-conj double-compl
(simp only: conj-disj-distrib conj-ac ac-simps)
show x ⊖ y = y ⊖ x
by (simp only: xor-def ac-simps)
show x ⊖ 0 = x
by (simp add: xor-def)
qed

lemma xor-def2:
THEORY “Boolean-Algebras”

\[ \langle x \ominus y = (x \cup y) \cap (-x \cup -y) \rangle \]

proof –

note xor-def [of x y]
also have (x \cap -y \cup -x \cap y = ((x \cup -x) \cap (-y \cup -x)) \cap (x \cap y) \cap (-y \cup y))
  by (simp add: ac-simps disj-conj-distrib)
also have ‘\ldots = (x \cup y) \cap (-x \cup -y)’
  by (simp add: ac-simps)
finally show \thesis. 
qed

lemma xor-one-right [simp]: \( x \ominus 1 = -x \)
  by (simp only: xor-def compl-one conj-zero-right conj-one-right disj-left-neutral)

lemma xor-one-left [simp]: \( 1 \ominus x = -x \)
  using xor-one-right [of x] by (simp add: ac-simps)

lemma xor-self [simp]: \( x \ominus (x \ominus y) = y \)
  by (simp only: xor-assoc [symmetric] xor-self xor.left-neutral)

lemma xor-compl-left [simp]: \( -x \ominus y = -(x \ominus y) \)
  by (simp add: ac-simps flip: xor-one-left)

lemma xor-compl-right [simp]: \( x \ominus y = -(x \ominus y) \)
  using xor.commute xor-compl-left by auto

lemma xor-cancel-right [simp]: \( x \ominus -x = 1 \)
  by (simp only: xor-cancel-right xor-cancel-left xor-self compl-zero)

lemma xor-cancel-left [simp]: \( -x \ominus x = 1 \)
  by (simp only: xor-cancel-left xor-self compl-zero)

lemma conj-xor-distrib: \( x \cap (y \ominus z) = (x \cap y) \ominus (x \cap z) \)
proof –

have \( \ast: (x \cap y \cap z) \cup (x \cap -y \cap -z) = (x \cap -y \cap -z) \cup (x \cap y \cap -z) \cup (x \cap y \cap z) \)
  by (simp only: conj-cancel-right conj-zero-right disj-left-neutral)
then show \( x \cap (y \ominus z) = (x \cap y) \ominus (x \cap z) \)
  by (simp (no_asm-use) only: xor-def de-Morgan-disj de-Morgan-conj double-compl
  conj-disj-distrib ac-simps)
qed

lemma conj-xor-distrib2: \( (y \ominus z) \cap x = (y \cap x) \ominus (z \cap x) \)
  by (simp add: conj.commute conj-xor-distrib)
lemmas conj-xor-distrib = conj-xor-distrib conj-xor-distrib2

end

7.3 Type classes

class boolean-algebra = distrib-lattice + bounded-lattice + minus + uminus +
assumes inf-compl-bot: (x ∩ − x = ⊥)
and sup-compl-top: (x ∪ − x = ⊤)
assumes diff-eq: (x − y = x ∩ − y)

begin

sublocale boolean-algebra: abstract-boolean-algebra (inf) (sup) uminus ⊥ ⊤
apply standard
apply (rule inf-sup-distrib1)
apply (rule sup-inf-distrib1)
apply (simp-all add: ac-simps inf-compl-bot sup-compl-top)
done

lemma compl-inf-bot: − x ∩ x = ⊥
by (fact boolean-algebra.conj-cancel-left)

lemma compl-sup-top: − x ∪ x = ⊤
by (fact boolean-algebra.disj-cancel-left)

lemma compl-unique:
assumes x ∩ y = ⊥
and x ∪ y = ⊤
shows − x = y
using assms by (rule boolean-algebra.compl-unique)

lemma double-compl: − (− x) = x
by (fact boolean-algebra.double-compl)

lemma compl-eq-compl-iff: − x = − y ↔ x = y
by (fact boolean-algebra.compl-eq-compl-iff)

lemma compl-bot-eq: − ⊥ = ⊤
by (fact boolean-algebra.compl-zero)

lemma compl-top-eq: − ⊤ = ⊥
by (fact boolean-algebra.compl-one)

lemma compl-inf: − (x ∩ y) = − x ∪ − y
by (fact boolean-algebra.de-Morgan-conj)

lemma compl-sup: − (x ∪ y) = − x ∩ − y
by (fact boolean-algebra.de-Morgan-disj)
lemma compl-mono:
  assumes $x \leq y$
  shows $-y \leq -x$
proof
  from assms have $x \sqcup y = y$ by (simp only: le_iff_sup)
  then have $-(x \sqcup y) = -y$ by simp
  then have $-x \sqcap -y = -y$ by simp
  then have $-y \sqcap -x = -y$ by (simp only: inf_commute)
  then show ?thesis by (simp only: le_iff_inf)
qed

lemma compl-le-compl-iff [simp]: $-x \leq -y \iff y \leq x$
  by (auto dest: compl-mono)

lemma compl-le-swap1:
  assumes $y \leq -x$
  shows $x \leq -y$
proof
  from assms have $-(\neg x) \leq -y$ by (simp only: compl-le-compl-iff)
  then show ?thesis by simp
qed

lemma compl-le-swap2:
  assumes $-y \leq x$
  shows $-x \leq -y$
proof
  from assms have $-x \leq -(\neg y)$ by (simp only: compl-le-compl-iff)
  then show ?thesis by simp
qed

lemma compl-less-compl-iff [simp]: $-x < -y \iff y < x$
  by (auto simp add: less_le)

lemma compl-less-swap1:
  assumes $y < -x$
  shows $x < -y$
proof
  from assms have $-(\neg x) < -y$ by (simp only: compl-less-compl-iff)
  then show ?thesis by simp
qed

lemma compl-less-swap2:
  assumes $-y < x$
  shows $-x < y$
proof
  from assms have $-x < -y$
    by (simp only: compl-less-compl-iff)
  then show ?thesis by simp
qed
lemma sup-cancel-left1: \( x \sqcup a \sqcup (\neg x \sqcup b) = \top \)
by (simp add: ac-simps)

lemma sup-cancel-left2: \( \neg x \sqcup a \sqcup (x \sqcup b) = \top \)
by (simp add: ac-simps)

lemma inf-cancel-left1: \( x \sqcap a \sqcap (\neg x \sqcap b) = \bot \)
by (simp add: ac-simps)

lemma inf-cancel-left2: \( \neg x \sqcap a \sqcap (x \sqcap b) = \bot \)
by (simp add: ac-simps)

lemma sup-compl-top-left1: \( \neg x \sqcup (x \sqcup y) = \top \)
by (simp add: sup-assoc [symmetric])

lemma sup-compl-top-left2: \( x \sqcup (\neg x \sqcup y) = \top \)
using sup-compl-top-left1 [of \( \neg x \ y \)] by simp

lemma inf-compl-bot-left1: \( \neg x \sqcap (x \sqcap y) = \bot \)
by (simp add: inf-assoc [symmetric])

lemma inf-compl-bot-left2: \( x \sqcap (\neg x \sqcap y) = \bot \)
using inf-compl-bot-left1 [of \( \neg x \ y \)] by simp

lemma inf-compl-bot-right: \( x \sqcap (y \sqcap \neg x) = \bot \)
by (subst inf-left-commute) simp

end

7.4 Lattice on bool

instantiation bool :: boolean-algebra
begin

definition bool-Compl-def [simp]: uminus = Not

definition bool-diff-def [simp]: A - B \(\iff\) A \(\land\) \(\neg\) B

definition [simp]: \( P \sqcap Q \iff P \land Q \)

definition [simp]: \( P \sqcup Q \iff P \lor Q \)

instance by standard auto

end

lemma sup-boolI1: \( P \implies P \sqcup Q \)
by simp
lemma sup-bool2: \( Q \Rightarrow P \sqcup Q \)
  by simp

lemma sup-boolE: \( P \sqcup Q \Rightarrow (P \Rightarrow R) \Rightarrow (Q \Rightarrow R) \Rightarrow R \)
  by auto

instance fun :: (type, boolean-algebra) boolean-algebra
  by standard (rule ext, simp-all add: inf-compl-bot sup-compl-top diff-eq)+

7.5 Lattice on unary and binary predicates

lemma inf1I: \( A x \Rightarrow B x \Rightarrow (A \sqcap B) x \)
  by (simp add: inf-fun-def)

lemma inf2I: \( A x y \Rightarrow B x y \Rightarrow (A \sqcap B) x y \)
  by (simp add: inf-fun-def)

lemma inf1E: \( (A \sqcap B) x \Rightarrow (A x \Rightarrow B x \Rightarrow P) \Rightarrow P \)
  by (simp add: inf-fun-def)

lemma inf2E: \( (A \sqcap B) x y \Rightarrow (A x y \Rightarrow B x y \Rightarrow P) \Rightarrow P \)
  by (simp add: inf-fun-def)

lemma inf1D1: \( (A \sqcap B) x \Rightarrow A x \)
  by (rule inf1E)

lemma inf2D1: \( (A \sqcap B) x y \Rightarrow A x y \)
  by (rule inf2E)

lemma inf1D2: \( (A \sqcap B) x \Rightarrow B x \)
  by (rule inf1E)

lemma inf2D2: \( (A \sqcap B) x y \Rightarrow B x y \)
  by (rule inf2E)

lemma sup1I: \( A x \Rightarrow (A \sqcup B) x \)
  by (simp add: sup-fun-def)

lemma sup2I: \( A x y \Rightarrow (A \sqcup B) x y \)
  by (simp add: sup-fun-def)

lemma sup1E: \( (A \sqcup B) x \Rightarrow (A x \Rightarrow P) \Rightarrow (B x \Rightarrow P) \Rightarrow P \)
by (simp add: sup-fun-def) iprover

lemma sup2E: 
  \((A \sqcup B) \times y \Rightarrow (A \times y \Rightarrow P) \Rightarrow (B \times y \Rightarrow P) \Rightarrow P\)
  by (simp add: sup-fun-def) iprover

Classical introduction rule: no commitment to \(A\) vs \(B\).

lemma sup1CI: 
  \((\neg B \times x \Rightarrow A \times x) \Rightarrow (A \sqcup B) \times x\)
  by (auto simp add: sup-fun-def)

lemma sup2CI: 
  \((\neg B \times y \Rightarrow A \times y) \Rightarrow (A \sqcup B) \times y\)
  by (auto simp add: sup-fun-def)

7.6 Simproc setup

locale boolean-algebra-cancel
begin

lemma sup1: 
  \((A::'a::semilattice-sup) \equiv sup k a \Rightarrow sup A b \equiv sup k (sup a b)\)
  by (simp only: ac-simps)

lemma sup2: 
  \((B::'a::semilattice-sup) \equiv sup k b \Rightarrow sup a B \equiv sup k (sup a b)\)
  by (simp only: ac-simps)

lemma sup0: 
  \((a::'a::bounded-semilattice-sup-bot) \equiv sup a bot\)
  by simp

lemma inf1: 
  \((A::'a::semilattice-inf) \equiv inf k a \Rightarrow inf A b \equiv inf k (inf a b)\)
  by (simp only: ac-simps)

lemma inf2: 
  \((B::'a::semilattice-inf) \equiv inf k b \Rightarrow inf a B \equiv inf k (inf a b)\)
  by (simp only: ac-simps)

lemma inf0: 
  \((a::'a::bounded-semilattice-inf-top) \equiv inf a top\)
  by simp

end

ML-file "Tools/boolean-algebra-cancel.ML"

simproc-setup boolean-algebra-cancel-sup (sup a b::'a::boolean-algebra) = 
  \(\langle K (K (\text{try Boolean-Algebra-Cancel.cancel-sup-conv}))\rangle\)

simproc-setup boolean-algebra-cancel-inf (inf a b::'a::boolean-algebra) = 
  \(\langle K (K (\text{try Boolean-Algebra-Cancel.cancel-inf-conv}))\rangle\)

context boolean-algebra
begin
lemma shunt1: \((x \sqcap y \leq z) \iff (x \leq -y \sqcup z)\)
proof
  assume \(x \sqcap y \leq z\)
  hence \(-y \sqcup (x \sqcap y) \leq -y \sqcup z\)
    using sup_mono by blast
  hence \(-y \sqcup x \leq -y \sqcup z\)
    by (simp add: sup-inf-distrib1)
  thus \(x \leq -y \sqcup z\)
    by simp
next
  assume \(x \leq -y \sqcup z\)
  hence \(x \sqcap y \leq (-y \sqcup z) \sqcap y\)
    using inf_mono by auto
  thus \(x \sqcap y \leq z\)
    using inf boundedE inf-sup-distrib2 by auto
qed

lemma shunt2: \((x \sqcap -y \leq z) \iff (x \leq y \sqcup z)\)
by (simp add: shunt1)

lemma inf-shunt: \((x \sqcap y = \bot) \iff (x \leq -y)\)
by (simp add: order_eq_iff shunt1)

lemma sup-shunt: \((x \sqcup y = \top) \iff (-x \leq y)\)
using inf-shunt [of \(-x\) \(-y\) symmetric]
by (simp flip: compl-sup compl-top-eq)

lemma diff-shunt-var[simp]: \((x - y = \bot) \iff (x \leq y)\)
by (simp add: diff_eq inf-shunt)

lemma diff-shunt[simp]: \((\bot = x - y) \iff (x \leq y)\)
by (auto simp flip: diff-shunt-var)

lemma sup-neg-inf:
  \(\langle p \leq q \sqcup r \iff p \sqcap -q \leq r \rangle\) (is \(?P \iff ?Q\))
proof
  assume \(?P\)
  then have \(p \sqcap -q \leq (q \sqcup r) \sqcap -q\)
    by (rule inf_mono) simp
  then show \(?Q\)
    by (simp add: inf-sup-distrib2)
next
  assume \(?Q\)
  then have \(p \sqcap -q \sqcup q \leq r \sqcup q\)
    by (rule sup_mono) simp
  then show \(?P\)
    by (simp add: sup-inf-distrib ac-simps)
qed
8 Set theory for higher-order logic

theory Set
  imports Lattices Boolean-Algebras
begin

8.1 Sets as predicates

typedcl 'a set

axiomatization Collect :: ('a ⇒ bool) ⇒ 'a set — comprehension
  and member :: 'a ⇒ 'a set ⇒ bool — membership

where mem-Coll-eq [iff, code-unfold]: member a (Collect P) = P a
  and Collect-mem-eq [simp]: Collect (λx. member x A) = A

notation member ('∈') and
  member ((/ / ∈) [51, 51] 50)

abbreviation not-member
  where not-member x A ≡ ¬(x ∈ A) — non-membership

notation not-member ('∉') and
  not-member ((/ / ∉) [51, 51] 50)

notation (ASCII)
  member ('\{') and
  member ((/ : - ) [51, 51] 50) and
  not-member ('\{\} ') and
  not-member ((/ / \∉) [51, 51] 50)

Set comprehensions

syntax
  -Coll :: pttrn ⇒ bool ⇒ 'a set ((1{-/-}))

translations
  {x. P} = CONST Collect (λx. P)

syntax (ASCII)
  -Collect :: pttrn ⇒ 'a set ⇒ bool ⇒ 'a set ((1{[-:-]-}))

syntax
  -Collect :: pttrn ⇒ 'a set ⇒ bool ⇒ 'a set ((1{[- / ∈] -}))

translations
  {p:A. P} → CONST Collect (λp. p ∈ A ∧ P)

lemma CollectI: P a ⇒ a ∈ {x. P x}
THEORY “Set”

by simp

lemma CollectD: a ∈ {x. P x} ⟹ P a
  by simp

lemma Collect-cong: (∀x. P x = Q x) ⟹ {x. P x} = {x. Q x}
  by simp

Simproc for pulling \( x = t \) in \{x. \ldots \land x = t \land \ldots \} to the front (and similarly for \( t = x \)):

simproc-setup defined-Collect {(x. P x ∧ Q x)} = 
K (Quantifier1.rearrange-Collect
(fn ctxt =>
  resolve-tac ctxt @{thms Collect-cong} 1 THEN
  resolve-tac ctxt @{thms iffI} 1 THEN
  ALLGOALS
    (EVERY' [REPEAT-DETERM o eresolve-tac ctxt @{thms conjE},
      DEPTH-SOLVE-1 o (assume-tac ctxt ORELSE' resolve-tac ctxt @{thms conjI})))))
).

lemmas CollectE = CollectD [elim-format]

lemma set-eqI:
  assumes \( \forall x. x ∈ A ←→ x ∈ B \)
  shows A = B
proof –
  from assms have \( \{x. x ∈ A\} = \{x. x ∈ B\} \)
  by simp
  then show ?thesis by simp
qed

lemma set-eq-iff: A = B ⟷ (∀x. x ∈ A ⟷ x ∈ B)
  by (auto intro:set-eqI)

lemma Collect-eqI:
  assumes \( \forall x. P x = Q x \)
  shows Collect P = Collect Q
  using assms by (auto intro: set-eqI)

Lifting of predicate class instances

instantiation set :: (type) boolean-algebra
begin

definition less-eq-set
  where A ≤ B ⟷ (λx. member x A) ≤ (λx. member x B)

definition less-set
  where A < B ⟷ (λx. member x A) < (λx. member x B)
definition \textit{inf-set}  
where \( A \cap B = \text{Collect} \((\lambda x. \text{member } x A) \cap (\lambda x. \text{member } x B)\) \)

definition \textit{sup-set}  
where \( A \cup B = \text{Collect} \((\lambda x. \text{member } x A) \cup (\lambda x. \text{member } x B)\) \)

definition \textit{bot-set}  
where \( \bot = \text{Collect } \bot \)

definition \textit{top-set}  
where \( \top = \text{Collect } \top \)

definition \textit{uminus-set}  
where \( A = \text{Collect} (- (\lambda x. \text{member } x A)) \)

definition \textit{minus-set}  
where \( A \setminus B = \text{Collect} ((\lambda x. \text{member } x A) \setminus (\lambda x. \text{member } x B)) \)

\text{instance}  
\text{by standard}  
\((\text{simp-all add: less-eq-set-def less-set-def inf-set-def sup-set-def}
\text{ bot-set-def top-set-def uminus-set-def minus-set-def}
\text{ less-le-not-le sup-inf-distrib1 diff-eq set-eqI fun-eq-iff}
\text{ del: inf-apply sup-apply bot-apply top-apply minus-apply uminus-apply})\)

end

Set enumerations

abbreviation \textit{empty} :: 'a set \((\{}\)\)  
where \( \{} \equiv \bot \)

definition \textit{insert} :: 'a ⇒ 'a set ⇒ 'a set  
where \( \text{insert-compr: } \text{insert } a B = \{x. x = a \lor x \in B\} \)

\text{syntax}  
-AFinset :: args ⇒ 'a set \((\{\}-)\)  
\text{translations}  
\( \{x, xs\} \Rightarrow \text{CONST insert } x \{xs\} \)
\( \{x\} \Rightarrow \text{CONST insert } x \{\} \)

8.2 Subsets and bounded quantifiers

abbreviation \textit{subset} :: 'a set ⇒ 'a set ⇒ bool  
where \( \text{subset } \equiv \text{less} \)

abbreviation \textit{subset-eq} :: 'a set ⇒ 'a set ⇒ bool  
where \( \text{subset-eq } \equiv \text{less-eq} \)
\[\text{THEORY "Set" 168}\]

\section*{Syntax}

\textbf{notation}

\begin{itemize}
  \item \texttt{subset} (\(\langle \subseteq \rangle\)) and
  \item \texttt{subset} (\(\langle / \subseteq \rangle\) [51, 51] 50) and
  \item \texttt{subset-eq} (\(\langle \subseteq \rangle\)) and
  \item \texttt{subset-eq} (\(\langle / \subseteq \rangle\) [51, 51] 50)
\end{itemize}

\section*{Abbreviation (Input)}

\texttt{supset :: 'a set \(\Rightarrow\) 'a set \(\Rightarrow\) bool where supset \equiv \text{greater}\n}

\section*{Abbreviation (Input)}

\texttt{supset-eq :: 'a set \(\Rightarrow\) 'a set \(\Rightarrow\) bool where supset-eq \equiv \text{greater-eq}\n}

\section*{Syntax (ASCII output)}

\begin{itemize}
  \item \texttt{subset} (\(\langle < \rangle\)) and
  \item \texttt{subset} (\(\langle / < \rangle\) [51, 51] 50) and
  \item \texttt{subset-eq} (\(\langle < \rangle\)) and
  \item \texttt{subset-eq} (\(\langle / \leq \rangle\) [51, 51] 50)
\end{itemize}

\section*{Definition}

\texttt{Ball :: 'a set \(\Rightarrow\) ('a \(\Rightarrow\) bool) \(\Rightarrow\) bool where Ball A P \(\iff\) (\(\forall\) x, x \(\in\) A \(\rightarrow\) P x) \(\rightarrow\) bounded universal quantifiers}

\section*{Definition}

\texttt{Bex :: 'a set \(\Rightarrow\) ('a \(\Rightarrow\) bool) \(\Rightarrow\) bool where Bex A P \(\iff\) (\(\exists\) x, x \(\in\) A \& P x) \(\rightarrow\) bounded existential quantifiers}

\section*{Syntax (ASCII)}

\begin{itemize}
  \item \texttt{-Ball :: pttrn \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) bool \(\rightarrow\)} (\(\texttt{3ALL (-/-)./ -}\) [0, 0, 10] 10)
  \item \texttt{-Bex :: pttrn \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) bool \(\rightarrow\)} (\(\texttt{3EX (-/-)./ -}\) [0, 0, 10] 10)
  \item \texttt{-Bex1 :: pttrn \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) bool \(\rightarrow\)} (\(\texttt{3EX! (-/-)./ -}\) [0, 0, 10] 10)
  \item \texttt{-Bleast :: id \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) 'a \(\rightarrow\)} (\(\texttt{3LEAST (-/-)./ -}\) [0, 0, 10] 10)
\end{itemize}

\section*{Syntax (Input)}

\begin{itemize}
  \item \texttt{-Ball :: pttrn \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) bool \(\rightarrow\)} (\(\texttt{3! (-/-)./ -}\) [0, 0, 10] 10)
  \item \texttt{-Bex :: pttrn \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) bool \(\rightarrow\)} (\(\texttt{3? (-/-)./ -}\) [0, 0, 10] 10)
  \item \texttt{-Bex1 :: pttrn \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) bool \(\rightarrow\)} (\(\texttt{3?! (-/-)./ -}\) [0, 0, 10] 10)
\end{itemize}

\section*{Syntax}

\begin{itemize}
  \item \texttt{-Ball :: pttrn \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) bool \(\rightarrow\)} (\(\texttt{3\forall (-/-)/ -}\) [0, 0, 10] 10)
  \item \texttt{-Bex :: pttrn \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) bool \(\rightarrow\)} (\(\texttt{3\exists (-/-)/ -}\) [0, 0, 10] 10)
  \item \texttt{-Bex1 :: pttrn \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) bool \(\rightarrow\)} (\(\texttt{3\exists! (-/-)/ -}\) [0, 0, 10] 10)
  \item \texttt{-Bleast :: id \(\Rightarrow\) 'a set \(\Rightarrow\) bool \(\Rightarrow\) 'a \(\rightarrow\)} (\(\texttt{3\LEAST (-/-)/ -}\) [0, 0, 10] 10)
\end{itemize}
translations

∀x∈A. P ⇒ CONST Ball A (∀x. P)
∃x∈A. P ⇒ CONST Bex A (∃x. P)
∃!x∈A. P ⇒ ∃!x. x ∈ A ∧ P
LEAST x.A. P ⇒ LEAST x. x ∈ A ∧ P

syntax (ASCII output)

-setlessAll :: [idt, 'a, bool] ⇒ bool ((∃ALL ->. / -) [0, 0, 10] 10)
-setlessEx :: [idt, 'a, bool] ⇒ bool ((∃EX ->. / -) [0, 0, 10] 10)
-selleAll :: [idt, 'a, bool] ⇒ bool ((∀ALL ->. / -) [0, 0, 10] 10)
-selleEx :: [idt, 'a, bool] ⇒ bool ((∀EX ->. / -) [0, 0, 10] 10)
-selleEx1 :: [idt, 'a, bool] ⇒ bool ((∃EX! ->. / -) [0, 0, 10] 10)

translations

∀A⊂B. P ⇒ ∀A. A ⊂ B ⇒ P
∃A⊂B. P ⇒ ∃A. A ⊂ B ∧ P
∀A⊂B. P ⇒ ∀A. A ⊂ B ⇒ P
∃!A⊂B. P ⇒ ∃!A. A ⊂ B ∧ P

print-translation :

let
val All-binder = Mixfix.binder-name const-syntax (All);
val Ex-binder = Mixfix.binder-name const-syntax (Ex);
val impl = const-syntax (HOL.implies);
val conj = const-syntax (HOL.conj);
val sbset = const-syntax (subset);
val sbset-eq = const-syntax (subset-eq);

val trans =
[(All-binder, impl, sbset),
 (Ex-binder, impl, sbset-eq),
 (All-binder, conj, sbset),
 (Ex-binder, conj, sbset-eq),
 (Ex-binder, conj, sbset-eq),
 (Ex-binder, conj, sbset-eq),
 (Ex-binder, conj, sbset-eq)];

fun mk v (v', T) c n P =
  if v = v' andalso not (Term.exists-subterm (fn Free (x, _) =>> x = v | - =>
  false) n)
  then Syntax.const c $ Syntax.Trans.mark-bound-body (v', T) $ n $ P
  else raise Match;

fun tr' q = (q, fn - =>}
(fn [Const (syntax-const (-bound), -) $ Free (v, Type (type-name (set, -))), Const (c, -) $ (Const (d, -) $ (Const (syntax-const (-bound), -) $ Free (v', T))) $ n) $ P] =>
  (case AList.lookup (=) trans (q, c, d) of
   NONE => raise Match
 | SOME l => mk (v', T) l n P)
 | _ => raise Match));

in [tr' All-binder, tr' Ex-binder]
end

Translate between \{ e | x1\ldots xn. P \} and \{ u. \exists x1\ldots xn. u = e \land P \}; \{ y. \exists x1\ldots xn. y = e \land P \} is only translated if \[0..n] \subseteq \text{bvs } e.

syntax
-Setcompr :: 'a \Rightarrow idts \Rightarrow bool \Rightarrow 'a set ((I[-]/-. -))

parse-translation
let
  val ex-tr = snd (Syntax-Trans.mk-binder-tr (EX, const-syntax (Ex)));

  fun nvars (Const (syntax-const (-idts), -) $ - $ idts) = nvars idts + 1
  | nvars - = 1;

  fun setcompr-tr ctxt [e, idts, b] =
    let
      fun check (Const (syntax-const (HOL.eq), -) $ Bound (nvars idts) $ e; val P = Syntax.const HOL.conj $ eq $ b; val exP = ex-tr ctxt [idts, P]; val eq = Syntax.const const-syntax (HOL.eq) $ Bound (nvars idts) $ e; val P = Syntax.const HOL.conj $ eq $ b; val exP = ex-tr ctxt [idts, P];
      in Syntax.const const-syntax (Collect) $ absdummy dummyT exP end;
    in [(syntax-const (-Setcompr), setcompr-tr)] end
  end;

print-translation :
[Syntax-Trans.preserve-binder-abs2-tr' const-syntax (Ball) syntax-const (-Ball), Syntax-Trans.preserve-binder-abs2-tr' const-syntax (Bex) syntax-const (-Bex)]

— to avoid eta-contraction of body

print-translation :
let
  val ex-tr' = snd (Syntax-Trans.mk-binder-tr' (const-syntax (Ex), DUMMY));

  fun setcompr-tr' ctxt [Abs (abs as (_, _, P))] =
    let
      fun check (Const (const-syntax (Ex), -) $ Abs (_, _, P), n) = check (P, n + 1)
      | check (Const (const-syntax (HOL.conj), -) $
(Const (const-syntax:HOL.eq, -) $ Bound m $ e) $ P, n) =
  n > 0 andalso m = n andalso not (loose-bvar1 (P, n)) andalso
  subset (=) (0 upto (n - 1), add-loose-bnos (e, 0, []))
| check = false;

fun tr' (- $ abs) =
  let val - $ idts $ (- $ - $ e) $ Q) = ex-tr' ctxt [abs]
      in Syntax.const syntax-const (-Setcompr $ e $ idts $ Q end)
  end
in
  if check (P, 0) then tr' P
  else
    let
      val (x as - $ Free(xN, -), t) = Syntax-Trans.atomic-abs-tr' abs;
      val M = Syntax.const syntax-const (-Coll $ x $ t);
      in
        case t of
          Const (const-syntax:HOL.conj, -) $
          (Const (const-syntax:Set.member), -) $
            (Const (syntax-const(-bound), -) $ Free (yN, -)) $ A) $ P =>>
            if xN = yN then Syntax.const syntax-const (-Collect) $ x $ A $ P else
            M
        end
    end;
  end
in [(const-syntax:Collect), setcompr-tr'] end

setup defined-Bex (∃ x∈A. P x ∧ Q x) = (K (Quantifier1.rearrange-Bex (fn ctxt =>> unfold-tac ctxt @{thms Bex-def})))

setup defined-All (∀ x∈A. P x → Q x) = (K (Quantifier1.rearrange-Ball (fn ctxt =>> unfold-tac ctxt @{thms Ball-def})))

lemma ballI [intro!]: (∀ x. x ∈ A ⇒ P x) ⇒ ∀ x∈A. P x
  by (simp add: Ball-def)

lemmas strip = impI allI ballI

lemma bspec [dest?!]: ∀ x∈A. P x ⇒ x ∈ A ⇒ P x
  by (simp add: Ball-def)

Gives better instantiation for bound:

setup (map-theory-claset (fn ctxt =>>
      ctxt addbefore (bspec, fn ctxt' =>> dresolve-tac ctxt' @{thms bspec} THEN' assume-tac ctxt')))

ML \\< \\
structure Simpdata = \\
  struct \\
  open Simpdata; \\
  val mksimps-pairs = [(const-name Ball, \{thms bspec\})] @ mksimps-pairs; \\
  end; \\
open Simpdata; \\
> \\
declaration \langle fn - => Simplifier.map-ss (Simplifier.set-mksimps (mksimps mk-simps-pairs)) \rangle \\
lemma ballE [elim]; \forall x \in A. P x \supset (P x \supset Q) \supset (x \notin A \supset Q) \supset Q \\
  unfolding Ball-def by blast \\
lemma bexI [intro]; P x \supset x \in A \supset \exists x \in A. P x \\
  — Normally the best argument order: P x constrains the choice of x \in A. \\
  unfolding Bex-def by blast \\
lemma rev-bexI [intro?]; x \in A \supset P x \supset \exists x \in A. P x \\
  — The best argument order when there is only one x \in A. \\
  unfolding Bex-def by blast \\
lemma bexCI: (\forall x \in A. \neg P x \supset P a) \supset a \in A \supset \exists x \in A. P x \\
  unfolding Bex-def by blast \\
lemma bexE [elim!]; \exists x \in A. P x \supset (\\\forall x. x \in A \supset P x \supset Q) \supset Q \\
  unfolding Bex-def by blast \\
lemma ball-triv [simp]; (\forall x \in A. P) \longleftrightarrow ((\exists x. x \in A) \supset P) \\
  — trivial rewrite rule. \\
  by (simp add: Ball-def) \\
lemma bex-triv [simp]; (\exists x \in A. P) \longleftrightarrow ((\exists x. x \in A) \land P) \\
  — Dual form for existentials. \\
  by (simp add: Bex-def) \\
lemma bex-triv-one-point1 [simp]; (\exists x \in A. x = a) \longleftrightarrow a \in A \\
  by blast \\
lemma bex-triv-one-point2 [simp]; (\exists x \in A. a = x) \longleftrightarrow a \in A \\
  by blast \\
lemma bex-one-point1 [simp]; (\exists x \in A. x = a \land P x) \longleftrightarrow a \in A \land P a \\
  by blast \\
lemma bex-one-point2 [simp]; (\exists x \in A. a = x \land P x) \longleftrightarrow a \in A \land P a
by blast

lemma ball-one-point1 [simp]: \( (\forall x \in A. \; x = a \rightarrow P x) \iff (a \in A \rightarrow P a) \)
by blast

lemma ball-one-point2 [simp]: \( (\forall x \in A. \; a = x \rightarrow P x) \iff (a \in A \rightarrow P a) \)
by blast

lemma ball-conj-distrib: \( (\forall x \in A. \; P x \land Q x) \iff (\forall x \in A. \; P x) \land (\forall x \in A. \; Q x) \)
by blast

lemma bex-disj-distrib: \( (\exists x \in A. \; P x \lor Q x) \iff (\exists x \in A. \; P x) \lor (\exists x \in A. \; Q x) \)
by blast

Congruence rules

lemma ball-cong:
\[
\begin{array}{ll}
[ & A = B; \quad \forall x. \; x \in B \implies P x \iff Q x ] \implies \\
(\forall x \in A. \; P x) & \iff (\forall x \in B. \; Q x)
\end{array}
\]
by (simp add: Ball-def)

lemma ball-cong-simp [cong]:
\[
\begin{array}{ll}
[ & A = B; \quad \forall x. \; x \in B \implies P x \iff Q x ] \implies \\
(\forall x \in A. \; P x) & \iff (\forall x \in B. \; Q x)
\end{array}
\]
by (simp add: simp-implies-def Ball-def)

lemma bex-cong:
\[
\begin{array}{ll}
[ & A = B; \quad \forall x. \; x \in B \implies P x \iff Q x ] \implies \\
(\exists x \in A. \; P x) & \iff (\exists x \in B. \; Q x)
\end{array}
\]
by (simp add: Bex-def cong: conj-cong)

lemma bex-cong-simp [cong]:
\[
\begin{array}{ll}
[ & A = B; \quad \forall x. \; x \in B \implies P x \iff Q x ] \implies \\
(\exists x \in A. \; P x) & \iff (\exists x \in B. \; Q x)
\end{array}
\]
by (simp add: simp-implies-def Bex-def cong: conj-cong)

lemma bex1-def: \( (\exists ! x \in X. \; P x) \iff (\exists x \in X. \; P x) \land (\forall x \in X. \; \forall y \in X. \; P x \rightarrow P y \implies x = y) \)
by auto

8.3 Basic operations
8.3.1 Subsets

lemma subsetI [intro!]: \( (\forall x. \; x \in A \implies x \in B) \implies A \subseteq B \)
by (simp add: less-eq-set-def le-fan-def)

Map the type ’a set \Rightarrow anything to just ’a; for overloading constants whose
first argument has type ’a set.

lemma subsetD [elim, intro?] : \( A \subseteq B \implies c \in A \Rightarrow c \in B \)
by (simp add: less-eq-set-def le-fun-def)
— Rule in Modus Ponens style.

lemma rev-subsetD [intro?, no-atp]:
c ∈ A ⇒ A ⊆ B ⇒ c ∈ B
— The same, with reversed premises for use with erule – cf. [?P; ?P → ?Q]
⇒ ?Q.
by (rule subsetD)

lemma subsetCE [elim, no-atp]:
A ⊆ B ⇒ (c ∉ A ⇒ P) ⇒ (c ∈ B ⇒ P)
⇒ P
— Classical elimination rule.
by (auto simp add: less-eq-set-def le-fun-def)

lemma subset-eq: A ⊆ B ⇔ (∀ x ∈ A. x ∈ B)
by blast

lemma contra-subsetD [no-atp]:
A ⊆ B ⇒ c /∈ B ⇒ c /∈ A
by blast

lemma subset-refl: A ⊆ A
by (fact order-refl)

lemma subset-trans: A ⊆ B ⇒ B ⊆ C ⇒ A ⊆ C
by (fact order-trans)

lemma subset-not-subset-eq [code]:
A ⊆ B ⇔ A ⊆ B ∧ ¬ B ⊆ A
by (fact less-le-not-le)

lemma eq-mem-trans: a = b ⇒ b ∈ A ⇒ a ∈ A
by simp

lemmas basic-trans-rules [trans] =
order-trans-rules rev-subsetD subsetD eq-mem-trans

8.3.2 Equality

lemma subset-antisym [intro!]:
A ⊆ B ⇒ B ⊆ A ⇒ A = B
— Anti-symmetry of the subset relation.
by (iprover intro: set-eqI subsetD)

Equality rules from ZF set theory – are they appropriate here?

lemma equalityD1: A = B ⇒ A ⊆ B
by simp

lemma equalityD2: A = B ⇒ B ⊆ A
by simp

Be careful when adding this to the claset as subset-empty is in the simpset:
A = {} goes to {} ⊆ A and A ⊆ {} and then back to A = {}!
lemma equalityE: \( A = B \implies (A \subseteq B \implies B \subseteq A \implies P) \implies P \)
by simp

lemma equalityCE [elim]: \( A = B \implies (c \in A \implies c \in B \implies P) \implies (c \notin A \implies c \notin B \implies P) \implies P \)
by blast

lemma eqset-imp-iff: \( A = B \implies x \in A \iff x \in B \)
by simp

lemma eqelem-imp-iff: \( x = y \implies x \in A \iff y \in A \)
by simp

8.3.3 The empty set

lemma empty-def: \( \{ \} = \{ x. \text{False} \} \)
by (simp add: bot-set-def bot-fun-def)

lemma empty-iff [simp]: \( c \in \{ \} \iff \text{False} \)
by (simp add: empty-def)

lemma emptyE [elim!]: \( a \in \{ \} \implies P \)
by simp

lemma empty-subsetI [iff]: \( \{ \} \subseteq A \)
— One effect is to delete the ASSUMPTION \( \{ \} \subseteq A \)
by blast

lemma equals0I: \( (\forall y. y \in A \implies \text{False}) \implies A = \{ \} \)
by blast

lemma equals0D: \( A = \{ \} \implies a \notin A \)
— Use for reasoning about disjointness: \( A \cap B = \{ \} \)
by blast

lemma ball-empty [simp]: Ball \( \{ \}\) P \iff True
by (simp add: Ball-def)

lemma bex-empty [simp]: Bex \( \{ \}\) P \iff False
by (simp add: Bex-def)

8.3.4 The universal set – UNIV

abbreviation UNIV :: 'a set
where UNIV \equiv top

lemma UNIV-def: UNIV = \( \{ x. \text{True} \} \)
by (simp add: top-set-def top-fun-def)

lemma UNIV-I [simp]: \( x \in \text{UNIV} \)
THEORY “Set”  

by (simp add: UNIV-def)

declare UNIV-I [intro] — unsafe makes it less likely to cause problems

lemma UNIV-witness [intro?]: \( \exists x. x \in \text{UNIV} \)
  by simp

lemma subset-UNIV: \( A \subseteq \text{UNIV} \)
  by (fact top-greatest)

Eta-contracting these two rules (to remove \( P \)) causes them to be ignored because of their interaction with congruence rules.

lemma ball-UNIV [simp]: Ball UNIV \( P \) ←→ All \( P \)
  by (simp add: Ball-def)

lemma bex-UNIV [simp]: Bex UNIV \( P \) ←→ Ex \( P \)
  by (simp add: Bex-def)

lemma UNIV-eq-I: \( (\forall x. x \in A) \Rightarrow \text{UNIV} = A \)
  by auto

lemma UNIV-not-empty [iff]: \( \text{UNIV} \neq {} \)
  by (blast elim: equalityE)

lemma empty-not-UNIV[simp]: \( {} \neq \text{UNIV} \)
  by blast

8.3.5 The Powerset operator – Pow

definition Pow :: ‘a set ⇒ ‘a set set
  where Pow-def: Pow \( A \) = \{ \( B. B \subseteq A \)\}

lemma Pow-iff [iff]: \( A \in \text{Pow} B \) ←→ \( A \subseteq B \)
  by (simp add: Pow-def)

lemma PowI: \( A \subseteq B \Rightarrow A \in \text{Pow} B \)
  by (simp add: Pow-def)

lemma PowD: \( A \in \text{Pow} B \Rightarrow A \subseteq B \)
  by (simp add: Pow-def)

lemma Pow-bottom: \( {} \in \text{Pow} B \)
  by simp

lemma Pow-top: \( A \in \text{Pow} A \)
  by simp

lemma Pow-not-empty: Pow \( A \) \( \neq {} \)
  using Pow-top by blast
8.3.6 Set complement

**lemma** Compl-iff \[\text{simp}]: c \in -A \iff c \notin A \\
by (simp add: fun-Compl-def uminus-set-def)

**lemma** ComplI \[\text{intro}!]: (c \in A \implies \text{False}) \implies c \in -A \\
by (simp add: fun-Compl-def uminus-set-def) blast

This form, with negated conclusion, works well with the Classical prover. Negated assumptions behave like formulae on the right side of the notional turnstile ...  

**lemma** ComplD \[\text{dest}!]: c \in -A \implies c \notin A \\
by simp

**lemmas** ComplE = ComplD [elim-format]

**lemma** Compl-eq: -A = \{x. \neg x \in A\} \\
by blast

8.3.7 Binary intersection

**abbreviation** inter :: 'a set \Rightarrow 'a set \Rightarrow 'a set (infixl \cap 70) \\
where (\cap) \equiv \text{inf}

**notation** (ASCII) 
inter (infixl Int 70)

**lemma** Int-def: A \cap B = \{x. x \in A \land x \in B\} \\
by (simp add: inf-set-def inf-fun-def)

**lemma** Int-iff \[\text{simp}]: c \in A \cap B \iff c \in A \land c \in B \\
unfolding Int-def by blast

**lemma** IntI \[\text{intro}!]: c \in A \implies c \in B \implies c \in A \cap B \\
by simp

**lemma** IntD1: c \in A \cap B \implies c \in A \\
by simp

**lemma** IntD2: c \in A \cap B \implies c \in B \\
by simp

**lemma** IntE \[\text{elim}!]: c \in A \cap B \implies (c \in A \implies c \in B \implies P) \implies P \\
by simp

8.3.8 Binary union

**abbreviation** union :: 'a set \Rightarrow 'a set \Rightarrow 'a set (infixl \cup 65) \\
where union \equiv \text{sup}
notation (ASCII)
  \text{union} (\text{infixl} \ Un \ 65)

\textbf{lemma} Un-def: \(A \cup B = \{x. \ x \in A \lor x \in B\}\)
  by (simp add: sup-set-def sup-fun-def)

\textbf{lemma} Un-iff [simp]: \(c \in A \cup B \iff c \in A \lor c \in B\)
  unfolding Un-def by blast

\textbf{lemma} UnI1 [elim?]: \(c \in A \Rightarrow c \in A \cup B\)
  by simp

\textbf{lemma} UnI2 [elim?]: \(c \in B \Rightarrow c \in A \cup B\)
  by simp

Classical introduction rule: no commitment to \(A\) vs. \(B\).

\textbf{lemma} UnCI [intro!]: \((c \notin B \Rightarrow c \in A) \Rightarrow c \in A \cup B\)
  by auto

\textbf{lemma} UnE [elim!]: \(c \in A \cup B \Rightarrow (c \in A \Rightarrow P) \Rightarrow (c \in B \Rightarrow P) \Rightarrow P\)
  unfolding Un-def by blast

\textbf{lemma} insert-def: \(\text{insert} \ a \ B = \{x. \ x = a\} \cup B\)
  by (simp add: insert-compr Un-def)

\subsection{8.3.9 Set difference}

\textbf{lemma} Diff-iff [simp]: \(c \in A - B \iff c \in A \land c \notin B\)
  by (simp add: minus-set-def fun-diff-def)

\textbf{lemma} DiffI [intro!]: \(c \in A \Rightarrow c \notin B \Rightarrow c \in A - B\)
  by simp

\textbf{lemma} DiffD1: \(c \in A - B \Rightarrow c \in A\)
  by simp

\textbf{lemma} DiffD2: \(c \in A - B \Rightarrow c \in B \Rightarrow P\)
  by simp

\textbf{lemma} DiffE [elim!]: \(c \in A - B \Rightarrow (c \in A \Rightarrow c \notin B \Rightarrow P) \Rightarrow P\)
  by simp

\textbf{lemma} set-diff-eq: \(A - B = \{x. \ x \in A \land x \notin B\}\)
  by blast

\textbf{lemma} Compl-eq-Diff-UNIV: \(-A = (\text{UNIV} - A)\)
  by blast
abbreviation sym-diff :: 'a set ⇒ 'a set ⇒ 'a set where
  sym-diff A B ≡ ((A - B) ∪ (B-A))

8.3.10 Augmenting a set – insert

lemma insert-iff [simp]: a ∈ insert b A ⇔ a = b ∨ a ∈ A
  unfolding insert-def by blast

lemma insertI1: a ∈ insert a B
  by simp

lemma insertI2: a ∈ B ⇒ a ∈ insert b B
  by simp

lemma insertE [elim!]: a ∈ insert b A ⇒ (a = b ⇒ P) ⇒ (a ∈ A ⇒ P) ⇒ P
  unfolding insert-def by blast

lemma insertCI [intro!]: (a ∉ B ⇒ a = b) ⇒ a ∈ insert b B
  — Classical introduction rule.
  by auto

lemma subset-insert-iff: A ⊆ insert x B ⇔
  (if x ∈ A then A - {x} ⊆ B else A ⊆ B)
  by auto

lemma set-insert:
  assumes x ∈ A
  obtains B where A = insert x B and x ∉ B
proof
  show A = insert x (A - {x}) using assms by blast
  show x ∉ A - {x} by blast
qed

lemma insert-ident: x ∉ A ⇒ x ∉ B ⇒
  insert x A = insert x B ⇔ A = B
  by auto

lemma insert-eq-iff:
  assumes a ∉ A b ∉ B
  shows insert a A = insert b B
  (if a = b then A = B else ∃ C. A = insert b C \& b ∉ C \& B = insert a C \& a ∉ C)
  (is ?L ⇔ ?R)
proof
  show ?R if ?L
  proof (cases a = b)
    case True
    with assms (?L) show ?R
    by (simp add: insert-ident)
next
case False
let ?C = A − {b}
have A = insert b ?C ∧ b /∈ ?C ∧ B = insert a ?C ∧ a /∈ ?C
  using assms (?L, (a ≠ b) by auto
then show ?R using (a ≠ b) by auto
qed
show ?L if ?R
  using that by (auto split: if-splits)
qed

lemma insert-UNIV[simp]: insert x UNIV = UNIV
by auto

8.3.11 Singletons, using insert

lemma singletonI [intro!]: a ∈ {a}
  — Redundant? But unlike insertCI, it proves the subgoal immediately!
  by (rule insertI1)

lemma singletonD [dest!]: b ∈ {a} =⇒ b = a
  by blast
lemmas singletonE = singletonD [elim-format]

lemma singleton-iff: b ∈ {a} ←→ b = a
  by blast

lemma singleton-inject [dest!]: {a} = {b} =⇒ a = b
  by blast

lemma singleton-insert-inj-eq [iff]: {b} = insert a A ←→ a = b ∧ A ⊆ {b}
  by blast

lemma singleton-insert-inj-eq' [iff]: insert a A = {b} ←→ a = b ∧ A ⊆ {b}
  by blast

lemma subset-singletonD: A ⊆ {x} =⇒ A = {} ∨ A = {x}
  by fast

lemma subset-singleton-iff: X ⊆ {a} ←→ X = {} ∨ X = {a}
  by blast

lemma subset-singleton-iff-Uniq: (∃ a. A ⊆ {a}) ←→ (∃ x. x ∈ A)
  unfolding Uniq-def by blast

lemma singleton-conv [simp]: {x. x = a} = {a}
  by blast
lemma singleton-conv2 [simp]: \{x. a = x\} = \{a\}
  by blast

lemma Diff-single-insert: \(A = \{x\} \subseteq B \implies A \subseteq insert\ x\ B\)
  by blast

lemma subset-Diff-insert: \(A \subseteq B - insert\ x\ C \iff A \subseteq B - C \land x \notin A\)
  by blast

lemma doubleton-eq-iff: \{a, b\} = \{c, d\} \iff a = c \land b = d \lor a = d \land b = c
  by (blast elim: equalityE)

lemma Un-singleton-iff: \(A \cup B = \{x\} \iff A = {} \land B = \{x\} \lor A = \{x\} \land B = {} \lor A = {} \land B = \{x\}\)
  by auto

lemma singleton-Un-iff: \(\{x\} = A \cup B \iff A = {} \land B = \{x\} \lor A = \{x\} \land B = {} \lor A = {} \land B = \{x\}\)
  by auto

8.3.12 Image of a set under a function

Frequently \(b\) does not have the syntactic form of \(f\ x\).

definition image :: ('a => 'b) => 'a set => 'b set  (infixr ' 90)
  where \(f\ ' A = \{y. \exists x \in A. y = f\ x\}\)

lemma image-eqI: \(b = f\ x\ = x \in A \implies b \in f\ ' A\)
  unfolding image-def by blast

lemma imageI: \(x \in A \implies f\ x \in f\ ' A\)
  by (rule image-eqI) (rule refl)

lemma rev-image-eqI: \(x \in A \implies b = f\ x \implies b \in f\ ' A\)
  — This version's more effective when we already have the required \(x\).
  by (rule image-eqI)

lemma imageE [elim!]:
  assumes \(b = (\lambda x. f\ x)\ ' A\) — The eta-expansion gives variable-name preservation.
  obtains \(x\ where\ b = f\ x\ and\ x \in A\)
  using assms unfolding image-def by blast

lemma Compr-image-eq: \(\{x \in f\ ' A. P\ x\} = f\ ' \{x \in A. P\ (f\ x)\}\)
  by auto

lemma image-Un: \(f\ ' (A \cup B) = f\ ' A \cup f\ ' B\)
  by blast

lemma image-iff: \(z \in f\ ' A \iff (\exists x \in A. z = f\ x)\)
  by blast
lemma image-subsetI: \((\forall x. x \in A \implies f x \in B) \implies f^A \subseteq B\)
— Replaces the three steps subsetI, imageE, hypsubst, but breaks too many existing proofs.

   by blast

lemma image-subset-iff: \(f^A \subseteq B \iff (\forall x \in A. f x \in B)\)
— This rewrite rule would confuse users if made default.

   by blast

lemma subset-imageE:
   assumes \(B \subseteq f^A\)
   obtains \(C\) where \(C \subseteq A\) and \(B = f^C\)
proof –
   from assms have \(B = f^\{a \in A. f a \in B\}\) by fast
   moreover have \(\{a \in A. f a \in B\} \subseteq A\) by blast
   ultimately show thesis by (blast intro: that)
qed

lemma subset-image-iff: \(B \subseteq f^A \iff (\exists AA \subseteq A. B = f^AA)\)
   by (blast elim: subset-imageE)

lemma image-ident [simp]: \(\lambda x. Y = Y\)
   by blast

lemma image-empty [simp]: \(f^\{\} = \{\}\)
   by blast

lemma image-insert [simp]: \(f^\{\} = \{\}\)
   by blast

lemma image-constant: \(x \in A \implies (\lambda x. c) \cdot A = \{c\}\)
   by auto

lemma image-constant-conv: \((\lambda x. c) \cdot A = (\text{if } A = \{\} \text{ then } \{\} \text{ else } \{c\})\)
   by auto

lemma image-image: \(f^\{(g \cdot A) = (\lambda x. f (g x)) \cdot A\)
   by blast

lemma insert-image [simp]: \(x \in A \implies \text{insert } f x) (f^A) = f^A\)
   by blast

lemma image-is-empty [iff]: \(f^A = \{\} \iff A = \{\}\)
   by blast

lemma empty-is-image [iff]: \(\{\} = f^A \iff A = \{\}\)
   by blast
**THEORY “Set”**

**lemma** image-Collect: $f \cdot \{x. \ P \ x\} = \{f \ x \mid x. \ P \ x\}$
— NOT suitable as a default simp rule: the RHS isn’t simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational properties than does the RHS.

  by blast

**lemma** if-image-distrib [simp]:

$(\lambda x. \text{if } P \ x \text{ then } f \ x \text{ else } g \ x) \cdot S = f \cdot (S \cap \{x. \ P \ x\}) \cup g \cdot (S \cap \{x. \neg P \ x\})$

  by auto

**lemma** image-cong:

$f \cdot M = g \cdot N \text{ if } M = N \land x. \ x \in N \implies f \ x = g \ x$

  using that by (simp add: image-def)

**lemma** image-cong-simp [cong]:

$f \cdot M = g \cdot N \text{ if } M = N \land x. \ x \in N = \text{simp} \implies f \ x = g \ x$

  using that image-cong [of $M \ N \ f \ g$] by (simp add: simp-implies-def)

**lemma** image-Int-subset: $f \cdot (A \cap B) \subseteq f \cdot A \cap f \cdot B$

  by blast

**lemma** image-diff-subset: $f \cdot A - f \cdot B \subseteq f \cdot (A - B)$

  by blast

**lemma** Setcompr-eq-image: \[
\{f \ x \mid x. \ x \in A\} = f \cdot A
\]

  by blast

**lemma** setcompr-eq-image: \[
\{f \ x \mid x. \ P \ x\} = f \cdot \{x. \ P \ x\}
\]

  by auto

**lemma** ball-imageD: \[\forall x \in f \cdot A. \ P x \implies \forall x \in A. \ P (f x)\]

  by simp

**lemma** bex-imageD: \[\exists x \in f \cdot A. \ P x \implies \exists x \in A. \ P (f x)\]

  by auto

**lemma** image-add-0 [simp]: \[(+) \ (0::_a::comm-monoid-add) \cdot S = S\]

  by auto

**theorem** Cantors-theorem: \[\#f. \ f \cdot A = \text{Pow } A\]

**proof**

  assume \(\exists f. \ f \cdot A = \text{Pow } A\)

  then obtain $f$ where $f \cdot A = \text{Pow } A$ \ldots

  let \(?X = \{a \in A. \ a \notin f \ a\}\)

  have \(?X \in \text{Pow } A\) by blast

  then have \(?X \in f \cdot A\) by (simp only: $f$)

  then obtain $x$ where $x \in A$ and $f x = ?X$ by blast

  then show False by blast

qed
Range of a function – just an abbreviation for image!

**abbreviation** range :: (′a ⇒ ′b) ⇒ ′b set — of function

**where** range f ≡ f "UNIV

**lemma** range-eqI: b = f x ⇒ b ∈ range f
  by simp

**lemma** rangeI: f x ∈ range f
  by simp

**lemma** rangeE [elim?]: b ∈ range (λx. f x) ⇒ (⋀x. b = f x ⇒ P) ⇒ P
  by (rule imageE)

**lemma** range-subsetD: range f ⊆ B ⇒ f i ∈ B
  by blast

**lemma** full-SetCompr-eq: {u. ∃x. u = f x} = range f
  by auto

**lemma** range-composition: range (λx. f (g x)) = f " range g
  by auto

**lemma** range-constant [simp]: range (λ-. x) = {x}
  by (simp add: image-constant)

**lemma** range-eq-singletonD: range f = {a} ⇒ f x = a
  by auto

### 8.3.13 Some rules with if

Elimination of \{x. \ldots \land x = t \land \ldots\}.

**lemma** Collect-conv-if: \{x. x = a \land P x\} = (if P a then \{a\} else \{\})
  by auto

**lemma** Collect-conv-if2: \{x. a = x \land P x\} = (if P a then \{a\} else \{\})
  by auto

Rewrite rules for boolean case-splitting: faster than if-split [split].

**lemma** if-split-eq1: (if Q then x else y) = b ⇔ (Q → x = b) ∧ (∼Q → y = b)
  by (rule if-split)

**lemma** if-split-eq2: a = (if Q then x else y) ⇔ (Q → a = x) ∧ (∼Q → a = y)
  by (rule if-split)

Split ifs on either side of the membership relation. Not for [simp] – can cause goals to blow up!
THEORY “Set” 185

lemma if-split-mem1: (if $Q$ then $x$ else $y$) ∈ $b$ ←→ $(Q \rightarrow x \in b) \land (\neg Q \rightarrow y \in b)$
by (rule if-split)

lemma if-split-mem2: ($a \in (if \ Q \ then \ x \ else \ y)$) ←→ $(Q \rightarrow a \in x) \land (\neg Q \rightarrow a \in y)$
by (rule if-split [where $P = \lambda S. \ a \in S$])

lemmas split-ifs = if-bool-eq-conj if-split-eq1 if-split-eq2 if-split-mem1 if-split-mem2

8.4 Further operations and lemmas

8.4.1 The “proper subset” relation

lemma psubsetI [intro!]: $A \subseteq B = \Rightarrow A \neq B = \Rightarrow A \subset B$
unfolding less-le by blast

lemma psubsetE [elim!]: $A \subset B = \Rightarrow (A \subseteq B = \Rightarrow \neg B \subseteq A = \Rightarrow R) = \Rightarrow R$
unfolding less-le by blast

lemma psubset-insert-iff: $A \subset insert x B = \iff (if \ x \in B \ then \ A \subset B \ else \ if \ x \in A \ then \ A - \{x\} \subset B \ else \ A \subseteq B)$
by (auto simp add: less-le subset-insert-iff)

lemma psubset-eq: $A \subset B = \iff A \subseteq B \land A \neq B$
by (simp only: less-le)

lemma psubset-imp-subset: $A \subset B = \Rightarrow A \subseteq B$
by (simp add: psubset-eq)

lemma psubset-trans: $A \subset B = \Rightarrow B \subseteq C = \Rightarrow A \subseteq C$
unfolding less-le by (auto dest: subset-antisym)

lemma psubsetD: $A \subset B = \Rightarrow c \in A = \Rightarrow c \in B$
unfolding less-le by (auto dest: subsetD)

lemma psubset-subset-trans: $A \subset B = \Rightarrow B \subseteq C = \Rightarrow A \subset C$
by (auto simp add: psubset-eq)

lemma subset-psubset-trans: $A \subseteq B = \Rightarrow B \subseteq C = \Rightarrow A \subset C$
by (auto simp add: psubset-eq)

lemma psubset-imp-ex-mem: $A \subset B = \Rightarrow \exists b. b \in B - A$
unfolding less-le by blast

lemma atomize-ball: $(\forall x. \ x \in A = \Rightarrow P \ x) = \text{Trueprop} (\forall x \in A. \ P \ x)$
by (simp only: Ball-def atomize-all atomize-imp)

lemmas [symmetric, rulify] = atomize-ball
and \[\text{defn} = \text{atomize-ball}\]

\textbf{lemma} \text{image-Pow-mono}: \( f \upharpoonright A \subseteq B \implies \text{image } f \upharpoonright \text{Pow } A \subseteq \text{Pow } B \)
by \text{blast}

\textbf{lemma} \text{image-Pow-surj}: \( f \upharpoonright A = B \implies \text{image } f \upharpoonright \text{Pow } A = \text{Pow } B \)
by (\text{blast elim: subset-imageE})

\textbf{8.4.2 Derived rules involving subsets.}

\textit{insert.}

\textbf{lemma} \text{subset-insertI}: \( B \subseteq \text{insert } a \ B \)
by (\text{rule subsetI}) (erule insertI2)

\textbf{lemma} \text{subset-insertI2}: \( A \subseteq B \implies A \subseteq \text{insert } a \ B \)
by \text{blast}

\textbf{lemma} \text{subset-insert}: \( x \notin A \implies A \subseteq \text{insert } x \ B \iff A \subseteq B \)
by \text{blast}

\textbf{Finite Union – the least upper bound of two sets.}

\textbf{lemma} \text{Un-upper1}: \( A \subseteq A \cup B \)
by (\text{fact sup-ge1})

\textbf{lemma} \text{Un-upper2}: \( B \subseteq A \cup B \)
by (\text{fact sup-ge2})

\textbf{lemma} \text{Un-least}: \( A \subseteq C \implies B \subseteq C \implies A \cup B \subseteq C \)
by (\text{fact sup-least})

\textbf{Finite Intersection – the greatest lower bound of two sets.}

\textbf{lemma} \text{Int-lower1}: \( A \cap B \subseteq A \)
by (\text{fact inf-le1})

\textbf{lemma} \text{Int-lower2}: \( A \cap B \subseteq B \)
by (\text{fact inf-le2})

\textbf{lemma} \text{Int-greatest}: \( C \subseteq A \implies C \subseteq B \implies C \subseteq A \cap B \)
by (\text{fact inf-greatest})

\textbf{Set difference.}

\textbf{lemma} \text{Diff-subset[simp]}: \( A - B \subseteq A \)
by \text{blast}

\textbf{lemma} \text{Diff-subset-conv}: \( A - B \subseteq C \iff A \subseteq B \cup C \)
by \text{blast}
8.4.3 Equalities involving union, intersection, inclusion, etc.

\{
\}

**lemma** *Collect-const* [simp]: \( \{ s. \ P \} = (if P \ then \ UNIV \ else \ \{\}) \)

— supersedes *Collect-False-empty*

by *auto*

**lemma** *subset-empty* [simp]: \( A \subseteq \{\} \iff A = \{\} \)

by (fact *bot-unique*)

**lemma** *not-psubset-empty* [iff]: \( \sim (A < \{\}) \)

by (fact *not-less-bot*)

**lemma** *Collect-subset* [simp]: \( \{ x \in A. \ P x \} \subseteq A \)

by *auto*

**lemma** *Collect-empty-eq* [simp]: \( \text{Collect} P = \{\} \iff (\forall x. \sim P x) \)

by *blast*

**lemma** *empty-Collect-eq* [simp]: \( \{\} = \text{Collect} P \iff (\forall x. \sim P x) \)

by *blast*

**lemma** *Collect-neg-eq*:
\[
\{ x. \sim P x \} = \sim \{ x. P x \}
\]

by *blast*

**lemma** *Collect-disj-eq*:
\[
\{ x. P x \lor Q x \} = \{ x. P x \} \cup \{ x. Q x \}
\]

by *blast*

**lemma** *Collect-imp-eq*:
\[
\{ x. P x \rightarrow Q x \} = \sim \{ x. P x \} \cup \{ x. Q x \}
\]

by *blast*

**lemma** *Collect-conj-eq*:
\[
\{ x. P x \land Q x \} = \{ x. P x \} \cap \{ x. Q x \}
\]

by *blast*

**lemma** *Collect-mono-iff*:
\[
\text{Collect} P \subseteq \text{Collect} Q \iff (\forall x. P x \rightarrow Q x)
\]

by *blast*

**insert.**

**lemma** *insert-is-Un*: *insert* \( a \) \( A = \{a\} \cup A \)

— NOT SUITABLE FOR REWRITING since \( \{a\} \equiv \text{insert} \ a \ \{\} \)

by *blast*

**lemma** *insert-not-empty* [simp]: *insert* \( a \) \( A \neq \{\} \)

and *empty-not-insert* [simp]: \( \{\} \neq \text{insert} \ a \ A \)

by *blast*

**lemma** *insert-absorb*: \( a \in A \Rightarrow \text{insert} \ a \ A = A \)

— [simp] causes recursive calls when there are nested inserts

— with quadratic running time

by *blast*
THEORY “Set”

lemma insert-absorb2 [simp]: insert x (insert x A) = insert x A
  by blast

lemma insert-commute: insert x (insert y A) = insert y (insert x A)
  by blast

lemma insert-subset [simp]: insert x A ⊆ B ←→ x ∈ B ∧ A ⊆ B
  by blast

lemma mk-disjoint-insert: a ∈ A ⇒ ∃ B. A = insert a B ∧ a /∈ B
  — use new B rather than A − {a} to avoid infinite unfolding
  by (rule exI [where x = A − {a}]) blast

lemma insert-Collect: insert a (Collect P) = {u. u ≠ a → P u}
  by auto

lemma insert-inter-insert [simp]: insert a A ∩ insert a B = insert a (A ∩ B)
  by blast

lemma insert-disjoint [simp]:
  insert a A ∩ B = {}
  {} = insert a A ∩ B = {}
  by auto

lemma disjoint-insert [simp]:
  B ∩ insert a A = {}
  {} = A ∩ insert b B
  by auto

Int

lemma Int-absorb: A ∩ A = A
  by (fact inf-idem)

lemma Int-left-absorb: A ∩ (A ∩ B) = A ∩ B
  by (fact inf-left-idem)

lemma Int-commute: A ∩ B = B ∩ A
  by (fact inf-commute)

lemma Int-left-commute: A ∩ (B ∩ C) = B ∩ (A ∩ C)
  by (fact inf-left-commute)

lemma Int-assoc: (A ∩ B) ∩ C = A ∩ (B ∩ C)
  by (fact inf-assoc)

lemmas Int-ac = Int-assoc Int-left-absorb Int-commute Int-left-commute
  — Intersection is an AC-operator
lemma Int-absorb1: $B \subseteq A \Longrightarrow A \cap B = B$
  by (fact inf-absorb2)

lemma Int-absorb2: $A \subseteq B \Longrightarrow A \cap B = A$
  by (fact inf-absorb1)

lemma Int-empty-left: $\{\} \cap B = \{\}$
  by (fact inf-bot-left)

lemma Int-empty-right: $A \cap \{\} = \{\}$
  by (fact inf-bot-right)

lemma disjoint-eq-subset-Compl: $A \cap B = \{\} \iff A \subseteq -B$
  by blast

lemma disjoint-iff: $A \cap B = \{\} \iff (\forall x. x \in A \rightarrow x \notin B)$
  by blast

lemma disjoint-iff-not-equal: $A \cap B = \{\} \iff (\forall x \in A. \forall y \in B. x \neq y)$
  by blast

lemma Int-UNIV-left: $\text{UNIV} \cap B = B$
  by (fact inf-top-left)

lemma Int-UNIV-right: $A \cap \text{UNIV} = A$
  by (fact inf-top-right)

lemma Int-Un-distrib: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
  by (fact inf-sup-distrib1)

lemma Int-Un-distrib2: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
  by (fact inf-sup-distrib2)

lemma Int-UNIV: $A \cap B = \text{UNIV} \iff A = \text{UNIV} \land B = \text{UNIV}$
  by (fact inf-eq-top-iff)

lemma Int-subset-iff: $C \subseteq A \cap B \iff C \subseteq A \land C \subseteq B$
  by (fact le-inf-iff)

lemma Int-Collect: $x \in A \cap \{x. P x\} \iff x \in A \land P x$
  by blast

lemma Un-absorb: $A \cup A = A$
  by (fact sup-idem)

lemma Un-left-absorb: $A \cup (A \cup B) = A \cup B$
  by (fact sup-left-idem)
lemma Un-commute: \( A \cup B = B \cup A \)
by (fact sup-commute)

lemma Un-left-commute: \( A \cup (B \cup C) = B \cup (A \cup C) \)
by (fact sup-left-commute)

lemma Un-assoc: \( (A \cup B) \cup C = A \cup (B \cup C) \)
by (fact sup-assoc)

lemmas Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute
— Union is an AC-operator

lemma Un-absorb1: \( A \subseteq B \Longrightarrow A \cup B = B \)
by (fact sup-absorb2)

lemma Un-absorb2: \( B \subseteq A \Longrightarrow A \cup B = A \)
by (fact sup-absorb1)

lemma Un-empty-left: \( \{\} \cup B = B \)
by (fact sup-bot-left)

lemma Un-empty-right: \( A \cup \{\} = A \)
by (fact sup-bot-right)

lemma Un-UNIV-left: \( \text{UNIV} \cup B = \text{UNIV} \)
by (fact sup-top-left)

lemma Un-UNIV-right: \( A \cup \text{UNIV} = \text{UNIV} \)
by (fact sup-top-right)

lemma Un-insert-left [simp]: \((\text{insert} \ a \ B) \cup C = \text{insert} \ a \ (B \cup C)\)
by blast

lemma Un-insert-right [simp]: \( A \cup (\text{insert} \ a \ B) = \text{insert} \ a \ (A \cup B) \)
by blast

lemma Int-insert-left: \( (\text{insert} \ a \ B) \cap C = (\text{if} \ a \in C \text{ then} \text{insert} \ a \ (B \cap C) \text{ else} \ B \cap C) \)
by auto

lemma Int-insert-left-if0 [simp]: \( a \notin C \Longrightarrow (\text{insert} \ a \ B) \cap C = B \cap C \)
by auto

lemma Int-insert-left-if1 [simp]: \( a \in C \Longrightarrow (\text{insert} \ a \ B) \cap C = \text{insert} \ a \ (B \cap C) \)
by auto

lemma Int-insert-right: \( A \cap (\text{insert} \ a \ B) = (\text{if} \ a \in A \text{ then} \text{insert} \ a \ (A \cap B) \text{ else} \ A \cap B) \)
by auto
lemma Int-insert-right-if0 [simp]: \( a \notin A \implies A \cap (\text{insert } a \ B) = A \cap B \)
by auto

lemma Int-insert-right-if1 [simp]: \( a \in A \implies A \cap (\text{insert } a \ B) = \text{insert } a \ (A \cap B) \)
by auto

lemma Un-Int-distrib: \( A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \)
by (fact sup-inf-distrib1)

lemma Un-Int-distrib2: \( (B \cap C) \cup A = (B \cup A) \cap (C \cup A) \)
by (fact sup-inf-distrib2)

lemma Un-Int-crazy: \( (A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A) \)
by blast

lemma subset-Un-eq: \( A \subseteq B \iff A \cup B = B \)
by (fact le-iff-sup)

lemma Un-empty [iff]: \( A \cup B = \{} \iff A = \{} \land B = \{} \)
by (fact sup-eq-bot-iff)

lemma Un-subset-iff: \( A \cup B \subseteq C \iff A \subseteq C \land B \subseteq C \)
by (fact le-sup-iff)

lemma Un-Diff-Int: \( (A - B) \cup (A \cap B) = A \)
by blast

lemma Diff-Int2: \( A \cap C - B \cap C = A \cap C - B \)
by blast

lemma subset-UnE:
assumes \( C \subseteq A \cup B \)
obtains \( A' \ B' \) where \( A' \subseteq A \ B' \subseteq B \ C = A' \cup B' \)
proof
show \( C \cap A \subseteq A \ C \cap B \subseteq B \ C = (C \cap A) \cup (C \cap B) \)
using assms by blast+
qed

lemma Un-Int-eq [simp]: \( (S \cup T) \cap S = S \) \( S \cup (S \cup T) = S \ T \cap (S \cup T) = T \)
by auto

lemma Int-Un-eq [simp]: \( (S \cap T) \cup S = S \) \( S \cap (S \cap T) \cup T = T \ S \cup (S \cap T) = S \)
by auto
Set complement

**lemma** Compl-disjoint [simp]: $A \cap -A = \{\}$
by (fact inf-compl-bot)

**lemma** Compl-disjoint2 [simp]: $-A \cap A = \{\}$
by (fact compl-inf-bot)

**lemma** Compl-partition: $A \cup -A = \text{UNIV}$
by (fact sup-compl-top)

**lemma** Compl-partition2: $-A \cup A = \text{UNIV}$
by (fact compl-sup-top)

**lemma** double-complement: $-(-A) = A$ for $A :: 'a set$
by (fact double-compl)

**lemma** Compl-Un: $-(A \cup B) = (-A) \cap (-B)$
by (fact compl-sup)

**lemma** Compl-Int: $-(A \cap B) = (-A) \cup (-B)$
by (fact compl-inf)

**lemma** subset-Compl-self-eq: $A \subseteq -A \iff A = \{\}$
by blast

**lemma** Un-Int-assoc-eq: $(A \cap B) \cup C = A \cap (B \cup C) \iff C \subseteq A$
— Halmos, Naive Set Theory, page 16.
by blast

**lemma** Compl-UNIV-eq: $-\text{UNIV} = \{\}$
by (fact compl-top-eq)

**lemma** Compl-empty-eq: $\{\} = \text{UNIV}$
by (fact compl-bot-eq)

**lemma** Compl-subset-Compl-iff [iff]: $A \subseteq -B \iff B \subseteq A$
by (fact compl-le-compl-iff)

**lemma** Compl-eq-Compl-iff [iff]: $-A = -B \iff A = B$
for $A B :: 'a set$
by (fact compl-eq-compl-iff)

**lemma** Compl-insert: $-\text{insert } x A = (-A) - \{x\}$
by blast

Bounded quantifiers.
The following are not added to the default simpset because (a) they duplicate
the body and (b) there are no similar rules for Int.
lemma ball-Un: \((\forall x \in A \cup B. \; P x) \leftrightarrow (\forall x \in A. \; P x) \land (\forall x \in B. \; P x)\)
by blast

lemma bex-Un: \((\exists x \in A \cup B. \; P x) \leftrightarrow (\exists x \in A. \; P x) \lor (\exists x \in B. \; P x)\)
by blast

Set difference.

lemma Diff-eq: \(A - B = A \cap (\neg B)\)
by (rule boolean-algebra-class.diff-eq)

lemma Diff-eq-empty-iff: \(A - B = \{\} \iff A \subseteq B\)
by (rule boolean-algebra-class.diff-shunt-var)

lemma Diff-cancel [simp]: \(A - A = \{\}\)
by blast

lemma Diff-idemp [simp]: \((A - B) - B = A - B\)
for \(A B :: 'a set\)
by blast

lemma Diff-triv: \(A \cap B = \{\} \implies A - B = A\)
by (blast elim: equalityE)

lemma empty-Diff [simp]: \(\{\} - A = \{\}\)
by blast

lemma Diff-empty [simp]: \(A - \{\} = A\)
by blast

lemma Diff-UNIV [simp]: \(A - UNIV = \{\}\)
by blast

lemma Diff-insert0 [simp]: \(x \notin A \implies A - insert x B = A - B\)
by blast

lemma Diff-insert: \(A - insert a B = A - B - \{a\}\)
— NOT SUITABLE FOR REWRITING since \(\{a\} \equiv insert a 0\)
by blast

lemma Diff-insert2: \(A - insert a B = A - \{a\} - B\)
— NOT SUITABLE FOR REWRITING since \(\{a\} \equiv insert a 0\)
by blast

lemma insert-Diff-if: \(insert x A - B = (if x \in B then A - B else insert x (A - B))\)
by auto

lemma insert-Diff1 [simp]: \(x \in B \implies insert x A - B = A - B\)
by blast
lemma insert-Diff-single[simp]: insert a \((A - \{a\})\) = insert a A
  by blast

lemma insert-Diff: \(a \in A \implies insert a (A - \{a\}) = A\)
  by blast

lemma Diff-insert-absorb: \(x \notin A \implies (insert x A) - \{x\} = A\)
  by blast

lemma Diff-disjoint [simp]: \(A \cap (B - A) = \{}\)
  by blast

lemma Diff-partition: \(A \subseteq B \implies A \cup (B - A) = B\)
  by blast

lemma double-diff: \(A \subseteq B \implies B \subseteq C \implies B - (C - A) = A\)
  by blast

lemma Un-Diff-cancel [simp]: \(A \cup (B - A) = A \cup B\)
  by blast

lemma Un-Diff-cancel2 [simp]: \((B - A) \cup A = B \cup A\)
  by blast

lemma Diff-Un: \(A - (B \cup C) = (A - B) \cap (A - C)\)
  by blast

lemma Diff-Int: \(A - (B \cap C) = (A - B) \cup (A - C)\)
  by blast

lemma Diff-Diff-Int: \(A - (A - B) = A \cap B\)
  by blast

lemma Un-Diff: \((A \cup B) - C = (A - C) \cup (B - C)\)
  by blast

lemma Int-Diff: \((A \cap B) - C = A \cap (B - C)\)
  by blast

lemma Diff-Int-distrib: \(C \cap (A - B) = (C \cap A) - (C \cap B)\)
  by blast

lemma Diff-Int-distrib2: \((A - B) \cap C = (A \cap C) - (B \cap C)\)
  by blast

lemma Diff-Compl [simp]: \(A - (-B) = A \cap B\)
  by auto
lemma Compl-Diff-eq [simp]: \( - (A - B) = - A \cup B \)
by blast

lemma subset-Compl-singleton [simp]: \( A \subseteq \{ b \} \iff b \notin A \)
by blast

Quantification over type bool.

lemma bool-induct: \( P True \implies P False \implies P x \)
by (cases x) auto

lemma all BOOL-eq: \( (\forall b. P b) \iff P True \land P False \)
by (auto intro: bool-induct)

lemma bool-contrapos: \( P x \implies \neg P False \implies P True \)
by (cases x) auto

lemma ex-BOOL-eq: \( (\exists b. P b) \iff P True \lor P False \)
by (auto intro: bool-contrapos)

lemma UNIV-BOOL: \( UNIV = \{ \text{False}, \text{True} \} \)
by (auto intro: bool-induct)

Pow

lemma Pow-empty [simp]: \( Pow \{ \} = \{ \{ \} \} \)
by (auto simp add: Pow-def)

lemma Pow-singleton iff [simp]: \( Pow X = \{ Y \} \iff X = \{ \} \land Y = \{ \} \)
by blast

lemma Pow-insert: \( Pow (insert a A) = Pow A \cup (insert a \setminus Pow A) \)
by (blast intro: image-eqI [where ?x = u \setminus \{ a \} for u])

lemma Pow-Compl: \( Pow (- A) = \{ - B \mid B, A \in Pow B \} \)
by (blast intro: exI [where \( ?x = - u \) for u])

lemma Pow-UNIV [simp]: \( Pow UNIV = UNIV \)
by blast

lemma Un-Pow-subset: \( Pow A \cup Pow B \subseteq Pow (A \cup B) \)
by blast

lemma Pow-Int-eq [simp]: \( Pow (A \cap B) = Pow A \cap Pow B \)
by blast

Miscellany.

lemma Int-Diff-disjoint: \( A \cap B \cap (A - B) = \{ \} \)
by blast
lemma Int-Diff-Un: $A \cap B \cup (A - B) = A$
  by blast

lemma set-eq-subset: $A = B \iff A \subseteq B \land B \subseteq A$
  by blast

lemma subset-iff: $A \subseteq B \iff (\forall t. t \in A \longrightarrow t \in B)$
  by blast

lemma subset-iff-psubset-eq: $A \subseteq B \iff A \subset B \lor A = B$
  unfolding less-le by blast

lemma all-not-in-conv [simp]: $(\forall x. x \notin A) \iff A = \{\}$
  by blast

lemma ex-in-conv: $(\exists x. x \in A) \iff A \neq \{\}$
  by blast

lemma ball-simps [simp, no-atp]:
  $\forall A \forall P. (\forall x \in A. P \lor Q) \iff (\forall x \in A. P) \lor Q$
  $\forall A \forall P. (\forall x \in A. P \land Q x) \iff (P \land (\forall x \in A. Q x))$
  $\forall A \forall P. (\forall x \in A. P \longrightarrow Q x) \iff (P \longrightarrow (\forall x \in A. Q x))$
  $\forall A \forall P. (\forall x \in A. P x \longrightarrow Q) \iff ((\exists x \in A. P x) \longrightarrow Q)$
  $\forall P. (\forall x \in \text{UNIV}. P x) \iff (\forall x. P x)$
  $\forall a B P. (\forall x \in \text{insert} a B. P x) \iff (P a \land (\forall x \in B. P x))$
  $\forall P Q. (\forall x \in \text{Collect} Q. P x) \iff (\forall x. Q x \longrightarrow P x)$
  $\forall A P f. (\forall x \in f'A. P x) \iff (\forall x \in A. P (f x))$
  $\forall A P. (\neg (\forall x \in A. P x)) \iff (\exists x \in A. \neg P x)$
  by auto

lemma bex-simps [simp, no-atp]:
  $\forall A \forall Q. (\exists x \in A. P x \land Q) \iff (\exists x \in A. P x) \land Q$
  $\forall A \forall P. (\exists x \in A. P x \land Q x) \iff (P \land (\exists x \in A. Q x))$
  $\forall P. (\exists x \in \text{UNIV}. P x) \iff (\exists x. P x)$
  $\forall a B P. (\exists x \in \text{insert} a B. P x) \iff (P a \lor (\exists x \in B. P x))$
  $\forall P Q. (\exists x \in \text{Collect} Q. P x) \iff (\exists x. Q x \land P x)$
  $\forall A P f. (\exists x \in f'A. P x) \iff (\exists x \in A. P (f x))$
  $\forall A P. (\neg (\exists x \in A. P x)) \iff (\forall x \in A. \neg P x)$
  by auto

lemma ex-image-cong-iff [simp, no-atp]:
  $(\exists x. x \in f'A) \iff A \neq \{\} \ (\exists x. x \in f'A \land P x) \iff (\exists x \in A. P (f x))$
  by auto
8.4.4 Monotonicity of various operations

**lemma** image-mono: \( A \subseteq B \implies f \circ A \subseteq f \circ B \)
by blast

**lemma** Pow-mono: \( A \subseteq B \implies \text{Pow} A \subseteq \text{Pow} B \)
by blast

**lemma** insert-mono: \( C \subseteq D \implies \text{insert} a C \subseteq \text{insert} a D \)
by blast

**lemma** Un-mono: \( A \subseteq C \implies B \subseteq D \implies A \cup B \subseteq C \cup D \)
by (fact sup-mono)

**lemma** Int-mono: \( A \subseteq C \implies B \subseteq D \implies A \cap B \subseteq C \cap D \)
by (fact inf-mono)

**lemma** Diff-mono: \( A \subseteq C \implies D \subseteq B \implies A - B \subseteq C - D \)
by blast

**lemma** Compl-anti-mono: \( A \subseteq B \implies -B \subseteq -A \)
by (fact compl-mono)

Monotonicity of implications.

**lemma** in-mono: \( A \subseteq B \implies x \in A \implies x \in B \)
by (rule impI) (erule subsetD)

**lemma** conj-mono: \( P1 \implies Q1 \implies P2 \implies Q2 \implies (P1 \land P2) \implies (Q1 \land Q2) \)
by iprover

**lemma** disj-mono: \( P1 \implies Q1 \implies P2 \implies Q2 \implies (P1 \lor P2) \implies (Q1 \lor Q2) \)
by iprover

**lemma** imp-mono: \( Q1 \implies P1 \implies P2 \implies Q2 \implies (P1 \implies P2) \implies (Q1 \implies Q2) \)
by iprover

**lemma** imp-refl: \( P \implies P \)

**lemma** not-mono: \( Q \implies P \implies \neg P \implies \neg Q \)
by iprover

**lemma** ex-mono: \( (\forall x. P x \implies Q x) \implies (\exists x. P x) \implies (\exists x. Q x) \)
by iprover

**lemma** all-mono: \( (\forall x. P x \implies Q x) \implies (\forall x. P x) \implies (\forall x. Q x) \)
by iprover

**lemma** Collect-mono: \( (\forall x. P x \implies Q x) \implies \text{Collect } P \subseteq \text{Collect } Q \)
THEORY “Set”

by blast

lemma Int-Collect-mono: $A \subseteq B \Longrightarrow (\forall x. x \in A \Longrightarrow P x \longrightarrow Q x) \Longrightarrow A \cap \text{Collect } P \subseteq B \cap \text{Collect } Q$
  by blast

lemmas basic-monomes =
  subset-refl imp-refl disj-mono conj-mono ex-mono Collect-mono in-mono

lemma eq-to-mono: $a = b \Longrightarrow c = d \Longrightarrow b \rightarrow d \Longrightarrow a \rightarrow c$
  by iprover

8.4.5 Inverse image of a function

definition vimage :: '('a ⇒ 'b) ⇒ 'b set ⇒ 'a set (infixr – 90)
  where $f –^’ B \equiv \{ x. f x \in B\}$

lemma vimage-eq [simp]: $a \in f –^’ B \iff f a \in B$
  unfolding vimage-def by blast

lemma vimage-singleton-eq: $a \in f –^’ \{ b\} \iff f a = b$
  by simp

lemma vimageI [intro]: $f a = b \Longrightarrow b \in B \Longrightarrow a \in f –^’ B$
  unfolding vimage-def by blast

lemma vimageI2: $f a \in A \Longrightarrow a \in f –^’ A$
  unfolding vimage-def by fast

lemma vimageE [elim!]: $a \in f –^’ B \Longrightarrow (\forall x. f a = x \Longrightarrow x \in B \Longrightarrow P) \Longrightarrow P$
  unfolding vimage-def by blast

lemma vimageD: $a \in f –^’ A \Longrightarrow f a \in A$
  unfolding vimage-def by fast

lemma vimage-empty [simp]: $f –^’ \{\} = \{\}$
  by blast

lemma vimage-Compl: $f –^’ (– A) = – (f –^’ A)$
  by blast

lemma vimage-Un [simp]: $f –^’ (A \cup B) = (f –^’ A) \cup (f –^’ B)$
  by blast

lemma vimage-Int [simp]: $f –^’ (A \cap B) = (f –^’ A) \cap (f –^’ B)$
  by fast

lemma vimage-Collect-eq [simp]: $f –^’ \text{Collect } P = \{ y. P (f y)\}$
  by blast
lemma vimage-Collect: (∀x. P (f x) = Q x) ⇒ f −' (Collect P) = Collect Q
  by blast

lemma vimage-insert: f −' (insert a B) = (f −' {a}) ∪ (f −' B)
  — NOT suitable for rewriting because of the recurrence of {a}.
  by blast

lemma vimage-Diff: f −' (A − B) = (f −' A) − (f −' B)
  by blast

lemma vimage-UNIV [simp]: f −' UNIV = UNIV
  by blast

lemma vimage-mono: A ⊆ B ⇒ f −' A ⊆ f −' B
  — monotonicity
  by blast

lemma vimage-image-eq: f −' (f ' A) = {y. ∃x ∈ A. f x = f y}
  by (blast intro: sym)

lemma image-vimage-subset: f ' (f −' A) ⊆ A
  by blast

lemma image-vimage-eq [simp]: f ' (f −' A) = A ∩ range f
  by blast

lemma image-subset-iff-subset-vimage: f ' A ⊆ B ⇔ A ⊆ f −' B
  by blast

lemma subset-vimage-iff: A ⊆ f −' B ⇔ (∀x ∈ A. f x ∈ B)
  by auto

lemma vimage-const [simp]: ((λx. c) −' A) = (if c ∈ A then UNIV else {})
  by auto

lemma vimage-if [simp]: ((λx. if x ∈ B then c else d) −' A) =
  (if c ∈ A then (if d ∈ A then UNIV else B)
   else if d ∈ A then − B else {})
  by (auto simp add: vimage-def)

lemma vimage-inter-cong: (∀w. w ∈ S ⇒ f w = g w) ⇒ f −' y ∩ S = g −' y ∩ S
  by auto

lemma vimage-ident [simp]: (λx. x) −' Y = Y
  by blast
8.4.6 Singleton sets

definition is-singleton :: "'a set ⇒ bool"
  where is-singleton A ⇔ (∃x. A = {x})

lemma is-singletonI [simp, intro!]: is-singleton {x}
  unfolding is-singleton-def by simp

lemma is-singletonI': A ≠ {} ⇒ (∀x y. x ∈ A ⇒ y ∈ A ⇒ x = y) ⇒
  is-singleton A
  unfolding is-singleton-def by blast

lemma is-singletonE: is-singleton A ⇒ (∀x. A = {x} ⇒ P) ⇒ P
  unfolding is-singleton-def by blast

8.4.7 Getting the contents of a singleton set

definition the-elem :: "'a set ⇒ 'a"
  where the-elem X = (THE x. X = {x})

lemma the-elem-eq [simp]: the-elem {x} = x
  by (simp add: the-elem-def)

lemma is-singleton-the-elem: is-singleton A ⇔ A = {the-elem A}
  by (auto simp: is-singleton-def)

lemma the-elem-image-unique:
  assumes A ≠ {} ∧ (∀y. y ∈ A ⇒ f y = f x)
  shows the-elem (f ' A) = f x
  unfolding the-elem-def
  proof (rule the1-equivalence)
    from ‹A ≠ {}› obtain y where y ∈ A by auto
    with * have f x = f y by simp
    with * have f x ∈ f ' A by blast
    with * show f ' A = {f x} by auto
    then show ∃!x. f ' A = {x} by auto
  qed

8.4.8 Monad operation

definition bind :: "'a set ⇒ ('a ⇒ 'b set) ⇒ 'b set"
  where bind A f = {x. ∃B ∈ f ' A. x ∈ B}

hide-const (open) bind

lemma bind-bind: Set.bind (Set.bind A B) C = Set.bind A (λx. Set.bind (B x) C)
  for A :: "'a set"
  by (auto simp: bind-def)
lemma empty-bind [simp]: \( \text{Set.bind} \{ \} \ f = \{ \} \)
  by (simp add: bind-def)

lemma nonempty-bind-const: \( A \neq \{ \} \) \implies \text{Set.bind} \( A \ (\lambda \cdot \ B) \) = \( B \)
  by (auto simp: bind-def)

lemma bind-const: \( \text{Set.bind} \ A \ (\lambda \cdot \ B) \) = \( \text{if} \ A = \{ \} \ \text{then} \ \{ \} \ \text{else} \ B \)
  by (auto simp: bind-def)

lemma bind-singleton-conv-image: \( \text{Set.bind} \ A \ (\lambda \cdot \ \{ f x \}) \) = \( f \mapsto A \)
  by (auto simp: bind-def)

8.4.9 Operations for execution

definition is-empty :: \('a\ \text{set}\) \Rightarrow \text{bool}
  where [code-abbrev]: is-empty \( A \) \longleftrightarrow \( A = \{ \} \)

hide-const (open) is-empty

definition remove :: \('a\ \Rightarrow \ 'a\ \text{set}\) \Rightarrow \text{bool}
  where [code-abbrev]: remove \( x \) \( A \) = \( A - \{ x \} \)

hide-const (open) remove

lemma member-remove [simp]: \( x \in \text{Set.remove} \ y \ A \longleftrightarrow x \in A \land x \neq y \)
  by (simp add: remove-def)

definition filter :: \('a \Rightarrow \text{bool}\) \Rightarrow \text{bool}
  where [code-abbrev]: filter \( P \) \( A \) = \( \{ a \in A. \ P a \} \)

hide-const (open) filter

lemma member-filter [simp]: \( x \in \text{Set.filter} \ P \ A \longleftrightarrow x \in A \land P x \)
  by (simp add: filter-def)

instantiation set :: (equal) equal
begin

definition HOL.equal \( A \ B \longleftrightarrow A \subseteq B \land B \subseteq A \)

instance by standard (auto simp add: equal-set-def)

end

Misc

definition pairwise :: \('a \Rightarrow \text{bool}\) \Rightarrow \text{bool}
  where pairwise \( R \ S \longleftrightarrow (\forall x \in S. \forall y \in S. \ x \neq y \rightarrow R \ x \ y) \)
lemma pairwise-alt: pairwise R S \iff (\forall x \in S. \forall y \in S - \{x\}. \ R x y)
by (auto simp add: pairwise-def)

lemma pairwise-trivial [simp]: pairwise (\lambda i. j \neq i) I
by (auto simp: pairwise-def)

lemma pairwiseI [intro?]:
pairwise R S if \forall x y. x \in S \implies y \in S \implies x \neq y \implies R x y
using that by (simp add: pairwise-def)

lemma pairwiseD:
R x y and R y x
if pairwise R S x \in S and y \in S and x \neq y
using that by (simp-all add: pairwise-def)

lemma pairwise-empty [simp]: pairwise P {}
by (simp add: pairwise-def)

lemma pairwise-singleton [simp]: pairwise P {A}
by (simp add: pairwise-def)

lemma pairwise-insert:
pairwise r (insert x s) \iff (\forall y. y \in s \land y \neq x \implies r x y \land y x) \land pairwise r s
by (force simp: pairwise-def)

lemma pairwise-subset: pairwise P S \implies T \subseteq S \implies pairwise P T
by (force simp: pairwise-def)

lemma pairwise-mono: [pairwise P A; \forall x y. P x y \implies Q x y; B \subseteq A] \implies pairwise Q B
by (fastforce simp: pairwise-def)

lemma pairwise-imageI:
pairwise P (f ' A)
if \forall x y. x \in A \implies y \in A \implies x \neq y \implies f x \neq f y \implies P (f x) (f y)
using that by (auto intro: pairwiseI)

lemma pairwise-image: pairwise r (f ' s) \iff pairwise (\lambda x y. (f x \neq f y) \implies r (f x) (f y)) s
by (force simp: pairwise-def)

definition disjnt :: 'a set \Rightarrow 'a set \Rightarrow bool
where disjnt A B \iff A \cap B = {}

lemma disjnt-self-iff-empty [simp]: disjnt S S \iff S = {}
by (auto simp: disjnt-def)

lemma disjnt-commute: disjnt A B = disjnt B A
by (auto simp: disjnt-def)
lemma disjnt-iff: disjnt A B \iff (\forall x. \neg (x \in A \land x \in B))
   by (force simp: disjnt-def)

lemma disjnt-sym: disjnt A B \implies disjnt B A
   using disjnt-iff by blast

lemma disjnt-empty1 [simp]: disjnt {} A and disjnt-empty2 [simp]: disjnt A {}
   by (auto simp: disjnt-def)

lemma disjnt-insert1 [simp]: disjnt (insert a X) Y \iff a \notin Y \land disjnt X Y
   by (simp add: disjnt-def)

lemma disjnt-insert2 [simp]: disjnt Y (insert a X) \iff a \notin Y \land disjnt Y X
   by (simp add: disjnt-def)

lemma disjnt-subset1: \[ (\text{disjnt } X Y; Z \subseteq X) \] \implies disjnt Z Y
   by (auto simp: disjnt-def)

lemma disjnt-subset2: \[ (\text{disjnt } X Y; Z \subseteq Y) \] \implies disjnt X Z
   by (auto simp: disjnt-def)

lemma disjnt-Un1 [simp]: disjnt (A \cup B) C \iff disjnt A C \land disjnt B C
   by (auto simp: disjnt-def)

lemma disjnt-Un2 [simp]: disjnt C (A \cup B) \iff disjnt C A \land disjnt C B
   by (auto simp: disjnt-def)

lemma disjnt-Diff1: disjnt (X - Y) (U - V) and disjnt-Diff2: disjnt (U - V) (X - Y)
   if X \subseteq V
   using that by (auto simp: disjnt-def)

lemma disjoint-image-subset: \[ \text{pairwise disjnt } A; \ \land X. X \in A \implies f X \subseteq X \] \implies pairwise disjnt (f 'A)
   unfolding disjnt-def pairwise-def by fast

lemma pairwise-disjnt-iff: pairwise disjnt A \iff (\forall x. \exists X. X \in A \land x \in X)
   by (auto simp: Uniq-def disjnt-iff pairwise-def)

lemma disjnt-insert:
   \langle \text{disjnt } (insert x M) N \rangle \iff \langle x \notin N \rangle \langle \text{disjnt } M N \rangle
   using that by (simp add: disjnt-def)

lemma Int-emptyI: \[ (\forall x. x \in A \implies x \in B \implies \text{False}) \implies A \cap B = \{\} \]
   by blast

lemma in-image-insert-iff:
   assumes \[ \land C. C \in B \implies x \notin C \]
   shows A \in insert x \cap B \iff x \in A \land A - \{x\} \in B (is ?P \iff ?Q)
proof
  assume ?P then show ?Q
  using assms by auto
next
  assume ?Q
  then have x ∈ A and A − {x} ∈ B
    by simp-all
from A − {x}∈ B have insert x (A − {x}) ∈ insert x ‘ B
  by (rule imageI)
also from x∈ A
  have insert x (A − {x}) = A
  by auto
finally show ?P .
qed

hide-const (open) member not-member

lemmas equalityI = subset-antisym
lemmas set-mp = subsetD
lemmas set-rev-mp = rev-subsetD

ML

val Ball-def = @{thm Ball-def}
val Bex-def = @{thm Bex-def}
val CollectD = @{thm CollectD}
val CollectE = @{thm CollectE}
val CollectI = @{thm CollectI}
val Collect-conj-eq = @{thm Collect-conj-eq}
val Collect-mem-eq = @{thm Collect-mem-eq}
val IntD1 = @{thm IntD1}
val IntD2 = @{thm IntD2}
val IntE = @{thm IntE}
val IntI = @{thm IntI}
val Int-Collect = @{thm Int-Collect}
val UNIV-I = @{thm UNIV-I}
val UNIV-witness = @{thm UNIV-witness}
val UnE = @{thm UnE}
val UnI1 = @{thm UnI1}
val UnI2 = @{thm UnI2}
val ballE = @{thm ballE}
val ballI = @{thm ballI}
val bexCI = @{thm bexCI}
val bexE = @{thm bexE}
val bexI = @{thm bexI}
val bex-triv = @{thm bex-triv}
val bspec = @{thm bspec}
val contra-subsetD = @{thm contra-subsetD}
val equalityCE = @{thm equalityCE}
val equalityD1 = @{thm equalityD1}
HOL type definitions

theory Typedef
imports Set
keywords
typedef :: thy-goal-defn and
morphisms :: quasi-command
begin

locale type-definition =
  fixes Rep and Abs and A
  assumes Rep: Rep x ∈ A
  and Rep-inverse: Abs (Rep x) = x
  and Abs-inverse: y ∈ A ⇒ Rep (Abs y) = y
  — This will be axiomatized for each typedef!
begin

lemma Rep-inject: Rep x = Rep y ⟷ x = y
proof
  assume Rep x = Rep y

then have \( \text{Abs} (\text{Rep} \, x) = \text{Abs} (\text{Rep} \, y) \) by (simp only:)
moreover have \( \text{Abs} (\text{Rep} \, x) = x \) by (rule \text{Rep-inverse})
moreover have \( \text{Abs} (\text{Rep} \, y) = y \) by (rule \text{Rep-inverse})
ultimately show \( x = y \) by simp
next
assume \( x = y \)
then show \( \text{Rep} \, x = \text{Rep} \, y \) by (simp only:)
qed

lemma \text{Abs-inject}:
assumes \( x \in A \) and \( y \in A \)
shows \( \text{Abs} \, x = \text{Abs} \, y \rightleftharpoons x = y \)
proof
assume \( \text{Abs} \, x = \text{Abs} \, y \)
then have \( \text{Rep} \, (\text{Abs} \, x) = \text{Rep} \, (\text{Abs} \, y) \) by (simp only:)
moreover from \( \langle x \in A \rangle \) have \( \text{Rep} \, (\text{Abs} \, x) = x \) by (rule \text{Rep-inverse})
moreover from \( \langle y \in A \rangle \) have \( \text{Rep} \, (\text{Abs} \, y) = y \) by (rule \text{Rep-inverse})
ultimately show \( x = y \) by simp
next
assume \( x = y \)
then show \( \text{Abs} \, x = \text{Abs} \, y \) by (simp only:)
qed

lemma \text{Rep-cases} [cases set]:
assumes \( y \in A \)
and \( \text{hyp}: \langle x. \, y = \text{Rep} \, x \Longrightarrow P \rangle \)
shows \( P \)
proof (rule \text{hyp})
from \( \langle y \in A \rangle \) have \( \text{Rep} \, (\text{Abs} \, y) = y \) by (rule \text{Abs-inverse})
then show \( y = \text{Rep} \, (\text{Abs} \, y) \) ..
qed

lemma \text{Abs-cases} [cases type]:
assumes \( r: \forall y. \, x = \text{Abs} \, y \Longrightarrow y \in A \Longrightarrow P \)
shows \( P \)
proof (rule \text{r})
have \( \text{Abs} \, (\text{Rep} \, x) = x \) by (rule \text{Rep-inverse})
then show \( x = \text{Abs} \, (\text{Rep} \, x) \) ..
show \( \text{Rep} \, x \in A \) by (rule \text{Rep})
qed

lemma \text{Rep-induct} [induct set]:
assumes \( y: \, y \in A \)
and \( \text{hyp}: \forall x. \, P \, (\text{Rep} \, x) \)
shows \( P \, y \)
proof
have \( P \, (\text{Rep} \, (\text{Abs} \, y)) \) by (rule \text{hyp})
moreover from \( y \) have \( \text{Rep} \, (\text{Abs} \, y) = y \) by (rule \text{Abs-inverse})
ultimately show \( P \, y \) by simp
qed

lemma Abs-induct [induct type]:
  assumes r: \( \forall y. y \in A \implies P (\text{Abs } y) \)
  shows P x
proof
  have Rep x \( \in \) A by (rule Rep)
  then have P (Abs (Rep x)) by (rule r)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  ultimately show P x by simp
qed

lemma Rep-range: range Rep = A
proof
  show range Rep \( \subseteq \) A using Rep by (auto simp add: image-def)
  show A \( \subseteq \) range Rep
proof
    fix x assume x \( \in \) A
    then have x = Rep (Abs x) by (rule Abs-inverse [symmetric])
    then show x \( \in \) range Rep by (rule range-eqI)
  qed
qed

lemma Abs-image: Abs ' A = UNIV
proof
  show Abs ' A \( \subseteq \) UNIV by (rule subset-UNIV)
  show UNIV \( \subseteq \) Abs ' A
proof
    fix x
    have x = Abs (Rep x) by (rule Rep-inverse [symmetric])
    moreover have Rep x \( \in \) A by (rule Rep)
    ultimately show x \( \in \) Abs ' A by (rule image-eqI)
  qed
qed

end

ML-file <Tools/typedef.ML>

end

10 Notions about functions

theory Fun
import Set
keywords functor :: thy-goal-defn
begin

lemma apply-inverse: \( f = u \implies (\forall x. P x \implies g (f x) = x) \implies P x \implies x = g \)
Uniqueness, so NOT the axiom of choice.

**Lemma uniq-choice:** \( \forall x. \exists! y. Q x y \implies \exists f. \forall x. Q x (f x) \)
by (force intro: theI')

**Lemma b-uniq-choice:** \( \forall x \in S. \exists! y. Q x y \implies \exists f. \forall x \in S. Q x (f x) \)
by (force intro: theI')

### 10.1 The Identity Function \( \text{id} \)

**Definition** \( \text{id} :: 'a \Rightarrow 'a \)
where \( \text{id} = (\lambda x. x) \)

**Lemma id-apply [simp]:** \( \text{id} x = x \)
by (simp add: id-def)

**Lemma image-id [simp]:** \( \text{image id} = \text{id} \)
by (simp add: id-def fun-eq-iff)

**Lemma vimage-id [simp]:** \( \text{vimage id} = \text{id} \)
by (simp add: id-def fun-eq-iff)

**Lemma eq-id-iff:** \( (\forall x. f x = x) \iff f = \text{id} \)
by auto

**Code-printing**
constant \( \text{id} \to (\text{Haskell}) \text{id} \)

### 10.2 The Composition Operator \( f \circ g \)

**Definition** \( \text{comp} :: ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c \) (infixl \( \circ \) 55)
where \( f \circ g = (\lambda x. f (g x)) \)

**Notation (ASCII)**
\[ \text{comp} \quad (\text{infixl} \circ 55) \]

**Lemma comp-apply [simp]:** \( (f \circ g) x = f (g x) \)
by (simp add: comp-def)

**Lemma comp-assoc:** \( (f \circ g) \circ h = f \circ (g \circ h) \)
by (simp add: fun-eq-iff)

**Lemma id-comp [simp]:** \( \text{id} \circ g = g \)
by (simp add: fun-eq-iff)

**Lemma comp-id [simp]:** \( f \circ \text{id} = f \)
by (simp add: fun-eq-iff)
lemma \textit{comp-eq-dest}: \(a \circ b = c \circ d \implies a \ (b \ v) = c \ (d \ v)\)
by (simp add: fun-eq-iff)

lemma \textit{comp-eq-clim}: \(a \circ b = c \circ d \implies (\forall v. \ a \ (b \ v) = c \ (d \ v)) \implies R \implies R\)
by (simp add: fun-eq-iff)

lemma \textit{comp-eq-dest-lhs}: \(a \circ b = c \implies a \ (b \ v) = c \ v\)
by clarsimp

lemma \textit{comp-eq-id-dest}: \(a \circ b = id \circ c \implies a \ (b \ v) = c \ v\)
by clarsimp

lemma \textit{image-comp}: \(f' (g' r) = (f \circ g)' r\)
by auto

lemma \textit{vimage-comp}: \(f'^{-} (g^{-} x) = (g' \circ f)' x\)
by auto

lemma \textit{image-eq-imp-comp}: \(f' A = g' B \implies (h \circ f)' A = (h \circ g)' B\)
by (auto simp: comp-def elim!: equalityE)

lemma \textit{image-bind}: \(f' (\text{Set.bind} \ A \ g) = \text{Set.bind} A (\ (' f \circ g))\)
by (auto simp add: Set.bind-def)

lemma \textit{bind-image}: \(\text{Set.bind} (f' A) \ g = \text{Set.bind} A (g \circ f)\)
by (auto simp add: Set.bind-def)

lemma (in \textit{group-add}) \textit{minus-comp-minus} [simp]: \(\text{uminus} \circ \text{uminus} = id\)
by (simp add: fun-eq-iff)

lemma (in \textit{boolean-algebra}) \textit{minus-comp-minus} [simp]: \(\text{uminus} \circ \text{uminus} = id\)
by (simp add: fun-eq-iff)

code-printing
  constant \textit{comp} \to (SML) infixl 5 o and (Haskell) \textit{infixr 9} .

10.3 The Forward Composition Operator \textit{fcomp}

definition \textit{fcomp} :: \('a \Rightarrow 'b\) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'c\ (\textit{infixl} \circ> 60)
where \(f \circ> g = (\lambda x. \ g \ (f \ x))\)

lemma \textit{fcomp-apply} [simp]: \((f \circ> g) \ x = g \ (f \ x)\)
by (simp add: fcomp-def)

lemma \textit{fcomp-assoc}: \((f \circ> g) \circ> h = f \circ> (g \circ> h)\)
by (simp add: fcomp-def)

lemma \textit{id-fcomp} [simp]: \(id \circ> g = g\)
by \((simp\ add:\ fcomp-def)\)

**lemma** \(fcomp-id\) \([simp]\): \(f\circ id = f\)
by \((simp\ add:\ fcomp-def)\)

**lemma** \(fcomp-comp\): \(fcomp\ f\ g = \comp\ g\ f\)
by \((simp\ add:\ ext)\)

code-printing
constant \(fcomp\) \(\mapsto\) (Eval) infixl 1 #>

no-notation \(fcomp\) (infixl \(\circ\) > 60)

### 10.4 Mapping functions

**definition** \(map-fun\) :: \('c\Rightarrow\ 'a\Rightarrow\ 'b\Rightarrow\ 'd\) ⇒ \('a\ set\Rightarrow\ 'b\ set\Rightarrow\ bool\) — injective
where \(map-fun\ f\ g\ h\ x = g\ (h\ (f\ x))\)
by \((simp\ add:\ map-fun-def)\)

### 10.5 Injectivity and Bijectivity

**definition** \(inj-on\) :: \('a\Rightarrow\ 'b\) ⇒ \('a\ set\⇒\ bool\) — injective
where \(inj-on\ f\ A\ =\ \forall\ x\in A.\ \forall\ y\in A.\ f\ x = f\ y\ \rightarrow\ x = y\)

**definition** \(bij-betw\) :: \('a\Rightarrow\ 'b\) ⇒ \('a\ set\⇒\ 'b\ set\⇒\ bool\) — bijective
where \(bij-betw\ f\ A\ B\ =\ inj-on\ f\ A\ ∧\ f\ A\ =\ B\)

A common special case: functions injective, surjective or bijective over the entire domain type.

**abbreviation** \(inj\) :: \('a\Rightarrow\ 'b\) ⇒ bool
where \(inj\ f\ =\ inj-on\ f\ UNIV\)

**abbreviation** \(surj\) :: \('a\Rightarrow\ 'b\) ⇒ bool
where \(surj\ f\ =\ range\ f\ =\ UNIV\)

translations — The negated case:
\!¬\ Const \(surj\ f\ =\ Const\ range\ f\ \neq\ Const\ UNIV\)

**abbreviation** \(bij\) :: \('a\Rightarrow\ 'b\) ⇒ bool
where \(bij\ f\ =\ bij-betw\ f\ UNIV\ UNIV\)

**lemma** \(inj-def\) :\ inj\ f\ \longleftrightarrow\ (∀\ x\ y.\ f\ x = f\ y\ \rightarrow\ x = y)\)
unfolding \(inj-on-def\) by \(blast\)

**lemma** \(injI\) :\ (∀\ x\ y.\ f\ x = f\ y\ \Longrightarrow\ x = y)\ \Longrightarrow\ inj\ f\)
unfolding \(inj-def\) by \(blast\)
theorem range-ex1-eq: inj f ⟹ b ∈ range f ⟷ (∃!x. b = f x)
unfolding inj-def by blast

lemma injD: inj f ⟹ f x = f y ⟹ x = y
by (simp add: inj-def)

lemma inj-on-eq-iff: inj-on f A ⟹ x ∈ A ⟹ y ∈ A ⟹ f x = f y ⟷ x = y
by (auto simp: inj-on-def)

lemma inj-on-cng: (∀a. a ∈ A ⟹ f a = g a) ⟹ inj-on f A ⟷ inj-on g A
by (auto simp: inj-on-def)

lemma image-strict-mono: inj-on f B ⟹ A ⊂ B ⟹ f ' A ⊂ f ' B
unfolding inj-on-def by blast

lemma inj-compose: inj f ⟹ inj g ⟹ inj (f ∘ g)
by (simp add: inj-def)

lemma inj-fun: inj f ⟹ inj (λx y. f x)
by (simp add: inj-def fun-eq-iff)

lemma inj-eq: inj f ⟹ f x = f y ⟷ x = y
by (simp add: inj-on-eq-iff)

lemma inj-on-iff-Uniq: inj-on f A ⟷ (∀x ∈ A. ∃y ∈ A. f x = f y)
by (auto simp: Uniq-def inj-on-def)

lemma inj-on-id[simp]: inj-on id A
by (simp add: inj-on-def)

lemma inj-on-id2[simp]: inj-on (λx. x) A
by (simp add: inj-on-def)

lemma inj-on-Int: inj-on f A ∨ inj-on f B ⟹ inj-on f (A ∩ B)
unfolding inj-on-def by blast

lemma surj-id: surj id
by simp

lemma bij-id[simp]: bij id
by (simp add: bij-betw-def)

lemma bij-uminus: bij (uminus :: 'a::group-add)
unfolding bij-betw-def inj-on-def
by (force intro: minus-minus [symmetric])

lemma bij-betwE: bij-betw f A B ⟹ ∀a ∈ A. f a ∈ B
unfolding bij-betw-def by auto
lemma inj-on [intro?]: \((\forall x y. x \in A \Rightarrow y \in A \Rightarrow f x = f y \Rightarrow x = y) \Rightarrow inj-on f A\)
  by (simp add: inj-on-def)

For those frequent proofs by contradiction

lemma inj-onCI: \((\forall x y. x \in A \Rightarrow y \in A \Rightarrow f x = f y \Rightarrow x \neq y \Rightarrow False) \Rightarrow inj-on f A\)
  by (force simp: inj-on-def)

lemma inj-on-inverseI: \((\forall x. x \in A \Rightarrow g (f x) = x) \Rightarrow inj-on f A\)
  unfolding inj-on-def by blast

lemma inj-on-subset:
  assumes inj-on f A
  and \(B \subseteq A\)
  shows inj-on f B
proof (rule inj-onI)
  fix \(a\) \(b\)
  assume \(a \in B\) and \(b \in B\)
  with assms have \(a \in A\) and \(b \in A\)
   by auto
  moreover assume \(f a = f b\)
  ultimately show \(a = b\)
   using assms by (auto dest: inj-onD)
qed

lemma comp-inj-on: inj-on f A \Rightarrow inj-on g (f ' A) \Rightarrow inj-on (g \circ f) A
  by (simp add: comp-def inj-on-def)

lemma inj-on-imageI: inj-on (g \circ f) A \Rightarrow inj-on g (f ' A)
  by (auto simp add: inj-on-def)

lemma inj-on-image-iff:
  \(\forall x \in A. \forall y \in A. g (f x) = g (f y) \leftrightarrow g x = g y \Rightarrow inj-on f A \Rightarrow inj-on g (f ' A) \leftrightarrow inj-on g A\)
  unfolding inj-on-def by blast

lemma inj-on-contraD: inj-on f A \Rightarrow \(x \neq y \Rightarrow x \in A \Rightarrow y \in A \Rightarrow f x \neq f y\)
  unfolding inj-on-def by blast

lemma inj-singleton [simp]: inj-on (\(\lambda x. \{x\})\) A
  by (simp add: inj-on-def)

lemma inj-on-empty [iff]: inj-on f {}
  by (simp add: inj-on-def)
THEORY "Fun"

lemma subset-inj-on: inj-on f B ⊆ B =⇒ inj-on f A
  unfolding inj-on-def by blast

lemma inj-on-Un: inj-on f (A ∪ B) =⇒ inj-on f A ∧ inj-on f B ∧ f '(A − B) ∩ f '(B − A) = {}
  unfolding inj-on-def by (blast intro: sym)

lemma inj-on-insert [iff]: inj-on f (insert a A) =⇒ inj-on f A ∧ f a /∈ f '(A − {a})
  unfolding inj-on-def by (blast intro: sym)

lemma inj-on-diff: inj-on f A =⇒ inj-on f (A − B)
  unfolding inj-on-def by blast

lemma comp-inj-on-iff: inj-on f A =⇒ inj-on (f' ◦ f) A
  by (auto simp: comp-inj-on inj-on-def)

lemma inj-on-imageI2: inj-on (f' ◦ f) A =⇒ inj-on f A
  by (auto simp: comp-inj-on inj-on-def)

lemma inj-img-insertE:
  assumes inj-on f A
  assumes x /∈ B and insert x B = f' A
  obtains x' A' where x' /∈ A' and A = insert x' A' and x = f x' and B = f' A'
  proof −
    from assms have x ∈ f' A by auto
    then obtain x' where : x' ∈ A x = f x' by auto
    then have A: A = insert x' (A − {x'}) by auto
    with assms have B: B = f' (A − {x'}) by (auto dest: inj-on-contraD)
    have x' /∈ A − {x'} by simp
    from this A x = f x' B show ?thesis ..
  qed

lemma linorder-inj-onI:
  fixes A :: 'a::order set
  assumes ne: ∀x y. [x < y; x∈A; y∈A] =⇒ f x ≠ f y and lin: ∀x y. [x∈A; y∈A]
  ⇒ x≤y ∨ y≤x
  shows inj-on f A
  proof (rule inj-onI)
    fix x y
    assume eq: f x = f y and x∈A y∈A
    then show x = y
      using lin [of x y] ne by (force simp: dual-order.order-iff-strict)
  qed

lemma linorder-inj-onI':
  fixes A :: 'a :: linorder set
assumes $\forall i, j. \ i \in A \Rightarrow j \in A \Rightarrow i < j \Rightarrow f i \neq f j$
shows $\text{inj} \ f \ A$
by (intro linorder-inj-onI) (auto simp add: assms)

lemma linorder-injI:
assumes $\forall x \ y :: A :: \text{linorder}. \ x < y \Rightarrow f x \neq f y$
shows $\text{inj} \ f$
— Courtesy of Stephan Merz
using assms by (simp add: linorder-inj-onI)

lemma inj-on-image-Pow: $\text{inj} \ f \ A \Rightarrow \text{inj} \ (\text{image} \ f) \ (\text{Pow} \ A)$
unfolding Pow-def inj-on-def by blast

lemma bij-betw-image-Pow: $\text{bij-betw} \ f \ A \ B \Rightarrow \text{bij-betw} \ (\text{image} \ f) \ (\text{Pow} \ A) \ (\text{Pow} \ B)$
by (auto simp add: bij-betw-def inj-on-image-Pow inj-on-Pow-surj)

lemma surj-def: $\text{surj} \ f \iff (\forall y. \ \exists x. \ y = f x)$
by auto

lemma surjI:
assumes $\forall x. \ g \ (f \ x) = x$
shows $\text{surj} \ g$
using assms [symmetric] by auto

lemma surjD: $\text{surj} \ f \Rightarrow \exists x. \ y = f x$
by (simp add: surj-def)

lemma surjE: $\text{surj} \ f \Rightarrow (\forall x. \ y = f x \Rightarrow C) \Rightarrow C$
by (simp add: surj-def) blast

lemma comp-surj: $\text{surj} \ f \Rightarrow \text{surj} \ g \Rightarrow \text{surj} \ (g \circ f)$
using image-comp [of g f UNIV] by simp

lemma bij-betw-imageI1: $\text{inj-on} \ f \ A \Rightarrow f \cdot A = B \Rightarrow \text{bij-betw} \ f \ A \ B$
unfolding bij-betw-def by clarify

lemma bij-betw-imp-surj-on: $\text{bij-betw} \ f \ A \ B \Rightarrow f \cdot A = B$
unfolding bij-betw-def by clarify

lemma bij-betw-imp-surj: $\text{bij-betw} \ f \ A \ UNIV \Rightarrow \text{surj} \ f$
unfolding bij-betw-def by auto

lemma bij-betw-emptyI: $\text{bij-betw} \ f \ \{\} \ A \Rightarrow A = \{\}$
unfolding bij-betw-def by auto

lemma bij-betw-empty2: $\text{bij-betw} \ f \ A \ \{\} \Rightarrow A = \{\}$
unfolding bij-betw-def by blast
**THEORY “Fun”**

lemma inj-on-imp-bij-betw: inj-on \( f \ A \implies\) bij-betw \( f \ A \) \((f \setminus A)\)
unfolding bij-betw-def by simp

lemma bij-betw-DiffI:
assumes \( \text{bij-betw } f \ A \ B \) \( \text{bij-betw } f \ C \ D \) \( C \subseteq A \) \( D \subseteq B \)
shows \( \text{bij-betw } f \ (A - C) \) \( (B - D) \)
using assms unfolding bij-betw-def inj-on-def by auto

lemma bij-betw-singleton-iff [simp]: \( \text{bij-betw } f \ \{x\} \ \{y\} \iff f \ x = y \)
by (auto simp: bij-betw-def)

lemma bij-betw-singletonI [intro]: \( f \ x = y \iff \text{bij-betw } f \ \{x\} \ \{y\} \)
by auto

lemma bij-betw-apply: \[ \text{bij-betw } f \ A \ B \implies f \ a \in B \]
unfolding bij-betw-def by auto

lemma bij-def: \( \text{bij } f \iff \text{inj } f \land \text{surj } f \)
by (rule bij-betw-def)

lemma bij-I: \( \text{inj } f \implies \text{surj } f \implies \text{bij } f \)
by (rule bij-betw-imageI)

lemma bij-is-inj: \( \text{bij } f \implies \text{inj } f \)
by (simp add: bij-def)

lemma bij-is-surj: \( \text{bij } f \implies \text{surj } f \)
by (simp add: bij-def)

lemma bij-betw-imp-inj-on: \( \text{bij-betw } f \ A \ B \implies \text{inj-on } f \ A \)
by (simp add: bij-betw-def)

lemma bij-betw-trans: \( \text{bij-betw } f \ A \ B \implies \text{bij-betw } g \ B \ C \implies \text{bij-betw } (g \circ f) \ A \ C \)
by (auto simp add: bij-betw-def comp-inj-on)

lemma bij-comp: \( \text{bij } f \implies \text{bij } g \implies \text{bij } (g \circ f) \)
by (rule bij-betw-trans)

lemma bij-betw-comp-iff: \( \text{bij-betw } f \ A \ A' \implies \text{bij-betw } f' \ A' \ A'' \iff \text{bij-betw } (f' \circ f) \ A \ A'' \)
by (auto simp add: bij-betw-def inj-on-def)

lemma bij-betw-Collect:
assumes \( \text{bij-betw } f \ A \ B \land x. x \in A \implies Q (f x) \iff P x \)
shows \( \text{bij-betw } f \ \{x\in A. P x\} \ \{y\in B. Q y\} \)
using assms by (auto simp add: bij-betw-def inj-on-def)

lemma bij-betw-comp-iff2:
assumes \( \text{bij: bij-betw } f' \ A' \ A'' \)
and \( \text{img} \): \( f' \ A \leq A' \)

shows \( \text{bij-betw } f \ A \ A' \rightleftharpoons \text{bij-betw } (f' \circ f) \ A \ A'' \) (is \( ?L \rightleftharpoons ?R \))

**proof**

assume \( ?L \)
then show \( ?R \)
   using assms by (auto simp add: bij-betw-comp-iff)

next

assume \( * \): \( ?R \)

have \( \text{inj-on } (f' \circ f) \ A \Longrightarrow \text{inj-on } f \ A \)
   using inj-on-imageI2 by blast

moreover have \( A' \subseteq f' \ A \)

proof

fix \( a' \)

assume \( ** \): \( a' \in A' \)

with \( \text{bij} \) have \( f' \ a' \in A'' \)
   unfolding bij-betw-def by force

with \( \text{img} \) have \( f \ a \in A' \) by auto

with \( \text{bij} \ ** \) \( 1 \) have \( f \ a = a' \)
   unfolding bij-betw-def inj-on-def by force

with \( 1 \) show \( a' \in f' \ A \) by auto

qed

ultimately show \( ?L \)
   using \( \text{img} \ * \) by (auto simp add: bij-betw-def)

qed

**lemma** \( \text{bij-betw-inv} \):

assumes \( \text{bij-betw } f \ A \ B \)

shows \( \exists \ g. \ \text{bij-betw } g \ B \ A \)

**proof** –

have \( i: \text{inj-on } f \ A \) and \( s: f \ A = B \)
   using assms by (auto simp: bij-betw-def)

let \( ?P = \lambda b \ a. \ a \in A \land f \ a = b \)
let \( ?g = \lambda b. \) The \( (?P \ b) \)

have \( g: \ ?g \ b = a \) if \( P: \ ?P \ b \ a \) for \( a \ b \)

proof –

from that \( s \) have \( \exists \ a. \ ?P \ b \ a \) by blast

then have \( \text{uex1: } \exists!a. \ ?P \ b \ a \) by (blast dest:inj-onD[OF \( i \)])

then show \( ?\text{thesis} \)
   using \( \text{the1-equality[OF uex1, OF } P \] P by simp \)

qed

have \( \text{inj-on } ?g \ B \)

proof (rule inj-onI)

fix \( x \ y \)

assume \( x \in B \ y \in B \ ?g \ x = ?g \ y \)

from \( s \ x \in B \) obtain \( a1 \) where \( a1: \ ?P \ x \ a1 \) by blast

from \( s \ y \in B \) obtain \( a2 \) where \( a2: \ ?P \ y \ a2 \) by blast

from \( g \ \{OF \ a1\} \ a1 \ g \ \{OF \ a2\} \ a2 \ \langle ?g \ x = ?g \ y \rangle \) show \( x = y \) by simp
qed
moreover have \(?g \cdot B = A
proof safe
  fix \(b\)
  assume \(b \in B\)
  with \(s\) obtain \(a\) where \(P: \exists P a b\) by blast
  with \(\overline{g[OF P]}\) show \(\exists b \in A\) by auto
next
  fix \(a\)
  assume \(a \in A\)
  with \(s\) obtain \(b\) where \(P: \exists P b a\) by blast
  with \(\overline{s}\) have \(b \in \overline{B}\) by blast
  with \(\overline{g[OF P]}\) have \(\exists b \in B. \ a = \ ?g b\) by blast
  then show \(a \in \ ?g \cdot B\)
    by auto
qed
ultimately show \(?\thesis\)
  by (auto simp: bij-betw-def)
qed

lemma bij-betw-cong: \((\forall a. a \in A \implies f a = g a) \implies \text{bij-betw } f A A' = \text{bij-betw } g A A'\)
  unfolding bij-betw-def inj-on-def by safe force+

lemma bij-betw-id [intro, simp]: \(\text{bij-betw } id A A\)
  unfolding bij-betw-def id-def by auto

lemma bij-betw-id-iff: \(\text{bij-betw } id A B \iff A = B\)
  by (auto simp add: bij-betw-def)

lemma bij-betw-combine:
  bij-betw \(f A B\) \(\implies\) bij-betw \(f C D\) \(\implies\) \(B \cap D = \{\}\) \(\implies\) bij-betw \(f (A \cup C) (B \cup D)\)
  unfolding bij-betw-def inj-on-Un image-Un by auto

lemma bij-betw-subset: \(\text{bij-betw } f A A' \implies B \subseteq A \implies f \cdot B = B' \implies \text{bij-betw } f B B'\)
  by (auto simp add: bij-betw-def)

lemma bij-betw-ball: \(\text{bij-betw } f A B \implies (\forall b \in B. \phi b) = (\forall a \in A. \phi (f a))\)
  unfolding bij-betw-def inj-on-def by blast

lemma bij-pointE:
  assumes \(bij f\)
  obtains \(x\) where \(y = f x\) and \(\land x'. y = f x' \implies x' = x\)
proof -
  from assms have \(inj f\) by (rule bij-is-inj)
moreover from assms have \(surj f\) by (rule bij-is-surj)
  then have \(y \in \text{range } f\) by simp
ultimately have $\exists ! x. \: y = f x$ by \((\text{simp add: range-ext-eq})\)

with \emph{that} show thesis by blast

qed

lemma bij-iff:
\(\langle \text{bij} \: f \: \longleftrightarrow \: (\forall x. \: \exists ! y. \: f \: y = x)\rangle\) \(\langle \text{is} \: \langle ?P \: \longleftrightarrow \: ?Q \rangle \rangle\)

proof
assume \(?P\)
then have \(\langle \text{inj} \: f, \: \text{surj} \: f \rangle\)
by \((\text{auto simp: bij-def})\)
show \(?Q\)
proof
fix \(y\)
from \(\langle \text{surj} \: f \rangle\) obtain \(x\) where \(\langle y = f \: x \rangle\)
by \((\text{auto simp: surj-def})\)
with \(\langle \text{inj} \: f \rangle\) show \(\langle \exists ! x. \: f \: x = y \rangle\)
by \((\text{auto simp: inj-def})\)
qed

next
assume \(?Q\)
then have \(\langle \text{inj} \: f \rangle\)
by \((\text{auto simp: inj-def})\)
moreover have \(\langle \exists x. \: y = f x \rangle\) for \(y\)
proof
from \(\langle ?Q \rangle\) obtain \(x\) where \(\langle f \: x = y \rangle\)
by blast
then have \(\langle y = f \: x \rangle\)
by simp
then show \(?\text{thesis} \). ..
qed

then have \(\langle \text{surj} \: f \rangle\)
by \((\text{auto simp: surj-def})\)
ultimately show \(?P\)
by \((\text{rule bijI})\)
qed

lemma bij-betw-partition:
\(\langle \text{bij-betw} \: f \: A \: B \rangle\)
if \(\langle \text{bij-betw} \: f \: (A \cup C) \: (B \cup D) \rangle\) \(\langle \text{bij-betw} \: f \: C \: D \rangle\) \(\langle A \cap C = \{ \} \rangle\) \(\langle B \cap D = \{ \} \rangle\)
proof
from that have \(\langle \text{inj-on} \: f \: (A \cup C) \rangle\) \(\langle \text{inj-on} \: f \: C \rangle\) \(\langle f ^ \cdot (A \cup C) = B \cup D \rangle\) \(\langle f ^ \cdot C = D \rangle\)
by \((\text{simp-all add: bij-betw-def})\)
then have \(\langle \text{inj-on} \: f \: A \rangle\) and \(\langle f ^ \cdot (A - C) \cap f ^ \cdot (C - A) = \{ \} \rangle\)
by \((\text{simp-all add: inj-on-Un})\)
with \(\langle A \cap C = \{ \} \rangle\) have \(\langle f ^ \cdot A \cap f ^ \cdot C = \{ \} \rangle\)
by auto
with \(\langle f ^ \cdot (A \cup C) = B \cup D \rangle\) \(\langle f ^ \cdot C = D \rangle\) \(\langle B \cap D = \{ \} \rangle\)
have \(\langle f ^ \cdot A = B \rangle\)
by blast
with ⟨inj-on f A⟩ show ?thesis
  by (simp add: bij-betw-def)
qed

lemma surj-image-vimage-eq: surj f ⟷ f (f' A) = A
  by simp

lemma surj-vimage-empty:
  assumes surj f
  shows f' A = {} ⟷ A = {}
  using surj-image-vimage-eq[OF ⟨surj f⟩, of A]
  by (intro iffI) fastforce+

lemma inj-vimage-image-eq: inj f ⟷ f' (f' A) = A
  unfolding inj-def by blast

lemma vimage-subsetD: surj f ⟷ f' B ⊆ A ⇒ B ⊆ f' A
  by (blast intro: sym)

lemma vimage-subsetI: inj f ⟷ B ⊆ f' A ⇒ f' B ⊆ A
  unfolding inj-def by blast

lemma vimage-subset-iff: bij f ⟷ f' B ⊆ A ⇒ B ⊆ f' A
  unfolding bij-def by (blast del: subsetI intro: vimage-subsetI vimage-subsetD)

lemma inj-on-image-eq-iff: inj-on f C ⟷ A ⊆ C ⇒ B ⊆ C ⇒ f' A = f' B
  ⟷ A = B
  by (fastforce simp: inj-on-def)

lemma inj-on-Un-image-eq-iff: inj-on f (A ∪ B) ⇒ f' (A ∪ B) = f' A = f' B
  ⟷ A = B
  by (erule inj-on-image-eq-iff) simp-all

lemma inj-on-image-Int: inj-on f C ⇒ A ⊆ C ⇒ B ⊆ C ⇒ f' (A ∩ B) = f' B
  unfolding inj-on-def by blast

lemma inj-on-image-set-diff: inj-on f C ⇒ A - B ⊆ C ⇒ B ⊆ C ⇒ f' (A - B) = f' A - f' B
  unfolding inj-on-def by blast

lemma image-Int: inj f ⇒ f' (A ∩ B) = f' A ∩ f' B
  unfolding inj-def by blast

lemma image-set-diff: inj f ⇒ f' (A - B) = f' A - f' B
  unfolding inj-def by blast

lemma inj-on-image-mem-iff: inj-on f B ⇒ a ∈ B ⇒ A ⊆ B ⇒ f a ∈ f' A
  ⟷ a ∈ A
by (auto simp: inj-on-def)

lemma inj_image_mem_iff: inj f ⇒ f a ∈ f ' A ⇔ a ∈ A
  by (blast dest: injD)

lemma inj_image_subset_iff: inj f ⇒ f ' A ⊆ f ' B ⇔ A ⊆ B
  by (blast dest: injD)

lemma inj_image_eq_iff: inj f ⇒ f ' A = f ' B ⇔ A = B
  by (blast dest: injD)

lemma surj_Compl_image_subset: surj f ⇒ − (f ' A) ⊆ f ' (− A)
  by auto

lemma inj_image_Compl_subset: inj f ⇒ f ' (− A) ⊆ − (f ' A)
  by (auto simp: inj-def)

lemma bij_image_Compl_eq: bij f ⇒ f ' (− A) = − (f ' A)
  by (simp add: bij=image-Compl-subset surj=Compl-image-subset equalityI)

lemma inj_image_singleton: inj f ⇒ f − {a} ⊆ {THE x. f x = a}
  — The inverse image of a singleton under an injective function is included in a singleton.
  by (simp add: inj-def) (blast intro: the-equality [symmetric])

lemma inj_on_image_singleton: inj_on f A ⇒ f − {a} ∩ A ⊆ {THE x. x ∈ A ∧ f x = a}
  by (auto simp add: inj-on_def intro: the-equality [symmetric])

lemma bij_betw_byWitness: assumes left: ∀ a ∈ A. f' (f a) = a
  and right: ∀ a' ∈ A'. f (f' a') = a'
  and f' A ⊆ A'
  and img2: f' ' A' ⊆ A
  shows bij_betw f A A'
  using assms
  unfolding bij_betw_def inj-on-def

proof safe
  fix a b
  assume a ∈ A b ∈ A
  with left have a = f' (f a) ∧ b = f' (f b) by simp
  moreover assume f a = f b
  ultimately show a = b by simp
next
  fix a' assume *: a' ∈ A'
  with img2 have f' a' ∈ A by blast
  moreover from * right have f' a' = f (f' a') by simp
  ultimately show a' ∈ f ' A by blast
qed
corollary notIn-Un-bij-betw:
assumes $b \notin A$
and $f \cdot b \notin A'$
and bij-betw $f \cdot A A'$
shows bij-betw $f \cdot (A \cup \{b\}) \cdot (A' \cup \{f \cdot b\})$
proof –
have bij-betw $f \cdot \{b\} \cdot \{f \cdot b\}$
unfolding bij-betw-def inj-on-def by simp
with asms show ?thesis
using bij-betw-combine[of $f \cdot A A'$ \{b\} \{f \cdot b\}] by blast
qed

lemma notIn-Un-bij-betw3:
assumes $b \notin A$
and $f \cdot b \notin A'$
shows $\text{bij-betw} \cdot f \cdot A A' = \text{bij-betw} \cdot f \cdot (A \cup \{b\}) \cdot (A' \cup \{f \cdot b\})$
proof
assume bij-betw $f \cdot A A'$
then show bij-betw $f \cdot (A \cup \{b\}) \cdot (A' \cup \{f \cdot b\})$
using asms notIn-Un-bij-betw[of $b \cdot A f A'$]
by blast
next
assume $*; \text{bij-betw} \cdot f \cdot (A \cup \{b\}) \cdot (A' \cup \{f \cdot b\})$
have $f' A = A'$
proof safe
fix $a$
assume $**; a \in A$
then have $f a \in A' \cup \{f \cdot b\}$
using $*$ unfolding bij-betw-def by blast
moreover
have False if $f a = f \cdot b$
proof –
have $a = b$
using $**$ that unfolding bij-betw-def inj-on-def by blast
with $\langle b \notin A \rangle$ $**$ show ?thesis by blast
qed
ultimately have $a \in A$ by blast
next
fix $a'$
assume $**; a' \in A'$
then have $a' \in f' \cdot (A \cup \{b\})$
using $*$ by (auto simp add: bij-betw-def)
then obtain $a$ where $1; a \in A \cup \{b\} \land f a = a'$ by blast
moreover
have False if $a = b$ using $1$ $**$ $\langle f \cdot b \notin A' \rangle$ that by blast
ultimately have $a \in A$ by blast
with $1$ show $a' \in f' \cdot A$ by blast
qed
then show bij-betw $f \cdot A A'$
using \texttt{* bij-betw-subset\{af f A \cup \{b\} - A\} by blast}

\textbf{qed}

\textbf{lemma inj-on-disjoint-Un:}
\textbf{assumes inj-on f A and inj-on g B}
\textbf{and} \( f \ ' A \cap g \ ' B = \{} \)
\textbf{shows inj-on} (\( \lambda x. \) \( \) if \( x \in A \) then \( f x \) else \( g x \) \( ) \) (\( A \cup B \))
\textbf{using assms by} (simp add: inj-on-def disjoint-iff) (blast)

\textbf{lemma bij-betw-disjoint-Un:}
\textbf{assumes bij-betw f A C and bij-betw g B D}
\textbf{and} \( A \cap B = \{} \)
\textbf{and} \( C \cap D = \{} \)
\textbf{shows bij-betw} (\( \lambda x. \) \( \) if \( x \in A \) then \( f x \) else \( g x \) \( ) \) (\( A \cup B \)) (\( C \cup D \))
\textbf{using assms by} (auto simp: inj-on-disjoint-Un bij-betw-def)

\textbf{lemma involuntary-imp-bij:
\langle bij f \rangle if \langle \( \forall x. f (f x) = x \) \rangle
\textbf{proof (rule bijI)}
from that \textbf{show} \langle surj f \rangle
by (rule surjI)
show \langle inj f \rangle
\textbf{proof (rule injI)}
fix \( x y \)
assume \langle \( f x = f y \) \rangle
then have \langle \( f (f x) = f (f y) \) \rangle
by simp
then show \langle \( x = y \) \rangle
by (simp add: that)
\textbf{qed}
\textbf{qed}

10.5.1 \textbf{Inj/surj/bij of Algebraic Operations}
\textbf{context cancel-semigroup-add begin}

\textbf{lemma inj-on-add [simp]:}
\textbf{inj-on} (\( \langle + \rangle a \) \( A \))
\textbf{by} (rule inj-onI) simp

\textbf{lemma inj-on-add' [simp]:}
\textbf{inj-on} (\( \lambda b. b + a \) \( A \))
\textbf{by} (rule inj-onI) simp

\textbf{lemma bij-betw-add [simp]:}
\textbf{bij-betw} (\( \langle + \rangle a \) \( A \) \( B \) \( \mapsto \) \( \langle + \rangle a \ \cdot \ A = B \))
\textbf{by} (simp add: bij-betw-def)
context group-add

lemma diff-left-imp-eq: $a - b = a - c \implies b = c$
unfolding add-uminus-conv-diff[symmetric]
by (drule local.add-left-imp-eq) simp

lemma inj-uminus[simp, intro]: inj-on uminus $A$
by (auto intro!: inj-onI)

lemma surj-uminus[simp]: surj uminus
using surjI minus-minus by blast

lemma surj-plus[simp]:
  surj $((+)\ a)$
proof (standard, simp, standard, simp)
  fix $x$
  have $x = a + (\neg a + x)$ by (simp add: add_assoc)
  thus $x \in \text{range} ((+)\ a)$ by blast
qed

lemma surj-plus-right[simp]:
  surj $(\lambda b.\ b+a)$
proof (standard, simp, standard, simp)
  fix $b$ show $b \in \text{range} (\lambda b.\ b+a)$
    using diff-add-cancel[of $b\ a$, symmetric] by blast
qed

lemma inj-on-diff-left[simp]:
  $\langle\text{inj-on}\ ((-)\ a)\ A\rangle$
by (auto intro: inj-onI dest!: diff-left-imp-eq)

lemma inj-on-diff-right[simp]:
  $\langle\text{inj-on}\ (\lambda b.\ b - a)\ A\rangle$
by (auto intro: inj-onI simp add: algebra-simps)

lemma surj-diff[simp]:
  surj $((-)\ a)$
proof (standard, simp, standard, simp)
  fix $x$
  have $x = a - (\neg x + a)$ by (simp add: algebra-simps)
  thus $x \in \text{range} ((-)\ a)$ by blast
qed

lemma surj-diff-right[simp]:
  surj $(\lambda x.\ x-a)$
proof (standard, simp, standard, simp)
fix $x$

have $x = x + a - a$ by simp

thus $x \in \text{range} \ (\lambda x. x - a)$ by fast

qed

lemma shows bij-plus: bij $((+\ a)\ )$ and bij-plus-right: bij $((\lambda x.x) + a)$

and bij-uminus: bij $uminus$

and bij-diff: bij $((\lambda x.x) - a)$ and bij-diff-right: bij $((\lambda x.x) - a)$

by(simp-all add: bij-def)

lemma translation-subtract-Compl:

$(\lambda x.x - a)\ '(-\ t) = - ((\lambda x.x - a)\ 't)$

by(rule bij-image-Compl-eq)

(auto simp add: bij-def surj-def inj-def diff-eq-eq intro add-diff-cancel[symmetric])

lemma translation-diff:

$(+\ a\ '(s - t)) = ((+\ a\ 's) - ((+\ a\ 't))$

by auto

lemma translation-subtract-diff:

$(\lambda x.x - a)\ '(s - t) = (((\lambda x.x - a)\ 's) - ((\lambda x.x - a)\ 't))$

by(rule image-set-diff)(simp add: inj-on-def diff-eq-eq)

lemma translation-Int:

$(+\ a\ '(s \cap t)) = ((+\ a\ 's) \cap ((+\ a\ 't))$

by auto

lemma translation-subtract-Int:

$(\lambda x.x - a)\ '(s \cap t) = (((\lambda x.x - a)\ 's) \cap ((\lambda x.x - a)\ 't))$

by(rule image-Int)(simp add: inj-on-def diff-eq-eq)

end

context ab-group-add

begin

lemma translation-Compl:

$(+\ a\ '(-\ t)) = - ((+\ a\ 't))$

proof (rule set-eqI)

fix $b$

show $b \in ([+] a\ '(-\ t)) \longleftrightarrow b \in (- (+\ a\ 't))$

by (auto simp: image-iff algebra-simps intro!: bexI [of - b - a])

qed

end
10.6 Function Updating

definition fun-upd :: '(a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a ⇒ 'b)
  where fun-upd f a b = (\x. if x = a then b else f x)

nonterminal updbinds and updbind

syntax
  -updbind :: 'a ⇒ 'a ⇒ updbind
  where updbind ⇒ updbinds

translations
  -Update f (-updbinds b bs) ≜ -Update (-Update f b) bs
  f(x:=y) ≜ CONST fun-upd f x y

lemma fun-upd-idem-iff: f(x:=y) = f ×⇒ f x = y
  unfolding fun-upd-def
  apply safe
  apply (erule subst)
  apply auto
  done

lemma fun-upd-idem: f x = y ⇒ f(x := y) = f
  by (simp only: fun-upd-idem-iff)

lemma fun-upd-triv [iff]: f(x := f x) = f
  by (simp only: fun-upd-idem)

lemma fun-upd-apply [simp]: (f(x := y)) z = (if z = x then y else f z)
  by (simp add: fun-upd-def)

lemma fun-upd-same: (f(x := y)) x = y
  by simp

lemma fun-upd-other: z ≠ x ⇒ (f(x := y)) z = f z
  by simp

lemma fun-upd-upd [simp]: f(x := y, x := z) = f(x := z)
  by (simp add: fun-eq-iff)

lemma fun-upd-twist: a ≠ c ⇒ (m(a := b))(c := d) = (m(c := d))(a := b)
  by auto

lemma inj-on-fun-upd: inj-on f A ⇒ y ∉ f ` A ⇒ inj-on (f(x := y)) A
  by (auto simp: inj-on-def)
lemma fun-upd-image: \( f(x := y) \cdot A = (\text{if } x \in A \text{ then } \text{insert } y (f' (A - \{x\})) \text{ else } f' A) \)
  by auto

lemma fun-upd-comp: \( f \circ (g(x := y)) = (f \circ g)(x := f y) \)
  by auto

lemma fun-upd-eqD: \( f(x := y) = g(x := z) \implies y = z \)
  by (simp add: fun-eq-iff split: if-split-asm)

10.7 override-on

definition override-on :: \('a ⇒ 'b) ⇒ ('a set ⇒ 'b) ⇒ ('a set ⇒ 'b)
  where override-on f g A = (\(\lambda a. \text{if } a \in A \text{ then } g a \text{ else } f a\))

lemma override-on-emptyset[simp]: override-on f g {} = f
  by (simp add: override-on-def)

lemma override-on-apply-notin[simp]: \( a \notin A \implies (\text{override-on } f g A) a = f a \)
  by (simp add: override-on-def)

lemma override-on-apply-in[simp]: \( a \in A \implies (\text{override-on } f g A) a = g a \)
  by (simp add: override-on-def)

lemma override-on-insert: override-on f g (insert x X) = (override-on f g X)(x := g x)
  by (simp add: override-on-def fun-eq-iff)

lemma override-on-insert': override-on f g (insert x X) = (override-on (f(x := g x)) g X)
  by (simp add: override-on-def fun-eq-iff)

10.8 Inversion of injective functions

definition the-inv-into :: \('a set ⇒ ('a ⇒ 'b) ⇒ ('b ⇒ 'a)\)
  where the-inv-into A f = (\(\lambda y. \text{THE } x. y \in A \wedge f y = x\))

lemma the-inv-into-f-f: inj-on f A \implies x \in A \implies the-inv-into A f (f x) = x
  unfolding the-inv-into-def inj-on-def by blast

lemma f-the-inv-into-f: inj-on f A \implies y \in f' A \implies f (the-inv-into A f y) = y
  unfolding the-inv-into-def
  by (rule the1I2; blast dest: inj-onD)

lemma f-the-inv-into-f-bij-betw:
  bij-betw f A B \implies (bij-betw f A B \implies x \in B) \implies f (the-inv-into A f x) = x
  unfolding bij-betw-def by (blast intro: f-the-inv-into-f)
lemma the-inv-into-into: inj-on f A \implies x \in f \cdot A \implies A \subseteq B \implies the-inv-into A f x \in B
  unfolding the-inv-into-def
  by (rule the1I2; blast dest: inj-onD)

lemma the-inv-into-onto [simp]: inj-on f A \implies the-inv-into A f \cdot (f \cdot A) = A
  by (fast intro: the-inv-into-into the-inv-into-f-f [symmetric])

lemma the-inv-into-f-eq: inj-on f A \implies f x = y \implies x \in A \implies the-inv-into A f y = x
  by (force simp add: the-inv-into-f-f)

lemma the-inv-into-comp:
  inj-on f (g \cdot A) \implies inj-on g A \implies x \in f \cdot g \cdot A \implies
  the-inv-into A (f \circ g) x = (the-inv-into A g \circ the-inv-into (g \cdot A) f) x
  apply (rule the-inv-into-f-eq)
  apply (fast intro: comp-inj-on)
  apply (simp add: f-the-inv-into-f the-inv-into-into)
  apply (simp add: the-inv-into-into)
  done

lemma inj-on-the-inv-into: inj-on f A \implies inj-on (the-inv-into A f) (f \cdot A)
  by (auto intro: inj-onI simp: the-inv-into-f-f)

lemma bij-betw-the-inv-into: bij-betw f A B \implies bij-betw (the-inv-into A f) B A
  by (auto simp add: bij-betw-def inj-on-the-inv-into the-inv-into-into)

lemma bij-betw-iff-bijections:
  bij-betw f A B \iff (\exists g. (\forall x \in A. f x \in B \land g(f x) = x) \land (\forall y \in B. g y \in A \land f(g y) = y))
  (is \ ?lhs = ?rhs)
  proof
    show ?lhs \implies ?rhs
      by (auto simp: bij-betw-def f-the-inv-into-f the-inv-into-f-f the-inv-into-into exI[where ?x=the-inv-into A f])
  next
    show ?rhs \implies ?lhs
      by (force intro: bij-betw-byWitness)
  qed

abbreviation the-inv :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)
  where the-inv f \equiv the-inv-into UNIV f

lemma the-inv-f-f: the-inv f (f x) = x if inj f
  using that UNIV-f by (rule the-inv-into-f-f)
10.9 Monotonicity

**Definition** monotone-on :: 'a set ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ bool

where monotone-on A orda ordb f ←→ (∀ x ∈ A. ∀ y ∈ A. orda x y ⇒ ordb (f x) (f y))

**Abbreviation** monotone :: ('a ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ bool

where monotone ≡ monotone-on UNIV

**Lemma** monotone-def [no-atp]: monotone orda ordb f ←→ (∀ x y. orda x y ⇒ ordb (f x) (f y))

by (simp add: monotone-on-def)

Lemma monotone-def is provided for backward compatibility.

**Lemma** monotone-onI:

(∀ x y. x ∈ A ⇒ y ∈ A ⇒ orda x y ⇒ ordb (f x) (f y)) ⇒ monotone-on A orda ordb f

by (simp add: monotone-on-def)

**Lemma** monotoneI[intro?]: (∀ x y. orda x y ⇒ ordb (f x) (f y)) ⇒ monotone orda ordb f

by (rule monotone-onI)

**Lemma** monotone-onD:

monotone-on A orda ordb f ⇒ x ∈ A ⇒ y ∈ A ⇒ orda x y ⇒ ordb (f x) (f y)

by (simp add: monotone-on-def)

**Lemma** monotoneD[dest?]: monotone orda ordb f ⇒ orda x y ⇒ ordb (f x) (f y)

by (rule monotone-onD[of UNIV, simplified])

**Lemma** monotone-on-subset: monotone-on A orda ordb f ⇒ B ⊆ A ⇒ monotone-on B orda ordb f

by (auto intro: monotone-onI dest: monotone-onD)

**Lemma** monotone-on-empty[simp]: monotone-on {} orda ordb f

by (auto intro: monotone-onI dest: monotone-onD)

**Lemma** monotone-on-o:

assumes

mono-f: monotone-on A orda ordb f and

mono-g: monotone-on B ordc orda g and

g ′ B ⊆ A

shows monotone-on B ordc ordb (f ◦ g)

**Proof** (rule monotone-onI)

fix x y assume x ∈ B and y ∈ B and ordc x y

hence orda (g x) (g y)

by (rule mono-g[THEN monotone-onD])
moreover from $g : B \subseteq A$, $x \in B$, $y \in B$, have $g \ x \in A$ and $g \ y \in A$

unfolding `image-subset-iff` by `simp-all`
ultimately show `ordb ((f o g) x) ((f o g) y)`
using `mono-f[THEN monotone-onD]` by `simp`

qed

10.9.1 Specializations For `ord` Type Class And More

context `ord` begin

abbreviation `mono-on` :: `'a set ⇒ ('a ⇒ 'b :: `ord`) ⇒ bool
  where `mono-on` A ≡ `monotone-on` A ($\leq$) ($\leq$)

abbreviation `strict-mono-on` :: `'a set ⇒ ('a ⇒ 'b :: `ord`) ⇒ bool
  where `strict-mono-on` A ≡ `monotone-on` A ($<$) ($<$)

abbreviation `antimono-on` :: `'a set ⇒ ('a ⇒ 'b :: `ord`) ⇒ bool
  where `antimono-on` A ≡ `monotone-on` A ($\leq$) ($\geq$)

abbreviation `strict-antimono-on` :: `'a set ⇒ ('a ⇒ 'b :: `ord`) ⇒ bool
  where `strict-antimono-on` A ≡ `monotone-on` A ($<$) ($>$)

lemma `mono-on-def[no-atp]`: `mono-on` A f ←→ ($\forall r \ s. \ r \in A \& \ s \in A \& \ r \leq s \to f \ r \leq f \ s$)
  by (auto simp add: `monotone-on-def`)

lemma `strict-mono-on-def[no-atp]`:
  `strict-mono-on` A f ←→ ($\forall r \ s. \ r \in A \& \ s \in A \& \ r < s \to f \ r < f \ s$)
  by (auto simp add: `monotone-on-def`)

Lemmas `mono-on-def` and `strict-mono-on-def` are provided for backward compatibility.

lemma `mono-onI`:
  ($\forall r \ s. \ r \in A \to s \in A \to r \leq s \to f \ r \leq f \ s$) \to `mono-on` A f
  by (rule `monotone-onI`)

lemma `strict-mono-onI`:
  ($\forall r \ s. \ r \in A \to s \in A \to r < s \to f \ r < f \ s$) \to `strict-mono-on` A f
  by (rule `monotone-onI`)

lemma `mono-onD`: `[`mono-on` A f; \ r \in A; \ s \in A; \ r \leq s] \to f \ r \leq f \ s
  by (rule `monotone-onD`)

lemma `strict-mono-onD`: `[strict-mono-on A f; \ r \in A; \ s \in A; \ r < s] \to f \ r < f \ s
  by (rule `monotone-onD`)

lemma `mono-on-subset`: `mono-on` A f \to B \subseteq A \to `mono-on` B f
  by (rule `monotone-on-subset`)
lemma mono-on-greaterD:
  assumes mono-on A g x ∈ A y ∈ A g x > (g (y::::linorder) :: - :: linorder)
  shows x > y
proof (rule ccontr)
  assume ¬x > y
  hence x ≤ y by (simp add: not-less)
  from assms(1-3) and this have g x ≤ g y by (rule mono-onD)
  with assms(4) show False by simp
qed

class order
begin
abbreviation mono :: ('a ⇒ 'b::order) ⇒ bool
  where mono ≡ mono-on UNIV

abbreviation strict-mono :: ('a ⇒ 'b::order) ⇒ bool
  where strict-mono ≡ strict-mono-on UNIV

abbreviation antimono :: ('a ⇒ 'b::order) ⇒ bool
  where antimono ≡ monotone (≤) (λx y. y ≤ x)

lemma mono-def[no-atp]: mono f ↔ (∀x y. x ≤ y → f x ≤ f y)
  by (simp add: monotone-on-def)

lemma strict-mono-def[no-atp]: strict-mono f ↔ (∀x y. x < y → f x < f y)
  by (simp add: monotone-on-def)

lemma antimono-def[no-atp]: antimono f ↔ (∀x y. x ≤ y → f x ≥ f y)
  by (simp add: monotone-on-def)

Lemmas mono-def, strict-mono-def, and antimono-def are provided for backward compatibility.

lemma monoI [intro?]: (∀x y. x ≤ y → f x ≤ f y) → mono f
  by (rule monotoneI)

lemma strict-monoI [intro?]: (∀x y. x < y → f x < f y) → strict-mono f
  by (rule monotoneI)

lemma antimonoI [intro?]: (∀x y. x ≤ y → f x ≥ f y) → antimono f
  by (rule monotoneI)

lemma monoD [dest?]: mono f → x ≤ y → f x ≤ f y
  by (rule monotoneD)

lemma strict-monoD [dest?]: strict-mono f → x < y → f x < f y
  by (rule monotoneD)
lemma antimonoD [dest?]: antimono f \(\Rightarrow\) \(x \leq y \Rightarrow f x \geq f y\) 
  by (rule monotoneD)

lemma monoE:
  assumes mono f
  assumes \(x \leq y\)
  obtains \(f x \leq f y\)
proof
  from assms show \(f x \leq f y\) by (simp add: mono-def)
qed

lemma antimonoE:
  fixes f :: 'a ⇒ 'b::order
  assumes antimono f
  assumes \(x \leq y\)
  obtains \(f x \geq f y\)
proof
  from assms show \(f x \geq f y\) by (simp add: antimono-def)
qed

lemma mono-imp-mono-on:
  mono f \(\Rightarrow\) mono-on A f 
  by (rule monotone-on-subset[OF subset_UNIV])

lemma strict-mono-mono [dest?]::
  assumes strict-mono f
  shows mono f
proof (rule monoI)
  fix x y
  assume \(x \leq y\)
  show \(f x \leq f y\)
  proof (cases \(x = y\))
    case True then show ?thesis by simp
  next
    case False with \(x \leq y\) have \(x < y\) by simp
    with assms strict-monoD have \(f x < f y\) by auto
    then show ?thesis by simp
  qed
qed

lemma mono-on-ident:
  mono-on S (\(\lambda x. x\))
  by (simp add: monotone-on-def)

lemma strict-mono-on-ident:
  strict-mono-on S (\(\lambda x. x\))
  by (simp add: monotone-on-def)

lemma mono-on-const:
  fixes a :: 'b::order shows mono-on S (\(\lambda x. a\))
  by (simp add: mono-on-def)
lemma antimono-on-const:
  fixes a :: 'b::order shows antimono-on S (λx. a)
  by (simp add: monotone-on-def)
end

context linorder begin

lemma mono-invE:
  fixes f :: 'a ⇒ 'b::order
  assumes mono f
  assumes f x < f y
  obtains x ≤ y
  proof
    show x ≤ y
    proof (rule ccontr)
      assume ¬ x ≤ y
      then have y ≤ x by simp
      with (mono f) obtain f y ≤ f x by (rule monoE)
      with (f x < f y) show False by simp
    qed
  qed

lemma mono-strict-invE:
  fixes f :: 'a ⇒ 'b::order
  assumes mono f
  assumes f x < f y
  obtains x < y
  proof
    show x < y
    proof (rule ccontr)
      assume ¬ x < y
      then have y ≤ x by simp
      with (mono f) obtain f y ≤ f x by (rule monoE)
      with (f x < f y) show False by simp
    qed
  qed

lemma strict-mono-eq:
  assumes strict-mono f
  shows f x = f y ←→ x = y
  proof
    assume f x = f y
    show x = y proof (cases x y rule: linorder-cases)
      case less with assms strict-monoD have f x < f y by auto
      with (f x = f y) show ?thesis by simp
    next
      case equal then show ?thesis .
next
case greater with assms strict-monoD have f y < f x by auto
  with f x = f y show ?thesis by simp
qed
qed simp

lemma strict-mono-less-eq:
  assumes strict-mono f
  shows f x ≤ f y ⇔ x ≤ y
proof
  assume x ≤ y
  with assms strict-mono monoD show f x ≤ f y by auto
next
  assume f x ≤ f y
  show x ≤ y
  proof (rule ccontr)
    assume ¬ x ≤ y
    then have y < x by simp
    with assms strict-monoD have f y < f x by auto
    with f x ≤ f y show False by simp
  qed
qed

lemma strict-mono-less:
  assumes strict-mono f
  shows f x < f y ⇔ x < y
  using assms by (auto simp add: less-le Orderings.less-le strict-mono-eq strict-mono-less-eq)
end

lemma strict-mono-inv:
  fixes f :: ('a::linorder) ⇒ ('b::linorder)
  assumes strict-mono f and surj f and inv: ∀x. g (f x) = x
  shows strict-mono g
proof
  fix x y :: 'b
  assume x < y
  from surj f obtain x' y' where [simp]; x = f x' y = f y' by blast
  with x < y and strict-mono f have x' < y' by (simp add: strict-mono-less)
  with inv show g x < g y by simp
qed

lemma strict-mono-on-imp-inj-on:
  assumes strict-mono-on A f :: (~ :: linorder) ⇒ (~ :: preorder)
  shows inj-on f A
proof (rule inj-onI)
  fix x y assume x ∈ A y ∈ A f x = f y
  thus x = y
  proof (cases x y rule: linorder-cases)
    (auto dest: strict-mono-onD[OF assms, of x y] strict-mono-onD[OF assms, of y x])
  qed
end
qed

lemma strict-mono-on-leD:
assumes strict-mono-on A (f :: (- :: linorder) ⇒ - :: preorder) x ∈ A y ∈ A x ≤ y
shows f x ≤ f y
proof (cases x = y)
case True
then show ?thesis by simp
next
case False
with assms have f x < f y
using strict-mono-onD[OF assms(1)] by simp
then show ?thesis by (rule less-imp-le)
qed

lemma strict-mono-on-eqD:
fixes f :: (- :: linorder) ⇒ (- :: preorder)
assumes strict-mono-on A f f x = f y x ∈ A y ∈ A
shows y = x
using assms by (cases rule: linorder-cases) (auto dest: strict-mono-onD)

lemma strict-mono-on-imp-mono-on:
strict-mono-on A (f :: (- :: linorder) ⇒ - :: preorder) =⇒ mono-on A f
by (rule mono-onI, rule strict-mono-on-leD)

lemma mono-imp-strict-mono:
fixes f :: 'a::order ⇒ 'b::order
shows [mono-on S f; inj-on f S] =⇒ strict-mono-on S f
by (auto simp add: monotone-on-def order-less-le inj-on-eq-iff)

lemma strict-mono-iff-mono:
fixes f :: 'a::linorder ⇒ 'b::order
shows strict-mono-on S f ⇐⇒ mono-on S f ∧ inj-on f S
proof
show strict-mono-on S f =⇒ mono-on S f ∧ inj-on f S
  by (simp add: strict-mono-on-imp-inj-on strict-mono-on-imp-mono-on)
qed (auto intro: mono-imp-strict-mono)

lemma antimono-imp-strict-antimono:
fixes f :: 'a::order ⇒ 'b::order
shows [antimono-on S f; inj-on f S] =⇒ strict-antimono-on S f
by (auto simp add: monotone-on-def order-less-le inj-on-eq-iff)

lemma strict-antimono-iff-antimono:
fixes f :: 'a::linorder ⇒ 'b::order
shows strict-antimono-on S f ⇐⇒ antimono-on S f ∧ inj-on f S
proof
show strict-antimono-on S f =⇒ antimono-on S f ∧ inj-on f S
  by (simp add: strict-antimono-on-imp-inj-on strict-antimono-on-imp-mono-on)
by (force simp add: monotone-on-def intro; linorder-inj-onI)
qed (auto intro: antimono-imp-strict-antimono)

lemma mono-compose: mono $Q \Rightarrow$ mono $(\lambda i. Q i \ (f \ x))$
  unfolding mono-def le-fun-def by auto

lemma mono-add:
  fixes $a :: 'a::{ordered-ab-semigroup-add}
  shows mono $((+) \ a)$
  by (simp add: add-left-mono monoI)

lemma (in semilattice-inf) mono-inf: mono $f \Rightarrow f \ (A \cap B) \leq f \ A \cap f \ B$
  for $f :: 'a \Rightarrow 'b::semilattice-inf$
  by (auto simp add: mono-def intro: Lattices.inf-greatest)

lemma (in semilattice-sup) mono-sup: mono $f \Rightarrow f \ (A \cup B) \leq f \ (A \cup B)$
  for $f :: 'a \Rightarrow 'b::semilattice-sup$
  by (auto simp add: mono-def intro: Lattices.sup-least)

lemma (in linorder) min-of-mono: mono $f \Rightarrow \min \ (f \ m \ (f \ n)) = f \ (\min \ m \ n)$
  by (auto simp: mono-def Orderings.min-def min-def intro: Orderings.antisym)

lemma (in linorder) max-of-mono: mono $f \Rightarrow \max \ (f \ m \ (f \ n)) = f \ (\max \ m \ n)$
  by (auto simp: mono-def Orderings.max-def max-def intro: Orderings.antisym)

lemma (in linorder)
  max-of-antimono: antimono $f \Rightarrow \max \ (f \ x \ (f \ y)) = f \ (\min \ x \ y)$ and
  min-of-antimono: antimono $f \Rightarrow \min \ (f \ x \ (f \ y)) = f \ (\max \ x \ y)$
  by (auto simp: antimono-Def Orderings.max-def max-def Orderings.min-def min-def intro!: antisym)

lemma (in linorder) strict-mono-imp-inj-on: strict-mono $f \Rightarrow inj-on f \ A$
  by (auto intro!: inj-onI dest: strict-mono-eq)

lemma mono-Int: mono $f \Rightarrow f \ (A \cap B) \subseteq f \ A \cap f \ B$
  by (fact mono-inf)

lemma mono-Un: mono $f \Rightarrow f \ A \cup f \ B \subseteq f \ (A \cup B)$
  by (fact mono-sup)

10.9.2 Least value operator

lemma Least-mono: mono $f \Rightarrow \exists x \in S. \ \forall y \in S. \ x \leq y \Rightarrow (\LEAST \ y. \ y \in f \ S) = f \ (\LEAST x. \ x \in S)$
  for $f :: 'a::order \Rightarrow 'b::order$
  — Courtesy of Stephan Merz
  apply clarify
  apply (erule-tac P = $\lambda x. \ x \in S$ in Least12-order)
  apply fast
apply (rule LeastI2-order)
apply (auto elim: monoD intro: order-antisym)
done

10.10 Setup

10.10.1 Proof tools

Simplify terms of the form \( f(\ldots, x := y, \ldots, x := z, \ldots) \) to \( f(\ldots, x := z, \ldots) \)

```
simproc-setup fun-upd2 \( f(v := w, x := y) \) = 
let 
  fun gen-fun-upd NONE T - - = NONE 
  | gen-fun-upd (SOME f) T x y = SOME (Const (const-name \( \text{fun-upd} \), T)) 

$ f \$ x \$ y 
fan dest-fun-T1 (Type (-, T :: Ts)) = T 
fan find-double \( t \) as Const (const-name \( \text{fun-upd} \), T) $ f \$ x \$ y = 
let 
  fun find (Const (const-name \( \text{fun-upd} \), T) $ g \$ v \$ w) = 
    if v aconv x then SOME g else gen-fun-upd (find g) T v w 
  | find t = NONE 
in (dest-fun-T1 T, gen-fun-upd (find f) T x y) end 

val ss = simpset-of context
fun proc ctxt ct = 
  let 
    val t = Thm.term-of ct 
  in 
    (case find-double t of 
      (T, NONE) => NONE 
    | (T, SOME rhs) => SOME (Goal.prove ctxt [] [] (Logic.mk_equals (t, rhs))) 
      (fn => 
        resolve-tac ctxt [eq-reflection] 1 THEN 
        resolve-tac ctxt @{thms ext} 1 THEN 
        simp-tac (put-simpset ss ctxt) 1)) 
    end 
  in K proc end 
```

10.10.2 Functorial structure of types

ML-file ⟨Tools/functor.ML⟩

```
functor map-fun: map-fun 
  by (simp-all add: fun-eq-iff)
```

```
functor vimage 
  by (simp-all add: fun-eq-iff vimage-comp)
```
Legacy theorem names

lemmas o-def = comp-def
lemmas o-apply = comp-apply
lemmas o-assoc = comp-assoc [symmetric]
lemmas id-o = id-comp
lemmas o-id = comp-id
lemmas o-eq-dest = comp-eq-dest
lemmas o-eq-elim = comp-eq-elim
lemmas o-eq-dest-lhs = comp-eq-dest-lhs
lemmas o-eq-id-dest = comp-eq-id-dest

end

11 Complete lattices

theory Complete-Lattices
  imports Fun
begin

11.1 Syntactic infimum and supremum operations

class Inf =
  fixes Inf :: 'a set ⇒ 'a (\\ [- 900] 900)

class Sup =
  fixes Sup :: 'a set ⇒ 'a (\\ [+ 900] 900)

syntax
  -INF1 :: pttrns ⇒ 'b ⇒ 'b ((\( - . / \) [0, 10] 10)
  -INF :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((\( - . / \) [0, 0, 10] 10)
  -SUP1 :: pttrns ⇒ 'b ⇒ 'b ((\( + . / \) [0, 10] 10)
  -SUP :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((\( + . / \) [0, 0, 10] 10)

syntax
  -INF1 :: pttrns ⇒ 'b ⇒ 'b ((\[ - . / \] [0, 10] 10)
  -INF :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((\[ - . / \] [0, 0, 10] 10)
  -SUP1 :: pttrns ⇒ 'b ⇒ 'b ((\[ + . / \] [0, 10] 10)
  -SUP :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((\[ + . / \] [0, 0, 10] 10)

translations
  \[ x y. f \] = \(\\[-\] \(\[ x. \] \[ y. f \])
  \[ x. f \] = \(\\[\] \(\[ CONST range (\lambda x. f) \])
  \[ x\in A. f \] = \( CONST Inf ((\lambda x. f) \cdot A)
  \[ x y. f \] = \(\\[ x. \] \[ y. f \])
  \[ x. f \] = \(\\[\] \(\[ CONST range (\lambda x. f) \])
  \[ x\in A. f \] = \( CONST Sup ((\lambda x. f) \cdot A)

context Inf
begin
lemma INF-image: \( \bigsqcap \ (g \circ f) \cdot A \) = \( \bigsqcap \ ((g \circ f) \cdot A) \)
by (simp add: image-comp)

lemma INF-identity-eq [simp]: \( \bigsqcap x \in A. x \) = \( \bigsqcap A \)
by simp

lemma INF-id-eq [simp]: \( \bigsqcap \ (id \cdot A) \) = \( \bigsqcap A \)
by simp

lemma INF-cong: \( A = B \implies (\bigsqcap x \in B \implies C x = D x) \implies (C \cdot A) = (D \cdot B) \)
by (simp add: image-def)

lemma INF-cong-simp:
\( A = B \implies (\bigsqcap x \in B = simp=> C x = D x) \implies (C \cdot A) = (D \cdot B) \)
unfolding simp-implies-def by (fact INF-cong)

end

context Sup
begin

lemma SUP-image: \( \bigsqcup \ (g \circ f) \cdot A \) = \( \bigsqcup \ ((g \circ f) \cdot A) \)
by(fact Inf.INF-image)

lemma SUP-identity-eq [simp]: \( \bigsqcup x \in A. x \) = \( \bigsqcup A \)
by(fact Inf.INF-identity-eq)

lemma SUP-id-eq [simp]: \( \bigsqcup (id \cdot A) \) = \( \bigsqcup A \)
by(fact Inf.INF-id-eq)

lemma SUP-cong: \( A = B \implies (\bigsqcup x \in B \implies C x = D x) \implies (C \cdot A) = (D \cdot B) \)
by (fact Inf.INF-cong)

lemma SUP-cong-simp:
\( A = B \implies (\bigsqcup x \in B = simp=> C x = D x) \implies (C \cdot A) = (D \cdot B) \)
by (fact Inf.INF-cong-simp)

end

11.2 Abstract complete lattices

A complete lattice always has a bottom and a top, so we include them into
the following type class, along with assumptions that define bottom and top
in terms of infimum and supremum.

class complete-lattice = lattice + Inf + Sup + bot + top +
assumes Inf-lower: \( x \in A \implies \bigsqcap A \leq x \)
and \( \text{Inf-greatest: } (\bigwedge x. x \in A \implies z \leq x) \implies z \leq \bigcap A \)

and \( \text{Sup-upper: } x \in A \implies x \leq \bigcup A \)

and \( \text{Sup-least: } (\bigwedge x. x \in A \implies x \leq z) \implies \bigcup A \leq z \)

and \( \text{Inf-empty [simp]: } \bigcap \{\} = \top \)

and \( \text{Sup-empty [simp]: } \bigcup \{\} = \bot \)

begin

subclass bounded-lattice
proof
  fix \( a \)
  show \( \bot \leq a \)
    by (auto intro: Sup-least simp only: Sup-empty [symmetric])

  show \( a \leq \top \)
    by (auto intro: Inf-greatest simp only: Inf-empty [symmetric])

qed

lemma dual-complete-lattice: class.complete-lattice sup Inf sup (\geq) (\geq) inf \top \bot
by (auto intro!: class.complete-lattice.intro dual-lattice)

(unfold-locales, (fact Inf-empty Sup-empty Sup-upper Sup-least Inf-lower Inf-greatest)+)

end

context complete-lattice
begin

lemma Sup-eqI:
  \( (\bigwedge y. y \in A \implies y \leq x) \implies (\bigwedge z. z \in A \implies z \leq y) \implies x \leq y) \implies \bigcup A = x \)

  by (blast intro: order.antisym Sup-least Sup-upper)

lemma Inf-eqI:
  \( (\bigwedge i. i \in A \implies x \leq i) \implies (\bigwedge y. (\bigwedge i. i \in A \implies y \leq i) \implies y \leq x) \implies \bigcap A = x \)

  by (blast intro: order.antisym Inf-greatest Inf-lower)

lemma SUP-eqI:
  \( (\bigwedge i. i \in A \implies f i \leq x) \implies (\bigwedge y. (\bigwedge i. i \in A \implies f i \leq y) \implies x \leq y) \implies (\bigsqcup i \in A. f i) = x \)

  using Sup-eqI [of f ' A x] by auto

lemma INF-eqI:
  \( (\bigwedge i. i \in A \implies x \leq f i) \implies (\bigwedge y. (\bigwedge i. i \in A \implies f i \geq y) \implies x \geq y) \implies (\bigsqcap i \in A. f i) = x \)

  using Inf-eqI [of f ' A x] by auto

lemma INF-lower: \( i \in A \implies (\bigsqcap i \in A. f i) \leq f i \)

  using Inf-lower [of - f ' A] by simp

lemma INF-greatest: \( (\bigwedge i. i \in A \implies u \leq f i) \implies u \leq (\bigsqcap i \in A. f i) \)

  using Inf-greatest [of f ' A] by auto
lemma SUP-upper: \( i \in A \implies f i \leq \bigcup i \in A. f i \)
using Sup-upper [of \( f \cdot A \)] by simp

lemma SUP-least: \( \forall i. i \in A \implies f i \leq u \implies \bigcup i \in A. f i \leq u \)
using Sup-least [of \( f \cdot A \)] by auto

lemma Inf-lower2: \( u \in A \implies u \leq v \implies \bigcap A \leq v \)
using Inf-lower [of \( u \cdot A \)] by auto

lemma INF-lower2: \( i \in A \implies f i \leq u \implies \bigcap i \in A. f i \leq u \)
using INF-lower [of \( i \cdot A \cdot f \)] by auto

lemma Sup-upper2: \( u \in A \implies v \leq u \implies v \leq \bigcup A \)
using Sup-upper [of \( u \cdot A \)] by auto

lemma SUP-upper2: \( i \in A \implies u \leq f i \implies u \leq \bigcup i \in A. f i \)
using SUP-upper [of \( i \cdot A \cdot f \)] by auto

lemma le-Inf-iff: \( b \leq \bigcap A \iff (\forall a \in A. b \leq a) \)
by (auto intro: Inf-greatest dest: Inf-lower)

lemma le-INF-iff: \( u \leq \bigcap i \in A. f i \iff (\forall i \in A. u \leq f i) \)
using le-Inf-iff [of \( f \cdot A \)] by simp

lemma Sup-le-iff: \( \bigcup A \leq b \iff (\forall a \in A. a \leq b) \)
by (auto intro: Sup-least dest: Sup-upper)

lemma SUP-le-iff: \( \bigcup i \in A. f i \leq u \iff (\forall i \in A. f i \leq u) \)
using SUP-le-iff [of \( f \cdot A \)] by simp

lemma Inf-insert [simp]: \( \bigcap (\text{insert } a \cdot A) = a \cap \bigcap A \)
by (auto intro: le-infI le-infII le-infI2 order.antisym Inf-greatest Inf-lower)

lemma INF-insert: \( (\forall x \in \text{insert } a \cdot A. f x) = f a \cap \bigcap (f \cdot A) \)
by simp

lemma Sup-insert [simp]: \( \bigcup (\text{insert } a \cdot A) = a \cup \bigcup A \)
by (auto intro: le-supI le-supII le-supI2 order.antisym Sup-greatest Sup-upper)

lemma SUP-insert: \( (\forall x \in \text{insert } a \cdot A. f x) = f a \cup \bigcup (f \cdot A) \)
by simp

lemma INF-empty: \( (\forall x \in \{\}. f x) = \top \)
by simp

lemma SUP-empty: \( (\exists x \in \{\}. f x) = \bot \)
by simp
lemma Inf-UNIV [simp]: \( \bigcap \text{UNIV} = \bot \)
  by (auto intro: order.antisym Inf-lower)

lemma Sup-UNIV [simp]: \( \bigcup \text{UNIV} = \top \)
  by (auto intro: order.antisym Sup-upper)

lemma Inf-eq-Sup: \( \forall A. \bigcap A = \bigvee \{b. \forall a \in A. b \leq a\} \)
  by (auto intro: order.antisym Inf-lower Inf-greatest Sup-upper Sup-least)

lemma Sup-eq-Inf: \( \bigvee A = \bigcap \{b. \forall a \in A. a \leq b\} \)
  by (auto intro: order.antisym Inf-lower Inf-greatest Sup-upper Sup-least)

lemma Inf-superset-mono: \( B \subseteq A \implies \bigcap A \leq \bigcap B \)
  by (auto intro: Inf-greatest Inf-lower)

lemma Sup-subset-mono: \( A \subseteq B \implies \bigvee A \leq \bigvee B \)
  by (auto intro: Sup-least Sup-upper)

lemma Inf-mono:
  assumes \( \forall b. \exists a \in A. a \leq b \)
  shows \( \bigcap A \leq \bigcap B \)
  proof (rule Inf-greatest)
    fix b assume b \in B
    with assms obtain a where a \in A and a \leq b by blast
    from \( a \in A \) have \( \bigcap A \leq a \) by (rule Inf-lower)
    with \( a \leq b \) show \( \bigcap A \leq b \) by auto
  qed

lemma INF-mono: \( \forall m. \exists n \in A. f n \leq g m \implies (\bigcap n \in A. f n) \leq (\bigcap n \in B. g n) \)
  using Inf-mono [of g \cdot B f \cdot A] by auto

lemma INF-mono': \( \forall x. f x \leq g x \implies (\bigcap x \in A. f x) \leq (\bigcap x \in B. g x) \)
  by (rule INF-mono) auto

lemma Sup-mono:
  assumes \( \forall a. \exists b \in B. a \leq b \)
  shows \( \bigcup A \leq \bigcup B \)
  proof (rule Sup-least)
    fix a assume a \in A
    with assms obtain b where b \in B and a \leq b by blast
    from \( b \in B \) have \( b \leq \bigcup B \) by (rule Sup-upper)
    with \( a \leq b \) show \( a \leq \bigcup B \) by auto
  qed

lemma SUP-mono: \( \forall n. \exists m \in B. f n \leq g m \implies (\bigcup n \in A. f n) \leq (\bigcup n \in B. g n) \)
  using Sup-mono [of f \cdot A g \cdot B] by auto
lemma \textit{SUP-mono}': \((\forall x. f x \leq g x) \Rightarrow (\bigsqcup x \in A. f x) \leq (\bigsqcup x \in A. g x)\)
by (rule \textit{SUP-mono}) auto

lemma \textit{INF-superset-mono}: \(B \subseteq A \Rightarrow (\forall x. x \in B \Rightarrow f x \leq g x) \Rightarrow (\bigsqcap x \in A. f x) \leq (\bigsqcap x \in A. g x)\)
— The last inclusion is POSITIVE!
by (blast intro: \textit{INF-mono} dest: \textit{subsetD})

lemma \textit{SUP-subset-mono}: \(A \subseteq B \Rightarrow (\forall x. x \in A \Rightarrow f x \leq g x) \Rightarrow (\bigsqcup x \in A. f x) \leq (\bigsqcup x \in B. g x)\)
by (blast intro: \textit{SUP-mono} dest: \textit{subsetD})

lemma \textit{Inf-less-eq}:
assumes \(\forall v. v \in A \Rightarrow v \leq u\)
and \(A \neq \{\}\)
shows \(\bigsqcap A \leq u\)
proof –
  from \(\langle A \neq \{\} \rangle\) obtain \(v\) where \(v \in A\) by blast
moreover from \(\langle v \in A \rangle\) \textit{assms}(1) have \(v \leq u\) by blast
ultimately show ?thesis by (rule \textit{Inf-lower2})
qed

lemma \textit{less-eq-Sup}:
assumes \(\forall v. v \in A \Rightarrow u \leq v\)
and \(A \neq \{\}\)
shows \(u \leq \bigsqcup A\)
proof –
  from \(\langle A \neq \{\} \rangle\) obtain \(v\) where \(v \in A\) by blast
moreover from \(\langle v \in A \rangle\) \textit{assms}(1) have \(u \leq v\) by blast
ultimately show ?thesis by (rule \textit{Sup-upper2})
qed

lemma \textit{INF-eq}:
assumes \(\forall i. i \in A \Rightarrow \exists j \in B. f i \geq g j\)
and \(\exists j. j \in B \Rightarrow \exists i \in A. g j \geq f i\)
shows \(\bigsqcap (f ' A) = \bigsqcap (g ' B)\)
by (intro order.antisym \textit{INF-greatest}) (blast intro: \textit{INF-lower2 dest: assms})+

lemma \textit{SUP-eq}:
assumes \(\forall i. i \in A \Rightarrow \exists j \in B. f i \leq g j\)
and \(\exists j. j \in B \Rightarrow \exists i \in A. g j \leq f i\)
shows \(\bigsqcup (f ' A) = \bigsqcup (g ' B)\)
by (intro order.antisym \textit{SUP-least}) (blast intro: \textit{SUP-upper2 dest: assms})+

lemma \textit{less-eq-Inf-inter}: \(\bigsqcap A \sqcup \bigsqcap B \leq \bigsqcap (A \cap B)\)
by (auto intro: \textit{Inf-greatest} \textit{Inf-lower})

lemma \textit{Sup-inter-less-eq}: \(\bigsqcup (A \cap B) \leq \bigsqcup A \sqcup \bigsqcup B\)
by (auto intro: \textit{Sup-least} \textit{Sup-upper})
lemma *Inf-union-distrib*: \( \bigcap (A \cup B) = \bigcap A \cap \bigcap B \)
by (rule order.antisym) (auto intro: Inf-greatest Inf-lower le-infI1 le-infI2)

lemma *INF-union*: \( \bigcap i \in A \cup B. M i \) = \( \bigcap i \in A. M i \cap \bigcap i \in B. M i \)
by (auto intro: order.antisym INF-mono intro: le-infI1 le-infI2 INF-greatest INF-lower)

lemma *Sup-union-distrib*: \( \bigvee (A \cup B) = \bigvee A \cup \bigvee B \)
by (rule order.antisym) (auto intro: Sup-least Sup-upper le-supI1 le-supI2)

lemma *SUP-union*: \( \bigvee i \in A \cup B. M i \) = \( \bigvee i \in A. M i \cup \bigvee i \in B. M i \)
by (auto intro: order.antisym SUP-mono intro: le-supI1 le-supI2 SUP-least SUP-upper)

lemma *INF-inf-distrib*: \( \bigcap a \in A. f a \cap \bigcap a \in A. g a \) = \( \bigcap a \in A. f a \cap g a \)
by (rule order.antisym) (rule INF-greatest, auto intro: le-infI1 le-infI2 INF-lower INF-mono)

lemma *SUP-sup-distrib*: \( \bigvee a \in A. f a \cup \bigvee a \in A. g a \) = \( \bigvee a \in A. f a \cup g a \)
(is ?L = ?R)
proof (rule order.antisym)
show ?L \leq ?R
  by (auto intro: le-supI1 le-supI2 SUP-upper SUP-mono)
show ?R \leq ?L
  by (rule SUP-least) (auto intro: le-supI1 le-supI2 SUP-upper)
qed

lemma *Inf-top-conv* [simp]:
\( \prod A = \top \iff (\forall x \in A. x = \top) \)
\( \top = \prod A \iff (\forall x \in A. x = \top) \)

proof
  show \( \prod A = \top \iff (\forall x \in A. x = \top) \)
  proof
    assume \( \forall x \in A. x = \top \)
    then have \( A = \{\} \lor A = \{\top\} \) by auto
    then show \( \prod A = \top \) by auto
  next
    assume \( \prod A = \top \)
    show \( \forall x \in A. x = \top \)
    proof (rule ccontr)
      assume \( \forall x \in A. x = \top \)
      then obtain \( x \) where \( x \in A \) and \( x \neq \top \) by blast
      then obtain \( B \) where \( A = \text{insert} \ x \ B \) by blast
      with \( \prod A = \top \) \( x \neq \top \) show \( \text{False} \) by simp
    qed
  qed
  then show \( \top = \prod A \iff (\forall x \in A. x = \top) \) by auto
  qed
THEORY “Complete-Lattices”

lemma INF-top-conv [simp]:
\[(\underset{x \in A}{\bigwedge} B x) = \top \iff (\forall x \in A. B x = \top)\]
\[\top = (\underset{x \in A}{\bigwedge} B x) \iff (\forall x \in A. B x = \top)\]
using Inf-top-conv [of B ' A] by simp-all

lemma Sup-bot-conv [simp]:
\[\bigvee A = \bot \iff (\forall x \in A. x = \bot)\]
\[\bot = \bigvee A \iff (\forall x \in A. x = \bot)\]
using dual-complete-lattice by (rule complete-lattice. Inf-top-conv)

lemma INF-constant [simp]:
\[(d y \in A. c) = (if A = {} then \top else c)\]
by (auto intro: order. antisym INF-lower INF-greatest)

lemma SUP-constant [simp]:
\[(\bigvee y \in A. c) = (if A = {} then \bot else c)\]
by (auto intro: order. antisym SUP-upper SUP-least)

lemma INF-commute: 
\[(d i \in A. d j \in B. f i j) = (d j \in B. d i \in A. f i j)\]
by (iprover intro: INF-lower INF-greatest order-trans order. antisym)

lemma SUP-commute: 
\[(\bigvee i \in A. \bigvee j \in B. f i j) = (\bigvee j \in B. \bigvee i \in A. f i j)\]
by (iprover intro: SUP-upper SUP-upper order-trans order.antisym)

lemma INF-absorb:
assumes k \in I
shows A k \cap (\bigcap i \in I. A i) = (\bigcap i \in I. A i)
proof -
  from assms obtain J where I = insert k J by blast
  then show \?thesis by simp
qed

lemma SUP-absorb:
assumes $k \in I$
shows $A \cup (\bigsqcup_{i \in I} A \ i) = (\bigsqcup_{i \in I} A \ i)$
proof –
from asms obtain $J$ where $I = \operatorname{insert} k J$ by blast
then show ?thesis by simp
qed

lemma INFINITY-inf-const1: $I \neq \{\} \Longrightarrow (\bigsqcap_{i \in I} \inf x (f \ i)) = \inf x (\bigsqcap_{i \in I} f \ i)$
by (intro order.antisym INFINITY-greatest inf-mono order-refl INFINITY-lower)
(auto intro: INFINITY-lower2 le-infl2 intro: INFINITY-mono)

lemma INFINITY-inf-const2: $I \neq \{\} \Longrightarrow (\bigsqcap_{i \in I} \inf (f \ i) x) = \inf (\bigsqcap_{i \in I} f \ i) x$
using INFINITY-inf-const1[of $I \ x f$] by (simp add: inf-commute)

lemma less-INF-D:
assumes $y < (\bigsqcap_{i \in A} f \ i) \ i \in A$
shows $y < f \ i$
proof –
note $\langle y < (\bigsqcap_{i \in A} f \ i) \ \rangle$
also have $\langle \bigsqcap_{i \in A} f \ i \leq f \ i \rangle$ using $\langle i \in A \rangle$
by (rule INFINITY-lower)
finally show $y < f \ i$.
qed

lemma SUP-lessD:
assumes $(\bigsqcup_{i \in A} f \ i) < y \ i \in A$
shows $f \ i < y$
proof –
have $f \ i \leq (\bigsqcup_{i \in A} f \ i)$
using $\langle i \in A \rangle$ by (rule SUP-upper)
also note $\langle \bigsqcup_{i \in A} f \ i \leq y \rangle$
finally show $f \ i < y$.
qed

lemma INFINITY-UNIV-bool-expand: $(\bigsqcap b. A \ b) = A \ True \cap A \ False$
by (simp add: UNIV-bool inf-commute)

lemma SUP-UNIV-bool-expand: $(\bigsqcup b. A \ b) = A \ True \cup A \ False$
by (simp add: UNIV-bool sup-commute)

lemma Inf-le-Sup: $A \neq \{\} \Longrightarrow \inf A \leq \sup A$
by (blast intro: SUP-upper2 Inf-lower ex-in-conv)

lemma INFINITY-le-SUP: $A \neq \{\} \Longrightarrow (\bigsqcap (f ^+ A) \leq \bigsqcup (f ^+ A)$
using Inf-le-Sup[of $f \ ^+ A$] by simp

lemma INFINITY-eq-const: $I \neq \{\} \Longrightarrow (\forall i. i \in I \Longrightarrow f \ i = x) \Longrightarrow (\bigsqcap (f ^+ I) = x$
by (auto intro: INFINITY-eqI)
lemma `SUP-eq-const`: \( I \neq \{ \} \implies (\forall i \in I \implies f_i = x) \implies \bigsqcup (f \cdot I) = x \)

by (auto intro: SUP-eqI)

lemma `INF-eq-iff`: \( I \neq \{ \} \implies (\forall i \in I \implies c \leq f_i) \implies \bigsqcup (f \cdot I) = c \iff \text{(\forall i \in I \implies c \leq f_i) \iff (\forall i \in I \implies f_i = c)} \)

by (auto intro: SUP-eq-const INF-lower order.antisym)

lemma `SUP-eq-iff`: \( I \neq \{ \} \implies (\forall i \in I \implies c \leq f_i) \implies \bigsqcup (f \cdot I) = c \iff \text{(\forall i \in I \implies c = f_i) \iff (\forall i \in I \implies f_i = c)} \)

by (auto intro: SUP-eq-const SUP-upper order.antisym)

end

class `complete-distrib-lattice` = `complete-lattice` +
assumes `Inf-Sup-le`: \( \Inf (\Sup ' A) \leq \Sup (\Inf ' \{ f \cdot A \mid f . (\forall Y \in A . f Y \in Y) \}) \)

begin

lemma `Inf-Sup`: \( \Inf (\Sup ' A) = \Sup (\Inf ' \{ f \cdot A \mid f . (\forall Y \in A . f Y \in Y) \}) \)

by (rule order.antisym, rule Inf-Sup-le, rule Sup-Inf-le)

subclass `distrib-lattice`
proof

fix \( a \, b \, c \)
show \( a \bigsqcup b \cap c = (a \bigsqcup b) \cap (a \bigsqcup c) \)
proof (rule order.antisym, simp-all)

show \( b \cap c \leq a \bigsqcup b \)
by (rule le-infI1, simp)

show \( b \cap c \leq a \bigsqcup c \)
by (rule le-infI2, simp)

have [simp]: \( a \cap c \leq a \bigsqcup b \cap c \)
by (rule le-infI1, simp)

have [simp]: \( b \cap a \leq a \bigsqcup a \bigsqcup b \cap c \)
by (rule le-infI2, simp)

have \( \prod (\Sup ' \{ \{ a, b \}, \{ a, c \} \}) = \)
\( \bigsqcup (f \cdot \{ \{ a, b \}, \{ a, c \} \} \mid f . \forall Y \in \{ \{ a, b \}, \{ a, c \} \}. f Y \in Y) \)
by (rule Inf-Sup)
from this show \( (a \bigsqcup b) \cap (a \bigsqcup c) \leq a \bigsqcup b \cap c \)
apply simp
by (rule SUP-least, safe, simp-all)

qed

end
context complete-lattice
begin

context complete-lattice

fixes f :: 'a ⇒ 'b::complete-lattice
assumes mono f
begin

lemma mono-Inf: f (⨅ A) ≤ (⨅ x∈A. f x)
using ⟨mono f⟩ by (auto intro: complete-lattice-class.INF-greatest Inf-lower dest: monoD)

lemma mono-Sup: (⨆ x∈A. f x) ≤ f (⨆ A)
using ⟨mono f⟩ by (auto intro: complete-lattice-class.SUP-least Sup-upper dest: monoD)

lemma mono-INF: f (∏ i∈I. A i) ≤ (∏ x∈I. f (A x))
by (intro complete-lattice-class.INF-greatest monoD[OF ⟨mono f⟩] INF-lower)

lemma mono-SUP: (∐ x∈I. f (A x)) ≤ f (∑ i∈I. A i)
by (intro complete-lattice-class.SUP-least monoD[OF ⟨mono f⟩] SUP-upper)

end

end

class complete-boolean-algebra = boolean-algebra + complete-distrib-lattice
begin

lemma uminus-Inf: − (∏ A) = ∑ (uminus ' A)
proof (rule order:antisym)
  show − ∏ A ≤ ∑ (uminus ' A)
  by (rule compl-le-swap2, rule Inf-greatest, rule compl-le-swap2, rule Sup-upper)
simp
  show ∑ (uminus ' A) ≤ − ∏ A
  by (rule Sup-least, rule compl-le-swap1, rule Inf-lower) auto
qed

lemma uminus-INF: − (⨅ x∈A. B x) = (⨅ x∈A. − B x)
by (simp add: uminus-Inf image-image)

lemma uminus-Sup: − (∪ A) = ⨆ (uminus ' A)
proof
  have ∪ A = ⨆ (uminus ' A)
  by (simp add: image-image uminus-INF)
  then show ?thesis by simp
qed

lemma uminus-SUP: − (⨆ x∈A. B x) = (∏ x∈A. − B x)
by (simp add: uminus-Sup image-image)

end

class complete-linorder = linorder + complete-lattice
begin

lemma dual-complete-linorder:
  class.complete-linorder Sup Inf sup (≥) (> inf ⊤ ⊥
by (rule class.complete-linorder.intro, rule dual-complete-lattice, rule dual-linorder)

lemma complete-linorder-inf-min: inf = min
by (auto intro: order.antisym simp add: min-def fun-eq-iff)

lemma complete-linorder-sup-max: sup = max
by (auto intro: order.antisym simp add: max-def fun-eq-iff)

lemma Inf-less-iff: d S < a ←→ (∃x∈S. x < a)
by (simp add: not-le [symmetric] le-Inf-iff)

lemma INF-less-iff: (∀i∈A. f i) < a ←→ (∃x∈A. f x < a)
by (simp add: Inf-less-iff [of f ' A])

lemma less-Sup-iff: a < (∀x∈A. x < i)
by (simp add: less-Sup-iff [of - f ' A])

lemma Sup-eq-top-iff [simp]: (∀x∈A. x < i)
proof
  assume *: (∀x∈A = T)
  show (∀x∈T. ∃x∈A. x < i)
    unfolding * [symmetric]
  proof (intro allI impI)
    fix x
    assume x < (∀x∈A
    then show ∃i∈A. x < i
      by (simp add: less-Sup-iff)
  qed
next
  assume *: (∀x<i. ∃x∈A. x < i)
  show (∀x∈A = T)
  proof (rule ccontr)
    assume ∄A = T
    with top-greatest [of A] have ∄A < T
    unfolding le-less by auto
    with * have ∄A < (∀x∈A
    unfolding less-Sup-iff by auto
then show False by auto
qed
qed

lemma SUP-eq-top-iff [simp]: (⨆\(\bigcup\) A. f i) = ⊤ ←→ (∀ x<⊤. ∃ i∈A. x < f i)
using Sup-eq-top-iff [of f ' A] by simp

lemma Inf-eq-bot-iff [simp]: d A = ⊥ ←→ (∀ x>⊥. ∃ i∈A. i < x)
using dual-complete-linorder
by (rule complete-linorder.Sup-eq-top-iff)

lemma INF-eq-bot-iff [simp]: (⨅\(\bigcap\) i∈A. f i) = ⊥ ←→ (∀ x>⊥. ∃ i∈A. x < f i)
using Inf-eq-bot-iff [of f ' A]
by simp

lemma Inf-le-iff [simp]: d A ≤ x ≤→ (∀ y>x. ∃ a∈A. y > a)
proof safe
fix y
assume x ≥ d A
then have y > d A by auto
qed (auto elim!: allE[of - d A] simp: not-le[symmetric] Sup-upper)

lemma le-Sup-iff [simp]: x ≤ ∨\(\bigvee\) A ≤→ (∀ y<x. ∃ a∈A. y < a)
proof safe
fix y
assume x ≤ ∨\(\bigvee\) A
then have y < ∨\(\bigvee\) A by auto
then show ∃ a∈A. y < a
unfolding less-Sup-iff .
qed (auto elim!: allE[of - ∨\(\bigvee\) A] simp: not-le[symmetric] Sup-upper)

lemma le-SUP-iff [simp]: x ≤ (⨅ f ' A) ≤→ (∀ y<x. ∃ i∈A. y < f i)
using le-Sup-iff [of f ' A] by simp

end

11.3 Complete lattice on bool

instantiation bool :: complete-lattice

begin

definition [simp, code]: ∨\(\bigvee\) A ≤→ False \notin A

definition [simp, code]: (⨅ f ' A) ≤→ True \in A

instance
  by standard (auto intro: bool-induct)
end

lemma not-False-in-image-Ball [simp]: False ∉ P ∈ A ↔ Ball A P
  by auto

lemma True-in-image-Bex [simp]: True ∈ P ∈ A ↔ Bex A P
  by auto

lemma INF-bool-eq [simp]: (λA f. \( \bigcap \{f : A} \)) = Ball
  by (simp add: fun-eq-iff)

lemma SUP-bool-eq [simp]: (λA f. \( \bigcup \{f : A} \)) = Bex
  by (simp add: fun-eq-iff)

instance bool :: complete-boolean-algebra
  by (standard, fastforce)

11.4 Complete lattice on - ⇒ -

instantiation fun :: (type, Inf) Inf
begin

definition \( \bigcap \) A = (λx. \( \bigcap \{f : A} f x) = \prod

lemma Inf-apply [simp, code]: (\( \bigcap \) A) x = (\( \bigcap \) f : A f x)
  by (simp add: Inf-fun-def)

instance ..
end

instantiation fun :: (type, Sup) Sup
begin

definition \( \bigcup \) A = (λx. \( \bigcup \{f : A} f x) = \prod

lemma Sup-apply [simp, code]: (\( \bigcup \) A) x = (\( \bigcup \) f : A f x)
  by (simp add: Sup-fun-def)

instance ..
end

instantiation fun :: (type, complete-lattice) complete-lattice
begin
instance
  by standard (auto simp add: le-fun-def intro: INF-lower INF-greatest SUP-upper SUP-least)
end

lemma INF-apply [simp]: ((\prod y \in A. f y) x = (\prod y \in A. f y x)
  by (simp add: image-comp)
lemma SUP-apply [simp]: ((\bigcup y \in A. f y) x = (\bigcup y \in A. f y x)
  by (simp add: image-comp)

11.5 Complete lattice on unary and binary predicates

lemma Inf1-I: (\forall P. P \in A \implies P a) \implies (\prod A) a
  by auto
lemma INF1-I: (\forall x. x \in A \implies B x b) \implies (\prod x \in A. B x) b
  by simp
lemma INF2-I: (\forall x. x \in A \implies B x b c) \implies (\prod x \in A. B x) b c
  by simp
lemma Inf2-I: (\forall r. r \in A \implies r a b) \implies (\prod A) a b
  by auto
lemma Inf1-D: (\prod A) a \implies P \in A \implies P a
  by auto
lemma INF1-D: (\prod x \in A. B x) b \implies a \in A \implies B a b
  by simp
lemma Inf2-D: (\prod A) a b \implies r \in A \implies r a b
  by auto
lemma INF2-D: (\prod x \in A. B x) b c \implies a \in A \implies B a b c
  by simp

lemma Inf1-E:
  assumes (\prod A) a
  obtains P a | P \notin A
  using assms by auto
lemma INF1-E:
  assumes (\prod x \in A. B x) b
  obtains B a b | a \notin A
  using assms by auto
lemma Inf2-E:
assumes \((\bigcap A)\) \(a\) \(b\)
obtains \(r\) \(a\) \(b\) \mid \(r \notin A\)
using assms by auto

lemma INF2-E:
assumes \((\bigcap x \in A. B x)\) \(b\) \(c\)
obtains \(B\) \(a\) \(b\) \(c\) \mid \(a \notin A\)
using assms by auto

lemma Sup1-I: \(P \in A \implies P\) \(a\) \implies (\(\bigcup A\)) \(a\)
by auto

lemma SUP1-I: \(a \in A \implies B\) \(a\) \(b\) \implies (\(\bigcup x \in A. B x\)) \(b\)
by auto

lemma Sup2-I: \(r \in A \implies r\) \(a\) \(b\) \implies (\(\bigcup A\)) \(a\) \(b\)
by auto

lemma SUP2-I: \(a \in A \implies B\) \(a\) \(b\) \(c\) \implies (\(\bigcup x \in A. B x\)) \(b\) \(c\)
by auto

lemma Sup1-E:
assumes (\(\bigcup A\)) \(a\)
obtains \(P\) where \(P \in A\) and \(P\) \(a\)
using assms by auto

lemma SUP1-E:
assumes (\(\bigcup x \in A. B x\)) \(b\)
obtains \(x\) where \(x \in A\) and \(B\) \(x\) \(b\)
using assms by auto

lemma Sup2-E:
assumes (\(\bigcup A\)) \(a\) \(b\)
obtains \(r\) where \(r \in A\) \(r\) \(a\) \(b\)
using assms by auto

lemma SUP2-E:
assumes (\(\bigcup x \in A. B x\)) \(b\) \(c\)
obtains \(x\) where \(x \in A\) \(B\) \(x\) \(b\) \(c\)
using assms by auto

11.6 Complete lattice on - set

instantiation set :: (type) complete-lattice
begin

definition \(\bigcap A\) = \(\{x. \bigcap((\lambda B. x \in B) ^{\cdot} A)\}\)

definition \(\bigcup A\) = \(\{x. \bigcup((\lambda B. x \in B) ^{\cdot} A)\}\)
instance
  by standard (auto simp add: less-eq-set-def Inf-set-def Sup-set-def le-fun-def)

end

11.6.1 Inter

abbreviation Inter :: "'a set ⇒ 'a set (⋂)
  where ⋂ S ≡ d S

lemma Inter-eq:
  ⋂ A = {x. ∀B ∈ A. x ∈ B}
proof (rule set-eqI)
  fix x
  have (∀Q∈{P. ∃B∈A. P ✈ x ∈ B}. Q) ✈ (∀B∈A. x ∈ B)
    by auto
  then show x ∈ ⋂ A ✈ x ∈ {x. ∀B ∈ A. x ∈ B}
    by (simp add: Inf-set-def image-def)
  qed

lemma Inter-iff [simp]: A ∈ ⋂ C ✈ (∀X ∈ C. A ∈ X)
proof (unfold Inter-eq)
  blast

lemma InterI [intro!]: (∀X. X ∈ C ✈ A ∈ X) ✈ A ∈ ⋂ C
proof (simp add: Inter-eq)

A "destruct" rule – every X in C contains A as an element, but A ∈ X can hold when X ∈ C does not! This rule is analogous to spec.

lemma InterD [elim, Pure.elim]: A ∈ ⋂ C ✈ X ∈ C ✈ A ∈ X
proof auto

lemma InterE [elim]: A ∈ ⋂ C ✈ (X ∉ C ✈ R) ✈ (A ∈ X ✈ R) ✈ R
  — "Classical" elimination rule – does not require proving X ∈ C.
unfolding Inter-eq by blast

lemma Inter-lower: B ∈ A ✈ ⋂ A ⊆ B
proof (fact Inf-lower)

lemma Inter-subset: (∀X. X ∈ A ✈ X ⊆ B) ✈ A ≠ {} ✈ ⋂ A ⊆ B
proof (fact Inf-less-eq)

lemma Inter-greatest: (∀X. X ∈ A ✈ C ⊆ X) ✈ C ⊆ ⋂ A
proof (fact Inf-greatest)

lemma Inter-empty: ⋂ {} = UNIV
proof (fact Inf-empty)

lemma Inter-UNIV: ⋂ UNIV = {}
proof (fact Inf-UNIV)
lemma Inter-insert: \( \bigcap (\text{insert } a B) = a \cap \bigcap B \)
by (fact Inf-insert)

lemma Inter-Un-subset: \( \bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B) \)
by (fact less-eq-Inf-inter)

lemma Inter-Un-distrib: \( \bigcap (A \cup B) = \bigcap A \cap \bigcap B \)
by (fact Inf-union-distrib)

lemma Inter-UNIV-conv [simp]:
\( \bigcap A = \text{UNIV} \iff (\forall x \in A. x = \text{UNIV}) \)
\( \text{UNIV} = \bigcap A \iff (\forall x \in A. x = \text{UNIV}) \)
by (fact Inf-top-conv)

lemma Inter-anti-mono:
\( B \subseteq A \Rightarrow \bigcap A \subseteq \bigcap B \)
by (fact Inf-superset-mono)

11.6.2 Intersections of families

syntax (ASCII)
\[-\text{INTER1} :: \text{pttrns} \Rightarrow 'b set \Rightarrow 'b set \ ((3\text{INT }-./) [0, 10] 10)\]
\[-\text{INTER} :: \text{pttrn} \Rightarrow 'a set \Rightarrow 'b set \Rightarrow 'b set \ ((3\text{INT }-\in-/ -) [0, 0, 10] 10)\]

syntax (latex output)
\[-\text{INTER1} :: \text{pttrns} \Rightarrow 'b set \Rightarrow 'b set \ ((3\bigcap \langle\text{unbreakable}\rangle -/) [0, 10] 10)\]
\[-\text{INTER} :: \text{pttrn} \Rightarrow 'a set \Rightarrow 'b set \Rightarrow 'b set \ ((3\bigcap \langle\text{unbreakable}\rangle -\in/ -) [0, 0, 10] 10)\]

translations
\( \bigcap x y. f = \bigcap x. \bigcap y. f \)
\( \bigcap x. f = \bigcap (\text{CONST range } (\lambda x. f)) \)
\( \bigcap x \in A. f = \text{CONST Inter } ((\lambda x. f) \cdot A) \)

lemma INTER-eq: \( (\bigcap x \in A. B x) = \{ y. \forall x \in A. y \in B x \} \)
by (auto intro!: INF-eql)

lemma INT-iff [simp]: \( b \in (\bigcap x \in A. B x) \iff (\forall x \in A. b \in B x) \)
using Inter-iff [of - B \cdot A] by simp

lemma INT-I [intro!]: \( (\forall x \in A \Rightarrow b \in B x) \Rightarrow b \in (\bigcap x \in A. B x) \)
by auto

lemma INT-D [elim, Pure.elim]: \( b \in (\bigcap x \in A. B x) \Rightarrow a \in A \Rightarrow b \in B a \)
by auto

lemma INT-E [elim]: \( b \in (\bigcap x \in A. B x) \implies (b \in B a \implies R) \implies (a \notin A \implies R) \)
\[ \implies R \]
— 'Classical' elimination – by the Excluded Middle on \( a \in A \).
by auto

lemma Collect-ball-eq: \( \{ x. \forall y \in A. P x y \} = (\bigcap y \in A. \{ x. P x y \}) \)
by blast

lemma Collect-all-eq: \( \{ x. \forall y. P x y \} = (\bigcap y. \{ x. P x y \}) \)
by blast

lemma INT-lower: \( a \in A \implies (\bigcap x \in A. B x) \subseteq B a \)
by (fact INF-lower)

lemma INT-greatest: \( (\forall x. x \in A \implies C \subseteq B x) \implies C \subseteq (\bigcap x \in A. B x) \)
by (fact INF-greatest)

lemma INT-empty: \( (\bigcap x \in \emptyset. B x) = \text{UNIV} \)
by (fact INF-empty)

lemma INT-absorb: \( k \in I \implies A k \cap (\bigcap i \in I. A i) = (\bigcap i \in I. A i) \)
by (fact INF-absorb)

lemma INT-subset-iff: \( B \subseteq (\bigcap i \in I. A i) \iff (\forall i \in I. B \subseteq A i) \)
by (fact le-INF-iff)

lemma INT-insert [simp]: \( (\bigcap x \in \text{insert} a A. B x) = B a \cap (\bigcap x \in A. B x) \)
by (fact INF-insert)

lemma INT-Un: \( (\bigcap i \in A \cup B. M i) = (\bigcap i \in A. M i) \cap (\bigcap i \in B. M i) \)
by (fact INF-union)

lemma INT-insert-distrib: \( a \in A \implies (\bigcap x \in A. \text{insert} a (B x)) = \text{insert} a (\bigcap x \in A. B x) \)
by blast

lemma INT-constant [simp]: \( (\bigcap y \in A. c) = (\text{if } A = \{ \} \text{ then } \text{UNIV} \text{ else } c) \)
by (fact INF-constant)

lemma INTER-UNIV-conv:
(UNIV = (\bigcap x \in A. B x)) = (\forall x \in A. B x = \text{UNIV})
((\bigcap x \in A. B x) = \text{UNIV}) = (\forall x \in A. B x = \text{UNIV})
by (fact INF-top-conv)

lemma INT-bool-eq: \( (\bigcap b. A b) = A \text{ True } \cap A \text{ False} \)
by (fact INF-UNIV-bool-expand)
lemma \textit{INT-anti-mono}: \( A \subseteq B \Rightarrow (\forall x. x \in A \Rightarrow f x \subseteq g x) \Rightarrow (\bigcap x \in B. f x) \subseteq (\bigcap x \in A. g x) \)

— The last inclusion is \textit{POSITIVE}!

by (fact INF-superset-mono)

lemma \textit{Pow-INT-eq}: \( \text{Pow} (\bigcap x \in A. B x) = (\bigcap x \in A. \text{Pow} (B x)) \)

by blast

lemma \textit{vimage-INT}: \( f \mapsto (\bigcap x \in A. B x) = (\bigcap x \in A. f \mapsto B x) \)

by blast

11.6.3 Union

abbreviation \textit{Union} :: 

where \( \bigcup S \equiv \bigsqcup S \)

lemma \textit{Union-eq}: \( \bigcup A = \{ x. \exists B \in A. x \in B \} \)

proof (rule set-eqI)

fix \( x \)

have \( (\exists Q \in \{ P. \exists B \in A. P \leftrightarrow x \in B \}. Q) \leftrightarrow (\exists B \in A. x \in B) \)

by auto

then show \( x \in \bigcup A \leftrightarrow x \in \{ x. \exists B \in A. x \in B \} \)

by (simp add: Sup-set-def image-def)

qed

lemma \textit{Union-iff \ [simp]}: \( A \in \bigcup C \leftrightarrow (\exists X \in C. A \in X) \)

by (unfold Union-eq) blast

lemma \textit{UnionI \ [intro]}: \( X \in C \Longrightarrow A \in X \Longrightarrow A \in \bigcup C \)

— The order of the premises presupposes that \( C \) is rigid; \( A \) may be flexible.

by auto

lemma \textit{UnionE \ [elim]}: \( A \in \bigcup C \Longrightarrow (\forall X. A \in X \Longrightarrow X \in C \Longrightarrow R) \Longrightarrow R \)

by auto

lemma \textit{Union-upper}: \( B \in A \Longrightarrow B \subseteq \bigcup A \)

by (fact Sup-upper)

lemma \textit{Union-least}: \( (\forall X. X \in A \Longrightarrow X \subseteq C) \Longrightarrow \bigcup A \subseteq C \)

by (fact Sup-least)

lemma \textit{Union-empty}: \( \bigcup \{ \} = \{ \} \)

by (fact Sup-empty)

lemma \textit{Union-UNIV}: \( \bigcup \text{UNIV} = \text{UNIV} \)

by (fact Sup-UNIV)

lemma \textit{Union-insert}: \( \bigcup (\text{insert} \ a \ B) = a \cup \bigcup B \)

by (fact Sup-insert)
lemma Union-Un-distrib [simp]: $\bigcup (A \cup B) = \bigcup A \cup \bigcup B$
  by (fact Sup-union-distrib)

lemma Union-Int-subset: $\bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B$
  by (fact Sup-inter-less-eq)

lemma Union-empty-conv: $(\bigcup A = \{\}) \iff (\forall x \in A. \ x = \{\})$
  by (fact Sup-bot-conv)

lemma empty-Union-conv: $(\{\} = \bigcup A) \iff (\forall x \in A. \ x = \{\})$
  by (fact Sup-bot-conv)

lemma subset-Pow-Union: $A \subseteq \Pow(\bigcup A)$
  by blast

lemma Union-Pow-eq [simp]: $\bigcup (\Pow A) = A$
  by blast

lemma Union-mono: $A \subseteq B \Rightarrow \bigcup A \subseteq \bigcup B$
  by (fact Sup-subset-mono)

lemma Union-subsetI: $(\forall x \in A \Rightarrow \exists y \in B \land x \subseteq y) \Rightarrow \bigcup A \subseteq \bigcup B$
  by blast

lemma disjnt-inj-on-iff:
  $\inj-on f (\bigcup A); X \in A; Y \in A \Rightarrow \disjnt (f ' X) (f ' Y) \iff \disjnt X Y$

  unfolding disjnt-def
  by safe (use inj-on-eq-iff in ‹fastforce›)

lemma disjnt-Union1 [simp]: $\disjnt (\bigcup A) B \iff (\forall A \in A. \ \disjnt A B)$
  by (auto simp: disjnt-def)

lemma disjnt-Union2 [simp]: $\disjnt B (\bigcup A) \iff (\forall A \in A. \ \disjnt B A)$
  by (auto simp: disjnt-def)

11.6.4 Unions of families

syntax (ASCII)
  -UNION1 :: pttrns => 'b set => 'b set  \((\text{UN} \ -\ /) \ [0, 10] \ 10)\)
  -UNION  :: pttrn => 'a set => 'b set => 'b set  \((\text{UN} \ -\ / \ -) \ [0, 0, 10] \ 10)\)

syntax
  -UNION1 :: pttrns => 'b set => 'b set  \((\text{UN} \ -\ /) \ [0, 10] \ 10)\)
  -UNION  :: pttrn => 'a set => 'b set => 'b set  \((\text{UN} \ -\ / \ -) \ [0, 0, 10] \ 10)\)

syntax (latex output)
  -UNION1 :: pttrns => 'b set => 'b set  \((\text{UN} \ (-)\ / \ -) \ [0, 10] \ 10)\)
-UNION :: pttrn => 'a set => 'b set ((\union{unbreakable}. _)/ -) [0, 0, 10] 10

translations
\union x y. f  :=  \union x. \union y. f
\union x. f  :=  \union(\CONST range (\lambda x. f))
\union x\in A. f  :=  \CONST Union ((\lambda x. f) ' A)

Note the difference between ordinary syntax of indexed unions and intersections (e.g. \union a\in A_1. B) and their \LaTeX rendition: \union a\in A_1 B.

lemma disjoint-UN-iff: disjoint A (\union i\in I. B i) \iff (\forall i\in I. disjoint A (B i))
  by (auto simp: disjoint-def)

lemma UNION-eq: (\union x\in A. B x) = \{ y. \exists x\in A. y \in B x\}
  by (auto intro!: SUP-eqI)

lemma bind-UNION [code]: Set.bind A f = \union (f ' A)
  by (simp add: bind-def UNION-eq)

lemma member-bind [simp]: x \in Set.bind A f \iff x \in \union (f ' A)
  by (simp add: bind-UNION)

lemma Union-SetCompr-eq: \union {f x| x. P x} = \{ a. \exists x. P x \land a \in f x\}
  by blast

lemma UN-iff [simp]: b \in (\union x\in A. B x) \iff (\exists x\in A. b \in B x)
  using Union-iff \[ of - B ' A \] by simp

lemma UN-I [intro]: a \in A \implies b \in B a \implies b \in (\union x\in A. B x)
  — The order of the premises presupposes that A is rigid; b may be flexible.
  by auto

lemma UN-E [elim!]: b \in (\union x\in A. B x) \implies (\forall x. x\in A \implies b \in B x \implies R) \implies R
  by auto

lemma UN-upper: a \in A \implies B a \subseteq (\union x\in A. B x)
  by (fact SUP-upper)

lemma UN-least: (\forall x. x\in A \implies B x \subseteq C) \implies (\union x\in A. B x) \subseteq C
  by (fact SUP-least)

lemma Collect-bex-eq: \{ x. \exists y\in A. P x y \} = (\union y\in A. \{ x. P x y\})
  by blast

lemma UN-insert-distrib: u \in A \implies (\union x\in A. insert a (B x)) = insert a (\union x\in A. B x)
  by blast
lemma **UN-empty**: \( \bigcup x \in \{ \}. B x \) = \{ \\
by (fact SUP-empty)

lemma **UN-empty2**: \( \bigcup x \in A. \{ \} \) = \{ \\
by (fact SUP-bot)

lemma **UN-absorb**: \( k \in I \implies A k \cup \bigcup i \in I. A i \) = (\( \bigcup i \in I. A i \) \\
by (fact SUP-absorb)

lemma **UN-insert**: \( \bigcup x \in insert a A. B x \) = B a \cup \bigcup (B \cdot A) \\
by (fact SUP-insert)

lemma **UN-Un**: \( \bigcup i \in A \cup B. M i \) = (\( \bigcup i \in A. M i \) \cup \( \bigcup i \in B. M i \) \\
by (fact SUP-union)

lemma **UN-UN-flatten**: \( \bigcup x \in (\bigcup y \in A. B y). C x \) = (\( \bigcup y \in A. \bigcup x \in B y. C x \) \\
by blast

lemma **UN-subset-iff**: (\( \bigcup i \in I. A i \subseteq B \) \( \iff \) (\( \forall i \in I. A i \subseteq B \) \\
by (fact SUP-le-iff)

lemma **UN-constant**: \( \bigcup y \in A. c \) = (if A = \{ \} then \{ \} else c) \\
by (fact SUP-constant)

lemma **UNION-singleton-eq-range**: \( \bigcup x \in A. \{ f x \} \) = f \cdot A \\
by blast

lemma **image-Union**: f \cdot \bigcup S = (\( \bigcup x \in S. f \cdot x \) \\
by blast

lemma **UNION-empty-conv**: \\
(\( \bigcup x \in A. B x \) \iff (\( \forall x \in A. B x \) = \{ \} \\
by (fact SUP-bot-conv)

lemma **Collect-ex-eq**: \{x. \exists y. P x y\} = (\( \bigcup y. \{ x. P x y \} \) \\
by blast

lemma **ball-UN**: (\( \forall z \in \bigcup (B \cdot A). P z \) \iff (\( \forall x \in A. \forall z \in B x. P z \) \\
by blast

lemma **bex-UN**: (\( \exists z \in \bigcup (B \cdot A). P z \) \iff (\( \exists x \in A. \exists z \in B x. P z \) \\
by blast

lemma **Un-eq-UN**: A \cup B = (\( \bigcup b. if b then A else B \) \\
by safe (auto simp add: if-split-mem2)

lemma **UN-bool-eq**: (\( \bigcup b. A b \) = (A True \cup A False) \\
by (fact SUP-UNIV-bool-expand)
lemma UN-Pow-subset: \((\bigcup x \in A. \text{Pow} (B x)) \subseteq \text{Pow} \ (\bigcup x \in A. B x)\)
by blast

lemma UN-mono:
\[A \subseteq B \implies (\bigwedge x. x \in A \implies f x \subseteq g x) \implies \]
\[(\bigcup x \in A. f x) \subseteq (\bigcup x \in B. g x)\]
by (fact SUP-subset-mono)

lemma vimage-Union: \(f ^-' (\bigcup A) = (\bigcup X \in A. f ^-' X)\)
by blast

lemma vimage-UN: \(f ^-' (\bigcup x \in A. B x) = (\bigcup x \in A. f ^-' B x)\)
by blast

lemma vimage-eq-UN: \(f ^-' B = (\bigcup y \in B. f ^-' \{y\})\)
— NOT suitable for rewriting
by blast

lemma image-UN: \(f ^ ' \bigcup (B ^ ' A) = (\bigcup x \in A. f ^ ' B x)\)
by blast

lemma UN-singleton [simp]: \((\bigcup x \in A. \{x\}) = A\)
by blast

lemma inj-on-image: inj-on f (\bigcup A) \implies inj-on (f) A
unfolding inj-on-def by blast

11.6.5 Distributive laws

lemma Int-Union: \(A \cap \bigcup B = (\bigcup C \in B. A \cap C)\)
by blast

lemma Un-Inter: \(A \cup \bigcap B = (\bigcap C \in B. A \cup C)\)
by blast

lemma Int-Union2: \(\bigcup B \cap A = (\bigcup C \in B. C \cap A)\)
by blast

lemma INT-Int-distrib: \((\bigcap i \in I. A i \cap B i) = (\bigcap i \in I. A i) \cap (\bigcap i \in I. B i)\)
by (rule sgm) (rule INF-inf-distrib)

lemma UN-Un-distrib: \((\bigcup i \in I. A i \cup B i) = (\bigcup i \in I. A i) \cup (\bigcup i \in I. B i)\)
by (rule sgm) (rule SUP-sup-distrib)

lemma Int-Inter-image: \((\bigcap x \in C. A x \cap B x) = \bigcap (A ^-' C) \cap \bigcap (B ^-' C)\)
by (simp add: INT-Int-distrib)

lemma Int-Inter-eq: \(A \cap \bigcap B = (\text{if } B=\{\} \text{ then } A \text{ else } (\bigcap B \in B. A \cap B))\)
THEORY “Complete-Lattices”

\[\bigcap B \cap A = (\text{if } B = \{\} \text{ then } A \text{ else } (\bigcap B \in B. B \cap A))\]

by auto

lemma Un-Union-image: \((\bigcup x \in C. A x \cup B x) = \bigcup (A \setminus C) \cup \bigcup (B \setminus C)\)
— Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:
— Union of a family of unions
by (simp add: UN-Un-distrib)

lemma Un-INT-distrib: \(B \cup (\bigcap i \in I. A i) = (\bigcap i \in I. B \cap A i)\)
by blast

lemma Int-UN-distrib: \(B \cap (\bigcup i \in I. A i) = (\bigcup i \in I. B \cup A i)\)
— Halmos, Naive Set Theory, page 35.
by blast

lemma Int-UN-distrib2: \((\bigcup i \in I. A i) \cap (\bigcup j \in J. B j) = (\bigcup i \in I. \bigcup j \in J. A i \cap B j)\)
by blast

lemma Un-INT-distrib2: \((\bigcap i \in I. A i) \cup (\bigcup j \in J. B j) = (\bigcup i \in I. \bigcap j \in J. A i \cup B j)\)
by blast

lemma Union-disjoint: \((\bigcup C \cap A = \{\}) \iff (\forall B \in C. B \cap A = \{\})\)
by blast

lemma SUP-UNION: \((\bigcup x \in (\bigcup y \in A. g y). f x) = (\bigcup y \in A. \bigcup x \in g y. f x :: - ::)\)
complete-lattice
by (rule order-antisym) (blast intro: SUP-least SUP-upper2)+

11.7 Injections and bijections

lemma inj-on-Inter: \(S \neq \{\} \Rightarrow (\forall A. A \in S \Rightarrow \text{inj-on } f A) \Rightarrow \text{inj-on } f (\bigcap S)\)
unfolding inj-on-def by blast

lemma inj-on-INTER: \(I \neq \{\} \Rightarrow (\forall i. i \in I \Rightarrow \text{inj-on } f (A i)) \Rightarrow \text{inj-on } f (\bigcap i \in I. A i)\)
unfolding inj-on-def by safe simp

lemma inj-on-UNION-chain:
assumes chain: \(\wedge i j. i \in I \Rightarrow j \in I \Rightarrow A i \leq A j \lor A j \leq A i\)
and inj: \(\forall i. i \in I \Rightarrow \text{inj-on } f (A i)\)
sf
shows inj-on f (\bigcup i \in I. A i)
proof -
have x = y
  if *: i \in I. j \in I
  and **: x \in A. i y \in A j
  and ***: f x = f y
for i j x y

using chain [OF *]
proof
assume A i ≤ A j
with * have x ∈ A j by auto
with inj * * show ?thesis
  by (auto simp add: inj-on-def)
next
assume A j ≤ A i
with * have y ∈ A i by auto
with inj * * show ?thesis
  by (auto simp add: inj-on-def)
qed
then show ?thesis
  by (unfold inj-on-def UNION-eq) auto
qed

lemma bij-betw-UNION-chain:
  assumes chain: ∏ i j. i ∈ I =⇒ j ∈ I =⇒ A i ≤ A j ∨ A j ≤ A i
  and bij: ∏ i. i ∈ I =⇒ bij-betw f (A i) (A' i)
  shows bij-betw f (∪ i ∈ I. A i) (∪ i ∈ I. A' i)
unfolding bij-betw-def
proof safe
  have ∏ i. i ∈ I =⇒ inj-on f (A i)
    using bij bij-betw-def[of f] by auto
  then show inj-on f (∪ (A' I))
    using chain inj-on-UNION-chain[of I A f] by auto
next
fix i x
assume *: i ∈ I x ∈ A i
with bij have f x ∈ A' i
  by (auto simp: bij-betw-def)
with * show f x ∈ ∪ (A' I) by blast
next
fix i x'
assume *: i ∈ I x' ∈ A' i
with bij have ∃ x ∈ A i. x' = f x
  unfolding bij-betw-def by blast
with * have ∃ j ∈ I. ∃ x ∈ A j. x' = f x
  by blast
then show x' ∈ f ∪ (A' I)
  by blast
qed

lemma image-INT: inj-on f C =⇒ ∃ x ∈ A. B x ⊆ C =⇒ j ∈ A =⇒ f' (j ∩ (B x))
= (∪ x ∈ A. f' B x)
  by (auto simp add: inj-on-def) blast

lemma bij-image-INT: bij f =⇒ f' (∩ (B x)) = (∪ x ∈ A. f' B x)
by (auto simp: bij-def inj-def surj-def) blast

lemma UNION-fun-upd: \( \bigcup (A(i := B) \cdot J) = \bigcup (A \cdot (J - \{i\})) \cup (if \ i \in \ J \ then \ B \ else \ \{\}) \)
by (auto simp add: set-eq-iff)

lemma bij-betw-Pow:
assumes bij-betw f A B
shows bij-betw (image f) (Pow A) (Pow B)
proof
from assms have inj-on f A
by (rule bij-betw-imp-inj-on)
then have inj-on f (\bigcup (Pow A))
by simp
then have inj-on (image f) (Pow A)
by (rule inj-on-image)
then have bij-betw (image f) (Pow A) (image f \cdot Pow A)
by (rule inj-on-imp-bij-betw)
moreover from assms have f \cdot A = B
by (rule bij-betw-imp-surj-on)
then have image f \cdot Pow A = Pow B
by (rule image-Pow-surj)
ultimately show \?thesis by simp
qed

11.7.1 Complement

lemma Compl-INT [simp]: \( - (\bigcap x \in A. B x) = (\bigcup x \in A. -B x) \)
by blast

lemma Compl-UN [simp]: \( - (\bigcup x \in A. B x) = (\bigcap x \in A. -B x) \)
by blast

11.7.2 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

lemma UN-simps [simp]:
\( \bigwedge a B C. (\bigcup x \in C. \ insert a (B x)) = (if C={\{\} then \{\} else insert a (\bigcup x \in C. B x)) \)
\( \bigwedge a B C. (\bigcup x \in C. A x \cup B) = ((if C={\{\} then \{\} else (\bigcup x \in C. A x) \cup B)) \)
\( \bigwedge a B C. (\bigcup x \in C. A x \cap B) = ((if C={\{\} then \{\} else A \cup (\bigcup x \in C. B x)) \)
\( \bigwedge a B C. (\bigcup x \in C. A x - B) = (A \cap (\bigcup x \in C. B x)) \)
\( \bigwedge \ A B C. (\bigcup x \in C. A x - B) = (\bigcup x \in C. A x) - B) \)
\( \bigwedge \ A B C. (\bigcup x \in C. A - B x) = (A - (\bigcup x \in C. B x)) \)
\( \bigwedge \ A B. (\bigcup x \in A. B x) = (\bigcup y \in A. \bigcup x \in y. B x) \)
\( \bigwedge \ A B C. (\bigcup z \in (\bigcup (B \cdot A)). C z) = (\bigcup x \in A. \bigcup z \in B x. C z) \)
\( \bigwedge \ A B f. (\bigcup x \in f A. B x) = (\bigcup a \in A. B (f a)) \)
by auto
THEORY "Complete-Lattices"

lemma INT-simps [simp]:
\( \forall A. B. C. (\bigcap x \in C. \ A \ x \cap B) = (\{ C = \{ \} \ \text{then \ UNIV} \ \text{else} \ (\bigcap x \in C. \ A \ x) \cap B) \)
\( \forall A. B. C. (\bigcap x \in C. \ A \cap B \ x) = (\{ C = \{ \} \ \text{then \ UNIV} \ \text{else} \ A \cap (\bigcap x \in C. \ B \ x)) \)
\( \forall A. B. C. (\bigcap x \in C. \ A \ x - B) = (\{ C = \{ \} \ \text{then \ UNIV} \ \text{else} \ (\bigcap x \in C. \ A \ x) - B) \)
\( \forall A. B. C. (\bigcap x \in C. \ A - B \ x) = (\{ C = \{ \} \ \text{then \ UNIV} \ \text{else} \ A - (\bigcup x \in C. \ B \ x)) \)
\( \forall a. B. C. (\bigcup x \in C. \ \text{insert} \ a \ (\bigcup x \in C. \ B \ x)) = \text{insert} \ a \ (\bigcup x \in C. \ B \ x) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \ x \cup B) = (\{ C = \{ \} \ \text{then \ UNIV} \ \text{else} \ (\bigcup x \in C. \ A \ x) \cup B) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \cup B \ x) = (\{ C = \{ \} \ \text{then \ UNIV} \ \text{else} \ (\bigcup x \in C. \ A \ x) \cup B \ x) \)
\( \forall A. B. C. (\bigcup x \in \bigcup A. \ B \ x) = (\{ \ y \in A. \ \bigcup x \in y. \ B \ x \) \)
\( \forall A. B. C. (\bigcup x \in (\bigcup (B \ A))). \ C \ z) = (\bigcup x \in A. \ \bigcap z \in B \ x. \ C \ z) \)
\( \forall A. B. f. (\bigcup x \in f' A. \ B \ x) = (\{ a \in A. \ B \ (f \ a)) \)
by auto

lemma UN-ball-bex-simps [simp]:
\( \forall A. B. P. (\forall x \in \bigcup A. \ P \ x) \leftrightarrow (\forall y \in A. \ \bigcup x \in y. \ P \ x) \)
\( \forall A. B. P. (\forall x \in (\bigcup (B \ A))). \ P \ x) = (\forall a \in A. \ \forall x \in B \ a. \ P \ x) \)
\( \forall A. B. P. (\exists x \in \bigcup A. \ P \ x) \leftrightarrow (\exists y \in A. \ \exists x \in y. \ P \ x) \)
\( \forall A. B. P. (\exists x \in (\bigcup (B \ A))). \ P \ x) \leftrightarrow (\exists a \in A. \ \exists x \in B \ a. \ P \ x) \)
by auto

Maxiscop: pulling out big Unions and Intersections.

lemma UN-extend-simps:
\( \forall a. B. C. \ \text{insert} \ a \ (\bigcup x \in C. \ B \ x) = (\{ C = \{ \} \ \text{then} \ \{ a \} \ \text{else} \ (\bigcup x \in C. \ \text{insert} \ a \ (\bigcup x \in C. \ B \ x)) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \ x \cup B) = (\{ C = \{ \} \ \text{then} \ B \ \text{else} \ (\bigcup x \in C. \ A \ x \cup B) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \ x \cap B) = (\{ C = \{ \} \ \text{then} \ A \ \text{else} \ (\bigcup x \in C. \ A \ x \cap B) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \ x \cap B \ x) = (\{ C = \{ \} \ \text{then} \ A \ \text{else} \ (\bigcup x \in C. \ A \ x \cap B \ x) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \ x - B) = (\{ C = \{ \} \ \text{then} \ A \ \text{else} \ (\bigcup x \in C. \ A \ x - B) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \ x - B \ x) = (\{ C = \{ \} \ \text{then} \ A \ \text{else} \ (\bigcup x \in C. \ A \ x - B \ x) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \ x) \cap B = (\{ C = \{ \} \ \text{then} \ B \ \text{else} \ (\bigcup x \in C. \ A \ x \cap B) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \ x \cap B \ x) = (\{ C = \{ \} \ \text{then} \ A \ \text{else} \ (\bigcup x \in C. \ A \ x \cap B \ x) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \ x) - B = (\{ C = \{ \} \ \text{then} \ UNIV \ \text{else} \ (\bigcup x \in C. \ A \ x - B) \)
\( \forall A. B. C. (\bigcup x \in C. \ A \ x) - B \ x) = (\{ C = \{ \} \ \text{then} \ A \ \text{else} \ (\bigcup x \in C. \ A \ x - B \ x) \)
\( \forall a. B. C. (\bigcup x \in C. \ A \ x \cap B) = (\{ C = \{ \} \ \text{then} \ B \ \text{else} \ (\bigcup x \in C. \ A \ x \cap B) \)
\( \forall a. B. C. (\bigcup x \in C. \ A \ x \cap B \ x) = (\{ C = \{ \} \ \text{then} \ A \ \text{else} \ (\bigcup x \in C. \ A \ x \cap B \ x) \)
\( \forall a. B. C. (\bigcup x \in C. \ A \ x) \cap B = (\{ C = \{ \} \ \text{then} \ B \ \text{else} \ (\bigcup x \in C. \ A \ x) \cap B) \)
\( \forall A. B. C. (\bigcup x \in (\bigcup (B \ A))). \ C \ z) = (\bigcup z \in (\bigcup (B \ A))). \ C \ z) \)
\( \forall A. B. f. (\bigcup x \in a' A. \ B \ (f \ a)) = (\bigcup x \in f' A. \ B \ x) \)
by auto
Finally

**lemmas mem-simps =**
insert-iff empty-iff Un-iff Int-iff Compl-iff Diff-iff
mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff
— Each of these has ALREADY been added [simp] above.

**end**

12 Wrapping Existing Freely Generated Type’s Constructors

theory *Ctr-Sugar*
imports HOL
keywords
  print-case-translations :: diag and
  free-constructors :: thy-goal
begin

consts
case-guard :: bool ⇒ 'a ⇒ ('a ⇒ 'b) ⇒ 'b
case-nil :: 'a ⇒ 'b
case-cons :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b
case-elem :: 'a ⇒ 'b ⇒ 'a ⇒ 'b
case-abs :: ('c ⇒ 'b) ⇒ 'b

declare [[coercion-args case-guard − + −]]
declare [[coercion-args case-cons − −]]
declare [[coercion-args case-abs −]]
declare [[coercion-args case-elem − +]]

ML-file ⟨Tools/Ctr-Sugar/case-translation.ML⟩

lemma iffI-np: \[ x \implies \neg y; \neg x \implies y \implies \neg x \iff y \]
by (erule iffI) (erule contrapos-pn)

lemma iff-contradict:
\neg P \implies P \iff Q \implies Q \implies R
\neg Q \implies P \iff Q \implies P \implies R
by blast+

ML-file ⟨Tools/Ctr-Sugar/ctr-sugar-util.ML⟩
ML-file ⟨Tools/Ctr-Sugar/ctr-sugar-tactics.ML⟩
ML-file ⟨Tools/Ctr-Sugar/ctr-sugar-code.ML⟩
ML-file ⟨Tools/Ctr-Sugar/ctr-sugar.ML⟩

Coinduction method that avoids some boilerplate compared with coinduct.
ML-file ⟨Tools/coinduction.ML⟩
13 Knaster-Tarski Fixpoint Theorem and inductive definitions

theory Inductive
imports Complete-Lattices Ctr-Sugar
keywords
  inductive coinductive inductive-cases inductive-simps :: thy-defn and
  monos and
  print-inductives :: diag and
  old-rep-datatype :: thy-goal and
  primrec :: thy-defn
begin

13.1 Least fixed points

context complete-lattice
begin

definition lfp :: ('a ⇒ 'a) ⇒ 'a
  where lfp f = Inf {u. f u ≤ u}

lemma lfp-lowerbound: f A ≤ A ⇒ lfp f ≤ A
  unfolding lfp-def by (rule Inf-lower) simp

lemma lfp-greatest: (∀ u. f u ≤ a ⇒ A ≤ u) ⇒ A ≤ lfp f
  unfolding lfp-def by (rule Inf-greatest) simp

end

lemma lfp-fixpoint:
  assumes mono f
  shows f (lfp f) = lfp f
  unfolding lfp-def
proof (rule order-antisym)
  let ?H = {u. f u ≤ u}
  let ?a = ⋂ ?H
  show f ?a ≤ ?a
proof (rule Inf-greatest)
  fix x
  assume x ∈ ?H
  then have ?a ≤ x by (rule Inf-lower)
  with (mono f) have f ?a ≤ f x ..
  also from (x ∈ ?H) have f x ≤ x ..
  finally show f ?a ≤ x .
qed
  show ?a ≤ f ?a
proof (rule Inf-lower)
  from (mono f) and (f ?a ≤ ?a) have f (f ?a) ≤ f ?a ..
  then show f ?a ∈ ?H ..
qed

lemma lfp-unfold: mono f ⇒ lfp f = f (lfp f)
  by (rule lfp-fixpoint [symmetric])

lemma lfp-const: lfp (λx. t) = t
  by (rule lfp-unfold) (simp add: mono-def)

lemma lfp-eqI: mono F ⇒ F x = x =⇒ (∀z. F z = z ⇒ x ≤ z) ⇒ lfp F = x
  by (rule antisym) (simp-all add: lfp-lowerbound lfp-unfold [symmetric])

13.2  General induction rules for least fixed points

lemma lfp-ordinal-induct [case-names mono step union]:
  fixes f :: 'a::complete-lattice ⇒ 'a
  assumes mono: mono f
  and P-f: ∀S. P S ⇒ S ≤ lfp f ⇒ P (f S)
  and P-Union: ∀M. ∀S∈M. P S ⇒ P (Sup M)
  shows P (lfp f)
proof –
  let ?M = {S. S ≤ lfp f ∧ P S}
  from P-Union have P (Sup ?M) by simp
  also have Sup ?M = lfp f
    proof (rule antisym)
      show Sup ?M ≤ lfp f
        by (blast intro: Sup-least)
      then have f (Sup ?M) ≤ (lfp f)
        by (rule mono [THEN monoD])
      then have f (Sup ?M) ≤ lfp f
        using mono [THEN lfp-unfold] by simp
      then have f (Sup ?M) ∈ ?M
        using P-Union by simp (intro P-f Sup-least, auto)
      then have f (Sup ?M) ≤ Sup ?M
        by (rule Sup-upper)
      then show lfp f ≤ Sup ?M
        by (rule lfp-lowerbound)
    qed
  finally show thesis .
qed

theorem lfp-induct:
  assumes mono: mono f
  and ind: f (inf (lfp f) P) ≤ P
  shows lfp f ≤ P
proof (induct rule: lfp-ordinal-induct)
case mono
show ?case by fact
next
  case (step S)
  then show ?case by (intro order-trans[OF - ind] monoD[OF mono]) auto
next
  case (union M)
  then show ?case by (auto intro: Sup-least)
qed

lemma lfp-induct-set:
  assumes lfp: \( a \in lfp f \)
  and mono: \( \text{mono } f \)
  and hyp: \( \forall x. x \in f (lfp f \cap \{ x. P x \}) \Rightarrow P x \)
  shows \( P a \)
  by (rule lfp-induct [THEN subsetD, THEN CollectD, OF mono - lfp]) (auto intro: hyp)

lemma lfp-ordinal-induct-set:
  assumes mono: \( \text{mono } f \)
  and P-f: \( \forall S. P S \Rightarrow P (f S) \)
  and P-Union: \( \forall M. \forall S \in M. P S \Rightarrow P (\bigcup M) \)
  shows \( P (lfp f) \)
  using assms by (rule lfp-ordinal-induct)

Definition forms of lfp-unfold and lfp-induct, to control unfolding.

lemma def-lfp-unfold:
  \( h \equiv lfp f \Rightarrow \text{mono } f \Rightarrow h = f h \)
  by (auto intro: lfp-unfold)

lemma def-lfp-induct:
  \( A \equiv lfp f \Rightarrow \text{mono } f \Rightarrow f (\inf A P) \leq P \Rightarrow A \leq P \)
  by (blast intro: lfp-induct)

lemma def-lfp-induct-set:
  \( A \equiv lfp f \Rightarrow \text{mono } f \Rightarrow a \in A \Rightarrow (\forall x. x \in f (A \cap \{ x. P x \}) \Rightarrow P x) \Rightarrow P a \)
  by (blast intro: lfp-induct-set)

Monotonicity of lfp!

lemma lfp-mono: \( (\forall Z. f Z \leq g Z) \Rightarrow lfp f \leq lfp g \)
  by (rule lfp-lowerbound [THEN lfp-greatest]) (blast intro: order-trans)

13.3 Greatest fixed points

context complete-lattice
begin

definition gfp :: \( \lambda a. \forall a \Rightarrow a \Rightarrow a \)
where \( \text{gfp} f = \text{Sup} \{ u. \ u \leq f u \} \)

**Lemma gfp-upperbound**: \( X \leq f X \Rightarrow X \leq \text{gfp} f \)
by (auto simp add: gfp-def intro: Sup-upper)

**Lemma gfp-least**: \((\forall u. u \leq f u \Rightarrow u \leq X) \Rightarrow \text{gfp} f \leq X\)
by (auto simp add: gfp-def intro: Sup-least)

end

**Lemma lfp-le-gfp**: \( \text{mono} f \Rightarrow \text{lfp} f \leq \text{gfp} f \)
by (rule gfp-upperbound) (simp add: lfp-fixpoint)

**Lemma gfp-fixpoint**:  
assumes \( \text{mono} f \)  
shows \( f (\text{gfp} f) = \text{gfp} f \)  
unfolding gfp-def  
proof (rule order-antisym)  
let \( ?H = \{ u. \ u \leq f u \} \)  
let \( ?a = \bigcup ?H \)  
show \( ?a \leq f ?a \)  
proof (rule Sup-least)  
fix \( x \)  
assume \( x \in ?H \)  
then have \( x \leq f x \) ..  
also from \( \langle x \in ?H \rangle \) have \( x \leq ?a \) by (rule Sup-upper)  
with \( \langle \text{mono} f \rangle \) have \( f x \leq f ?a \) ..  
finally show \( x \leq f ?a \).  
qed  
show \( f ?a \leq ?a \)  
proof (rule Sup-upper)  
from \( \langle \text{mono} f \rangle \) and \( \langle ?a \leq f ?a \rangle \) have \( f ?a \leq f (f ?a) \) ..  
then show \( f ?a \in ?H \) ..  
qed  
qed

**Lemma gfp-unfold**: \( \text{mono} f \Rightarrow \text{gfp} f = f (\text{gfp} f) \)
by (rule gfp-fixpoint [symmetric])

**Lemma gfp-const**: \( \text{gfp} (\lambda x. \ t) = t \)
by (rule gfp-unfold) (simp add: mono-def)

**Lemma gfp-eqI**: \( \text{mono} F \Rightarrow F x = x \Rightarrow (\forall z. F z = z \Rightarrow z \leq x) \Rightarrow \text{gfp} F = x \)
by (rule antisym) (simp-all add: gfp-upperbound gfp-unfold[symmetric])

13.4 Coinduction rules for greatest fixed points

Weak version.
**THEORY “Inductive”**

**lemma** weak-coinduct: \( a \in X \implies X \subseteq f X \implies a \in \text{gfp } f \)
  by (rule gfp-upperbound [THEN subsetD]) auto

**lemma** weak-coinduct-image: \( a \in X \implies g'X \subseteq f (g'X) \implies g a \in \text{gfp } f \)
  apply (erule gfp-upperbound [THEN subsetD])
  apply (erule imageI)
  done

**lemma** coinduct-lemma: \( X \leq f (\sup X (\text{gfp } f)) \implies \text{mono } f \implies \sup X (\text{gfp } f) \leq f (\sup X (\text{gfp } f)) \)
  apply (frule gfp-unfold [THEN eq-refl])
  apply (drule mono-sup)
  apply (rule le-supI)
  apply assumption
  apply (rule order-trans)
  apply assumption
  apply (rule sup-ge2)
  apply assumption
  done

Strong version, thanks to Coen and Frost.

**lemma** coinduct-set: \( \text{mono } f \implies a \in X \implies X \subseteq f (X \cup \text{gfp } f) \implies a \in \text{gfp } f \)
  by (rule weak-coinduct[rotated], rule coinduct-lemma) blast+

**lemma** gfp-fun-UnI2: \( \text{mono } f \implies a \in \text{gfp } f \implies a \in f (X \cup \text{gfp } f) \)
  by (blast dest: gfp-fixpoint mono-Un)

**lemma** gfp-ordinal-induct[case-names mono step union]:
  fixes \( f : 'a::\text{complete-lattice} \Rightarrow 'a \)
  assumes mono: \( \text{mono } f \)
  and P-f: \( \forall S. P S \implies \text{gfp } f \leq S \implies P (f S) \)
  and P-Union: \( \forall M. \forall S \in M. P S \implies P (\text{Inf } M) \)
  shows \( P (\text{gfp } f) \)
proof –
  let \( ?M = \{ S. \text{gfp } f \leq S \land P S \} \)
  from P-Union have \( P (\text{Inf } ?M) \) by simp
  also have \( \text{Inf } ?M = \text{gfp } f \)
proof (rule antisym)
  show \( \text{gfp } f \leq \text{Inf } ?M \)
    by (blast intro: Inf-greatest)
  then have \( f (\text{gfp } f) \leq f (\text{Inf } ?M) \)
    by (rule mono [THEN monoD])
  then have \( \text{gfp } f \leq f (\text{Inf } ?M) \)
    using mono [THEN gfp-unfold] by simp
  then have \( f (\text{Inf } ?M) \in ?M \)
    using P-Union by simp (intro P-f Inf-greatest, auto)
  then have \( \text{Inf } ?M \leq f (\text{Inf } ?M) \)
    by (rule Inf-lower)
then show \( \inf \ ?M \leq \gfp f \)
by (rule \( \gfp \)-upperbound)
qed
finally show \( \vdash \thesis \).
qed

lemma coinduct:
assumes mono: \( \text{mono } f \)
and ind: \( X \leq f \ (\sup X \ (\gfp f)) \)
shows \( X \leq \gfp f \)
proof (induct rule: \( \gfp \)-ordinal-induct)
  case mono
  then show \( \vdash \case \) by fact
next
  case (step \( S \))
  then show \( \vdash \case \) by (intro order-trans[OF ind -] monoD[OF mono]) auto
next
  case (union \( M \))
  then show \( \vdash \case \) by (auto intro: mono \( \inf \)-greatest)
qed

13.5 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition \( X \subseteq f X \) to one expressed using both \( \lfp \) and \( \gfp \)

lemma coinduct3-mono-lemma: \( \text{mono } f \Rightarrow \text{mono } (\lambda x. \ f x \cup X \cup B) \)
by (iprover intro: subset-refl monoI Un-mono monoD)

lemma coinduct3-lemma:
\( X \subseteq f \ (\lfp (\lambda x. \ f x \cup X \cup \gfp f)) \Rightarrow \text{mono } f \Rightarrow \)
\( \lfp (\lambda x. \ f x \cup X \cup \gfp f) \subseteq f \ (\lfp (\lambda x. \ f x \cup X \cup \gfp f)) \)
apply (rule subset-trans)
apply (erule coinduct3-mono-lemma [THEN \( \lfp \)-unfold [THEN eq-refl]])
apply (rule Un-least [THEN Un-least])
apply (rule subset-refl, assumption)
apply (rule \( \gfp \)-unfold [THEN equalityD1, THEN subset-trans], assumption)
apply (rule monoD, assumption)
apply (subst coinduct3-mono-lemma [THEN \( \lfp \)-unfold], auto)
done

lemma coinduct3: \( \text{mono } f \Rightarrow \ a \in X \Rightarrow X \subseteq f \ (\lfp (\lambda x. \ f x \cup X \cup \gfp f)) \Rightarrow \)
a \in \gfp f
apply (rule coinduct3-lemma [THEN [2] \text{weak-coinduct}])
apply (rule coinduct3-mono-lemma [THEN \( \lfp \)-unfold, THEN ss subst])
apply simp-all
done

Definition forms of \( \gfp \)-unfold and coinduct, to control unfolding.
lemma def-gfp-unfold: $A \equiv \text{gfp } f \implies \text{mono } f \implies A = f A$
by (auto intro!: gfp-unfold)

lemma def-coinduct: $A \equiv \text{gfp } f \implies \text{mono } f \implies X \leq f (\sup X A) \implies X \leq A$
by (iprover intro!: coinduct)

lemma def-coinduct-set: $A \equiv \text{gfp } f \implies \text{mono } f \implies a \in X \implies X \subseteq f (X \cup A)$
by (auto intro!: coinduct-set)

lemma def-Collect-coinduct:
$A \equiv \text{gfp } (\lambda w. \text{Collect } (P w)) \implies \text{mono } (\lambda w. \text{Collect } (P w)) \implies a \in X \implies \bigwedge z. z \in X \implies P (X \cup A) z \implies a \in A$
by (erule def-coinduct-set) auto

lemma def-coinduct3: $A \equiv \text{gfp } f \implies \text{mono } f \implies a \in X \implies X \subseteq f (\text{lfp } (\lambda x. f x \cup X \cup A)) \implies a \in A$
by (auto intro!: coinduct3)

Monotonicity of gfp!

lemma gfp-mono: $(\bigwedge Z. f Z \leq g Z) \implies \text{gfp } f \leq \text{gfp } g$
by (rule gfp-upperbound [THEN gfp-least]) (blast intro: order-trans)

13.6 Rules for fixed point calculus

lemma lfp-rolling:
assumes mono g mono f
shows $g (\text{lfp } (\lambda x. f (g x))) = \text{lfp } (\lambda x. g (f x))$
proof (rule antisym)
  have \*: mono (\lambda x. f (g x))
  using assms by (auto simp: mono-def)
  show \text{lfp } (\lambda x. g (f x)) \leq g (\text{lfp } (\lambda x. f (g x)))
  by (rule lfp-lowerbound) (simp add: lfp-unfold[OF \*, symmetric])
  show $g (\text{lfp } (\lambda x. f (g x))) \leq \text{lfp } (\lambda x. g (f x))$
  proof (rule lfp-greatest)
    fix u
    assume u: $g (f u) \leq u$
    then have $g (\text{lfp } (\lambda x. f (g x))) \leq g (f u)$
    by (intro assms[THEN monoD] lfp-lowerbound)
    with u show $g (\text{lfp } (\lambda x. f (g x))) \leq u$
    by auto
  qed
qed

lemma lfp-lfp:
assumes $\bigwedge x y w z. x \leq y \implies w \leq z \implies f x w \leq f y z$
shows $\text{lfp } (\lambda x. \text{lfp } (f x)) = \text{lfp } (\lambda x. f x x)$
proof (rule antisym)
  have \*: mono (\lambda x. f x x)
by (blast intro: monoI f)

show \( \text{lfp} (\lambda x. \text{lfp} (f x)) \leq \text{lfp} (\lambda x. f x) \)
by (intro lfp-lowerbound) (simp add: lfp-unfold[OF \(*\), symmetric])

show \( \text{lfp} (\lambda x. \text{lfp} (f x)) \geq \text{lfp} (\lambda x. f x) \) (is \(?F \geq \))
proof (intro lfp-lowerbound)

have \(*\): \(?F = \text{lfp} (f ?F)\)
by (rule lfp-unfold) (blast intro: monoI lfp-mono f)

also have \ldots = f ?F (lfp (f ?F))
by (rule lfp-unfold) (blast intro: monoI lfp-mono f)

finally show \(?F \leq ?F\)
by (simp add: \(*\) [symmetric])
qed

qed

lemma gfp-rolling:
assumes mono g mono f
shows \( g \circ \text{gfp} (\lambda x. f (g x)) = \text{gfp} (\lambda x. g (f x)) \)
proof (rule antisym)

have \(*\): mono (\lambda x. f (g x))
using assms by (auto simp: mono-def)

show \( g \circ \text{gfp} (\lambda x. f (g x)) \leq \text{gfp} (\lambda x. g (f x)) \)
by (rule gfp-upperbound) (simp add: gfp-unfold[OF \(*\), symmetric])

show \( \text{gfp} (\lambda x. g (f x)) \leq g \circ \text{gfp} (\lambda x. f (g x)) \)
proof (rule gfp-least)

fix u
assume u: u \leq g (f u)
then have g (f u) \leq g \circ \text{gfp} (\lambda x. f (g x))
by (intro assms[THEN monoD] gfp-upperbound)

with u show u \leq g \circ \text{gfp} (\lambda x. f (g x))
by auto
qed

qed

lemma gfp-gfp:
assumes f: \( x y w z. x \leq y \Longrightarrow w \leq z \Longrightarrow f x w \leq f y z \)
sows \( \text{gfp} (\lambda x. \text{gfp} (f x)) = \text{gfp} (\lambda x. f x) \)
proof (rule antisym)

have \(*\): mono (\lambda x. f x)
by (blast intro: monoI f)

show \( \text{gfp} (\lambda x. f x) \leq \text{gfp} (\lambda x. \text{gfp} (f x)) \)
by (intro gfp-upperbound) (simp add: gfp-unfold[OF \(*\), symmetric])

show \( \text{gfp} (\lambda x. \text{gfp} (f x)) \leq \text{gfp} (\lambda x. f x) \) (is \(?F \leq \))
proof (intro gfp-upperbound)

have \(*\): \(?F = \text{gfp} (f ?F)\)
by (rule gfp-unfold) (blast intro: monoI gfp-mono f)

also have \ldots = f ?F (gfp (f ?F))
by (rule gfp-unfold) (blast intro: monoI gfp-mono f)

finally show \(?F \leq ?F \) \( ?F \)
by (simp add: \(*\) [symmetric])
13.7 Inductive predicates and sets

Package setup.

lemmas basic-monos =
subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
Collect-mono in-mono vimage-mono

lemma le-rel-bool-arg-iff: \( X \leq Y \iff X \text{ False} \leq Y \text{ False} \land X \text{ True} \leq Y \text{ True} \)
unfolding le-fun-def le-bool-def using bool-.induct by auto

lemma imp-conj-iff: \( (P \implies Q) \land P \) = \( P \land Q \)
by blast

lemma meta-fun-cong: \( P \equiv Q \implies P \ a \equiv Q \ a \)
by auto

ML-file \langle Tools/inductive.ML\rangle

lemmas [mono] =
imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
imp-mono not-mono
Ball-def Bex-def
induct-rulify-fallback

13.8 The Schroeder-Bernstein Theorem

See also:

- \$ISABELLE_HOME/src/HOL/ex/Set_Theory.thy
- http://planetmath.org/proofofschroederbernsteintheoremusingtarskiknastertheorem
- Springer LNCS 828 (cover page)

theorem Schroeder-Bernstein:
fixes f : 'a \Rightarrow 'b and g : 'b \Rightarrow 'a
and A : 'a set and B : 'b set
assumes inj1: inj-on f A and sub1: \( f \cdot A \subseteq B \)
and inj2: inj-on g B and sub2: \( g \cdot B \subseteq A \)
shows \( \exists h. \ bij-betw h A B \)
proof (rule exI, rule bij-betw-IMAGEI)
define X where X = lfp (\( \lambda X. A - (g \cdot (B - (f \cdot X))) \))
define g' where g' = the-inv-into (B - (f \cdot X)) g
let \( \exists h = \lambda z. \text{ if } z \in X \text{ then } f z \text{ else } g' z \)
have $X = A - (g' \cdot (B - (f' \cdot X)))$

unfolding $X$-def by (rule lfp-unfold) (blast intro: monoI)
then have $X$-compl: $A - X = g' \cdot (B - (f' \cdot X))$
using sub2 by blast

from $inj2$ have $inj2'$: $inj$-on $g \cdot (B - (f' \cdot X))$
by (rule inj-on-subset) auto
with $X$-compl have $*: g' \cdot (A - X) = B - (f' \cdot X)$
by (simp add: $g'$-def)

from $X$ have $X$-sub: $X \subseteq A$ by auto
from $X$ sub1 have $fX$-sub: $f' \cdot X \subseteq B$ by auto

show $?h' \cdot A = B$
proof -
from $X$-sub have $?h' \cdot A = ?h' \cdot (X \cup (A - X))$ by auto
also have $\ldots = ?h' \cdot X \cup ?h' \cdot (A - X)$ by (simp only: image-Un)
also have $?h' \cdot X = f' \cdot X$ by auto
also from $*$ have $?h' \cdot (A - X) = B - (f' \cdot X)$ by auto
also from $fX$-sub have $f' \cdot X \cup (B - f' \cdot X) = B$ by blast
finally show $?thesis$.

qed

show $inj$-on $?h$ $A$
proof -
from $inj1$ $X$-sub have on-$X$: $inj$-on $f \cdot X$
by (rule subset-inj-on)

have $on$-$X$-compl: $inj$-on $g' \cdot (A - X)$
unfolding $g'$-def $X$-compl
by (rule inj-on-the-inv-into) (rule $inj2'$)

have impossible: False if eq: $f \cdot a = g' \cdot b$ and $a: a \in X$ and $b: b \in A - X$ for $a$ $b$
proof -
from $a$ have fa: $f \cdot a \in f' \cdot X$ by (rule imageI)
from $b$ have $g' \cdot b \in g' \cdot (A - X)$ by (rule imageI)
with $*$ have $g' \cdot b \in - (f' \cdot X)$ by simp
with eq fa show False by simp

qed

show $?thesis$
proof (rule inj-onI)
fix $a$ $b$
assume $h$: $?h a = ?h b$
assume $a: a \in A$ and $b: b \in A$
then consider $a \in X$ $b \in X$ $a: a \in A - X$ $b: b \in A - X$
$| a \in X$ $b: b \in A - X$ $a: a \in A - X$ $b: b \in X$
by blast
then show $a = b$
proof cases
  case 1 with h on-X show ?thesis by (simp add: inj-on-eq-iff)
next
case 2 with h on-X-compl show ?thesis by (simp add: inj-on-eq-iff)
next
case 3 with h impossible [of a b] have False by simp
  then show ?thesis ..
next
case 4 with h impossible [of b a] have False by simp
  then show ?thesis..
qed
qed
qed

13.9 Inductive datatypes and primitive recursion

Package setup.

ML-file‹Tools/Old-Datatype/old-datatype-aux.ML›
ML-file‹Tools/Old-Datatype/old-datatype-prop.ML›
ML-file‹Tools/Old-Datatype/old-datatype-data.ML›
ML-file‹Tools/Old-Datatype/old-rep-datatype.ML›
ML-file‹Tools/Old-Datatype/old-datatype-codegen.ML›
ML-file‹Tools/BNF/bnf-lfp-rec-sugar-util.ML›
ML-file‹Tools/Old-Datatype/old-primrec.ML›
ML-file‹Tools/BNF/bnf-lfp-rec-sugar.ML›

Lambda-abstractions with pattern matching:
syntax (ASCII)
  -lam-pats-syntax :: cases-syn ⇒ 'a ⇒ 'b ((% -) 10)
syntax
  -lam-pats-syntax :: cases-syn ⇒ 'a ⇒ 'b ((λ -) 10)
parse-translation〈
  let
    fun fun-tr ctxt [cs] =
      let
        val x = Syntax.free (fst (Term.declare-term-frees cs
          Name.context)));
        val ft = Case-Translation.case-tr true ctxt [x, cs];
      in
        lambda x ft end
      end
    in
      ([syntax-const (-lam-pats-syntax, fun-tr)]
```
14 Cartesian products

theory Product-Type
  imports Typedef Inductive Fun
  keywords inductive-set coinductive-set :: thy-defn
begin

14.1 bool is a datatype

free-constructors (discs-sels) case-bool for True | False
  by auto

Avoid name clashes by prefixing the output of old-rep-datatype with old.

setup ⟨Sign.mandatory-path old⟩

old-rep-datatype True False by (auto intro: bool-induct)

setup ⟨Sign.parent-path⟩

But erase the prefix for properties that are not generated by free-constructors.

setup ⟨Sign.mandatory-path bool⟩

lemmas induct = old.bool.induct
lemmas inducts = old.bool.inducts
lemmas rec = old.bool.rec
lemmas simps = bool.distinct bool.case bool.rec

setup ⟨Sign.parent-path⟩

declare case-split [cases type: bool]
  — prefer plain propositional version

lemma [code]: HOL.equal False P ↔ P
  and [code]: HOL.equal True P ↔ P
  and [code]: HOL.equal P False ↔ P
  and [code]: HOL.equal P True ↔ P
  and [code ndc]: HOL.equal P P ↔ True
  by (simp-all add: equal)

lemma If-case-cert:
  assumes CASE ≡ (λb. If b f g)
  shows (CASE True ≡ f) &&& (CASE False ≡ g)
  using assms by simp-all

setup ⟨Code.declare-case-global 0{thm If-case-cert}⟩

code-printing
  constant HOL.equal :: bool ⇒ bool ⇒ bool → (Haskell) infix 4 ==
  | class-instance bool :: equal → (Haskell) –
14.2 The unit type

typedef unit = {True} by auto

definition Unity :: unit ('())
  where () = Abs-unit True

lemma unit-eq [no-atp]: u = () by (induct u) (simp add: Unity-def)

Simplification procedure for unit-eq. Cannot use this rule directly — it loops!
simproc-setup unit-eq (x::unit) =  
  (K (K (fn ct =>
    if HOLogic.is-unit (Thm.term_of ct) then NONE
    else SOME (mk-meta-eq @{thm unit-eq}))))

free-constructors case-unit for () by auto

Avoid name clashes by prefixing the output of old-rep-datatype with old.

setup  (Sign.mandatory-path old)

old-rep-datatype () by simp

setup  (Sign.parent-path)

But erase the prefix for properties that are not generated by free-constructors.

setup  (Sign.mandatory-path unit)

lemmas induct = old.unit.induct
lemmas inducts = old.unit.inducts
lemmas rec = old.unit.rec
lemmas simps = unit.case unit.rec

setup  (Sign.parent-path)

lemma unit-all-eq1: (\x::unit. PROP P x) \iff PROP P () by simp

lemma unit-all-eq2: (\x::unit. PROP P) \iff PROP P  
  by (rule triv-forall-equality)

This rewrite counters the effect of simproc unit-eq on \x::unit. f u, replacing it by f rather than by \x. f ()

lemma unit-abs-eta-conv [simp]: (\x::unit. f ()) = f  
  by (rule ext) simp
THEORY "Product-Type"

lemma UNIV-unit: UNIV = {()}
  by auto

instantiation unit :: default
begin

definition default = ()

instance ..
end

instantiation unit :: {complete-boolean-algebra, complete-linorder, wellorder}
begin

definition less-eq-unit :: unit ⇒ unit ⇒ bool
  where (- :: unit) ≤ - ↔ True

lemma less-eq-unit [iff]: u ≤ v for u v :: unit
  by (simp add: less-eq-unit-def)

definition less-unit :: unit ⇒ unit ⇒ bool
  where (- :: unit) < - ↔ False

lemma less-unit [iff]: ¬ u < v for u v :: unit
  by (simp-all add: less-eq-unit-def less-unit-def)

definition bot-unit :: unit
  where [code-unfold]: ⊥ = ()

definition top-unit :: unit
  where [code-unfold]: ⊤ = ()

definition inf-unit :: unit ⇒ unit ⇒ unit
  where [simp]: - ⊓ - = ()

definition sup-unit :: unit ⇒ unit ⇒ unit
  where [simp]: - ⊔ - = ()

definition Inf-unit :: unit set ⇒ unit
  where [simp]: ⋂ - = ()

definition Sup-unit :: unit set ⇒ unit
  where [simp]: ⋃ - = ()

definition uminus-unit :: unit ⇒ unit
  where [simp]: - - = ()
\begin{enumerate}
\item \textbf{declare} less-eq-unit-def [abs-def, code-unfold]
\item \textbf{less-unit-def} [abs-def, code-unfold]
\item \textbf{inf-unit-def} [abs-def, code-unfold]
\item \textbf{sup-unit-def} [abs-def, code-unfold]
\item \textbf{Inf-unit-def} [abs-def, code-unfold]
\item \textbf{Sup-unit-def} [abs-def, code-unfold]
\item \textbf{uminus-unit-def} [abs-def, code-unfold]
\end{enumerate}

\begin{proof}
\begin{enumerate}
\item \textbf{instance} by intro-classes auto
\end{enumerate}
\end{proof}

\begin{definition}
\textbf{lemma} [code]: $HOL.equal \; u \; v \leftrightarrow \text{True}$ for $u \; v :: \text{unit}$
\textbf{unfolding} $\text{equal \; unit-eq \; [of \; u]} \; \text{unit-eq \; [of \; v]}$ by (rule \textit{iffI \; TrueI \; refl})+
\end{definition}

\begin{code-printing}
\textbf{type-constructor} unit $\rightarrow$
\begin{enumerate}
\item \textbf{(SML)} \texttt{unit}
\item \textbf{(OCaml)} \texttt{unit}
\item \textbf{(Haskell)} \texttt{()}
\item \textbf{(Scala)} \texttt{Unit}
\end{enumerate}
\textbf{| constant} \texttt{Unity} $\rightarrow$
\begin{enumerate}
\item \textbf{(SML)} \texttt{()}
\item \textbf{(OCaml)} \texttt{()}
\item \textbf{(Haskell)} \texttt{()}
\item \textbf{(Scala)} \texttt{()}
\end{enumerate}
\textbf{| class-instance} \texttt{unit :: equal} $\rightarrow$
\begin{enumerate}
\item \textbf{(Haskell)} --
\end{enumerate}
\textbf{| constant} \texttt{HOL.equal :: unit $\Rightarrow$ unit $\Rightarrow$ bool} $\rightarrow$
\begin{enumerate}
\item \textbf{(Haskell)} $\text{infix} \; 4 \; ==$
\end{enumerate}

\begin{code-reserved}
\textbf{SML}
\begin{enumerate}
\item \texttt{unit}
\end{enumerate}
\textbf{OCaml}
\begin{enumerate}
\item \texttt{unit}
\end{enumerate}
\textbf{Scala}
\begin{enumerate}
\item \texttt{Unit}
\end{enumerate}
\end{code-reserved}

\section{The product type}

\subsection{Type definition}

\begin{definition}
\textbf{definition} \texttt{Pair-Rep :: 'a $\Rightarrow$ 'b $\Rightarrow$ 'a $\Rightarrow$ 'b $\Rightarrow$ bool}
\textbf{where} \texttt{Pair-Rep \; a \; b = (\lambda x \; y. \; x = a \wedge y = b)}
\end{definition}

\begin{definition}
\textbf{definition} \texttt{prod} = \{ f. \exists a \; b. \; f = \text{Pair-Rep} \; (a::'a) \; (b::'b) \}
\end{definition}
typedef \( (\cdot, \cdot) \) prod \( \left( (- \times/ -) [21, 20] 20 \right) = \) prod :: \( (\cdot \Rightarrow \cdot \Rightarrow \text{bool}) \) set 

unfolding prod-def by auto

type-notation (ASCII)

\( \text{prod} \) (infixr \( \ast 20 \))

definition \( \text{Pair} :: \cdot \Rightarrow \cdot \Rightarrow \cdot \times \cdot \)

where \( \text{Pair} a b = \text{Abs-prod} (\text{Pair-Rep} a b) \)

lemma prod-cases: \( \forall a b. P (\text{Pair} a b) \implies P p \)

by (cases p) (auto simp add: prod-def Pair-def Pair-Rep-def)

free-constructors case-prod for \( \text{Pair} \) \( \text{fst} \) \( \text{snd} \)

proof

--

fix \( P :: \text{bool} \) and \( p :: \cdot \times \cdot \)

show \( \forall x1 x2. p = \text{Pair} x1 x2 \implies P \implies P \)

by (cases p) (auto simp add: prod-def Pair-def Pair-Rep-def)

next

fix a c :: \cdot \) and \( b d :: \cdot \)

have \( \text{Pair-Rep} a b = \text{Pair-Rep} c d \iff a = c \land b = d \)

by (auto simp add: Pair-Rep-def fun-eq-iff)

moreover have \( \text{Pair-Rep} a b \in \text{prod} \) and \( \text{Pair-Rep} c d \in \text{prod} \)

by (auto simp add: prod-def)

ultimately show \( \text{Pair} a b = \text{Pair} c d \iff a = c \land b = d \)

by (simp add: Pair-def Abs-prod-inject)

qed

Avoid name clashes by prefixing the output of old-rep-datatype with old.

setup (Sign.mandatory-path old)

old-rep-datatype Pair

by (erule prod-cases) (rule prod.inject)

setup (Sign.parent-path)

But erase the prefix for properties that are not generated by free-constructors.

setup (Sign.mandatory-path prod)

declare old.prod.inject [iff del]

lemmas induct = old.prod.induct

lemmas inducts = old.prod.inducts

lemmas rec = old.prod.rec

lemmas simps = prod.inject prod.case prod.rec

setup (Sign.parent-path)

declare prod.case [nitpick-simp del]

declare old.prod.case cong-weak [cong del]
THEORY “Product-Type”

declare prod.case-eq-if [mono]
declare prod.split [no-atp]
declare prod.split-asn [no-atp]

prod.split could be declared as [split] done after the Splitter has been speeded up significantly; precompute the constants involved and don’t do anything unless the current goal contains one of those constants.

14.3.2 Tuple syntax

Patterns – extends pre-defined type pttrn used in abstractions.

nonterminal tuple-args and patterns
syntax
- tuple :: 'a ⇒ tuple-args ⇒ 'a × 'b
- tuple-arg :: 'a ⇒ tuple-args
- tuple-args :: 'a ⇒ tuple-args ⇒ tuple-args
- pattern :: pttrn ⇒ patterns ⇒ pttrn
- patterns :: pttrn ⇒ patterns ⇒ patterns
- unit :: pttrn

translations
(x, y) ⇔ CONST Pair x y
- pattern x y ⇔ CONST Pair x y
- patterns x y ⇔ CONST Pair x y
- tuple x y z ⇔ -tuple x (-tuple-arg (-tuple y z))
λ(x, y, zs). b ⇔ CONST case-prod (λx y. b)
- abs (CONST Pair x y) t ⇒ λx y. t
- abs (CONST Unity) t ⇒ λ(). t

print case-prod f as case-prod f and case-prod f as case-prod f
Reconstruct pattern from (nested) case-prods, avoiding eta-contraction of body; required for enclosing "let", if "let" does not avoid eta-contraction, which has been observed to occur.

print-translation : 
  let 
    fun case-prod-tr\' [Abs (x, T, t as (Abs abs))] = 
      (* case-prod (\x y. t) \Rightarrow \lambda(x, y) t *) 
      let 
      val (y, t') = Syntax.Trans.atomic-abs-tr\' abs; 
      val (x', t'') = Syntax.Trans.atomic-abs-tr\' (x, T, t'); 
      in 
      Syntax.const syntax-const (-abs) $ 
      (Syntax.const syntax-const (-patterns) $ x' $ y) $ t'' 
      end 
    | case-prod-tr\' [Abs (x, T, (s as Const (const-syntax \case-prod\-, -) $ t))] = 
      (* case-prod (\x. (case-prod (\y z. t))) \Rightarrow \lambda(x, y, z) t *) 
      let 
      val Const (syntax-const (-abs), -) $ 
      (Const (syntax-const (-patterns) $ y $ z) $ t' = 
      case-prod-tr\' [t]; 
      val (x', t'') = Syntax.Trans.atomic-abs-tr\' (x, T, t'); 
      in 
      Syntax.const syntax-const (-abs) $ 
      (Syntax.const syntax-const (-patterns) $ x' $ 
      (Syntax.const syntax-const (-patterns) $ y $ z)) $ t'' 
      end 
    | case-prod-tr\' [Const (const-syntax \case-prod\-, -) $ t] = 

THEORY “Product-Type”

(* case-prod (case-prod (λ x y z. t)) ⇒ λ((x, y), z). t *)
case-prod-tr’ [case-prod-tr’ [t]]

(* inner case-prod-tr’ creates next pattern *)

| case-prod-tr’ [Const (syntax-const (-abs), -) $ x-y $ Abs abs] =
| (case-prod (λ x-y z. t) ⇒ λ(x-y, z. t *))
| let val (x, y) = Syntax.Trans.atomic-abs-tr’ abs in
| Syntax.const syntax-const (-pattern) $ x-y $ z) $ t
|
end

14.3.3 Code generator setup

code-printing
type-constructor prod ⇀
| (SML) infix 2 *
| and (OCaml) infix 2 *
| and (Haskell) !((-,/ (-))
| and (Scala) !((-,/ (-))
| constant Pair ⇀
| (SML) !((-,/ (-))
| and (OCaml) !((-,/ (-))
| and (Haskell) !((-,/ (-))
| and (Scala) !((-,/ (-))
| class-instance prod :: equal ⇀
| (Haskell) −
| constant HOL.equal :: 'a × 'b ⇒ 'a × 'b ⇒ bool ⇀
| (Haskell) infix 4 ==
| constant fst ⇀ (Haskell) fst
| constant snd ⇀ (Haskell) snd

14.3.4 Fundamental operations and properties

lemma Pair-inject: (a, b) = (a', b') ⇒ (a = a' ⇒ b = b' ⇒ R) ⇒ R
| by simp

lemma surj-pair [simp]: \exists x y. p = (x, y)
| by (cases p) simp

lemma fst-eqD: fst (x, y) = a ⇒ x = a
| by simp

lemma snd-eqD: snd (x, y) = a ⇒ y = a
| by simp

lemma case-prod-unfold [nitpick-unfold]: case-prod = (λc p. c (fst p) (snd p))
| by (simp add: fun-eq-iff split: prod.split)
lemma case-prod-conv \(\text{simp, code}\): (case \((a, b)\) of \((c, d)\) \(\Rightarrow\) \(f\) \(c\) \(d\)) = \(f\) \(a\) \(b\)
by (fact prod.case)

lemmas surjective-pairing = prod-collapse [symmetric]

lemma prod-eq-iff: \(s = t \iff \text{fst } s = \text{fst } t \land \text{snd } s = \text{snd } t\)
by (cases \(s\), cases \(t\)) simp

lemma prod-eqI \(\text{intro}\): \(\text{fst } p = \text{fst } q \Rightarrow \text{snd } p = \text{snd } q \Rightarrow p = q\)
by (cases \(s\), cases \(t\)) simp

lemma prod-eqD: (case \((a, b)\) of \((c, d)\) \(\Rightarrow\) \(f\) \(c\) \(d\)) = \(\Rightarrow\) \(f\) \(a\) \(b\)
by (rule prod.case \[THEN\] iffD1)

lemma case-prod-comp: (case \(x\) of \((a, b)\) \(\Rightarrow\) \((f \circ g)\) \(a\) \(b\)) = \(f\) \((g\) \((\text{fst } x)\)) \((\text{snd } x)\)
by (cases \(x\)) simp

lemma The-case-prod: The (case-prod \(P\)) = (THE \(xy\). \(P\) \((\text{fst } xy)\) \((\text{snd } xy)\))
by (simp add: case-prod-unfold)

lemma cond-case-prod-eta: \((\lambda x, y. f x y = g) \((x, y)\) \(\Rightarrow\) \((\lambda x, y. f x y) = g\)
— Subsumes the old split-Pair when \(f\) is the identity function.
by (simp add: fun-eq-iff split: prod.split)

lemma split-paired-all \(\text{no-atp}\): \((\forall x. \text{PROP } P x) \equiv (\forall a b. \text{PROP } P \((a, b)\))
proof
  fix \(a\) \(b\)
  assume \(\forall x. \text{PROP } P x\)
  then show \(\text{PROP } P \((a, b)\)\).
next
  fix \(x\)
  assume \(\forall a b. \text{PROP } P \((a, b)\)
  from \(\text{PROP } P \((\text{fst } x, \text{snd } x)\)\) show \(\text{PROP } P x\) by simp
qed

The rule split-paired-all does not work with the Simplifier because it also affects premises in congruence rules, where this can lead to premises of the form \(\forall a b. \ldots = ?P(a, b)\) which cannot be solved by reflexivity.
lemmas split-tupled-all = split-paired-all unit-all-eq2

ML

(* replace parameters of product type by individual component parameters *)
local (* filtering with exists-paired-all is an essential optimization *)

fun exists-paired-all (Const (const-name (Pure.all), _) $ Abs (_, T, t)) =
  can HOLogic.dest-prodT T orelse exists-paired-all t
  | exists-paired-all (t $ u) = exists-paired-all t orelse exists-paired-all u
  | exists-paired-all (Abs (_, -, t)) = exists-paired-all t
  | exists-paired-all _ = false;

val ss =
  simpset_of
  (put-simpset HOL-basic-ss context
diffsimps [@{thm split-paired-all}, @{thm unit-all-eq2}, @{thm unit-abs-eta-conv}]
diffsimps [simpl proc (unit-eq)]);

in

fun split-all-tac ctxt = SUBGOAL (fn (t, i) =>
  if exists-paired-all t then safe-full-simp-tac (put-simpset ss ctxt) i else no-tac);

fun unsafe-split-all-tac ctxt = SUBGOAL (fn (t, i) =>
  if exists-paired-all t then full-simp-tac (put-simpset ss ctxt) i else no-tac);

fun split-all ctxt th =
  if exists-paired-all (Thm.prop-of th)
  then full-simplify (put-simpset ss ctxt) th else th;
end


setup (map-theory-claset (fn ctxt => ctxt addSbefore (split-all-tac, split-all-tac)));

lemma split-paired-All [simp, no-atp]: (\forall x. P x) \iff (\forall a b. P (a, b))
  — [iff] is not a good idea because it makes blast loop
by fast

lemma split-paired-Ex [simp, no-atp]: (\exists x. P x) \iff (\exists a b. P (a, b))
by fast

lemma split-paired-The [no-atp]: (THE x. P x) = (THE (a, b). P (a, b))
  — Can't be added to simpset: loops!
by (simp add: case-prod-eta)

Simplification procedure for cond-case-prod-eta. Using case-prod-eta as a
rewrite rule is not general enough, and using cond-case-prod-eta directly
would render some existing proofs very inefficient; similarly for prod.case-eq-if.

ML

local
val cond-case-prod-eta-ss =
  simpset-of (put-simpset HOL-basic-ss context diffsimps @{thms cond-case-prod-eta});
fun Pair-pat k 0 (Bound m) = (m = k)
| Pair-pat k i (Const (const-name Pair, -) $ Bound m $ t) = 
| i > 0 andalso m = k + i andalso Pair-pat k (i - 1) t 
| Pair-pat - - - = false;

fun no-args k i (Abs (\ - , - , t)) = no-args (k + 1) i t 
| no-args k i (t $ u) = no-args k i t andalso no-args k i u 
| no-args k i (Bound m) = m < k orelse m > k + i 
| no-args - - - = true;

fun split-pat tp i (Abs (\ - , - , t)) = if tp 0 i t then SOME (i, t) else NONE 
| split-pat tp i (Const (const-name case-prod, -) $ Abs (\ - , - , t)) = split-pat tp (i + 1) t 
| split-pat tp i - - - = NONE;

fun metaeq ctxt lhs rhs = mk-meta-eq (Goal.prove ctxt [] [] 
  (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs))) 
  (K (simp-tac (put-simpset cond-case-prod-eta-ss ctxt 1))));

fun beta-term-pat k i (Abs (\ - , - , t)) = beta-term-pat (k + 1) i t 
| beta-term-pat k i (t $ u) = 
  Pair-pat k i (t $ u) orelse beta-term-pat k i t andalso beta-term-pat k i u 
| beta-term-pat k i t = no-args k i t;

fun eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg 
| eta-term-pat - - - = false;

fun subst arg k i (Abs (x, T, t)) = Abs (x, T, subst arg (k+1) i t) 
| subst arg k i (t $ u) = 
  if Pair-pat k i (t $ u) then incr-boundvars k arg 
  else (subst arg k i t $ subst arg k i u) 
| subst arg k i t = t;

fun beta-proc ctxt (s as Const (const-name case-prod, -) $ Abs (\ - , - , t) $ arg) = 
  (case split-pat beta-term-pat 1 t of 
    SOME (i, f) => SOME (metaeq ctxt s (subst arg 0 i f)) 
  | NONE => NONE) 
| beta-proc - - - = NONE;

fun eta-proc ctxt (s as Const (const-name case-prod, -) $ Abs (\ - , - , t)) = 
  (case split-pat eta-term-pat 1 t of 
    SOME (\ - , ft) => SOME (metaeq ctxt s (let val f $ - = ft in f end)) 
  | NONE => NONE) 
| eta-proc - - - = NONE;
end;

simproc-setup case-prod-beta (case-prod f z) = 
  K (fn ctxt => fn ct => beta-proc ctxt (Thm.term-of ct));
simproc-setup case-prod-eta (case-prod f) = 
  K (fn ctxt => fn ct => eta-proc ctxt (Thm.term-of ct));

lemma case-prod-beta\': (\(x,y). f x y) = (\x. f (fst x) (snd x)) 
  by (auto simp: fun-eq-iff)

case-prod used as a logical connective or set former.
These rules are for use with \textit{blast}; could instead call \textit{simp} using \textit{prod.split} as rewrite.

\textbf{lemma} case-prodI2:
\[
\forall p. \left( \forall a \ b. \ p = (a, b) \Rightarrow c \ a \ b \right) \Rightarrow \text{case } p \text{ of } (a, b) \Rightarrow c \ a \ b
\]
by (\textit{simp add: split-tupled-all})

\textbf{lemma} case-prodI2':
\[
\forall p. \left( a, b \right) = p \Rightarrow c \ a \ b \ x \Rightarrow \text{case } p \text{ of } (a, b) \Rightarrow c \ a \ b \ x
\]
by (\textit{simp add: split-tupled-all})

\textbf{lemma} case-prodE [\textit{elim}!]:
\[
\text{case } p \text{ of } (a, b) \Rightarrow c \ a \ b \ \Rightarrow \left( \forall x \ y. \ p = (x, y) \Rightarrow c \ x \ y \Rightarrow Q \right) \Rightarrow Q
\]
by (\textit{induct } p \textit{ simp})

\textbf{lemma} case-prodE' [\textit{elim}!]:
\[
\text{case } p \text{ of } (a, b) \Rightarrow c \ a \ b \ z \Rightarrow \left( \forall x \ y. \ p = (x, y) \Rightarrow c \ x \ y \ z \Rightarrow Q \right) \Rightarrow Q
\]
by (\textit{induct } p \textit{ simp})

\textbf{lemma} case-prodE2:
\begin{itemize}
  \item \textit{assumes } q: Q \left( \text{case } z \text{ of } (a, b) \Rightarrow P \ a \ b \right)
  \item \textit{and } r: \left( \forall x \ y. \ z = (x, y) \Rightarrow Q \left( P \ x \ y \right) \Rightarrow R \right)
\end{itemize}
\textit{shows } R
\textit{proof (rule } r \textit{)}
\begin{itemize}
  \item \textit{show } z = (\text{fst } z, \text{snd } z) \text{ by } \textit{simp}
  \item \textit{then show } Q \left( P \left( \text{fst } z \right) \left( \text{snd } z \right) \right)
  \item \textit{using } q \text{ by } (\textit{simp add: case-prod-unfold})
\end{itemize}
\textit{qed}

\textbf{lemma} case-prodD': \left( \text{case } (a, b) \text{ of } (c, d) \Rightarrow R \ c \ d \right) \Rightarrow R \ a \ b \ c

by \textit{simp}

\textbf{lemma} mem-case-prodI: \left( z \in c \ a \ b \Rightarrow \text{case } (a, b) \text{ of } (d, e) \Rightarrow c \ d \ e \right)

by \textit{simp}

\textbf{lemma} mem-case-prodI2 [\textit{intro}!]:
\[
\forall p. \left( \forall a \ b. \ p = (a, b) \Rightarrow z \in c \ a \ b \right) \Rightarrow \left( \text{case } p \text{ of } (a, b) \Rightarrow c \ a \ b \right)
\]
by (\textit{simp only: split-tupled-all})

\textbf{declare} mem-case-prodI [\textit{intro}!] — postponed to maintain traditional declaration order!

\textbf{declare} case-prodI2' [\textit{intro}!] — postponed to maintain traditional declaration order!

\textbf{declare} case-prodI2 [\textit{intro}!] — postponed to maintain traditional declaration order!

\textbf{declare} case-prodI [\textit{intro}!] — postponed to maintain traditional declaration order!

\textbf{lemma} mem-case-prodE [\textit{elim}!]:
\begin{itemize}
  \item \textit{assumes } z \in \text{case-prod } c \ p
  \item \textit{obtains } x \ y \text{ where } p = (x, y) \text{ and } z \in c \ x \ y
  \item \textit{using } assms \text{ by } (\textit{rule case-prodE2})
\end{itemize}
ML

local (* filtering with exists-p-split is an essential optimization *)

fun exists-p-split (Const (const-name case-prod, -) $ - $(Const (const-name Pair, -)$ - $)) = true
| exists-p-split (t $ u) = exists-p-split t orelse exists-p-split u
| exists-p-split (Abs (_, _, t)) = exists-p-split t
| exists-p-split - = false;
in
fun split-conv-tac ctxt = SUBGOAL (fn (t, i) =>
if exists-p-split t
then safe-full-simp-tac (put-simpset HOL-basic-ss ctxt addsimps @ {ths case-prod-conv}) i
else no-tac;
) end;

setup (map-theory-claset (fn ctxt =
ctxt addSbefore (split-conv-tac, split-conv-tac)));

lemma split-eta-SetCompr [simp, no-atp]: 
(\lambda u. \exists x y. u = (x, y) \land P x y)
by (rule ext) fast

lemma split-eta-SetCompr2 [simp, no-atp]: 
(\lambda u. \exists x y. u = (x, y) \land P x y) = case-prod P
by (rule ext) fast

lemma split-part [simp]: 
(\lambda (a, b). P \land Q a b) = (\lambda a b. P \land case-prod Q a b)
— Allows simplifications of nested splits in case of independent predicates.
by (rule ext) blast

lemma split-comp-eq:
fixes f :: 'a ⇒ 'b ⇒ 'c
and g :: 'd ⇒ 'a
shows (\lambda u. f (g (fst u)) (snd u)) = case-prod (\lambda x. f (g x))
by (rule ext) auto

lemma pair-imageI [intro]: 
(a, b) ∈ A ⇒ f a b ∈ (\lambda (a, b). f a b) 'A
by (rule image-eqI [where x = (a, b)]) auto

lemma Collect-const-case-prod[ simp]: 
{(a, b). P} = (if P then UNIV else { })
by auto

lemma The-split-eq [simp]: 
(THE (x', y'). x = x' ∧ y = y') = (x, y)
by blast
lemma case-prod-beta: case-prod f p = f (fst p) (snd p)
  by (fact prod.case-eq-if)

lemma prod-cases3 [cases type]:
  obtains (fields) a b c where y = (a, b, c)
proof (cases y)
  case (Pair a b)
  with that show ?thesis
  by (cases b) blast
qed

lemma prod-induct3 [case-names fields, induct type]:
  (∀a b c. P (a, b, c)) ⇒ P x
  by (cases x) blast

lemma prod-cases4 [cases type]:
  obtains (fields) a b c d where y = (a, b, c, d)
proof (cases y)
  case (fields a b c)
  with that show ?thesis
  by (cases c) blast
qed

lemma prod-induct4 [case-names fields, induct type]:
  (∀a b c d. P (a, b, c, d)) ⇒ P x
  by (cases x) blast

lemma prod-cases5 [cases type]:
  obtains (fields) a b c d e where y = (a, b, c, d, e)
proof (cases y)
  case (fields a b c d)
  with that show ?thesis
  by (cases d) blast
qed

lemma prod-induct5 [case-names fields, induct type]:
  (∀a b c d e. P (a, b, c, d, e)) ⇒ P x
  by (cases x) blast

lemma prod-cases6 [cases type]:
  obtains (fields) a b c d e f where y = (a, b, c, d, e, f)
proof (cases y)
  case (fields a b c d e)
  with that show ?thesis
  by (cases e) blast
qed

lemma prod-induct6 [case-names fields, induct type]:
THEORY “Product-Type”

\[(\forall a\ b\ c\ d\ e\ f.\ P\ (a,\ b,\ c,\ d,\ e,\ f)) \implies P\ x\]
by (cases \(x\)) blast

lemma prod-cases7 [cases type]:
  obtains (fields) \(a\ b\ c\ d\ e\ f\ g\) where \(y = (a,\ b,\ c,\ d,\ e,\ f,\ g)\)
proof (cases \(y\))
  case (fields \(a\ b\ c\ d\ e\ f\))
  with \(\dots\) that show ?thesis
  by (cases \(f\)) blast
qed

lemma prod-induct7 [case-names fields, induct type]:
\[(\forall a\ b\ c\ d\ e\ f\ g.\ P\ (a,\ b,\ c,\ d,\ e,\ f,\ g)) \implies P\ x\]
by (cases \(x\)) blast

definition internal-case-prod :: \(('a\Rightarrow'b\Rightarrow'c)\Rightarrow'a\times'b\Rightarrow'c\)
where internal-case-prod \(\equiv\) case-prod

glemma internal-case-prod-conv [simp]: internal-case-prod \(c\ (a,\ b) = c\ a\ b\)
by (simp only: internal-case-prod-def case-prod-conv)

ML-file <Tools/split-rule.ML>

hide-const internal-case-prod

14.3.5 Derived operations

definition curry :: \(('a\times'b\Rightarrow'c)\Rightarrow'a\Rightarrow'b\Rightarrow'c\)
where curry = (\(\lambda\ x\ y.\ c\ (x,\ y)\))

lemma curry-conv [simp, code]: curry \(f\ a\ b = f\ (a,\ b)\)
by (simp add: curry-def)

lemma curryI [intro!]: \(f\ (a,\ b) \implies curry\ f\ a\ b\)
by (simp add: curry-def)

lemma curryD [dest!]: \(curry\ f\ a\ b \implies f\ (a,\ b)\)
by (simp add: curry-def)

lemma curryE: curry \(f\ a\ b \implies (f\ (a,\ b) \implies Q) \implies Q\)
by (simp add: curry-def)

lemma curry-case-prod [simp]: curry \((case-prod\ f) = f\)
by (simp add: curry-def case-prod-unfold)

lemma case-prod-curry [simp]: case-prod \((curry\ f) = f\)
by (simp add: curry-def case-prod-unfold)
**THEORY “Product-Type”**

**lemma** curry-K : curry (\(\lambda x. \ c\)) = (\(\lambda y. \ c\))
by (simp add: fun-eq-iff)

The composition-uncurry combinator.

**definition** scomp :: ('a ⇒ 'b × 'c) ⇒ ('b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'd (infixl \(\circ\rightarrow\) 60)
where \(f \circ\rightarrow g = (\lambda x. \ case-prod \ g \ (f \ x))\)

**no-notation** scomp (infixl \(\circ\rightarrow\) 60)

**bundle** state-combinator-syntax
begin

**notation** fcomp (infixl \(\circ\>) 60)
**notation** scomp (infixl \(\circ\rightarrow\) 60)

end

code-printing
constant scomp ⇀ (Eval) infixl 3 \(\#\rightarrow\)

**map-prod** — action of the product functor upon functions.

**definition** map-prod :: ('a ⇒ 'c) ⇒ ('b ⇒ 'd) ⇒ 'a × 'b ⇒ 'c × 'd
where \( \text{map-prod} \ f \ g = (\lambda(x, y). \ (f \ x, \ g \ y)) \)

lemma \( \text{map-prod-simp} \) [simp, code]: \( \text{map-prod} \ f \ g \ (a, b) = (f \ a, \ g \ b) \)
by (simp add: map-prod-def)

functor \( \text{map-prod} \): \( \text{map-prod} \)
by (auto simp add: split-paired-all)

lemma \( \text{fst-map-prod} \) [simp]: \( \text{fst} \ (\text{map-prod} \ f \ g \ x) = f \ (\text{fst} \ x) \)
by (cases \( x \)) simp-all

lemma \( \text{snd-map-prod} \) [simp]: \( \text{snd} \ (\text{map-prod} \ f \ g \ x) = g \ (\text{snd} \ x) \)
by (cases \( x \)) simp-all

lemma \( \text{fst-comp-map-prod} \) [simp]: \( \text{fst} \circ (\text{map-prod} \ f \ g) = f \circ \text{fst} \)
by (rule ext) simp-all

lemma \( \text{snd-comp-map-prod} \) [simp]: \( \text{snd} \circ (\text{map-prod} \ f \ g) = g \circ \text{snd} \)
by (rule ext) simp-all

lemma \( \text{map-prod-compose} \): \( \text{map-prod} \ (f1 \circ f2) \ (g1 \circ g2) = (\text{map-prod} \ f1 \ g1 \circ \text{map-prod} \ f2 \ g2) \)
by (rule ext) (simp add: map-prod-compositionality comp-def)

lemma \( \text{map-prod-ident} \) [simp]: \( \text{map-prod} \ (\lambda x. \ x) \ (\lambda y. \ y) = (\lambda z. \ z) \)
by (rule ext) (simp add: map-prod.identity)

lemma \( \text{map-prod-imageI} \) [intro]: \((a, b) \in R \Longrightarrow (f \ a, \ g \ b) \in \text{map-prod} \ f \ g \ ' R \)
by (rule image-eqI) simp-all

lemma \( \text{prod-fun-imageE} \) [elim!]:
  assumes major: \( c \in \text{map-prod} \ f \ g \ ' R \)
  and cases: \( \forall x \ y. \ c \ = \ (f \ x, \ g \ y) \Longrightarrow (x, \ y) \in R \Longrightarrow P \)
  shows \( P \)
proof (rule major [THEN imageE])
fix \( x \)
assume \( c = \text{map-prod} \ f \ g \ x \ x \in R \)
then show \( P \)
  using cases by (cases \( x \)) simp
qed

definition \( \text{apfst} :: \ (a \Rightarrow c) \Rightarrow a \times b \Rightarrow c \times b \)
where \( \text{apfst} \ f \ = \ \text{map-prod} \ f \ \text{id} \)

definition \( \text{apsnd} :: \ (b \Rightarrow c) \Rightarrow a \times b \Rightarrow a \times c \)
where \( \text{apsnd} \ f \ = \ \text{map-prod} \ \text{id} \ f \)

lemma \( \text{apfst-conv} \) [simp, code]: \( \text{apfst} \ f \ (x, \ y) = (f \ x, \ y) \)
by (simp add: apfst-def)
**THEORY “Product-Type”**

**lemma** apsnd-conv \[simp, code\]: \( \text{apsnd } f (x, y) = (x, f y) \)
\[\text{by (simp add: apsnd-def)}\]

**lemma** fst-apfst \[simp\]: \( \text{fst } (\text{apfst } f x) = f (\text{fst } x) \)
\[\text{by (cases } x \text{ simp)}\]

**lemma** fst-comp-apfst \[simp\]: \( \text{fst } \circ \text{apfst } f = f \circ \text{fst} \)
\[\text{by (simp add: fun-eq-iff)}\]

**lemma** fst-apsnd \[simp\]: \( \text{fst } \text{apsnd } f x = \text{fst } x \)
\[\text{by (cases } x \text{ simp)}\]

**lemma** fst-comp-apsnd \[simp\]: \( \text{fst } \circ \text{apsnd } f = \text{fst} \)
\[\text{by (simp add: fun-eq-iff)}\]

**lemma** snd-apfst \[simp\]: \( \text{snd } (\text{apfst } f x) = \text{snd } x \)
\[\text{by (cases } x \text{ simp)}\]

**lemma** snd-comp-apfst \[simp\]: \( \text{snd } \circ \text{apfst } f = \text{snd} \)
\[\text{by (simp add: fun-eq-iff)}\]

**lemma** snd-apsnd \[simp\]: \( \text{snd } \text{apsnd } f x = f (\text{snd } x) \)
\[\text{by (cases } x \text{ simp)}\]

**lemma** snd-comp-apsnd \[simp\]: \( \text{snd } \circ \text{apsnd } f = \text{f } \circ \text{snd} \)
\[\text{by (simp add: fun-eq-iff)}\]

**lemma** apfst-compose: \( \text{apfst } f (\text{apfst } g x) = \text{apfst } (f \circ g) x \)
\[\text{by (cases } x \text{ simp)}\]

**lemma** apsnd-compose: \( \text{apsnd } f (\text{apsnd } g x) = \text{apsnd } (f \circ g) x \)
\[\text{by (cases } x \text{ simp)}\]

**lemma** apfst-apsnd \[simp\]: \( \text{apfst } f (\text{apsnd } g x) = (f \text{ (fst } x\text{)}, \text{g (snd } x\text{)}) \)
\[\text{by (cases } x \text{ simp)}\]

**lemma** apsnd-apfst \[simp\]: \( \text{apsnd } f (\text{apfst } g x) = (\text{g (fst } x\text{)}, \text{f (snd } x\text{)}) \)
\[\text{by (cases } x \text{ simp)}\]

**lemma** apfst-id \[simp\]: \( \text{apfst } id = id \)
\[\text{by (simp add: fun-eq-iff)}\]

**lemma** apsnd-id \[simp\]: \( \text{apsnd } id = id \)
\[\text{by (simp add: fun-eq-iff)}\]

**lemma** apfst-eq-conv \[simp\]: \( \text{apfst } f x = \text{apfst } g x \iff f \text{ (fst } x\text{)} = g \text{ (fst } x\text{)} \)
\[\text{by (cases } x \text{ simp)}\]
lemma apsnd-eq-conv [simp]: apsnd f x = apsnd g x ⟷ f (snd x) = g (snd x)
  by (cases x) simp

lemma apsnd-apfst-commute: apsnd f (apfst g p) = apfst g (apsnd f p)
  by simp

class begin

local-setup (Local-Theory.map-background-naming (Name-Space.mandatory-path prod))

definition swap :: 'a × 'b ⇒ 'b × 'a
  where swap p = (snd p, fst p)
end

lemma swap-simp [simp]: prod.swap (x, y) = (y, x)
  by (simp add: prod.swap-def)

lemma swap-swap [simp]: prod.swap (prod.swap p) = p
  by (cases p) simp

lemma swap-comp-swap [simp]: prod.swap ∘ prod.swap = id
  by (simp add: fun-eq-iff)

lemma pair-in-swap-image [simp]: (y, x) ∈ prod.swap ' A ⟷ (x, y) ∈ A
  by (auto intro: image-eqI)

lemma inj-swap [simp]: inj-on prod.swap A
  by (rule inj-onI) auto

lemma swap-inj-on: inj-on (λ(i, j). (j, i)) A
  by (rule inj-onI) auto

lemma surj-swap [simp]: surj prod.swap
  by (rule surjI [of - prod.swap]) simp

lemma bij-swap [simp]: bij prod.swap
  by (simp add: bij-def)

lemma case-swap [simp]: (case prod.swap p of (y, x) ⇒ f x y) = (case p of (x, y) ⇒ f x y)
  by (cases p) simp

lemma fst-swap [simp]: fst (prod.swap x) = snd x
  by (cases x) simp

lemma snd-swap [simp]: snd (prod.swap x) = fst x
THEORY “Product-Type” 296

by (cases x) simp

lemma split-pairs:
(A,B) = X ↔ fst X = A ∧ snd X = B and X = (A,B) ↔ fst X = A ∧ snd X = B
by auto

Disjoint union of a family of sets – Sigma.
definition Sigma :: 'a set ⇒ ('a ⇒ 'b set) ⇒ ('a × 'b) set
where Sigma A B ≡ ∪ x∈A. ∪ y∈B x. {Pair x y}

abbreviation Times :: 'a set ⇒ 'b set ⇒ ('a × 'b) set
(infixr "×" 80)
where A × B ≡ Sigma A (λ-. B)

hide-const (open) Times

bundle no-Set-Product-syntax begin
no-notation Product-Type.Times (infixr "×" 80)
end
bundle Set-Product-syntax begin
notation Product-Type.Times (infixr "×" 80)
end

syntax
-Sigma :: pttrn ⇒ 'a set ⇒ 'b set ⇒ ('a × 'b) set
((3SIGMA :-/ -) [0, 0, 10] 10)
translations
SIGMA x:A. B ⇔ CONST Sigma A (λx. B)

lemma SigmaI [intro!]: a ∈ A ⇒ b ∈ B a ⇒ (a, b) ∈ Sigma A B
unfolding Sigma-def by blast

lemma SigmaE [elim!]: c ∈ Sigma A B ⇒ (∀x y. x ∈ A ⇒ y ∈ B x ⇒ c = (x, y) ⇒ P) ⇒ P
— The general elimination rule.
unfolding Sigma-def by blast

Elimination of (a, b) ∈ A × B – introduces no eigenvariables.
lemma SigmaD1: (a, b) ∈ Sigma A B ⇒ (a, b) ∈ Sigma A B
by blast

lemma SigmaD2: (a, b) ∈ Sigma A B ⇒ b ∈ B a
by blast

lemma SigmaE2: (a, b) ∈ Sigma A B ⇒ (a ∈ A ⇒ b ∈ B a ⇒ P) ⇒ P
by blast

lemma Sigma-cong: A = B ⇒ (∀x. x ∈ B ⇒ C x = D x) ⇒ (SIGMA x:A. C x) = (SIGMA x:B. D x)
by auto

lemma Sigma-mono: \( A \subseteq C \implies (\forall x. x \in A \implies B x \subseteq D x) \implies \Sigma A B \subseteq \Sigma C D \)
by blast

lemma Sigma-empty1 [simp]: \( \Sigma \{\} B = \{\} \)
by blast

lemma Sigma-empty2 [simp]: \( A \times \{\} = \{\} \)
by blast

lemma UNIV-Times-UNIV [simp]: \( \text{UNIV} \times \text{UNIV} = \text{UNIV} \)
by auto

lemma Compl-Times-UNIV1 [simp]: \(- (\text{UNIV} \times A) = \text{UNIV} \times (-A) \)
by auto

lemma Compl-Times-UNIV2 [simp]: \(- (A \times \text{UNIV}) = (-A) \times \text{UNIV} \)
by auto

lemma mem-Sigma-iff [iff]: \((a, b) \in \Sigma A B \iff a \in A \land b \in B \)
by blast

lemma mem-Times-iff: \( x \in A \times B \iff \text{fst } x \in A \land \text{snd } x \in B \)
by (induct x) simp

lemma Sigma-empty-iff: \( (\forall i \in I. X i) = \{\} \iff (\forall i \in I. X i = \{\}) \)
by auto

lemma Times-subset-cancel2: \( x \in C \implies A \times C \subseteq B \times C \iff A \subseteq B \)
by blast

lemma Times-eq-cancel2: \( x \in C \implies A \times C = B \times C \iff A = B \)
by (blast elim: equalityE)

lemma Collect-case-prod-Sigma: \( \{\langle x, y \rangle. \ P x \land Q x y\} = (\Sigma x.\ P. \ \text{Collect } (Q x)) \)
by blast

lemma Collect-case-prod [simp]: \( \{\langle a, b \rangle. \ P a \land Q b\} = \text{Collect } P \times \text{Collect } Q \)
by (fact Collect-case-prod-Sigma)

lemma Collect-case-prodD: \( x \in \text{Collect } (\text{case-prod } A) \implies A (\text{fst } x) (\text{snd } x) \)
by auto

lemma Collect-case-prod-mono: \( A \leq B \implies \text{Collect } (\text{case-prod } A) \subseteq \text{Collect } (\text{case-prod } B) \)
by auto (auto elim!: le-funE)
THEORY “Product-Type”

lemma Collect-split-mono-strong:
  \[ X = \text{fst} \, 'A \Rightarrow Y = \text{snd} \, 'A \Rightarrow \forall a \in X. \forall b \in Y. \ P a b \rightarrow Q a b \]
  \[ \Rightarrow A \subseteq \text{Collect} \, (\text{case-prod} \ P) \Rightarrow A \subseteq \text{Collect} \, (\text{case-prod} \ Q) \]
  by fastforce

lemma UN-Times-distrib: \[
\bigcup \,(a, b) \in A \times B. E a \times F b) \bigcup \,(E \,' A) \times \bigcup \,(F \,' B)
\] — Suggested by Pierre Chartier
  by blast

lemma split-paired-Ball-Sigma [simp, no-atp]: \[
(\forall z \in \Sigma A B. P z) \iff (\forall x \in A. \forall y \in B . P (x, y))
\]
  by blast

lemma split-paired-Bex-Sigma [simp, no-atp]: \[
(\exists z \in \Sigma A B. P z) \iff (\exists x \in A. \exists y \in B . P (x, y))
\]
  by blast

lemma Sigma-Un-distrib1: \[
\Sigma (I \cup J) C = \Sigma I C \cup \Sigma J C
\]
  by blast

lemma Sigma-Un-distrib2: \[
(\Sigma i : I. A i \cup B i) = \Sigma I A \cup \Sigma I B
\]
  by blast

lemma Sigma-Int-distrib1: \[
\Sigma (I \cap J) C = \Sigma I C \cap \Sigma J C
\]
  by blast

lemma Sigma-Int-distrib2: \[
(\Sigma i : I. A i \cap B i) = \Sigma I A \cap \Sigma I B
\]
  by blast

lemma Sigma-Diff-distrib1: \[
\Sigma (I - J) C = \Sigma I C - \Sigma J C
\]
  by blast

lemma Sigma-Diff-distrib2: \[
(\Sigma i : I. A i - B i) = \Sigma I A - \Sigma I B
\]
  by blast

lemma Sigma-Union: \[
\Sigma (\bigcup X) B = (\bigcup A \in X. \Sigma A B)
\]
  by blast

lemma Pair-vimage-Sigma: \[
\text{Pair} \ x - ' \Sigma A f = (\text{if} \ x \in A \text{ then} \ f \ x \text{ else} \ {} )
\]
  by auto

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma Times-Un-distrib1: \[
(A \cup B) \times C = A \times C \cup B \times C
\]
  by (fact Sigma-Un-distrib1)

lemma Times-Int-distrib1: \[
(A \cap B) \times C = A \times C \cap B \times C
\]
  by (fact Sigma-Int-distrib1)
lemma Times-Diff-distrib1: 
\((A - B) \times C = A \times C - B \times C\)
by (fact Sigma-Diff-distrib1)

lemma Times-empty [simp]: 
\(A \times B = \{\}\) \iff \(A = \{\} \lor B = \{\}\)
by auto

lemma times-subset-iff:
\((A \times C) \subseteq B \times D) \iff (A = \{\} \lor C = \{\} \lor A \subseteq B \land C \subseteq D\)
by blast

lemma times-eq-iff:
\((A \times B) = (C \times D) \iff (A = C \land B = D) \lor (A = \{\} \lor B = \{\})) \land (C = \{\} \lor D = \{\})\)
by auto

lemma fst-image-times [simp]: 
\(\text{fst} \ ' (A \times B) = (\text{if } B = \{\} \text{ then } \{\} \text{ else } A)\)
by force

lemma snd-image-times [simp]: 
\(\text{snd} \ ' (A \times B) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } B)\)
by force

lemma fst-image-Sigma: 
\(\text{fst} \ ' (\Sigma A B) = \{x \in A. B(x) \neq \{\}\}\)
by force

lemma snd-image-Sigma: 
\(\text{snd} \ ' (\Sigma A B) = (\bigcup x \in A. B x)\)
by force

lemma vimage-fst: 
\(\text{fst} -\ ' A = A \times \text{UNIV}\)
by auto

lemma vimage-snd: 
\(\text{snd} -\ ' A = \text{UNIV} \times A\)
by auto

lemma insert-Times-insert [simp]:
\(\text{insert} a A \times \text{insert} b B = \text{insert} (a,b) (A \times \text{insert} b B \cup \{a\} \times B)\)
by blast

lemma sing-Times-sing: 
\(\{x\} \times \{y\} = \{(x,y)\}\)
by simp

lemma vimage-Times: 
\(f -\ ' (A \times B) = (\text{fst} \circ f) -\ ' (A \cap (\text{snd} \circ f) -\ ' B)\)
proof (rule set-eqI)
show \(x \in f -\ ' (A \times B) \iff x \in (\text{fst} \circ f) -\ ' (A \cap (\text{snd} \circ f) -\ ' B)\) for \(x\)
by (cases \(f x\)) (auto split: prod.split)
qed

lemma Times-Int-Times: 
\((A \times B) \cap C \times D = (A \cap C) \times (B \cap D)\)
by auto

lemma image-paired-Times:
\[(\lambda(x,y). (f x, g y)) \cdot (A \times B) \equiv (f \cdot A) \times (g \cdot B)\]

by auto

**lemma** product-swap: prod.swap \cdot (A \times B) = B \times A

by (auto simp add: set-eq-iff)

**lemma** swap-product: (\lambda(i, j). (j, i)) \cdot (A \times B) = B \times A

by (auto simp add: set-eq-iff)

**lemma** image-split-eq-Sigma: (\lambda x. (f x, g x)) \cdot A = \Sigma(f \cdot A) (\lambda x. g \cdot (f - \cdot \{x\} \cap A))

**proof** (safe intro!: imageI)

fix a b

assume *: a \in A b \in A and eq: f a = f b

show (f b, g a) \in (\lambda x. (f x, g x)) \cdot A

using * eq[symmetric] by auto

qed simp-all

**lemma** subset-fst-snd: A \subseteq (fst \cdot A \times snd \cdot A)

by force

**lemma** inj-on-apsnd [simp]: inj (apsnd f) (UNIV \times A) \iff inj f A

by (auto simp add: inj-on-def)

**lemma** inj-apfst [simp]: inj (apfst f) (A \times UNIV) \iff inj f

using inj-on-apsnd[of f UNIV] by simp

**lemma** inj-apfst [simp]: inj (apfst f) \iff inj f

using inj-on-apfst[of f UNIV] by simp

context

begin

qualified definition product :: 'a set \Rightarrow 'b set \Rightarrow ('a \times 'b) set

where [code-abbrev]: product A B = A \times B

**lemma** member-product: x \in Product-Type.product A B \iff x \in A \times B

by (simp add: product-def)

end

The following map-prod lemmas are due to Joachim Breitner:

**lemma** map-prod-inj-on:

assumes inj-on f A

and inj-on g B

shows inj-on (map-prod f g) (A \times B)
proof (rule inj-onI)
  fix x :: 'a × 'c
  fix y :: 'a × 'c
  assume x ∈ A × B
  then have fst x ∈ A and snd x ∈ B by auto
  assume y ∈ A × B
  then have map-prod f g x = map-prod f g y
  then have f (fst x) = f (fst y) by (cases x, cases y) auto
  with (inj-on f A) and (fst x ∈ A) and (fst y ∈ A) have fst x = fst y
  by (auto dest: inj-onD)
moreover from (map-prod f g x = map-prod f g y)
  have snd (map-prod f g x) = snd (map-prod f g y) by auto
  then have g (snd x) = g (snd y) by (cases x, cases y) auto
  with (inj-on g B) and (snd x ∈ B) and (snd y ∈ B) have snd x = snd y
  by (auto dest: inj-onD)
ultimately show x = y by (rule prod-eqI)
qed

lemma map-prod-surj:
  fixes f :: 'a ⇒ 'b
  and g :: 'c ⇒ 'd
  assumes surj f and surj g
  shows surj (map-prod f g)
  unfolding surj-def
proof
  fix y :: 'b × 'd
  from (surj f) obtain a where fst y = f a
  by (auto elim: surjE)
moreover
  from (surj g) obtain b where snd y = g b
  by (auto elim: surjE)
ultimately have (fst y, snd y) = map-prod f g (a,b)
  by auto
then show ∃ x. y = map-prod f g x
  by auto
qed

lemma map-prod-surj-on:
  assumes f :: 'A = A' and g :: 'B = B'
  shows map-prod f g :: (A × B) = A' × B'
  unfolding image-def
proof (rule set-eqI, rule iffI)
  fix x :: 'a × 'c
  assume x ∈ { y::'a × 'c. ∃ x::'b × 'd∈ A × B. y = map-prod f g x}
then obtain y where y ∈ A × B and x = map-prod f g y
  by blast
from (image f A = A') and (y ∈ A × B) have f (fst y) ∈ A'

qed
by auto
moreover from (image g B = B' \land \forall y \in A \times B. \text{have } g (\text{snd } y) \in B'
by auto
ultimately have (f (\text{fst } y), g (\text{snd } y)) \in (A' \times B')
by auto
with \(x = \text{map-prod } f g y\) show \(x \in A' \times B'
by (cases y) auto

next
fix \(x :: 'a \times 'c\)
assume \(x \in A' \times B'\)
then have \(\text{fst } x \in A' \land \text{snd } x \in B'\)
by auto
from (image f A = A') \land (\text{fst } x \in A') \text{ have } \text{fst } x \in \text{image } f A
by auto
then obtain \(a\) where \(a \in A\) \land \(\text{fst } x = f a\)
by (rule imageE)
moreover from (image g B = B' \land (\text{snd } x \in B') \text{ obtain } \(b\) where \(b \in B\)
and \(\text{snd } x = g b\)
by auto
ultimately have (\text{fst } x, \text{snd } x) = \text{map-prod } f g (a, b)
by auto
moreover from \((a \in A) \land (b \in B) \text{ have } (a \land b) \in A \times B\)
by auto
ultimately have \(\exists y \in A \times B. x = \text{map-prod } f g y\)
by auto
then show \(x \in \{x. \exists y \in A \times B. x = \text{map-prod } f g y\}\)
by auto
qed

14.4 Simproc for rewriting a set comprehension into a point-free expression

ML-file (Tools/set-comprehension-pointfree.ML)

simproc-setup passive set-comprehension (Collect P) = 
\(<K\text{-Set-Comprehension-Pointfree.code-proc}>\)

setup Code-Preproc.map-pre (fn ctxt => ctxt addsimprocs [simple proc (set-comprehension)])

14.5 Lemmas about disjointness

lemma disjnt-Times1-iff [simp]: disjnt \((C \times A) (C \times B) \iff C = \{\} \lor \text{disjnt } A B\)
by (auto simp: disjnt-def)

lemma disjnt-Times2-iff [simp]: disjnt \((A \times C) (B \times C) \iff C = \{\} \lor \text{disjnt } A B\)
by (auto simp: disjnt-def)
lemma disjnt-Sigma-iff: disjnt \( (\Sigma A C) \) \( (\Sigma B C) \) \( \iff \) \( (\forall i \in A \cap B. \ C i = \{\}) \ \vee \ \text{disjnt} \ A \ B \) by (auto simp: disjnt-def)

14.6 Inductively defined sets

simpsetup Collect-mem (Collect t) = 
\( K \ (fn \ ct x => \ fn \ ct y => \) \( (\text{case} \ \text{Thm} \ . \ \text{term-of} \ ct \ \text{of} \) Const (const-name\( \langle \text{Collect} \rangle \), Type (type-name\( \langle \text{fun} \rangle \), [-, T])) \$ t => \) 
let val \( (u, -, ps) = \) HOLogic.strip_ptupleabs \( t \) in 
\( (\text{case} \ u \ \text{of} \) Const (const-name\( \langle \text{Set.member} \rangle \), -) \$ q \$ S' => \) 
\( (\text{case} \ \text{try} \ \text{HOLogic} \ . \ \text{strip_ptuple} \ ps \ q \ \text{of} \) NONE => \) NONE 
\| SOME \( ts \) => 
if not \( (\text{Term} \ . \ \text{is_open} \ S') \) \( \text{andalso} \) 
\( ts = \) map Bound \( (\text{length} \ ps \ \text{downto} \ 0) \) 
then 
let val simp = 
full-simp-tac \( \text{put_simpset} \ \text{HOL_basic_ss} \) ctxt 
\( \text{add_simps} \ \{\{\text{thm \ split-paired-all}\}, \{\text{thm \ case_prod_conv}\}\} \) \( 1 \) 
in 
SOME (Goal.prove ctxt [] [] 
(\text{Const} \ (\text{const-name} \langle \text{Pure.eq} \rangle \), T ---\ T ---\ propT) \$ S \$ S') \( K \) \( \text{EVERY} \) 
[resolve_tac ctxt [eq_reflection] \( 1 \), 
\text{resolve_tac} \ ctxt \ \{\text{thms \ subset-antisym}\} \ 1, 
\text{resolve_tac} \ ctxt \ \{\text{thms \ subsetI}\} \ 1, 
\text{resolve_tac} \ ctxt \ \{\text{thms \ CollectD}\} \ 1, \ simp, 
\text{resolve_tac} \ ctxt \ \{\text{thms \ CollectI}\} \ 1, 
\text{resolve_tac} \ ctxt \ \{\text{thms \ CollectI}\} \ 1, \ simp])])
end 
else NONE) 
| - => NONE) 
end 
| - => NONE))
)

ML-file ⟨Tools/inductive-set.ML⟩

14.7 Legacy theorem bindings and duplicates

lemmas fst-conv = prod.sel(1)
lemmas snd-conv = prod.sel(2)
lemmas split-def = case_prod-unfold
lemmas split-beta' = case_prod-beta'
lemmas split-beta = prod.case_eq-if
lemmas split-conv = case_prod-conv
lemmas split = case-prod-conv

hide-const (open) prod

end

15 The Disjoint Sum of Two Types

theory Sum-Type
  imports Typedef Inductive Fun
begin

15.1 Construction of the sum type and its basic abstract operations

definition Inl-Rep :: 'a ⇒ 'a ⇒ 'b ⇒ bool ⇒ bool
  where Inl-Rep a x y p ←→ x = a ∧ p

definition Inr-Rep :: 'b ⇒ 'a ⇒ 'b ⇒ bool ⇒ bool
  where Inr-Rep b x y p ←→ y = b ∧ ¬ p

definition sum = { f. (∃ a. f = Inl-Rep (a::'a)) ∨ (∃ b. f = Inr-Rep (b::'b))}

typedef ('a, 'b) sum (infixr + 10) = sum :: ('a ⇒ 'b ⇒ bool ⇒ bool) set

unfolding sum-def by auto

lemma Inl-RepI: Inl-Rep a ∈ sum
  by (auto simp add: sum-def)

lemma Inr-RepI: Inr-Rep b ∈ sum
  by (auto simp add: sum-def)

lemma inj-on-Abs-sum: A ⊆ sum ⇒ inj-on Abs-sum A
  by (rule inj-on-inverseI, rule Abs-sum-inverse) auto

lemma Inl-Rep-inject: inj-on Inl-Rep A
  proof (rule inj-onI)
    show ∀ a c. Inl-Rep a = Inl-Rep c ⇒ a = c
      by (auto simp add: Inl-Rep-def fun-eq-iff)
  qed

lemma Inr-Rep-inject: inj-on Inr-Rep A
  proof (rule inj-onI)
    show ∀ b d. Inr-Rep b = Inr-Rep d ⇒ b = d
      by (auto simp add: Inr-Rep-def fun-eq-iff)
  qed

lemma Inl-Rep-not-Inr-Rep: Inl-Rep a ≠ Inr-Rep b
  by (auto simp add: Inl-Rep-def Inr-Rep-def fun-eq-iff)
definition Inl :: 'a ⇒ 'a + 'b
  where Inl = Abs-sum ◦ Inl-Rep

definition Inr :: 'b ⇒ 'a + 'b
  where Inr = Abs-sum ◦ Inr-Rep

lemma inj-Inl [simp]: inj-on Inl A
  by (auto simp add: Inl-def intro: comp-inj-on Inl-Rep-inject inj-on-Abs-sum Inl-RepI)

lemma Inl-inject: Inl x = Inl y ⟹ x = y
  using inj-Inl by (rule injD)

lemma inj-Inr [simp]: inj-on Inr A
  by (auto simp add: Inr-def intro: comp-inj-on Inr-Rep-inject inj-on-Abs-sum Inr-RepI)

lemma Inr-inject: Inr x = Inr y ⟹ x = y
  using inj-Inr by (rule injD)

lemma Inl-not-Inr: Inl a ≠ Inr b
proof –
  have {Inl-Rep a, Inr-Rep b} ⊆ sum
    using Inl-RepI [of a] Inr-RepI [of b] by auto
  with inj-on-Abs-sum have inj-on Abs-sum {Inl-Rep a, Inr-Rep b}.
  with Inl-Rep-not-Inr-Rep [of a b] inj-on-contraD have Abs-sum (Inl-Rep a) ≠
    Abs-sum (Inr-Rep b)
    by auto
  then show ?thesis
    by (simp add: Inl-def Inr-def)
qed

lemma Inr-not-Inl: Inr b ≠ Inl a
  using Inl-not-Inr by (rule not-sym)

lemma sumE:
  assumes \( \forall x::'a. s = \text{Inl } x \implies P \)
  and \( \forall y::'b. s = \text{Inr } y \implies P \)
  shows P
proof (rule Abs-sum-cases [of s])
  fix f
  assume s = Abs-sum f and f ∈ sum
  with assms show P
  by (auto simp add: sum-def Inl-def Inr-def)
qed

free-constructors case-sum for
  isl: Inl projl
THEORY “Sum-Type”

| Inr projr
| by (erule sumE, assumption) (auto dest: Inl-inject Inr-inject simp add: Inl-not-Inr)

Avoid name clashes by prefixing the output of old-rep-datatype with old.

setup ⟨Sign.mandatory-path old⟩

old-rep-datatype Inl Inr
proof −
fix P
fix s :: 'a + 'b
assume x: ∀x::'a. P (Inl x) and y: ∀y::'b. P (Inr y)
then show P s by (auto intro: sumE [of s])
qed (auto dest: Inl-inject Inr-inject simp add: Inl-not-Inr)

setup ⟨Sign.parent-path⟩

But erase the prefix for properties that are not generated by free-constructors.

setup ⟨Sign.mandatory-path sum⟩

declare
old.sum.inject[iff del]
old.sum.distinct(1)[simp del, induct-simp del]

lemmas induct = old.sum.induct
lemmas inducts = old.sum.inducts
lemmas rec = old.sum.rec
lemmas simps = sum.inject sum.distinct sum.case sum.rec

setup ⟨Sign.parent-path⟩

primrec map-sum :: ('a ⇒ 'c) ⇒ ('b ⇒ 'd) ⇒ 'a + 'b ⇒ 'c + 'd
where
map-sum f1 f2 (Inl a) = Inl (f1 a)
| map-sum f1 f2 (Inr a) = Inr (f2 a)

functor map-sum: map-sum
proof −
show map-sum f g ∘ map-sum h i = map-sum (f ∘ h) (g ∘ i) for f g h i
proof
show (map-sum f g ∘ map-sum h i) s = map-sum (f ∘ h) (g ∘ i) s for s
by (cases s) simp-all
qed
show map-sum id id = id
proof
show map-sum id id s = id s for s
by (cases s) simp-all
qed

qed
lemma split-sum-all: \((\forall x. P x) \iff (\forall x. P (Inl x)) \land (\forall x. P (Inr x))\)
by (auto intro: sum.induct)

lemma split-sum-ex: \((\exists x. P x) \iff (\exists x. P (Inl x)) \lor (\exists x. P (Inr x))\)
using split-sum-all[of \(\lambda x. \neg P x\)] by blast

15.2 Projections

lemma case-sum-KK [simp]: case-sum \((\lambda x. a) \ (\lambda x. a) = (\lambda x. a)\)
by (rule ext) (simp split: sum.split)

lemma surjective-sum: case-sum \((\lambda x:\cdot a. f (Inl x)) \ (\lambda y:\cdot b. f (Inr y)) = f\)
proof
  fix \(s::\cdot a + \cdot b\)
  show case s of Inl \((x::\cdot a) \Rightarrow f (Inl x) | Inr \((y::\cdot b) \Rightarrow f (Inr y)) = f \) \(s\)
    by (cases s) simp-all
qed

lemma case-sum-inject:
  assumes \(a::\cdot a + \cdot b\)
  shows \(P\)
proof (rule \(r\))
  show \(f1 = g1\)
  proof
    fix \(x::\cdot a\)
    from \(a\) have case-sum \(f1 f2\) \((Inl x) = case-sum g1 g2 \(Inl x)\) by simp
    then show \(f1 x = g1 x\) by simp
  qed
  show \(f2 = g2\)
  proof
    fix \(y::\cdot b\)
    from \(a\) have case-sum \(f1 f2\) \((Inr y) = case-sum g1 g2 \(Inr y)\) by simp
    then show \(f2 y = g2 y\) by simp
  qed
qed

primrec Suml :: \(\cdot a \Rightarrow \cdot c\) \(\Rightarrow \cdot a + \cdot b \Rightarrow \cdot c\)
where \(Suml f \ (Inl x) = f x\)

primrec Sumr :: \(\cdot b \Rightarrow \cdot c\) \(\Rightarrow \cdot a + \cdot b \Rightarrow \cdot c\)
where \(Sumr f \ (Inr x) = f x\)

lemma Suml-inject:
  assumes \(Suml f = Suml g\)
  shows \(f = g\)
proof
  fix \(x::\cdot a\)
  let \(\ ?s = Inl x::\cdot a + \cdot b\)
from assms have Suml f ?s = Suml g ?s by simp
then show f x = g x by simp
qed

lemma Sumr-inject:
assumes Sumr f = Sumr g
shows f = g
proof
fix x :: 'b
let ?s = Inr x :: 'a + 'b
from assms have Sumr f ?s = Sumr g ?s by simp
then show f x = g x by simp
qed

15.3 The Disjoint Sum of Sets

definition Plus :: 'a set ⇒ 'b set ⇒ ('a + 'b) set (infixr <+>) 65
  where A <+> B = Inl ' A ∪ Inr ' B

hide-const (open) Plus — Valuable identifier

lemma InlI [intro!]: a ∈ A ⇒ Inl a ∈ A <+> B
  by (simp add: Plus-def)

lemma InrI [intro!]: b ∈ B ⇒ Inr b ∈ A <+> B
  by (simp add: Plus-def)

Exhaustion rule for sums, a degenerate form of induction

lemma PlusE [elim!]:
  u ∈ A <+> B ⇒ (\x. x ∈ A ⇒ u = Inl x ⇒ P) ⇒ (\y. y ∈ B ⇒ u = Inr y ⇒ P) ⇒ P
  by (auto simp add: Plus-def)

lemma Plus-eq-empty-conv [simp]: A <+> B = {} ←→ A = {} ∧ B = {}
  by auto

lemma UNIV-Plus-UNIV [simp]: UNIV <+> UNIV = UNIV
  proof (rule set-eqI)
  fix u :: 'a + 'b
  show u ∈ UNIV <+> UNIV ←→ u ∈ UNIV by (cases u) auto
  qed

lemma UNIV-sum: UNIV = Inl ' UNIV ∪ Inr ' UNIV
  proof
  have x ∈ range Inl if x ∉ range Inr for x :: 'a + 'b
  using that by (cases x) simp-all
  then show ?thesis by auto
  qed
16 Rings

theory Rings
  imports Groups Set Fun
begin

16.1 Semirings and rings

class semiring = ab-semigroup-add + semigroup-mult +
  assumes distrib-right [algebra-simps, algebra-split-simps]: (a + b) * c = a * c + b * c
  assumes distrib-left [algebra-simps, algebra-split-simps]: a * (b + c) = a * b + a * c
begin

For the combine-numerals simproc

lemma combine-common-factor: a * e + (b * e + c) = (a + b) * e + c
  by (simp add: distrib-right ac-simps)
end

class mult-zero = times + zero +
  assumes mult-zero-left [simp]: 0 * a = 0
  assumes mult-zero-right [simp]: a * 0 = 0
begin

lemma mult-not-zero: a * b ≠ 0 → a ≠ 0 ∧ b ≠ 0
  by auto
end

class semiring-0 = semiring + comm-monoid-add + mult-zero

class semiring-0-cancel = semiring + cancel-comm-monoid-add
begin

subclass semiring-0
proof
  fix a :: 'a
  have 0 * a + 0 * a = 0 * a + 0
    by (simp add: distrib-right [symmetric])
  then show 0 * a = 0
    by (simp only: add-left-cancel)
  have a * 0 + a * 0 = a * 0 + 0
    by (simp add: distrib-left [symmetric])
end
then show \( a \ast 0 = 0 \)
  by (simp only: add-left-cancel)
qed

end

class comm-semiring = ab-semigroup-add + ab-semigroup-mult +
  assumes distrib: \((a + b) \ast c = a \ast c + b \ast c\)
begin

subclass semiring
proof
  fix a b c :: 'a
  show \((a + b) \ast c = a \ast c + b \ast c\)
    by (simp add: distrib)
  have a \ast \((b + c) = (b + c) \ast a\)
    by (simp add: ac-simps)
  also have \(\ldots = b \ast a + c \ast a\)
    by (simp only: distrib)
  also have \(\ldots = a \ast b + a \ast c\)
    by (simp add: ac-simps)
  finally show a \ast \((b + c) = a \ast b + a \ast c\)
    by blast
qed

end

class comm-semiring-0 = comm-semiring + comm-monoid-add + mult-zero
begin

subclass semiring-0 ..
end

class comm-semiring-0-cancel = comm-semiring + cancel-comm-monoid-add
begin

subclass semiring-0-cancel ..
subclass comm-semiring-0 ..
end

class zero-neq-one = zero + one +
  assumes zero-neq-one [simp]: \(0 \neq 1\)
begin

lemma one-neq-zero [simp]: \(1 \neq 0\)
  by (rule not-sym) (rule zero-neq-one)
definition of-bool :: bool ⇒ 'a
  where of-bool p = (if p then 1 else 0)

lemma of-bool-eq [simp, code]:
  of-bool False = 0
  of-bool True = 1
  by (simp-all add: of-bool-def)

lemma of-bool-eq-iff: of-bool p = of-bool q ⟷ p = q
  by (simp add: of-bool-def)

lemma split-of-bool [split]: P (of-bool p) ⟷ (p → P 1) ∧ (∼ p → P 0)
  by (cases p) simp-all

lemma split-of-bool-asn: P (of-bool p) ⟷ (∼ (p ∧ ∼ P 1 ∨ ∼ p ∧ ∼ P 0))
  by (cases p) simp-all

lemma of-bool-eq-0-iff [simp]:
  of-bool P = 0 ⟷ ∼ P,
  by simp

lemma of-bool-eq-1-iff [simp]:
  of-bool P = 1 ⟷ P,
  by simp

end

class semiring-1 = zero-neq-one + semiring-0 + monoid-mult
begin

lemma of-bool-conj:
  of-bool (P ∧ Q) = of-bool P * of-bool Q
  by auto

end

lemma lambda-zero: (∀ h::'a::mult-zero. 0) = (∗ 0
  by auto

lemma lambda-one: (∀ x::'a::monoid-mult. x) = (∗ 1
  by auto

16.2 Abstract divisibility

class dvd = times
begin

definition dvd :: 'a ⇒ 'a ⇒ bool (infix dvd 50)
where \( b \) dvd \( a \) \( \iff \) (\( \exists k. a = b \ast k \) )

**Lemma dvdI [intro?]:** \( a = b \ast k \implies b \) dvd \( a \)

unfolding dvd-def ..

**Lemma dvdE [elim]:** \( b \) dvd \( a \) \( \implies (\forall k. a = b \ast k \implies P) \implies P \)

unfolding dvd-def by blast

end

color context comm-monoid-mult
begin

subclass dvd .

**Lemma dvd-refl [simp]:** \( a \) dvd \( a \)

proof
do show \( a = a \ast 1 \) by simp

qed

**Lemma dvd-trans [trans]:**

assumes \( a \) dvd \( b \) and \( b \) dvd \( c \)

shows \( a \) dvd \( c \)

proof —
do from assms obtain \( v \) where \( b = a \ast v \)

by auto

moreover from assms obtain \( w \) where \( c = b \ast w \)

by auto

ultimately have \( c = a \ast (v \ast w) \)

by (simp add: mult.assoc)

then show \?thesis ..

qed

**Lemma subset-divisors-dvd: \{ c. \ c \) dvd \( a \}\} \subseteq \{ c. \ c \) dvd \( b \}\} \iff a \) dvd \( b \)

by (auto simp add: subset-iff intro: dvd-trans)

**Lemma strict-subset-divisors-dvd: \{ c. \ c \) dvd \( a \}\) \subset \{ c. \ c \) dvd \( b \}\) \iff a \) dvd \( b \land \neg b \)

dvd \( a \)

by (auto simp add: subset-iff intro: dvd-trans)

**Lemma one-dvd [simp]:** \( 1 \) dvd \( a \)

by (auto intro: dvdI)

**Lemma dvd-mult [simp]:** \( a \) dvd \( (b \ast c) \) if \( a \) dvd \( c \)

using that by (auto intro: mult.left-commute dvdI)

**Lemma dvd-mult2 [simp]:** \( a \) dvd \( (b \ast c) \) if \( a \) dvd \( b \)

using that dvd-mult [of \( a \) \( b \) \( c \)] by (simp add: ac-simps)
lemma dvd-triv-right [simp]: a dvd b * a
  by (rule dvd-mult) (rule dvd-refl)

lemma dvd-triv-left [simp]: a dvd a * b
  by (rule dvd-mult2) (rule dvd-refl)

lemma mult-dvd-mono:
  assumes a dvd b
  and c dvd d
  shows a * c dvd b * d
proof –
  from ‹a dvd b› obtain b' where b = a * b' ..
  moreover from ‹c dvd d› obtain d' where d = c * d' ..
  ultimately have b * d = (a * c) * (b' * d')
    by (simp add: ac-simps)
  then show ?thesis ..
qed

lemma dvd-mult-left: a * b dvd c ⇒ a dvd c
  by (simp add: dvd-def mult.assoc) blast

lemma dvd-mult-right: a * b dvd c ⇒ b dvd c
  using dvd-mult-left [of b a c] by (simp add: ac-simps)
end

class comm-semiring-1 = zero-neq-one + comm-semiring-0 + comm-monoid-mult
begin
subclass semiring-1 ..

lemma dvd-0-left-iff [simp]: 0 dvd a ←→ a = 0
  by auto

lemma dvd-0-right [iff]: a dvd 0
proof
  show 0 = a * 0 by simp
qed

lemma dvd-0-left: 0 dvd a ⇒ a = 0
  by simp

lemma dvd-add [simp]:
  assumes a dvd b and a dvd c
  shows a dvd (b + c)
proof –
  from ‹a dvd b› obtain b' where b = a * b' ..
  moreover from ‹a dvd c› obtain c' where c = a * c' ..
  ultimately have b + c = a * (b' + c')

by (simp add: distrib-left)
then show ?thesis ..
qed

end

class semiring-1-cancel = semiring + cancel-comm-monoid-add
+ zero-neq-one + monoid-mult
begin
subclass semiring-0-cancel ..
subclass semiring-1 ..
end

class comm-semiring-1-cancel =
comm-semiring + cancel-comm-monoid-add + zero-neq-one + comm-monoid-mult
+
assumes right-diff-distrib’ [algebra-simps, algebra-split-simps]:
a * (b - c) = a * b - a * c
begin
subclass semiring-1-cancel ..
subclass comm-semiring-0-cancel ..
subclass comm-semiring-1 ..

lemma left-diff-distrib’ [algebra-simps, algebra-split-simps]:
(b - c) * a = b * a - c * a
by (simp add: algebra-simps)

lemma dvd-add-times-triv-left-iff [simp]: a dvd c * a + b ↔ a dvd b
proof
have a dvd a * c + b ↔ a dvd b (is ?P ↔ ?Q)
proof
assume ?Q
then show ?P by simp
next
assume ?P
then obtain d where a * c + b = a * d ..
then have a * c + b - a * c = a * d - a * c by simp
then have b = a * (d - c) by (simp add: algebra-simps)
then show ?Q ..
qed
then show a dvd c * a + b ↔ a dvd b by (simp add: ac-simps)
qed

lemma dvd-add-times-triv-right-iff [simp]: a dvd b + c * a ↔ a dvd b
using dvd-add-times-triv-left-iff [of a c b] by (simp add: ac-simps)

lemma dvd-add-triv-left-iff [simp]: a dvd a + b ←→ a dvd b
using dvd-add-times-triv-left-iff [of a 1 b] by simp

lemma dvd-add-triv-right-iff [simp]: a dvd b + a ←→ a dvd b
using dvd-add-times-triv-right-iff [of a b 1] by simp

lemma dvd-add-right-iff:
assumes a dvd b
shows a dvd c + c ←→ a dvd c (is ?P ←→ ?Q)
proof
assume ?P
then obtain d where b + c = a * d ..
moreover from (a dvd b) obtain e where b = a * e ..
ultimately have a * e + c = a * d by simp
then have a * e + c - a * e = a * d - a * e by simp
then have c = a * (d - c) by (simp add: algebra-simps)
then show ?Q ..
next
assume ?Q
with assms show ?P by simp
qed

lemma dvd-add-left-iff: a dvd c + c ←→ a dvd c
using dvd-add-right-iff [of a c b] by (simp add: ac-simps)

end

class ring = semiring + ab-group-add

begin

subclass semiring-0-cancel ..

Distribution rules

lemma minus-mult-left: -(a * b) = - a * b
by (rule minus-unique) (simp add: distrib-right [symmetric])

lemma minus-mult-right: -(a * b) = a * - b
by (rule minus-unique) (simp add: distrib-left [symmetric])

Extract signs from products

lemmas mult-minus-left [simp] = minus-mult-left [symmetric]
lemmas mult-minus-right [simp] = minus-mult-right [symmetric]

lemma minus-mult-minus [simp]: - a * - b = a * b
by simp
lemma minus-mult-commute: 
by simp

lemma right-diff-distrib [algebra-simps, algebra-split-simps]: 
a * (b - c) = a * b - a * c
using distrib-left [of a b - c] by simp

lemma left-diff-distrib [algebra-simps, algebra-split-simps]: 
(a - b) * c = a * c - b * c
using distrib-right [of a - b c] by simp

lemmas ring-distribs = distrib-left distrib-right left-diff-distrib right-diff-distrib

lemma eq-add-iff1: 
a * e + c = b * e + d  \iff  (a - b) * e + c = d
by (simp add: algebra-simps)

lemma eq-add-iff2: 
a * e + c = b * e + d  \iff  c = (b - a) * e + d
by (simp add: algebra-simps)

end

lemmas ring-distribs = distrib-left distrib-right left-diff-distrib right-diff-distrib

class comm-ring = comm-semiring + ab-group-add
begin

subclass ring ..
subclass comm-semiring-0-cancel ..

lemma square-diff-square-factored: 
x * x - y * y = (x + y) * (x - y)
by (simp add: algebra-simps)

end

class ring-1 = ring + zero-neq-one + monoid-mul
begin

subclass semiring-1-cancel ..

lemma of-bool-not-iff:
(of-bool (\sim P) = 1 - of-bool P)
by simp

lemma square-diff-one-factored: 
x * x - 1 = (x + 1) * (x - 1)
by (simp add: algebra-simps)

end

class comm-ring-1 = comm-ring + zero-neq-one + comm-monoid-mul
begin

subclass ring-1 ..
subclass comm-semiring-1-cancel
  by standard (simp add; algebra-simps)

lemma dvd-minus-iff [simp]: \( x \divides y \iff x \divides y \)
proof
  assume \( x \divides y \)
  then have \( x \divides 1 \ast - y \) by (rule dvd-mult)
  then show \( x \divides y \) by simp
next
  assume \( x \divides y \)
  then have \( x \divides 1 \ast y \) by (rule dvd-mult)
  then show \( x \divides - y \) by simp
qed

lemma minus-dvd-iff [simp]: \(- x \divides y \iff x \divides y \)
proof
  assume \(- x \divides y \)
  then obtain \( k \) where \( y = - x \ast k \) ..
  then have \( y = x \ast - k \) by simp
  then show \( x \divides y \) ..
next
  assume \( x \divides y \)
  then obtain \( k \) where \( y = x \ast k \) ..
  then have \( y = - x \ast - k \) by simp
  then show \(- x \divides y \) ..
qed

lemma dvd-diff [simp]: \( x \divides y \Longrightarrow x \divides z \Longrightarrow x \divides (y - z) \)
  using dvd-add [of \( x \ y \) - \( z \)] by simp

end

16.3 Towards integral domains

class semiring-no-zero-divisors = semiring-0 +
  assumes no-zero-divisors: \( a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a \ast b \neq 0 \)
begins

lemma divisors-zero:
  assumes \( a \ast b = 0 \)
  shows \( a = 0 \lor b = 0 \)
proof (rule classical)
  assume \( \neg \)thesis
  then have \( a \neq 0 \) and \( b \neq 0 \) by auto
  with no-zero-divisors have \( a \ast b \neq 0 \) by blast
  with assms show \( \)thesis by simp
lemma mult-eq-0-iff [simp]: a * b = 0 ←→ a = 0 ∨ b = 0
proof (cases a = 0 ∨ b = 0)
  case False
    then have a ≠ 0 and b ≠ 0 by auto
    then show ?thesis using no-zero-divisors by simp
next
  case True
    then show ?thesis by auto
qed

class semiring-1-no-zero-divisors = semiring-1 + semiring-no-zero-divisors

class semiring-no-zero-divisors-cancel = semiring-no-zero-divisors +
  assumes mult-cancel-right [simp]: a * c = b * c ←→ c = 0 ∨ a = b
  and mult-cancel-left [simp]: c * a = c * b ←→ c = 0 ∨ a = b
begin

  lemma mult-left-cancel: c ≠ 0 ⇒ c * a = c * b ←→ a = b
    by simp

  lemma mult-right-cancel: c ≠ 0 ⇒ a * c = b * c ←→ a = b
    by simp

end

class ring-no-zero-divisors = ring + semiring-no-zero-divisors
begin

subclass semiring-no-zero-divisors-cancel
proof
  fix a b c
  have a * c = b * c ←→ (a - b) * c = 0
    by (simp add: algebra-simps)
  also have . . . ←→ c = 0 ∨ a = b
    by auto
  finally show a * c = b * c ←→ c = 0 ∨ a = b .
  have c * a = c * b ←→ c * (a - b) = 0
    by (simp add: algebra-simps)
  also have . . . ←→ c = 0 ∨ a = b
    by auto
  finally show c * a = c * b ←→ c = 0 ∨ a = b .
qed

end
class ring-1-no-zero-divisors = ring-1 + ring-no-zero-divisors
begin

subclass semiring-1-no-zero-divisors ..

lemma square-eq-1-iff: \( x \times x = 1 \iff x = 1 \lor x = -1 \)
proof
  have \((x - 1) \times (x + 1) = x \times x - 1\)
  by (simp add: algebra-simps)
  then have \( x \times x = 1 \iff (x - 1) \times (x + 1) = 0 \)
  by simp
  then show \(?thesis\)
  by (simp add: eq-neg-iff-add-eq-0)
qed

lemma mult-cancel-right1 [simp]: \( c = b \times c \iff c = 0 \lor b = 1 \)
using mult-cancel-right [of 1 c b] by auto

lemma mult-cancel-right2 [simp]: \( a \times c = c \iff c = 0 \lor a = 1 \)
using mult-cancel-right [of a c 1] by simp

lemma mult-cancel-left1 [simp]: \( c = c \times b \iff c = 0 \lor b = 1 \)
using mult-cancel-left [of c 1 b] by force

lemma mult-cancel-left2 [simp]: \( c \times a = c \iff c = 0 \lor a = 1 \)
using mult-cancel-left [of c a 1] by simp

end

class semidom = comm-semiring-1-cancel + semiring-no-zero-divisors
begin

subclass semiring-1-no-zero-divisors ..

end

class idom = comm-ring-1 + semiring-no-zero-divisors
begin

subclass semidom ..

subclass ring-1-no-zero-divisors ..

lemma dvd-mult-cancel-right [simp]: \( a \times c \text{ dvd } b \times c \iff c = 0 \lor a \text{ dvd } b \)
proof
  have \( a \times c \text{ dvd } b \times c \iff (\exists k. b \times c = (a \times k) \times c) \)
  by (auto simp add: ac-simps)
  also have \((\exists k. b \times c = (a \times k) \times c) \iff c = 0 \lor a \text{ dvd } b \)
by auto
finally show ?thesis.
qed

lemma dvd-mult-cancel-left [simp]:
c * a dvd c * b \iff c = 0 \lor a dvd b
using dvd-mult-cancel-right [of a c b] by (simp add: ac-simps)

lemma square-eq-iff: a * a = b * b \iff a = b \lor a = - b
proof
assume a * a = b * b
then have (a - b) * (a + b) = 0
by (simp add: algebra-simps)
then show a = b \lor a = - b
by (simp add: eq-neg-iff-add-eq-0)
next
assume a = b \lor a = - b
then show a * a = b * b by auto
qed

lemma inj-mult-left [simp]: \langle inj ((*) a) \iff a \neq 0 \rangle (is \langle ?P \iff ?Q \rangle)
proof
assume ?P
show ?Q
proof
assume \langle a = 0 \rangle
with \langle ?P \rangle have inj ((*) 0)
by simp
moreover have 0 * 0 = 0 * 1
by simp
ultimately have 0 = 1
by (rule injD)
then show False
by simp
qed
next
assume ?Q then show ?P
by (auto intro: injI)
qed

end

class idom-abs-sgn = idom + abs + sgn +
assumes sgn-mult-abs: sgn a * |a| = a
and sgn-sgn [simp]: sgn (sgn a) = sgn a
and abs-abs [simp]: ||a|| = |a|
and abs-0 [simp]: |0| = 0
and sgn-0 [simp]: sgn 0 = 0
and sgn-1 [simp]: sgn 1 = 1
and sgn-minus-1: \( \text{sgn} (-1) = -1 \)
and sgn-mult: \( \text{sgn} (a \ast b) = \text{sgn} a \ast \text{sgn} b \)

begin

lemma sgn-eq-0-iff:
\( \text{sgn} a = 0 \iff a = 0 \)
proof –
\{ assume \( \text{sgn} a = 0 \)
then have \( \text{sgn} a \ast \vert a \vert = 0 \)
by simp
then have \( a = 0 \)
by (simp add: sgn-mult-abs)
\} then show \(?\)thesis
by auto
qed

lemma abs-eq-0-iff:
\( \vert a \vert = 0 \iff a = 0 \)
proof –
\{ assume \( \vert a \vert = 0 \)
then have \( \text{sgn} a \ast \vert a \vert = 0 \)
by simp
then have \( a = 0 \)
by (simp add: sgn-mult-abs)
\} then show \(?\)thesis
by auto
qed

lemma abs-mult-sgn:
\( \vert a \vert \ast \text{sgn} a = a \)
using sgn-mult-abs [of a] by (simp add: ac-simps)

lemma abs-1 [simp]:
\( \vert 1 \vert = 1 \)
using sgn-mult-abs [of 1] by simp

lemma sgn-abs [simp]:
\( \vert \text{sgn} a \vert = \text{of-bool} (a \neq 0) \)
using sgn-mult-abs [of \( \text{sgn} a \vert \) mult-cancel-left [of \( \text{sgn} a \) \( \text{sgn} a \vert 1 \)]
by (auto simp add: sgn-eq-0-iff)

lemma abs-sgn [simp]:
\( \vert \text{sgn} a \vert = \text{of-bool} (a \neq 0) \)
using sgn-mult-abs [of \( \vert a \vert \) mult-cancel-right [of \( \text{sgn} \vert a \vert \) \( \vert a \vert \) \( 1 \)]
by (auto simp add: abs-eq-0-iff)

lemma abs-mult:
\( \vert a \ast b \vert = \vert a \vert \ast \vert b \vert \)
proof (cases \( a = 0 \lor b = 0 \))
THEORY “Rings”

case True
then show ?thesis
  by auto
next
case False
then have *: sgn (a * b) ≠ 0
  by (simp add: sgn-eq-0-iff)
from abs-mult-sgn [of a * b] abs-mult-sgn [of a] abs-mult-sgn [of b]
have |a * b| * sgn (a * b) = |a| * sgn a * |b| * sgn b
  by (simp add: ac-simps)
then have |a * b| * sgn (a * b) = |a| * |b| * sgn (a * b)
  by (simp add: sgn-mult ac-simps)
with * show ?thesis
  by simp
qed

lemma sgn-minus [simp]:
  sgn (− a) = − sgn a
proof −
  from sgn-minus-1 have sgn (− 1 * a) = − 1 * sgn a
    by (simp only: sgn-mult)
then show ?thesis
  by simp
qed

lemma abs-minus [simp]:
  |− a| = |a|
proof −
  have [simp]: |− 1| = 1
    using sgn-mult-abs [of − 1] by simp
then have |− 1 * a| = 1 * |a|
    by (simp only: abs-mult)
then show ?thesis
  by simp
qed

end

16.4 (Partial) Division

class divide =
  fixes divide :: 'a ⇒ 'a ⇒ 'a (infixl div 70)
setup (Sign.add-const-constraint (const-name divide), SOME typ ('a ⇒ 'a ⇒ 'a))

context semiring
begin
lemma [field-simps, field-split-simps]:
shows distrib-left-NO-MATCH: NO-MATCH (x div y) a → a * (b + c) = a * b + a * c
and distrib-right-NO-MATCH: NO-MATCH (x div y) c → (a + b) * c = a * c + b * c
by (rule distrib-left distrib-right)+
end

context ring
begin
lemma [field-simps, field-split-simps]:
shows left-diff-distrib-NO-MATCH: NO-MATCH (x div y) c → (a - b) * c = a * c - b * c
and right-diff-distrib-NO-MATCH: NO-MATCH (x div y) a → a * (b - c) = a * b - a * c
by (rule left-diff-distrib right-diff-distrib)+
end

setup (Sign.add-const-constraint (const-name divide, SOME typ ('a::divide ⇒ 'a ⇒ 'a)));

class divide-trivial = zero + one + divide +
assumes div-by-0 [simp]: ⟨a div 0 = 0⟩
and div-by-1 [simp]: ⟨a div 1 = a⟩
and div-0 [simp]: ⟨0 div a = 0⟩

Algebraic classes with division

class semidom-divide = semidom + divide +
assumes nonzero-mult-div-cancel-right [simp]: ⟨b ≠ 0 ⇒ (a * b) div b = a⟩
assumes semidom-div-by-0: ⟨a div 0 = 0⟩
begin

lemma nonzero-mult-div-cancel-left [simp]: ⟨a ≠ 0 ⇒ (a * b) div a = b⟩
using nonzero-mult-div-cancel-right [of a b] by (simp add: ac-simps)

subclass divide-trivial
proof
show [simp]: ⟨a div 0 = 0⟩ for a
by (fact semidom-div-by-0)
show ⟨a div 1 = a⟩ for a
using nonzero-mult-div-cancel-right [of 1 a] by simp
show ⟨0 div a = 0⟩ for a
using nonzero-mult-div-cancel-right [of a 0] by (cases ⟨a = 0⟩) simp-all
qed

subclass semiring-no-zero-divisors-cancel
proof
  show \(*\): \(a \ast c = b \ast c \iff c = 0 \lor a = b\) for \(a\ b\ c\)
  proof (cases \(c = 0\))
    case True
    then show ?thesis by simp
  next
    case False
    have \(a = b\) if \(a \ast c = b \ast c\)
    proof
      from that have \(a \ast c \div c = b \ast c \div c\)
      by simp
      with False show ?thesis
      by simp
    qed
    then show ?thesis by auto
  qed

show \(*\): \(c \ast a = c \ast b \iff c = 0 \lor a = b\) for \(a\ b\ c\)
  using \(*\) [of \(a\ c\ b\)] by (simp add: ac-simps)
qed

lemma \texttt{div-self} [simp]: \(a \neq 0 \implies a \div a = 1\)
  using nonzero-mult-div-cancel-left [of \(a\ 1\)] by simp

lemma \texttt{dvd-div-eq-0-iff}:
  assumes \(b\ dvd\ a\)
  shows \(a \div b = 0 \iff a = 0\)
  using assms by (elim dvdE, cases \(b = 0\)) simp-all

lemma \texttt{dvd-div-eq-cancel}:
  \(a \div c = b \div c \implies c \dvd a \implies c \dvd b \implies a = b\)
  by (elim dvdE, cases \(c = 0\)) simp-all

lemma \texttt{dvd-div-eq-iff}:
  \(c \dvd a \implies c \dvd b \implies a \div c = b \div c \iff a = b\)
  by (elim dvdE, cases \(c = 0\)) simp-all

lemma \texttt{inj-on-mult}:
  \texttt{inj-on} \((\ast)\ a\) \(A\) if \(a \neq 0\)
  proof (rule inj-on1)
    fix \(b\ c\)
    assume \(a \ast b = a \ast c\)
    then have \(a \ast b \div a = a \ast c \div a\)
    by (simp only:)
    with that show \(b = c\)
    by simp
  qed

end
class idom-divide = idom + semidom-divide
begin

lemma dvd-neg-div:
  assumes b dvd a
  shows - a div b = -(a div b)
proof (cases b = 0)
  case True
  then show ?thesis by simp
next
case False
  from assms obtain c where a = b * c ..
  then have - a div b = (b * - c) div b
    by simp
  from False also have .. = - c
    by (rule nonzero-mult-div-cancel-left)
  with False a = b * c show ?thesis
    by simp
qed

lemma dvd-div-neg:
  assumes b dvd a
  shows a div - b = -(a div b)
proof (cases b = 0)
  case True
  then show ?thesis by simp
next
case False
  then have - b ≠ 0
    by simp
  from assms obtain c where a = b * c ..
  then have a div - b = (- b * - c) div - b
    by simp
  from (- b ≠ 0) also have .. = - c
    by (rule nonzero-mult-div-cancel-left)
  with False a = b * c show ?thesis
    by simp
qed

end

class algebraic-semidom = semidom-divide
begin

Class algebraic-semidom enriches a integral domain by notions from algebra,
like units in a ring. It is a separate class to avoid spoiling fields with notions
which are degenerated there.

lemma dvd-times-left-cancel-iff [simp]:
  assumes a ≠ 0


shows $a \cdot b \text{ dvd } a \cdot c \longleftrightarrow b \text{ dvd } c$

(is $\text{lhs} \longleftrightarrow \text{rhs}$)
proof
assume $\text{lhs}$
then obtain $d$ where $a \cdot c = a \cdot b \cdot d$ ..
with assms have $c = b \cdot d$ by (simp add: ac-simps)
then show $\text{rhs}$ ..
next
assume $\text{rhs}$
then obtain $d$ where $c = b \cdot d$ ..
then have $a \cdot c = a \cdot b \cdot d$ by (simp add: ac-simps)
then show $\text{lhs}$ ..
qed

lemma dvd-times-right-cancel-iff [simp]:
assumes $a \neq 0$
shows $b \cdot a \text{ dvd } c \longleftrightarrow b \text{ dvd } c$
using dvd-times-left-cancel-iff [of $a$ $b$ $c$] assms by (simp add: ac-simps)

lemma div-dvd-iff-mult:
assumes $b \neq 0$ and $b \text{ dvd } a$
shows $a \text{ div } b \text{ dvd } c \longleftrightarrow a \text{ dvd } c \cdot b$
proof
from $\langle b \text{ dvd } a \rangle$ obtain $d$ where $a = b \cdot d$ ..
with $\langle b \neq 0 \rangle$ show $\text{thesis}$ by (simp add: ac-simps)
qed

lemma dvd-div-iff-mult:
assumes $c \neq 0$ and $c \text{ dvd } b$
shows $a \text{ dvd } b \text{ div } c \longleftrightarrow a \cdot c \text{ dvd } b$
proof
from $\langle c \text{ dvd } b \rangle$ obtain $d$ where $b = c \cdot d$ ..
with $\langle c \neq 0 \rangle$ show $\text{thesis}$ by (simp add: mult.commute [of $a$])
qed

lemma div-dvd-div [simp]:
assumes $a \text{ dvd } b$ and $a \text{ dvd } c$
shows $b \text{ div } a \text{ dvd } c \longleftrightarrow b \text{ dvd } c$
proof (cases $a = 0$)
case True
with assms show $\text{thesis}$ by simp
next
case False
moreover from assms obtain $k$ $l$ where $b = a \cdot k$ and $c = a \cdot l$
by blast
ultimately show $\text{thesis}$ by simp
qed

lemma div-add [simp]:
assumes $c \mid a$ and $c \mid b$
shows $(a + b) \div c = a \div c + b \div c$
proof (cases $c = 0$)
  case True
  then show ?thesis by simp
next
  case False
moreover from assms obtain $k \ l$ where $a = c \ast k$ and $b = c \ast l$
  by blast
moreover have $c \ast k + c \ast l = c \ast (k + l)$
  by (simp add: algebra-simps)
ultimately show ?thesis
  by simp
qed

lemma div-mult-div-if-dvd:
  assumes $b \mid a$ and $d \mid c$
shows $(a \div b) \ast (c \div d) = (a \ast c) \div (b \ast d)$
proof (cases $b = 0 \lor c = 0$)
  case True
  with assms show ?thesis by auto
next
  case False
moreover from assms obtain $k \ l$ where $a = b \ast k$ and $c = d \ast l$
  by blast
moreover have $b \ast k \ast (d \ast l) \div (b \ast d) = (b \ast d) \ast (k \ast l) \div (b \ast d)$
  by (simp add: ac-simps)
ultimately show ?thesis by simp
qed

lemma dvd-div-eq-mult:
  assumes $a \neq 0$ and $a \mid b$
shows $b \div a = c \iff b = c \ast a$
(is ?lhs $\iff$ ?rhs)
proof
  assume ?rhs
  then show ?lhs by (simp add: assms)
next
  assume ?lhs
  then have $b \div a = c \iff b = c \ast a$ by simp
moreover from assms have $b \div a \ast a = b$
  by (auto simp add: ac-simps)
ultimately show ?rhs by simp
qed

lemma dvd-div-mult-self [simp]: $a \mid b \Longrightarrow b \div a \ast a = b$
  by (cases $a = 0$) (auto simp add: ac-simps)
lemma dvd-mult-div-cancel [simp]: $a \mid b \Longrightarrow a \ast (b \div a) = b$
using dvd-div-mul-self [of a b] by (simp add: ac-simps)

lemma div-mult-swap:
  assumes c dvd b
  shows a * (b div c) = (a * b) div c
proof (cases c = 0)
  case True
  then show ?thesis by simp
next
  case False
  from assms obtain d where b = c * d ..
  moreover from False have a * divide (d * c) c = ((a * d) * c) div c
    by simp
  ultimately show ?thesis by (simp add: ac-simps)
qed

lemma dvd-div-mult: c dvd b \implies b div c * a = (b * a) div c
using div-mult-swap [of c b a] by (simp add: ac-simps)

lemma dvd-div-mult2-eq:
  assumes b * c dvd a
  shows a div (b * c) = a div b div c
proof
  from assms obtain k where a = b * c * k ..
  then show ?thesis
    by (cases b = 0 \lor c = 0) (auto, simp add: ac-simps)
qed

lemma dvd-div-div-eq-mult:
  assumes a \neq 0 c \neq 0 and a dvd b c dvd d
  shows b div a = d div c \iff b * c = a * d
(is ?lhs \iff ?rhs)
proof
  from assms have a * c \neq 0 by simp
  then have ?lhs \iff b div a * (a * c) = d div c * (a * c)
    by simp
  also have \dots \iff (a * (b div a)) * c = (c * (d div c)) * a
    by (simp add: ac-simps)
  also have \dots \iff (a * b div a) * c = (c * d div c) * a
    using assms by (simp add: div-mult-swap)
  also have \dots \iff ?rhs
    using assms by (simp add: ac-simps)
  finally show ?thesis .
qed

lemma dvd-mult-imp-div:
  assumes a * c dvd b
  shows a dvd b div c
proof (cases c = 0)
case True then show ?thesis by simp

next
case False
from \( a \ast c \) dvd b obtain d where \( b = a \ast c \ast d \)
with False show ?thesis
  by (simp add: mult.commute [of a] mult.assoc)
qed

lemma div-div-eq-right:
  assumes c dvd b b dvd a
  shows \( a \div (b \div c) = a \div b \ast c \)
proof (cases \( c = 0 \lor b = 0 \))
case True
  with assms show ?thesis
  by auto
next
case False
from assms obtain r s where \( b = c \ast r \) and \( a = c \ast r \ast s \)
  by blast
moreover with False have \( r \neq 0 \)
  by auto
ultimately show ?thesis
  by simp (simp add: mult.assoc mult.commute [of c])
qed

lemma div-div-div-same:
  assumes d dvd b b dvd a
  shows \( (a \div d) \div (b \div d) = a \div b \)
proof (cases \( b = 0 \lor d = 0 \))
case True
  with assms show ?thesis
  by auto
next
case False
from assms obtain r s
  where \( a = d \ast r \ast s \) and \( b = d \ast r \)
  by blast
with False show ?thesis
  by simp (simp add: ac-simps)
qed

Units: invertible elements in a ring

abbreviation is-unit :: 'a :: ring => bool
  where is-unit a \equiv a dvd 1

lemma not-is-unit-0 [simp]; \( \neg \) is-unit 0
  by simp

lemma unit-imp-dvd [dest]: is-unit b \implies b dvd a
lemma unit-dvdE:
assumes is-unit a
obtains c where a ≠ 0 and b = a * c
proof –
  from assms have a dvd b by auto
  then obtain c where b = a * c ..
  moreover from assms have a ≠ 0 by auto
  ultimately show thesis using that by blast
qed

lemma dvd-unit-imp-unit: a dvd b ⇒ is-unit b ⇒ is-unit a by (rule dvd-trans)

lemma unit-div-1-unit [simp, intro]:
assumes is-unit a
shows is-unit (1 div a)
proof –
  from assms have 1 = 1 div a * a by simp
  then show is-unit (1 div a) by (rule dvdI)
qed

lemma is-unitE [elim?!]:
assumes is-unit a
obtains b where a ≠ 0 and b ≠ 0
  and is-unit b and 1 div a = b and 1 div b = a
  and a * b = 1 and c div a = c * b
proof (rule that)
  define b where b = 1 div a
  then show 1 div a = b by simp
  from assms b-def show is-unit b by simp
  with assms show a ≠ 0 and b ≠ 0 by auto
  from assms b-def show a * b = 1 by simp
  then have 1 = a * b ..
  with b-def ⟨b ≠ 0⟩ show 1 div b = a by simp
  from assms have a dvd c ..
  then obtain d where c = a * d ..
  with ⟨a ≠ 0⟩ ⟨a * b = 1⟩ show c div a = c * b
    by (simp add: mult.assoc mult.left-commute [of a])
qed

lemma unit-prod [intro]: is-unit a ⇒ is-unit b ⇒ is-unit (a * b)
  by (subst mult-1-left [of 1, symmetric]) (rule mult-dvd-mono)

lemma is-unit-mult-iff: is-unit (a * b) ←→ is-unit a ∧ is-unit b
  by (auto dest: dvd-mult-left dvd-mult-right)

lemma unit-div [intro]: is-unit a ⇒ is-unit b ⇒ is-unit (a div b)
by (erule is-unitE[of b a]) (simp add: ac-simps unit-prod)

lemma mult-unit-dvd-iff:
assumes is-unit b
shows \( a \cdot b \mid c \iff a \mid c \)
proof
  assume \( a \cdot b \mid c \)
  with assms have \( c = (a \cdot b) \cdot (1 \div b \cdot k) \)
  by (simp add: mult-ac)
  then show \( a \cdot b \mid c \) by (rule dvdI)
next
  assume \( a \mid c \)
  then obtain \( k \) where \( c = a \cdot k \)
  with assms have \( c = (a \cdot b) \cdot (1 \div b \cdot k) \)
  by (simp add: mult-ac)
  then show \( a \cdot b \mid c \) by (rule dvdI)
qed

lemma mult-unit-dvd-iff': is-unit a \( \Rightarrow (a \cdot b) \mid c \iff b \mid c \)
using mult-unit-dvd-iff[of a b c] by (simp add: ac-simps)

lemma dvd-mult-unit-iff:
assumes is-unit b
shows \( a \mid c \cdot b \iff a \mid c \)
proof
  assume \( a \mid c \cdot b \)
  with assms have \( c \cdot b \mid c \cdot (b \cdot (1 \div b)) \)
  by (subst mult-assoc[symmetric]) simp
  also from assms have \( b \cdot (1 \div b) = 1 \)
  by (rule is-unitE) simp
  finally have \( c \cdot b \mid c \) by simp
  with \( a \mid c \cdot b \) show \( a \mid c \) by (rule dvd-trans)
next
  assume \( a \mid c \)
  then show \( a \mid c \cdot b \) by simp
qed

lemma dvd-mult-unit-iff': is-unit b \( \Rightarrow a \mid b \cdot c \iff a \mid c \)
using dvd-mult-unit-iff[of b a c] by (simp add: ac-simps)

lemma div-unit-dvd-iff: is-unit b \( \Rightarrow a \div b \mid c \iff a \mid c \)
by (erule is-unitE[of - a]) (auto simp add: mult-unit-dvd-iff)

lemma dvd-div-unit-iff: is-unit b \( \Rightarrow a \mid c \div b \iff a \mid c \)
by (erule is-unitE[of - c]) (simp add: dvd-mult-unit-iff)

lemmas unit-dvd-iff = mult-unit-dvd-iff mult-unit-dvd-iff'
dvd-mult-unit-iff dvd-mult-unit-iff'
div-unit-dvd-iff dvd-div-unit-iff
lemma unit-mult-div-div [simp]: is-unit a \implies b \cdot (1 \div a) = b \div a
  by (erule is-unitE [of - b]) simp

lemma unit-div-mult-self [simp]: is-unit a \implies b \div a \cdot a = b
  by (rule dvd-div-mult-self) auto

lemma unit-div-1-div-1 [simp]: is-unit a \implies 1 \div (1 \div a) = a
  by (erule is-unitE)

lemma unit-mult-left-cancel: is-unit a \implies a \cdot b = a \cdot c \iff b = c
  using unit-mult-left-cancel [of a b c] by (auto simp add: ac-simps)

lemma unit-mult-right-cancel: is-unit a \implies b \cdot a = c \cdot a \iff b = c
  using unit-mult-right-cancel [of a b c] by (auto simp add: ac-simps)

lemma unit-div-cancel: assumes is-unit a
  shows b \div a = c \div a \iff b = c
  proof
    from assms have is-unit (1 \div a) by simp
    then have b \cdot (1 \div a) = c \cdot (1 \div a) \iff b = c
      by (rule unit-mult-right-cancel)
    with assms show ?thesis by simp
  qed

lemma is-unit-div-mult2-eq: assumes is-unit b and is-unit c
  shows a \div (b \cdot c) = a \div b \div c
  proof
    from assms have is-unit (b \cdot c)
      by (simp add: unit-proof)
    then have b \cdot c dvd a
      by (rule unit-imp-dvd)
    then show ?thesis
      by (rule dvd-div-mult2-eq)
  qed
**THEORY “Rings”**

**lemma** is-unit-div-mult-cancel-left:
- **assumes** $a \neq 0$ and is-unit $b$
- **shows** $a \div (a \ast b) = 1 \div b$

**proof**
- **from** assms **have** $a \div (a \ast b) = a \div a \div b$
- **by** (simp add: mult-unit-dvd-iff dvd-div-mult2-eq)
- **with** assms **show** ?thesis **by** simp

**qed**

**lemma** is-unit-div-mult-cancel-right:
- **assumes** $a \neq 0$ and is-unit $b$
- **shows** $a \div (b \ast a) = 1 \div b$
- using assms is-unit-div-mult-cancel-left [of $a$ $b$]
- **by** (simp add: ac-simps)

**lemma** unit-div-eq-0-iff:
- **assumes** is-unit $b$
- **shows** $a \div b = 0 \iff a = 0$
- using assms by (simp add: dvd-div-eq-0-iff unit-imp-dvd)

**lemma** div-mult-unit2:
- is-unit $c$ if coprime $a$ $b$ and $c \div a$ and $c \div b$ and $\neg$ is-unit $c$
- **by** (rule dvd-div-mul2-eq) (simp-all add: mult-unit-dvd-iff)

**Coprimality**

**definition** coprime :: ‘a ⇒ ‘a ⇒ bool
- where coprime $a$ $b$ **if** (∀ $c$. $c \div a$ **⇒** $c \div b$ **⇒** is-unit $c$)

**lemma** coprimeI:
- **assumes** $\forall c. c \div a \Rightarrow c \div b \Rightarrow$ is-unit $c$
- **shows** coprime $a$ $b$
- using assms **by** (auto simp: coprime-def)

**lemma** not-coprimeI:
- **assumes** $c \div a$ and $c \div b$ and $\neg$ is-unit $c$
- **shows** $\neg$ coprime $a$ $b$
- using assms **by** (auto simp: coprime-def)

**lemma** coprime-common-divisor:
- is-unit $c$ if coprime $a$ $b$ and $c \div a$ and $c \div b$
- **using** that **by** (auto simp: coprime-def)

**lemma** not-coprimeE:
- **assumes** $\neg$ coprime $a$ $b$
- **obtains** $c$ where $c \div a$ and $c \div b$ and $\neg$ is-unit $c$
- **using** assms **by** (auto simp: coprime-def)

**lemma** coprime-imp-coprime:
- coprime $a$ $b$ if coprime $c$ $d$
- and $\forall e. \neg$ is-unit $e \Rightarrow e \div a \Rightarrow e \div b \Rightarrow e \div c$
theory "Rings"

and \( \forall e. \neg \text{is-unit } e \implies e \ dvd a \implies e \ dvd b \implies e \ dvd d \)

proof (rule coprimeI)
  fix \( e \)
  assume \( e \ dvd a \) and \( e \ dvd b \)
  with that have \( e \ dvd c \) and \( e \ dvd d \)
  by (auto intro: dvd-trans)
  with \( \langle \text{coprime } e \rangle \) show \( \text{is-unit } e \)
  by (rule coprime-common-divisor)

qed

lemma coprime-divisors:
  coprime \( a \) \( b \) if \( a \ dvd c \) \( b \ dvd d \) and \( \text{coprime } c \) \( d \)
  using \( \langle \text{coprime } c \rangle \) proof (rule coprime-imp-coprime)
  fix \( e \)
  assume \( e \ dvd a \) then show \( e \ dvd c \)
  using \( \langle a \ dvd c \rangle \) by (rule dvd-trans)
  assume \( e \ dvd b \) then show \( e \ dvd d \)
  using \( \langle b \ dvd d \rangle \) by (rule dvd-trans)

qed

lemma coprime-self [simp]:
  coprime \( a \) \( a \) \iff \( \text{is-unit } a \) (is \(?P\) \iff \(?Q\))
  proof
    assume \(?P\)
    then show \(?Q\)
      by (rule coprime-common-divisor) simp-all
  next
    assume \(?Q\)
    show \(?P\)
      by (rule coprimeI) (erule dvd-unit-imp-unit, rule \(?Q\))

qed

lemma coprime-commute [ac-simps]:
  coprime \( b \) \( a \) \iff coprime \( a \) \( b \)
  unfolding coprime-def by auto

lemma is-unit-left-imp-coprime:
  coprime \( a \) \( b \) if \( \text{is-unit } a \)
  proof (rule coprimeI)
    fix \( c \)
    assume \( c \ dvd a \)
    with that show \( \text{is-unit } c \)
    by (auto intro: dvd-unit-imp-unit)
  qed

lemma is-unit-right-imp-coprime:
  coprime \( a \) \( b \) if \( \text{is-unit } b \)
  using that is-unit-left-imp-coprime [of \( b \) \( a \)] by (simp add: ac-simps)
lemma coprime-1-left [simp]:
coprime 1 a
by (rule coprimeI)

lemma coprime-1-right [simp]:
coprime a 1
by (rule coprimeI)

lemma coprime-0-left-iff [simp]:
coprime 0 a ←→ is-unit a
by (auto intro: coprimeI dvd-unit-imp-unit coprime-common-divisor [of 0 a a])

lemma coprime-0-right-iff [simp]:
coprime a 0 ←→ is-unit a
using coprime-0-left-iff [of a] by (simp add: ac-simps)

lemma coprime-mult-self-left-iff [simp]:
coprime (c * a) (c * b) ←→ is-unit c ∧ coprime a b
by (auto intro: coprime-common-divisor)
(rule coprimeI, auto intro: coprime-common-divisor simp add: dvd-mult-unit-iff')+

lemma coprime-mult-self-right-iff [simp]:
coprime (a * c) (b * c) ←→ is-unit c ∧ coprime a b
using coprime-mult-self-left-iff [of a b] by (simp add: ac-simps)

lemma coprime-absorb-left:
assumes x dvd y
shows coprime x y ←→ is-unit x
using assms coprime-common-divisor is-unit-imp-coprime by auto

lemma coprime-absorb-right:
assumes y dvd x
shows coprime x y ←→ is-unit y
using assms coprime-common-divisor is-unit-right-imp-coprime by auto

end

class unit-factor =
fixes unit-factor :: 'a ⇒ 'a

class semidom-divide-unit-factor = semidom-divide + unit-factor +
assumes unit-factor-0 [simp]: unit-factor 0 = 0
and is-unit-unit-factor: a dvd 1 ⇒ unit-factor a = a
and unit-factor-is-unit: a ≠ 0 ⇒ unit-factor a dvd 1
and unit-factor-mult-unit-left: a dvd 1 ⇒ unit-factor (a * b) = a * unit-factor b
— This fine-grained hierarchy will later on allow lean normalization of polynomials
begin
lemma unit-factor-mult-unit-right: a dvd 1 ⇒ unit-factor (b * a) = unit-factor b * a  
  using unit-factor-mult-unit-left[of a b] by (simp add: mult-ac)

lemmas [simp] = unit-factor-mult-unit-left unit-factor-mult-unit-right
end

class normalization-semidom = algebraic-semidom + semidom-divide-unit-factor  
  fixes normalize :: 'a ⇒ 'a
  assumes unit-factor-mult-normalize [simp]: unit-factor a * normalize a = a  
    and normalize-0 [simp]: normalize 0 = 0
begin

Class normalization-semidom cultivates the idea that each integral domain can be split into equivalence classes whose representants are associated, i.e. divide each other. normalize specifies a canonical representant for each equivalence class. The rationale behind this is that it is easier to reason about equality than equivalences, hence we prefer to think about equality of normalized values rather than associated elements.

declare unit-factor-is-unit [iff]

lemma unit-factor-dvd [simp]: a ≠ 0 ⇒ unit-factor a dvd b  
  by (rule unit-imp-dvd) simp

lemma unit-factor-self [simp]: unit-factor a dvd a  
  by (cases a = 0) simp-all

lemma normalize-mult-unit-factor [simp]: normalize a * unit-factor a = a  
  using unit-factor-mult-normalize [of a] by (simp add: ac-simps)

lemma normalize-eq-0-iff [simp]: normalize a = 0 ↔ a = 0  
  (is ?lhs ↔ ?rhs)
proof
  assume ?lhs
  moreover have unit-factor a * normalize a = a by simp
  ultimately show ?rhs by simp
next
  assume ?rhs
  then show ?lhs by simp
qed

lemma unit-factor-eq-0-iff [simp]: unit-factor a = 0 ↔ a = 0  
  (is ?lhs ↔ ?rhs)
proof
  assume ?lhs
  moreover have unit-factor a * normalize a = a by simp
  ultimately show ?rhs by simp

next
  assume ?rhs
  then show ?lhs by simp
qed

lemma div-unit-factor [simp]: a div unit-factor a = normalize a
proof (cases a = 0)
  case True
  then show ?thesis by simp
next
  case False
  then have unit-factor a ≠ 0
    by simp
  with nonzero-mult-div-cancel-left
  have unit-factor a * normalize a div unit-factor a = normalize a
    by blast
  then show ?thesis by simp
qed

lemma normalize-div [simp]: normalize a div a = 1 div unit-factor a
proof (cases a = 0)
  case True
  then show ?thesis by simp
next
  case False
  have normalize a div a = normalize a div (unit-factor a * normalize a)
    by simp
  also have ... = 1 div unit-factor a
    using False by (subst is-unit-div-mult-cancel-right) simp-all
  finally show ?thesis .
qed

lemma is-unit-normalize:
  assumes is-unit a
  shows normalize a = 1
proof —
  from assms have unit-factor a = a
    by (rule is-unit-unit-factor)
  moreover from assms have a ≠ 0
    by auto
  moreover have normalize a = a div unit-factor a
    by simp
  ultimately show ?thesis
    by simp
qed

lemma unit-factor-1 [simp]: unit-factor 1 = 1
  by (rule is-unit-unit-factor) simp
THEORY “Rings”

lemma normalize-1 [simp]: normalize 1 = 1
by (rule is-unit-normalize) simp

lemma normalize-1-iff: normalize a = 1 ←→ is-unit a
(is ?lhs ←→ ?rhs)
proof
  assume ?rhs
  then show ?lhs by (rule is-unit-normalize)
next
  assume ?lhs
  then have unit-factor a * normalize a = unit-factor a * 1
      by simp
  then have unit-factor a = a
      by simp
  moreover from ?lhs have a ≠ 0 by auto
  then have is-unit (unit-factor a) by simp
  ultimately show ?rhs by simp
qed

lemma div-normalize [simp]: a div normalize a = unit-factor a
proof (cases a = 0)
  case True
  then show ?thesis by simp
next
  case False
  then have normalize a ≠ 0 by simp
  with nonzero-mult-div-cancel-right
  have unit-factor a * normalize a div normalize a = unit-factor a by blast
  then show ?thesis by simp
qed

lemma mult-one-div-unit-factor [simp]: a * (1 div unit-factor b) = a div unit-factor b
by (cases b = 0) simp-all

lemma inv-unit-factor-eq-0-iff [simp]:
  1 div unit-factor a = 0 ←→ a = 0
(is ?lhs ←→ ?rhs)
proof
  assume ?lhs
  then have a * (1 div unit-factor a) = a * 0
      by simp
  then show ?rhs
      by simp
next
  assume ?rhs
  then show ?lhs by simp
qed
lemma unit-factor-idem [simp]: unit-factor (unit-factor a) = unit-factor a
  by (cases a = 0) (auto intro: is-unit-unit-factor)

lemma normalize-unit-factor [simp]: a ≠ 0 ⟹ normalize (unit-factor a) = 1
  by (rule is-unit-normalize) simp

lemma normalize-mult-unit-left [simp]:
  assumes a dvd 1
  shows normalize (a * b) = normalize b
proof (cases b = 0)
  case False
  have a * unit-factor b * normalize (a * b) = unit-factor (a * b) * normalize (a * b)
    using assms by (subst unit-factor-mult-unit-left) auto
  also have . . . = a * b by simp
  also have b = unit-factor b * normalize b by simp
  hence a * b = a * unit-factor b * normalize b
    by (simp only: mult-ac)
  finally show ?thesis
    using assms False by auto
qed auto

lemma normalize-mult-unit-right [simp]:
  assumes b dvd 1
  shows normalize (a * b) = normalize a
using assms by (subst mult.commute) auto

lemma normalize-idem [simp]: normalize (normalize a) = normalize a
proof (cases a = 0)
  case False
  have normalize a = normalize (unit-factor a * normalize a)
    by simp
  also from False have . . . = normalize (normalize a)
    by (subst normalize-mult-unit-left) auto
  finally show ?thesis ..
qed auto

lemma unit-factor-normalize [simp]:
  assumes a ≠ 0
  shows unit-factor (normalize a) = 1
proof -
  from assms have *: normalize a ≠ 0
    by simp
  have unit-factor (normalize a) * normalize (normalize a) = normalize a
    by (simp only: unit-factor-mult-normalize)
  then have unit-factor (normalize a) * normalize a = normalize a
    by simp
  with * have unit-factor (normalize a) * normalize a div normalize a = normalize
a \text{ div} \text{ normalize} \ a \\
\text{by} \ \text{simp} \\
\text{with} * \text{ show} \ \text{thesis} \\
\text{by} \ \text{simp} \\
\text{qed}

\text{lemma} \text{ normalize-dvd-iff} \ [\text{simp}]: \text{ normalize} \ a \ \text{dvd} \ b \iff \text{ a dvd} \ b \\
\text{proof} - \\
\text{have} \text{ normalize} \ a \ \text{dvd} \ b \iff \text{ unit-factor} \ a \ * \ \text{normalize} \ a \ \text{dvd} \ b \\
\text{using} \text{ mult-unit-dvd-iff} \ [\text{of} \ \text{unit-factor} \ a \ \text{normalize} \ a \ b] \\
\text{by} \ (\text{cases} \ a = 0) \ \text{simp-all} \\
\text{then show} \ \text{thesis} \ \text{by} \ \text{simp} \\
\text{qed}

\text{lemma} \text{ dvd-normalize-iff} \ [\text{simp}]: \text{ a dvd normalize} \ b \iff \text{ a dvd} \ b \\
\text{proof} - \\
\text{have} \text{ a dvd normalize} \ b \iff \text{ a dvd normalize} \ b * \ \text{unit-factor} \ b \\
\text{using} \text{ dvd-mult-unit-iff} \ [\text{of} \ \text{unit-factor} \ b \ a \ \text{normalize} \ b] \\
\text{by} \ (\text{cases} \ b = 0) \ \text{simp-all} \\
\text{then show} \ \text{thesis} \ \text{by} \ \text{simp} \\
\text{qed}

\text{lemma} \text{ normalize-idem-imp-unit-factor-eq:} \\
\text{assumes} \text{ normalize} \ a = a \\
\text{shows} \text{ unit-factor} \ a = \text{ of-bool} \ (a \neq 0) \\
\text{proof} (\text{cases} \ a = 0) \\
\text{case} \ \text{True} \\
\text{then show} \ \text{thesis} \\
\text{by} \ \text{simp} \\
\text{next} \\
\text{case} \ \text{False} \\
\text{then show} \ \text{thesis} \\
\text{using} \text{ assms} \ \text{unit-factor-normalize} \ [\text{of} \ a] \ \text{by} \ \text{simp} \\
\text{qed}

\text{lemma} \text{ normalize-idem-imp-is-unit-iff:} \\
\text{assumes} \text{ normalize} \ a = a \\
\text{shows} \text{ is-unit} \ a \iff a = 1 \\
\text{using} \text{ assms} \text{ by} \ (\text{cases} \ a = 0) \ (\text{auto dest: is-unit-normalize})

\text{lemma} \text{ coprime-normalize-left-iff} \ [\text{simp}]; \\
\text{coprime} \ (\text{normalize} \ a) \ b \iff \text{ coprime} \ a \ b \\
\text{by} \ (\text{rule} \ \text{iffI}; \ \text{rule} \ \text{coprimeI}) \ (\text{auto intro: coprime-common-divisor})

\text{lemma} \text{ coprime-normalize-right-iff} \ [\text{simp}]; \\
\text{coprime} \ a \ (\text{normalize} \ b) \iff \text{ coprime} \ a \ b \\
\text{using} \text{ coprime-normalize-left-iff} \ [\text{of} \ b \ a] \ \text{by} \ (\text{simp add: ac-simps})

\text{We avoid an explicit definition of associated elements but prefer explicit nor-}
malisation instead. In theory we could define an abbreviation like associated $a \ b = (\text{normalize } a = \text{normalize } b)$ but this is counterproductive without suggestive infix syntax, which we do not want to sacrifice for this purpose here.

**lemma** associatedI:
- **assumes** $a \ dvd \ b$ and $b \ dvd \ a$
- **shows** normalize $a = \text{normalize } b$

**proof** (cases $a = 0 \lor b = 0$)
- case True
  - with assms show ?thesis by auto
- next
  - case False
  - from $\langle a \ dvd \ b \rangle$ obtain $c$ where $b = a \ast c$ ..
  - moreover from $\langle b \ dvd \ a \rangle$ obtain $d$ where $a = b \ast d$ ..
  - ultimately have $b \ast 1 = b \ast (c \ast d)$
    - by (simp add: ac-simps)
  - with False have $1 = c \ast d$
    - unfolding mult-cancel-left by simp
  - then have is-unit $c$ and is-unit $d$
    - by auto
  - with $a \ b$ show ?thesis
    - by (simp add: is-unit-normalize)

**qed**

**lemma** associatedD1: normalize $a = \text{normalize } b \Longrightarrow a \ dvd \ b$
- **using** dvd-normalize-iff [of $b$, symmetric] normalize-dvd-iff [of $a -$, symmetric]
- by simp

**lemma** associatedD2: normalize $a = \text{normalize } b \Longrightarrow b \ dvd \ a$
- **using** dvd-normalize-iff [of $a -$, symmetric] normalize-dvd-iff [of $b -$, symmetric]
- by simp

**lemma** associated-unit: normalize $a = \text{normalize } b \Longrightarrow \text{is-unit } a \Longrightarrow \text{is-unit } b$
- **using** dvd-unit-imp-unit by (auto dest!: associatedD1 associatedD2)

**lemma** associated-iff-dvd: normalize $a = \text{normalize } b \longleftrightarrow a \ dvd \ b \land b \ dvd \ a$
- (is ?lhs $\longleftrightarrow$ ?rhs)

**proof**
- assume ?rhs
  - then show ?lhs by (auto intro!: associatedI)
- next
  - assume ?lhs
  - then have unit-factor $a \ast \text{normalize } a = \text{unit-factor } a \ast \text{normalize } b$
    - by simp
  - then have $\ast:\text{normalize } b \ast \text{unit-factor } a = a$
    - by (simp add: ac-simps)
  - show ?rhs
    - proof (cases $a = 0 \lor b = 0$)
      - case True
with \(?lhs\) show \(?thesis\) by auto

next

  case False

  then have \( b \) \( \text{dvd} \) \( \text{normalize} \ \( b \) \text{ and } \text{unit-factor} \ \( a \) \) \( \text{and} \) \( \text{normalize} \ \( b \) \text{ and } \text{unit-factor} \ \( a \) \) \( \text{dvd} \) \( b \) 
  by (simp-all add: mult-unit-dvd-iff dvd-mult-unit-iff)

  with \(*\) show \(?thesis\) by simp

qed

lemma associated-eqI:

assumes \( a \) \( \text{dvd} \) \( b \) \text{ and } \( b \) \( \text{dvd} \) \( a \)

assumes \( \text{normalize} \ \( a \) \text{ and } \text{normalize} \ \( b \) \)

shows \( a \) \( = \) \( b \)

proof −

  from assms have \( \text{normalize} \ \( a \) \text{ = } \text{normalize} \ \( b \) \)

  unfolding associated-iff-dvd by simp

  with \( \langle \text{normalize} \ \( a \) \text{ = } \text{a} \text{; have} \ \( a \) \text{ = } \text{normalize} \ \( b \) \text{ by simp} \text{; with } \langle \text{normalize} \ \( b \) \text{ = } \text{b} \text{ show} \ \( a \) \text{ = } \text{b} \text{ by simp} \)

qed

lemma normalize-unit-factor-eqI:

assumes \( \text{normalize} \ \( a \) \text{ = } \text{normalize} \ \( b \) \) \text{ and } \( \text{unit-factor} \ \( a \) \text{ = } \text{unit-factor} \ \( b \) \)

shows \( a \) \( = \) \( b \)

proof −

  from assms have \( \text{unit-factor} \ \( a \) \text{ * } \text{normalize} \ \( a \) \text{ = } \text{unit-factor} \ \( b \) \text{ * } \text{normalize} \ \( b \) \)

  by simp

  then show \(?thesis\)

  by simp

qed

lemma normalize-mult-normalize-left [simp]: \( \text{normalize} \ (\text{normalize} \ \( a \) \text{ * } \text{b}) \) = \( \text{normalize} \ (\text{a} \text{ * } \text{b}) \)

by (rule associated-eqI) (auto intro: mult-dvd-mono)

lemma normalize-mult-normalize-right [simp]: \( \text{normalize} \ (\text{a} \text{ * } \text{normalize} \ \( b \)) \) = \( \text{normalize} \ (\text{a} \text{ * } \text{b}) \)

by (rule associated-eqI) (auto intro: mult-dvd-mono)

end

class normalization-semidom-multiplicative = normalization-semidom +

assumes unit-factor-mult: \( \text{unit-factor} \ (\text{a} \text{ * } \text{b}) \) = \( \text{unit-factor} \ \( a \text{ * } \text{unit-factor} \ \( b \) \)

begin
lemma normalize-mult: normalize (a * b) = normalize a * normalize b
proof (cases a = 0 ∨ b = 0)
  case True
  then show ?thesis by auto
next
  case False
  have unit-factor (a * b) * normalize (a * b) = a * b
    by (rule unit-factor-mult-normalize)
  then have normalize (a * b) = a * b div unit-factor (a * b)
    by simp
  also have . . . = a * b div unit-factor (b * a)
    by (simp add: ac-simps)
  also have . . . = a * b div unit-factor b div unit-factor a
    using False by (simp add: unit-factor-mult is-unit-div-mult2-eq [symmetric])
  also have . . . = a * (b div unit-factor b) div unit-factor a
    using False by (subst unit-div-mult-swap) simp-all
  also have . . . = normalize a * normalize b
    using False
    by (simp add: mult.commute [of a] mult.commute [of normalize a] unit-div-mult-swap [symmetric])
  finally show ?thesis.
qed

lemma dvd-unit-factor-div:
  assumes b dvd a
  shows unit-factor (a div b) = unit-factor a div unit-factor b
proof –
  from assms have a = a div b * b
    by simp
  then have unit-factor a = unit-factor (a div b * b)
    by simp
  then show ?thesis
    by (cases b = 0) (simp-all add: unit-factor-mult)
qed

lemma dvd-normalize-div:
  assumes b dvd a
  shows normalize (a div b) = normalize a div normalize b
proof –
  from assms have a = a div b * b
    by simp
  then have normalize a = normalize (a div b * b)
    by simp
  then show ?thesis
    by (cases b = 0) (simp-all add: normalize-mult)
qed

end

Syntactic division remainder operator
class modulo = dvd + divide +
  fixes modulo :: 'a ⇒ 'a ⇒ 'a (infixl mod 70)

Arbitrary quotient and remainder partitions

class semiring-modulo = comm-semiring-1-cancel + divide + modulo +
  assumes div-mult-mod-eq: ‹a div b * b + a mod b = a›

begin

lemma mod-div-decomp:
  fixes a b
  obtains q r where q = a div b and r = a mod b
  and a = q * b + r
  proof –
    from div-mult-mod-eq have a = a div b * b + a mod b by simp
    moreover have a div b = a div b ..
    moreover have a mod b = a mod b ..
    note that ultimately show thesis by blast
  qed

lemma mult-div-mod-eq: b * (a div b) + a mod b = a
  using div-mult-mod-eq [of a b] by (simp add: ac-simps)

lemma mod-div-mult-eq: a mod b + b * (a div b) = a
  using div-mult-mod-eq [of a b] by (simp add: ac-simps)

lemma mult-div-mult-eq: (a * b) * (a div b) = a
  using div-mult-mod-eq [of a b] by (simp add: ac-simps)

lemma minus-div-mult-eq-mod: a − a div b * b = a mod b
  by (rule add-implies-diff [symmetric]) (fact mod-div-mult-eq)

lemma minus-mult-div-eq-mod: a − b * (a div b) = a mod b
  by (rule add-implies-diff [symmetric]) (fact mod-mult-div-eq)

lemma minus-mod-eq-div-mult: a − a mod b = a div b * b
  by (rule add-implies-diff [symmetric]) (fact div-mult-mod-eq)

lemma minus-mod-eq-mult-div: a − a mod b = b * (a div b)
  by (rule add-implies-diff [symmetric]) (fact mult-div-mod-eq)

lemma mod-0-imp-dvd [dest!]:
  b dvd a if a mod b = 0
  proof –
    have b dvd (a div b) * b by simp
    also have (a div b) * b = a
      using div-mult-mod-eq [of a b] by (simp add: that)
    finally show ‹thesis›.
  qed
lemma [nitpick-unfold]:
\[ a \mod b = a - a \div b \times b \]
by (fact minus-div-mult-eq-mod [symmetric])

end

class semiring-modulo-trivial = semiring-modulo + divide-trivial
begin

lemma mod-0 [simp]:
\langle 0 \mod a = 0 \rangle
using div-mult-mod-eq [of 0 a] by simp

lemma mod-by-0 [simp]:
\langle a \mod 0 = a \rangle
using div-mult-mod-eq [of a 0] by simp

lemma mod-by-1 [simp]:
\langle a \mod 1 = 0 \rangle
proof -
  have \langle a + a \mod 1 = a \rangle
    using div-mult-mod-eq [of a 1] by simp
  then have \langle a + a \mod 1 = a + 0 \rangle
    by simp
  then show ?thesis
    by (rule add-left-imp-eq)
qed

end

16.5 Quotient and remainder in integral domains

class semidom-modulo = algebraic-semidom + semiring-modulo
begin

subclass semiring-modulo-trivial ..

lemma mod-self [simp]:
a \mod a = 0
using div-mult-mod-eq [of a a] by simp

lemma dvd-imp-mod-0 [simp]:
b \mod a = 0 if a dvd b
using that minus-div-mult-eq-mod [of b a] by simp

lemma mod-eq-0-iff-dvd:
a \mod b = 0 ↔ b dvd a
by (auto intro: mod-0-imp-dvd)
lemma dvd-eq-mod-eq-0 [nitpick-unfold, code]:
\[ a \text{ dvd } b \iff b \mod a = 0 \]
by (simp add: mod-eq-0-iff-dvd)

lemma dvd-mod-iff:
  assumes c dvd b
  shows c dvd a mod b \iff c dvd a
proof -
  from assms have \((c \text{ dvd } a \mod b) \iff (c \text{ dvd } ((a \div b) \ast b + a \mod b))\)
    by (simp add: dvd-add-right-iff)
  also have \((a \div b) \ast b + a \mod b = a\)
    using div-mult-mod-eq [of a b] by simp
  finally show ?thesis .
qed

lemma dvd-mod-imp-dvd:
  assumes c dvd a mod b and c dvd b
  shows \(c \text{ dvd } a\)
  using assms dvd-mod-iff [of c b a] by simp

lemma dvd-minus-mod [simp]:
\[
\begin{align*}
b \text{ dvd } a - a \mod b \\
& \text{by (simp add: minus-mod-eq-div-mult)}
\end{align*}
\]

lemma cancel-div-mod-rules:
\[
\begin{align*}
((a \div b) \ast b + a \mod b) + c &= a + c \\
(b \ast (a \div b) + a \mod b) + c &= a + c
\end{align*}
\]
by (simp-all add: div-mult-mod-eq mult-div-mod-eq)

end

class idom-modulo = idom + semidom-modulo
begin

subclass idom-divide ..

lemma div-diff [simp]:
\[
\begin{align*}
c \text{ dvd } a &\Longrightarrow c \text{ dvd } b \Longrightarrow (a - b) \text{ div } c = a \text{ div } c - b \text{ div } c
\end{align*}
\]
using div-add [of \(-\) \(-\) \(-\)] by (simp add: dvd-neg-div)

end

16.6 Interlude: basic tool support for algebraic and arithmetic calculations

named-theorems arith arith facts --- only ground formulas
ML-file ⟨Tools/arith-data.ML⟩

ML-file ⟨~/src/Provers/Arith/cancel-div-mod.ML⟩
ML

structure Cancel-Div-Mod-Ring = Cancel-Div-Mod
{
  val div-name = const-name (divide);
  val mod-name = const-name (modulo);
  val mk-binop = HOLogic.mk-binop;
  val mk-sum = Arith-Data.mk-sum;
  val dest-sum = Arith-Data.dest-sum;

  val div-mod-eqs = map mk-meta-eq @{thms cancel-div-mod-rules};

  val prove-eq-sums = Arith-Data.prove-conv2 all-tac
    (Arith-Data.simp-all-tac
     @{thms diff-conv-add-uminus add-0-left add-0-right ac-simps})
}

simproc-setup cancel-div-mod-int ((a::a::semidom-modulo) + b) =
  {K Cancel-Div-Mod-Ring.proc}

16.7 Ordered semirings and rings

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society, 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press, 1963

Most of the used notions can also be looked up in

- [http://www.mathworld.com](http://www.mathworld.com) by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer

class ordered-semiring = semiring + ordered-comm-monoid-add +

assumes mult-left-mono: \( a \leq b \Rightarrow 0 \leq c \Rightarrow c \cdot a \leq c \cdot b \)
assumes mult-right-mono: \( a \leq b \Rightarrow 0 \leq c \Rightarrow a \cdot c \leq b \cdot c \)

begin

lemma mult-mono: \( a \leq b \Rightarrow c \leq d \Rightarrow 0 \leq b \Rightarrow 0 \leq c \Rightarrow a \cdot c \leq b \cdot d \)
apply (erule (1) mult-right-mono [THEN order-trans])
apply (erule (1) mult-left-mono)
done

lemma mult-mono': \( a \leq b \Rightarrow c \leq d \Rightarrow 0 \leq a \Rightarrow 0 \leq c \Rightarrow a \cdot c \leq b \cdot d \)
by (rule mult-mono) (fast intro: order-trans)
lemma mono-mult:
  fixes a :: 'a::ordered-semiring
  shows a ≥ 0 ⇒ mono ((*) a)
  by (simp add: mono-def mult-left-mono)

class ordered-semiring-0 = semiring-0 + ordered-semiring
begin

lemma mult-nonneg-nonneg [simp]: 0 ≤ a ⇒ 0 ≤ b ⇒ 0 ≤ a * b
  using mult-left-mono [of 0 b a] by simp

lemma mult-nonneg-nonpos: 0 ≤ a ⇒ b ≤ 0 ⇒ a * b ≤ 0
  using mult-left-mono [of b 0 a] by simp

lemma mult-nonpos-nonneg: a ≤ 0 ⇒ 0 ≤ b ⇒ a * b ≤ 0
  using mult-right-mono [of a 0 b] by simp

Legacy – use mult-nonpos-nonneg.

lemma mult-nonneg-nonpos2: 0 ≤ a ⇒ b ≤ 0 ⇒ b * a ≤ 0
  by (drule mult-right-mono [of b 0]) auto

lemma split-mult-neg-le: (0 ≤ a ∧ b ≤ 0) ∨ (a ≤ 0 ∧ 0 ≤ b) ⇒ a * b ≤ 0
  by (auto simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)

end

class ordered-cancel-semiring = ordered-semiring + cancel-comm-monoid-add
begin

subclass semiring-0-cancel ..

subclass ordered-semiring-0 ..

end

class linordered-semiring = ordered-semiring + linordered-cancel-ab-semigroup-add
begin

subclass ordered-cancel-semiring ..

subclass ordered-cancel-comm-monoid-add ..

subclass ordered-ab-semigroup-monoid-add-imp-le ..

lemma mult-left-less-imp-less: c * a < c * b ⇒ 0 ≤ c ⇒ a < b
  by (force simp add: mult-left-mono not-le [symmetric])
lemma mult-right-less-imp-less: \( a \cdot c < b \cdot c \implies 0 \leq c \implies a < b \)
   by (force simp add: mult-right-mono not-le [symmetric])

end

class zero-less-one = order + zero + one +
   assumes zero-less-one [simp]: \( 0 < 1 \)
begin
  subclass zero-neq-one
  by standard (simp add: less-imp-neq)

lemma zero-le-one [simp]:
  \( \langle 0 \leq 1 \rangle \) by (rule less-imp-le) simp
end

class linordered-semiring-1 = linordered-semiring + semiring-1 + zero-less-one
begin

lemma convex-bound-le:
   assumes \( x \leq a \) \( y \leq a \) \( 0 \leq u \) \( 0 \leq v \) \( u + v = 1 \)
   shows \( u \cdot x + v \cdot y \leq a \)
proof -
  from assms have \( u \cdot x + v \cdot y \leq u \cdot a + v \cdot a \)
    by (simp add: add-mono mult-left-mono)
  with assms show \(?thesis\)
    unfolding distrib-right[symmetric] by simp
qed

end

class linordered-semiring-strict = semiring + comm-monoid-add + linordered-cancel-ab-semigroup-add +
   assumes mult-strict-left-mono: \( a < b \implies 0 < c \implies c \cdot a < c \cdot b \)
   assumes mult-strict-right-mono: \( a < b \implies 0 < c \implies a \cdot c < b \cdot c \)
begin
  subclass semiring-0-cancel ..
  subclass linordered-semiring
proof
  fix \( a \ b \ c :: \'a \)
  assume \( a \leq b \) \( 0 \leq c \)
  then show \( c \cdot a \leq c \cdot b \)
    unfolding le-less
    using mult-strict-left-mono by (cases \( c = 0 \)) auto
  from \( \ast \) show \( a \cdot c \leq b \cdot c \)
    unfolding le-less
qend
using mult-strict-right-mono by (cases c = 0) auto

qed

lemma mult-left-le-imp-le: c * a ≤ c * b ⊢ 0 < c ⊢ a ≤ b
by (auto simp add: mult-strict-left-mono -not-less [symmetric])

lemma mult-right-le-imp-le: a * c ≤ b * c ⊢ 0 < c ⊢ a ≤ b
by (auto simp add: mult-strict-right-mono not-less [symmetric])

lemma mult-pos-pos[simp]: 0 < a ⇒ 0 < b ⇒ 0 < a * b
using mult-strict-left-mono [of 0 b a] by simp

lemma mult-pos-neg: 0 < a ⇒ b < 0 ⇒ a * b < 0
using mult-strict-left-mono [of b 0 a] by simp

lemma mult-neg-pos: a < 0 ⇒ 0 < b ⇒ a * b < 0
using mult-strict-right-mono [of a 0 b] by simp

Legacy – use mult-neg-pos.

lemma mult-pos-neg2: 0 < a ⇒ b < 0 ⇒ b * a < 0
by (drule mult-strict-right-mono [of b 0]) auto

lemma zero-less-mult-pos:
assumes 0 < a * b 0 < a shows 0 < b
proof (cases b ≤ 0)
case True
then show ?thesis
using assms by (auto simp: le-less dest: less-not-sym mult-pos-neg [of a b])
qed (auto simp add: le-less not-less)

lemma zero-less-mult-pos2:
assumes 0 < b * a 0 < a shows 0 < b
proof (cases b ≤ 0)
case True
then show ?thesis
using assms by (auto simp: le-less dest: less-not-sym mult-pos-neg2 [of a b])
qed (auto simp add: le-less not-less)

Strict monotonicity in both arguments

lemma mult-strict-mono:
assumes a < b c < d 0 < b 0 ≤ c
shows a * c < b * d
proof (cases c = 0)
case True
with assms show ?thesis
by simp
next
case False
with assms have \( a \cdot c < b \cdot c \)
  by (simp add: mult-strict-right-mono [OF \( a < b \)])
also have \( \ldots < b \cdot d \)
  by (simp add: assms mult-strict-left-mono)
finally show \(?thesis\).
qed

This weaker variant has more natural premises

lemma mult-strict-mono':
  assumes \( a < b \) and \( c < d \) and \( 0 \leq a \) and \( 0 \leq c \)
  shows \( a \cdot c < b \cdot d \)
  using assms by (auto simp add: mult-strict-mono)

lemma mult-less-le-imp-less:
  assumes \( a < b \) and \( c \leq d \) and \( 0 \leq a \) and \( 0 < c \)
  shows \( a \cdot c < b \cdot d \)
proof
  have \( a \cdot c < b \cdot c \)
    by (simp add: assms mult-strict-right-mono)
  also have \( \ldots \leq b \cdot d \)
    by (intro mult-left-mono) (use assms in auto)
finally show \(?thesis\).
qed

lemma mult-le-less-imp-less:
  assumes \( a \leq b \) and \( c < d \) and \( 0 < a \) and \( 0 \leq c \)
  shows \( a \cdot c < b \cdot d \)
proof
  have \( a \cdot c \leq b \cdot d \)
    by (simp add: assms mult-right-mono)
  also have \( \ldots < b \cdot d \)
    by (intro mult-strict-left-mono) (use assms in auto)
finally show \(?thesis\).
qed

end

class linordered-semiring-1-strict = linordered-semiring-strict + semiring-1 + zero-less-one
begin
subclass linordered-semiring-1 ..

lemma convex-bound-lt:
  assumes \( x < a \) \( y < a \) \( 0 \leq u \) \( 0 \leq v \) \( u + v = 1 \)
  shows \( u \cdot x + v \cdot y < a \)
proof
  from assms have \( u \cdot x + v \cdot y < u \cdot a + v \cdot a \)
    by (cases \( u = 0 \)) (auto intro: add-less-le_mono mult-strict-left-mono mult-left-mono)
  with assms show \(?thesis\)
unfolding \textit{distrib-right[symmetric]} by simp

qed

end

class \textit{ordered-comm-semiring} = \textit{comm-semiring-0} + \textit{ordered-ab-semigroup-add} +
  assumes \textit{comm-mult-left-mono}: \( a \leq b \Rightarrow 0 \leq c \Rightarrow c \cdot a \leq c \cdot b \)

begin

subclass \textit{ordered-semiring}
proof
  fix \( a \, b \, c :: 'a \)
  assume \( a \leq b \, 0 \leq c \)
  then show \( c \cdot a \leq c \cdot b \) by (rule \textit{comm-mult-left-mono})
  then show \( a \cdot c \leq b \cdot c \) by (simp only: \textit{mult.commute})

qed

end

class \textit{ordered-cancel-comm-semiring} = \textit{ordered-comm-semiring} + \textit{cancel-comm-monoid-add} begin

subclass \textit{comm-semiring-0-cancel} ..
subclass \textit{ordered-comm-semiring} ..
subclass \textit{ordered-cancel-semiring} ..

end

class \textit{linordered-comm-semiring-strict} = \textit{comm-semiring-0} + \textit{linordered-cancel-ab-semigroup-add} +
  assumes \textit{comm-mult-strict-left-mono}: \( a < b \Rightarrow 0 < c \Rightarrow c \cdot a < c \cdot b \)

begin

subclass \textit{linordered-semiring-strict}
proof
  fix \( a \, b \, c :: 'a \)
  assume \( a < b \, 0 < c \)
  then show \( c \cdot a < c \cdot b \)
    by (rule \textit{comm-mult-strict-left-mono})
  then show \( a \cdot c < b \cdot c \)
    by (simp only: \textit{mult.commute})

qed

subclass \textit{ordered-cancel-comm-semiring}
proof
  fix \( a \, b \, c :: 'a \)
  assume \( a \leq b \, 0 \leq c \)
  then show \( c \cdot a \leq c \cdot b \)
    unfolding \textit{le-less}
using mult-strict-left-mono by (cases c = 0) auto

qed

end

class ordered-ring = ring + ordered-cancel-semiring
begin

subclass ordered-ab-group-add ..

lemma less-add-iff1: a * e + c < b * e + d ⇔ (a - b) * e + c < d
by (simp add: algebra-simps)

lemma less-add-iff2: a * e + c < b * e + d ⇔ c < (b - a) * e + d
by (simp add: algebra-simps)

lemma le-add-iff1: a * e + c ≤ b * e + d ⇔ (a - b) * e + c ≤ d
by (simp add: algebra-simps)

lemma le-add-iff2: a * e + c ≤ b * e + d ⇔ c ≤ (b - a) * e + d
by (simp add: algebra-simps)

lemma mult-left-mono-neg: b ≤ a ⇒ c ≤ 0 ⇒ c * a ≤ c * b
by (auto dest: mult-left-mono [of - - c])

lemma mult-right-mono-neg: b ≤ a ⇒ c ≤ 0 ⇒ a * c ≤ b * c
by (auto dest: mult-right-mono [of - - c])

lemma mult-nonpos-nonpos: a ≤ 0 ⇒ b ≤ 0 ⇒ 0 ≤ a * b
using mult-right-mono-neg [of a 0 b] by simp

lemma split-mult-pos-le: (0 ≤ a ∧ 0 ≤ b) ∨ (a ≤ 0 ∧ b ≤ 0) ⇒ 0 ≤ a * b
by (auto simp add: mult-nonpos-nonpos)

end

class abs-if = minus + uminus + ord + zero + abs +
assumes abs-if: |a| = (if a < 0 then - a else a)

class linordered-ring = ring + linordered-semiring + linordered-ab-group-add + abs-if
begin

subclass ordered-ring ..

subclass ordered-ab-group-add-abs
proof
  fix a b
  show |a + b| ≤ |a| + |b|

end
by (auto simp add: abs-if not-le not-less algebra-simps
    simp del: add.commute dest: add-neg-neg add-nonneg-nonneg)
qed (auto simp: abs-if)

lemma zero-le-square [simp]: \(0 \leq a \cdot a\)
  using linear [of 0 a] by (auto simp add: mult-nonpos-nonpos)

lemma not-square-less-zero [simp]: \(\neg (a \cdot a < 0)\)
  by (simp add: not-less)

proposition abs-eq-iff : \(|x| = |y| \iff x = y \lor x = -y\)
  by (auto simp add: abs-if split: if-split-asm)

lemma abs-eq-iff' :
  \(|a| = b \iff b \geq 0 \amp (a = b \lor a = -b)\)
  by (cases a \geq 0) auto

lemma eq-abs-iff' :
  \(a = |b| \iff a \geq 0 \amp (b = a \lor b = -a)\)
  by auto

lemma sum-squares-ge-zero: \(0 \leq x \cdot x + y \cdot y\)
  by (intro add-nonneg-nonneg zero-le-square)

lemma not-sum-squares-lt-zero: \(\neg x \cdot x + y \cdot y < 0\)
  by (simp add: not-less sum-squares-ge-zero)
end

class linordered-ring-strict = ring + linordered-semiring-strict
  + ordered-ab-group-add + abs-if
begin

subclass linordered-ring ..

lemma mult-strict-left-mono-neg: \(b < a \Rightarrow c < 0 \Rightarrow c \cdot a < c \cdot b\)
  using mult-strict-left-mono [of b a - c] by simp

lemma mult-strict-right-mono-neg: \(b < a \Rightarrow c < 0 \Rightarrow a \cdot c < b \cdot c\)
  using mult-strict-right-mono [of b a - c] by simp

lemma mult-neg-neg: \(a < 0 \Rightarrow b < 0 \Rightarrow 0 < a \cdot b\)
  using mult-strict-right-mono-neg [of a 0 b] by simp

subclass ring-no-zero-divisors
proof
  fix a b
  assume a \neq 0
  then have a: \(a < 0 \lor 0 < a\) by (simp add: neq-iff)
assume $b \neq 0$
then have $b: b < 0 \lor 0 < b$ by (simp add: neq_iff)
have $a \cdot b < 0 \lor 0 < a \cdot b$
proof (cases $a < 0$)
case $True$
show $?thesis$
proof (cases $b < 0$)
case $True$
with $\langle a < 0 \rangle$ show $?thesis$
by (auto dest: mult-neg-neg)
next
case $False$
with $b$ have $0 < b$ by auto
with $\langle a < 0 \rangle$ show $?thesis$
by (auto dest: mult-strict-right-mono)
qed
next
case $False$
with $a$ have $0 < a$ by auto
show $?thesis$
proof (cases $b < 0$)
case $True$
by (auto dest: mult-strict-right-mono-neg)
next
case $False$
with $b$ have $0 < b$ by auto
with $\langle 0 < a \rangle$ show $?thesis$
by auto
qed
then show $a \cdot b \neq 0$
by (simp add: neq_iff)
qed

lemma zero-less-mult-iff [algebra-split-simps, field-split-simps]:
$0 < a \cdot b \leftrightarrow 0 < a \land 0 < b \lor a < 0 \land b < 0$
by (cases $a \ 0 \ b \ 0$ rule: linorder-cases(case-product linorder-cases))
(auto simp add: mult-neg-neg not-less le-less dest: zero-less-mult-pos zero-less-mult-pos2)

lemma zero-le-mult-iff [algebra-split-simps, field-split-simps]:
$0 \leq a \cdot b \leftrightarrow 0 \leq a \land 0 \leq b \lor a \leq 0 \land b \leq 0$
by (auto simp add: eq-commute [of 0] le-less not-less zero-less-mult-iff)

lemma mult-less-0-iff [algebra-split-simps, field-split-simps]:
$a \cdot b < 0 \leftrightarrow 0 < a \land b < 0 \lor a < 0 \land 0 < b$
using zero-less-mult-iff [of $-a \ b$] by auto

lemma mult-le-0-iff [algebra-split-simps, field-split-simps]:
$a \cdot b \leq 0 \leftrightarrow 0 \leq a \land b \leq 0 \lor a \leq 0 \land 0 \leq b$
using zero-le-mult-iff [of $-a \ b$] by auto

Cancellation laws for $c \cdot a < c \cdot b$ and $a \cdot c < b \cdot c$, also with the relations
\( \leq \) and equality.

These “disjunction” versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

**Lemma** \( \text{mult-less-cancel-right-disj} \): \( a \cdot c < b \cdot c \iff 0 < c \land a < b \lor c < 0 \land b < a \)

**Proof** (cases \( c = 0 \))

**Case** False

Show \( \text{thesis (is \( \text{lhs} \rightleftharpoons \text{rhs} \))} \)

**Proof**

Assume \( \text{lhs} \)

Then have \( c < 0 \implies b < a \land c > 0 \implies b > a \)

By (auto simp flip: not-le intro: mult-right-mono mult-right-mono-neg)

With False show \( \text{rhs} \)

By (auto simp add: neq-iff)

Next

Assume \( \text{rhs} \)

With False show \( \text{lhs} \)

By (auto simp add: mult-strict-right-mono mult-strict-right-mono-neg)

Qed

Qed auto

**Lemma** \( \text{mult-less-cancel-left-disj} \): \( c \cdot a < c \cdot b \iff 0 \leq c \land a < b \lor c \leq 0 \land b < a \)

**Proof** (cases \( c = 0 \))

**Case** False

Show \( \text{thesis (is \( \text{lhs} \rightleftharpoons \text{rhs} \))} \)

**Proof**

Assume \( \text{lhs} \)

Then have \( c < 0 \implies b < a \land c > 0 \implies b > a \)

By (auto simp flip: not-le intro: mult-left-mono mult-left-mono-neg)

With False show \( \text{rhs} \)

By (auto simp add: neq-iff)

Next

Assume \( \text{rhs} \)

With False show \( \text{lhs} \)

By (auto simp add: mult-strict-left-mono mult-strict-left-mono-neg)

Qed

Qed auto

The “conjunction of implication” lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

**Lemma** \( \text{mult-less-cancel-right} \): \( a \cdot c < b \cdot c \iff (0 \leq c \implies a < b) \land (c \leq 0 \implies b < a) \)

**Using** \( \text{mult-less-cancel-right-disj [of \( a \cdot c \cdot b \)] by auto} \)

**Lemma** \( \text{mult-less-cancel-left} \): \( c \cdot a < c \cdot b \iff (0 \leq c \implies a < b) \land (c \leq 0 \implies b < a) \)

**Using** \( \text{mult-less-cancel-left-disj [of \( c \cdot a \cdot b \)] by auto} \)
lemma mult-le-cancel-right: $a \cdot c \leq b \cdot c \iff (0 < c \rightarrow a \leq b) \land (c < 0 \rightarrow b \leq a)$
  by (simp add: not-less [symmetric] mult-less-cancel-right-disj)

lemma mult-le-cancel-left: $c \cdot a \leq c \cdot b \iff (0 < c \rightarrow a \leq b) \land (c < 0 \rightarrow b \leq a)$
  by (simp add: not-less [symmetric] mult-less-cancel-left-disj)

lemma mult-le-cancel-left-pos: $0 < c \implies c \cdot a \leq c \cdot b \iff a \leq b$
  by (auto simp: mult-le-cancel-left)

lemma mult-le-cancel-left-neg: $c < 0 \implies c \cdot a \leq c \cdot b \iff b \leq a$
  by (auto simp: mult-le-cancel-left)

lemma mult-less-cancel-left-pos: $0 < c \implies c \cdot a < c \cdot b \iff a < b$
  by (auto simp: mult-less-cancel-left)

lemma mult-less-cancel-left-neg: $c < 0 \implies c \cdot a < c \cdot b \iff b < a$
  by (auto simp: mult-less-cancel-left)

lemma mult-le-cancel-right-pos: $0 < c \implies a \cdot c \leq b \cdot c \iff a \leq b$
  by (auto simp: mult-le-cancel-right)

lemma mult-le-cancel-right-neg: $c < 0 \implies a \cdot c \leq b \cdot c \iff b \leq a$
  by (auto simp: mult-le-cancel-right)

lemma mult-less-cancel-right-pos: $0 < c \implies a \cdot c < b \cdot c \iff a < b$
  by (auto simp: mult-less-cancel-right)

lemma mult-less-cancel-right-neg: $c < 0 \implies a \cdot c < b \cdot c \iff b < a$
  by (auto simp: mult-less-cancel-right)

end

lemmas mult-sign-intros =
  mult-nonneg-nonneg mult-nonneg-nonpos
  mult-nonpos-nonneg mult-nonpos-nonpos
  mult-pos-pos mult-pos-neg
  mult-neg-pos mult-neg-neg

class ordered-comm-ring = comm-ring + ordered-comm-semiring
begin
subclass ordered-ring ..
subclass ordered-cancel-comm-semiring ..
end
class linordered-nonzero-semiring = ordered-comm-semiring + monoid-mult + linorder + zero-less-one +
  assumes add-mono1: a < b \implies a + 1 < b + 1
begin

subclass zero-neq-one
  by standard

subclass comm-semiring-1
  by standard (rule mult-1-left)

lemma zero-le-one [simp]: 0 ≤ 1
  by (rule zero-less-one [THEN less-imp-le])

lemma not-one-le-zero [simp]: ¬ 1 ≤ 0
  by (simp add: not-le)

lemma not-one-less-zero [simp]: ¬ 1 < 0
  by (simp add: not-less)

lemma of-bool-less-eq-iff [simp]:
  of-bool P ≤ of-bool Q \iff (P \rightarrow Q);
  by auto

lemma of-bool-less-iff [simp]:
  of-bool P < of-bool Q \iff \neg P \land Q;
  by auto

lemma mult-left-le: c ≤ 1 \implies 0 ≤ a \implies a * c ≤ a
  using mult-left-mono[of c 1 a] by simp

lemma mult-le-one: a ≤ 1 \implies 0 ≤ b \implies b ≤ 1 \implies a * b ≤ 1
  using mult-mono[of a 1 b 1] by simp

lemma zero-less-two: 0 < 1 + 1
  using add-pos-pos[OF zero-less-one zero-less-one] .

end

class linordered-semidom = semidom + linordered-comm-semiring-strict + zero-less-one +
  assumes le-add-diff-inverse2 [simp]: b ≤ a \implies a - b + b = a
begin

subclass linordered-nonzero-semiring
proof
  show a + 1 < b + 1 if a < b for a b
  proof (rule ccontr)
    assume ¬ a + 1 < b + 1
moreover with that have \( a + 1 < b + 1 \)
by simp
ultimately show False
by contradiction
qed
qed

lemma zero-less-eq-of-bool [simp]:
\( \langle 0 \leq \text{of-bool } P \rangle \)
by simp

lemma zero-less-of-bool-iff [simp]:
\( \langle 0 < \text{of-bool } P \longleftrightarrow P \rangle \)
by simp

lemma of-bool-less-eq-one [simp]:
\( \langle \text{of-bool } P \leq 1 \rangle \)
by simp

lemma of-bool-less-one-iff [simp]:
\( \langle \text{of-bool } P < 1 \longleftrightarrow \neg P \rangle \)
by simp

lemma of-bool-or-iff [simp]:
\( \langle \text{of-bool } (P \lor Q) = \max (\text{of-bool } P) (\text{of-bool } Q) \rangle \)
by (simp add: max-def)

Addition is the inverse of subtraction.

lemma le-add-diff-inverse [simp]: \( b \leq a \Longrightarrow b + (a - b) = a \)
by (frule le-add-diff-inverse2) (simp add: add.commute)

lemma add-diff-inverse: \( \neg a < b \Longrightarrow b + (a - b) = a \)
by simp

lemma add-le-imp-le-diff:
assumes \( i + k \leq n \) shows \( i \leq n - k \)
proof
have \( n - (i + k) + i + k = n \)
  by (simp add: assms add.assoc)
with assms add-implies-diff have \( i + k \leq n - k + k \)
  by fastforce
then show \?thesis
  by simp
qed

lemma add-le-add-imp-diff-le:
assumes \( 1: i + k \leq n \)
and \( 2: n \leq j + k \)
shows \( i + k \leq n \Longrightarrow n \leq j + k \Longrightarrow n - k \leq j \)
proof  
  have \( n - (i + k) + i + k = n \)
  using 1 by (simp add: add.assoc)
moreover have \( n - k = n - k - i + i \)
  using 1 by (simp add: add-le-imp-le-diff)
ultimately show \(?thesis \)
  using 2 add-le-imp-le-diff [of n k j k]
  by (simp add: add.commute diff-diff-add)
qed

lemma less-1-mult: \( 1 < m \implies 1 < n \implies 1 < m * n \)
  using mult-strict-mono [of 1 m 1 n]
  by (simp add: less-trans [OF zero-less-one])

end

class linordered-idom = comm-ring-1 + linordered-comm-semiring-strict +
  ordered-ab-group-add + abs-if + sgn +
assumes sgn-if: sgn \( x = (\text{if } x = 0 \text{ then } 0 \text{ else if } 0 < x \text{ then } 1 \text{ else } -1) \)
begin
subclass linordered-ring-strict ..
subclass linordered-semiring-1-strict
proof
  have \( 0 \leq 1 * 1 \)
    by (fact zero-le-square)
then show \( 0 < 1 \)
  by (simp add: le-less)
qed

subclass ordered-comm-ring ..
subclass idom ..

subclass linordered-semidom
by standard simp

subclass idom-abs-sgn
by standard
  (auto simp add: sgn-if abs-if zero-less-mult-iff)

lemma abs-bool-eq [simp]:
  \(|\text{of-bool } P| = \text{of-bool } P|\)
  by simp

lemma linorder-neqE-linordered-idom:
  assumes \( x \not= y \)
  obtains \( x < y \ | \ y < x \)
  using assms by (rule neqE)

These cancellation simp rules also produce two cases when the comparison
is a goal.

lemma mult-cancel-right1: \( c \leq b \cdot c \iff (0 < c \implies 1 \leq b) \land (c < 0 \implies b \leq 1) \)
  using mult-cancel-right [of 1 c b] by simp

lemma mult-cancel-right2: \( a \cdot c \leq c \iff (0 < c \implies a \leq 1) \land (c < 0 \implies 1 \leq a) \)
  using mult-cancel-right [of a c 1] by simp

lemma mult-cancel-left1: \( c \leq c \cdot b \iff (0 < c \implies 1 \leq b) \land (c < 0 \implies b \leq 1) \)
  using mult-cancel-left [of c 1 b] by simp

lemma mult-cancel-left2: \( c \cdot a \leq c \iff (0 < c \implies a \leq 1) \land (c < 0 \implies 1 \leq a) \)
  using mult-cancel-left [of c a 1] by simp

lemma mult-less-cancel-right1: \( c < b \cdot c \iff (0 \leq c \implies 1 < b) \land (c \leq 0 \implies b < 1) \)
  using mult-less-cancel-right [of 1 c b] by simp

lemma mult-less-cancel-right2: \( a \cdot c < c \iff (0 \leq c \implies a < 1) \land (c \leq 0 \implies 1 < a) \)
  using mult-less-cancel-right [of a c 1] by simp

lemma mult-less-cancel-left1: \( c < c \cdot b \iff (0 \leq c \implies 1 < b) \land (c \leq 0 \implies b < 1) \)
  using mult-less-cancel-left [of c 1 b] by simp

lemma mult-less-cancel-left2: \( c \cdot a < c \iff (0 \leq c \implies a < 1) \land (c \leq 0 \implies 1 < a) \)
  using mult-less-cancel-left [of c a 1] by simp

lemma sgn-0-0: \( sgn a = 0 \iff a = 0 \)
  by (fact sgn-eq-0-iff)

lemma sgn-1-pos: \( sgn a = 1 \iff a > 0 \)
  unfolding sgn-if by simp

lemma sgn-1-neg: \( sgn a = -1 \iff a < 0 \)
  unfolding sgn-if by auto

lemma sgn-pos [simp]: \( 0 < a \implies sgn a = 1 \)
  by (simp only: sgn-1-pos)

lemma sgn-neg [simp]: \( a < 0 \implies sgn a = -1 \)
  by (simp only: sgn-1-neg)

lemma abs-sgn: \(|k| = k \cdot sgn k\)
unfolding sgn-if abs-if by auto

lemma sgn-greater [simp]: $0 < \text{sgn} \ a \iff 0 < a$
unfolding sgn-if by auto

lemma sgn-less [simp]: $\text{sgn} \ a < 0 \iff a < 0$
unfolding sgn-if by auto

lemma abs-sgn-eq-1 [simp]:
$a \neq 0 \implies |\text{sgn} \ a| = 1$
by simp

lemma abs-sgn-eq: $|\text{sgn} \ a| = (\text{if} \ a = 0 \text{ then } 0 \text{ else } 1)$
by (simp add: sgn-if)

lemma sgn-mult-self-eq [simp]:
$\text{sgn} \ a \ast \text{sgn} \ a = \text{of-bool} \ (a \neq 0)$
by (cases $a > 0$) simp-all

lemma left-sgn-mult-self-eq [simp]:
$(\text{sgn} \ a \ast (\text{sgn} \ a \ast b) = \text{of-bool} \ (a \neq 0) \ast b)$
by (simp flip: mult.assoc)

lemma abs-mult-self-eq [simp]:
$|a| \ast |a| = a \ast a$
by (cases $a > 0$) simp-all

lemma same-sgn-sgn-add:
$\text{sgn} \ (a + b) = \text{sgn} \ a \text{ if } \text{sgn} \ b = \text{sgn} \ a$
proof (cases $a \ 0$ rule: linorder-cases)
case equal
  with that show ?thesis
  by simp
next
case less
  with that have $b < 0$
  by (simp add: sgn-1-neg)
  with $a < 0 \cdot$ have $a + b < 0$
  by (rule add-neg-neg)
  with $a < 0 \cdot$ show ?thesis
  by simp
next
case greater
  with that have $b > 0$
  by (simp add: sgn-1-pos)
  with $a > 0 \cdot$ have $a + b > 0$
  by (rule add-pos-pos)
  with $a > 0 \cdot$ show ?thesis
  by simp
lemma same-sgn-abs-add:
\[ |a + b| = |a| + |b| \text{ if } \text{sgn } b = \text{sgn } a \]

proof
- have \( a + b = \text{sgn } a \times |a| + \text{sgn } b \times |b| \)
  by (simp add: sgn-mult-abs)
also have \( \ldots = \text{sgn } a \times (|a| + |b|) \)
  using that by (simp add: algebra-simps)
finally show \( ?\text{thesis} \)
  by (auto simp add: abs-mult)

qed

lemma sgn-not-eq-imp:
\[ \text{sgn } a = - \text{sgn } b \text{ if } \text{sgn } b \neq \text{sgn } a \text{ and } \text{sgn } a \neq 0 \text{ and } \text{sgn } b \neq 0 \]
using that by (cases a < 0) (auto simp add: sgn-0-0 sgn-1-pos sgn-1-neg)

lemma abs-dvd-iff [simp]:
\[ |m| \text{ dvd } k \iff m \text{ dvd } k \]
by (simp add: abs-if)

lemma dvd-abs-iff [simp]:
\[ m \text{ dvd } |k| \iff m \text{ dvd } k \]
by (simp add: abs-if)

lemma dvd-if-abs-eq:
\[ |l| = |k| \Rightarrow l \text{ dvd } k \]
by (subst abs-dvd-iff [symmetric]) simp

The following lemmas can be proven in more general structures, but are dangerous as simp rules in absence of \(- ?a = ?a\) = \((- ?a \leq ?a) = ((?a < ?a) < ?a), (- ?a < ?a) = ((?a::'a < ?a), (- ?a \leq ?a) = ((0::'a) \leq ?a).

lemma equation-minus-iff-1 [simp, no-atp]:
\[ 1 = - a \iff a = - 1 \]
by (fact equation-minus-iff)

lemma minus-equation-iff-1 [simp, no-atp]:
\[ - a = 1 \iff a = - 1 \]
by (subst minus-equation-iff, auto)

lemma le-minus-iff-1 [simp, no-atp]:
\[ 1 \leq - b \iff b \leq - 1 \]
by (fact le-minus-iff)

lemma minus-le-iff-1 [simp, no-atp]:
\[ - a \leq 1 \iff - 1 \leq a \]
by (fact minus-le-iff)

lemma less-minus-iff-1 [simp, no-atp]:
\[ 1 < - b \iff b < - 1 \]
by (fact less-minus-iff)

lemma minus-less-iff-1 [simp, no-atp]:
\[ - a < 1 \iff - 1 < a \]
by (fact minus-less-iff)

lemma add-less-zeroD:
shows \( x + y < 0 \Rightarrow x < 0 \lor y < 0 \)
by (auto simp: not-less intro; le-less-trans[of -x+y])

Is this really better than just rewriting with abs-if?

lemma abs-split [no-atp]: \( \langle P |a| \longleftrightarrow (0 \leq a \rightarrow P a) \land (a < 0 \rightarrow P (-a)) \rangle \)
  by (force dest: order-less-le-trans simp add: abs-if linorder-not-less)

end

class discrete-linordered-semidom = linordered-semidom +
  assumes less-iff-succ-less-eq: \( \langle a < b \longleftrightarrow a + 1 \leq b \rangle \)
begin

lemma less-eq-iff-succ-less:
  \( \langle a \leq b \longleftrightarrow a < b + 1 \rangle \)
  using less-iff-succ-less-eq [of a \( b + 1 \)] by simp

end

Reasoning about inequalities with division
context linordered-semidom
begin

lemma less-add-one: \( a < a + 1 \)
proof
  have \( a + 0 < a + 1 \)
    by (blast intro: zero-less-one add-strict-left-mono)
  then show \( \text{thesis} \) by simp
qed

end

context linordered-idom
begin

lemma mult-right-le-one-le: \( 0 \leq x \Rightarrow 0 \leq y \Rightarrow y \leq 1 \Rightarrow x \ast y \leq x \)
  by (rule mult-left-le)

lemma mult-left-le-one-le: \( 0 \leq x \Rightarrow 0 \leq y \Rightarrow y \leq 1 \Rightarrow y \ast x \leq x \)
  by (auto simp add: mult-le-cancel-right2)

end

Absolute Value
context linordered-idom
begin

lemma mult-sgn-abs: \( \text{sgn} \ x \ast \|x\| = x \)
  by (fact sgn-mult-abs)

end
lemma abs-one: |1| = 1
by (fact abs-1)

end

class ordered-ring-abs = ordered-ring + ordered-ab-group-add-abs +
assumes abs-eq-mult:
(0 ≤ a ∨ a ≤ 0) ∧ (0 ≤ b ∨ b ≤ 0) → |a * b| = |a| * |b|

context linordered-idom
begin

subclass ordered-ring-abs
by standard (auto simp: abs-if not-less mult-less-0-iff)

lemma abs-mult-self: |a| * |a| = a * a
by (fact abs-mult-self-eq)

lemma abs-mult-less:
assumes ac: |a| < c
and bd: |b| < d
shows |a| * |b| < c * d
proof –
from ac have 0 < c
by (blast intro: le-less-trans abs-ge-zero)
with bd show ?thesis by (simp add: ac mult-strict-mono)
qed

lemma abs-less-iff: |a| < b ←→ a < b ∧ − a < b
by (simp add: less-le abs-le-iff) (auto simp add: abs-if)

lemma abs-mult-pos: 0 ≤ x → |y| * x = |y * x|
by (simp add: abs-mult)

lemma abs-mult-pos': 0 ≤ x → x * |y| = |x * y|
by (simp add: abs-mult)

lemma abs-diff-less-iff: |x − a| < r ←→ a − r < x ∧ x < a + r
by (auto simp add: diff-less-eq ac-simps abs-less-iff)

lemma abs-diff-le-iff: |x − a| ≤ r ←→ a − r ≤ x ∧ x ≤ a + r
by (auto simp add: diff-le-eq ac-simps abs-le-iff)

lemma abs-add-one-gt-zero: 0 < 1 + |x|
by (auto simp: abs-if not-less intro: zero-less-one add-strict-increasing less-trans)
end
16.8 Dioids

Dioids are the alternative extensions of semirings, a semiring can either be a ring or a dioid but never both.

```plaintext
class dioid = semiring-1 + canonically-ordered-monoid-add
begin

subclass ordered-semiring
  by standard (auto simp: le-iff-add distrib-left distrib-right)

end

hide-fact (open) comm-mult-left-mono comm-mult-strict-left-mono distrib
```

17 Natural numbers

```plaintext
theory Nat
imports Inductive Typedef Fun Rings
begin

17.1 Type ind
typedecl ind

axiomatization Zero-Rep :: ind and Suc-Rep :: ind ⇒ ind
— The axiom of infinity in 2 parts:
  where Suc-Rep-inject: Suc-Rep x = Suc-Rep y ⇒ x = y

17.2 Type nat
Type definition

inductive Nat :: ind ⇒ bool
where
  Zero-Repl: Nat Zero-Rep
| Suc-Repl: Nat i ⇒ Nat (Suc-Rep i)

typedef nat = {n. Nat n}
morphisms Rep-Nat Abs-Nat
using Nat.Zero-Repl by auto

lemma Nat-Rep-Nat: Nat (Rep-Nat n)
```
using Rep-Nat by simp

lemma Nat-Abs-Nat-inverse: Nat n ⇒ Rep-Nat (Abs-Nat n) = n
  using Abs-Nat-inverse by simp

lemma Nat-Abs-Nat-inject: Nat n ⇒ Nat m ⇒ Abs-Nat n = Abs-Nat m ←→ n = m
  using Abs-Nat-inject by simp

instantiation nat :: zero
begin

definition Zero-nat-def: 0 = Abs-Nat Zero-Rep

instance ..

end

definition Suc :: nat ⇒ nat
  where Suc n = Abs-Nat (Suc-Rep (Rep-Nat n))

lemma Suc-not-Zero: Suc m ≠ 0
  by (simp add: Zero-nat-def Suc-def Suc-RepI Zero-RepI

lemma Zero-not-Suc: 0 ≠ Suc m
  by (rule not-sym) (rule Suc-not-Zero)

lemma Suc-Rep-inject': Suc-Rep x = Suc-Rep y ←→ x = y
  by (rule iffI, rule Suc-Rep-inject) simp-all

lemma nat-induct0:
  assumes P 0 and \(\forall n. P n \Rightarrow P (Suc n)\)
  shows P n
proof −
  have P (Abs-Nat (Rep-Nat n))
    using assms unfolding Zero-nat-def Suc-def
        [THEN subst])
  then show ?thesis
    by (simp add: Rep-Nat-inverse)
qed

free-constructors case-nat for 0 :: nat | Suc pred
  where pred (0 :: nat) = (0 :: nat)
proof atomize-elim
  fix n
  show n = 0 ∨ (∃m. n = Suc m)
    by (induction n rule: nat-induct0) auto
next
  fix n m
  show \((\text{Suc} \ n) = \text{Suc} \ m\) = \((n = m)\)
next
  fix n
  show \(0 \neq \text{Suc} \ n\)
    by (simp add: Suc-not-Zero)
qed

— Avoid name clashes by prefixing the output of old-rep-datatype with old.
setup (Sign.mandatory-path old)

old-rep-datatype \(0 :: \text{nat} \ \text{Suc}\)
  by (erule nat-induct0) auto

setup (Sign.parent-path)

— But erase the prefix for properties that are not generated by free-constructors.
setup (Sign.mandatory-path nat)

declare old.nat.inject[iff del]
  and old.nat.distinct(1)[simp del, induct-simp del]

lemmas induct = old.nat.induct
lemmas inducts = old.nat.inducts
lemmas rec = old.nat.rec
lemmas simps = nat.inject nat.distinct nat.case nat.rec

setup (Sign.parent-path)

abbreviation rec-nat :: '\(a\) ⇒ (nat ⇒ '\(a\) ⇒ '\(a\)) ⇒ nat ⇒ '\(a\)
  where rec-nat ≡ old.rec-nat

declare nat.sel[code del]

hide-const (open) Nat.pred — hide everything related to the selector
hide-fact
  nat.case-eq-if
  nat.collapse
  nat.expand
  nat.sel
  nat.exhaust-sel
  nat.split-sel
  nat.split-sel-asn

lemma nat-exhaust [case-names 0 Suc, cases type: nat]:
THEORY "Nat"

(y = 0 → P) → (∀nat. y = Suc nat → P) → P
— for backward compatibility – names of variables differ
by (rule old.nat.exhaust)

lemma nat-induct [case-names 0 Suc, induct type: nat]:
fixes n
assumes P 0 and ∀n. P n → P (Suc n)
shows P n
— for backward compatibility – names of variables differ
using assms by (rule nat.induct)

hide-fact
nat-exhaust
nat-induct0

ML
val nat-basic-lfp-sugar =
  let
    val ctr-sugar = the (Ctr-Sugar.ctr-sugar-of-global theory type-name (nat));
    val recx = Logic.varify-types-global term (rec-nat);
    val C = body-type (fastype-of recx);
in
    { T = HOLogic.natT, fp-res-index = 0, C = C, fun-arg-Tss = [[]], [[HOLogic.natT, C]],
      ctr-sugar = ctr-sugar, recx = recx, rec-thms = @ {ths nat.rec} }
  end;

setup
val basic-lfp-sugars-of = [typ (nat)] - - ctxt =
  let
    fun basic-lfp-sugars-of - [typ (nat)] - - ctxt =
      ( [[]], [0], [nat-basic-lfp-sugar], [[]], [], [], TrueI (*dummy*), [], false, ctxt)
    | basic-lfp-sugars-of bs arg-Ts callers callssss ctxt =
      BNF-LFP.Rec-Sugar.default-basic-lfp-sugars-of bs arg-Ts callers callssss ctxt;
in
BNF-LFP.Rec-Sugar.register-lfp-rec-extension
{ nested-simps = [], special-endgame-tac = K (K (K no-tac)), is-new-datatype
  = K (K true),
  basic-lfp-sugars-of = basic-lfp-sugars-of, rewrite-nested-rec-call = NONE }
end

Injectiveness and distinctness lemmas

lemma inj-Suc [simp]:
inj-on Suc N
by (simp add: inj-on-def)

lemma bij-betw-Suc [simp]:
bij-betw Suc M N ↔ Suc ' M = N
THEORY "Nat"

by (simp add: bij-betw-def)

lemma Suc-neq-Zero: Suc m = 0 \implies R
by (rule notE) (rule Suc-not-Zero)

lemma Zero-neq-Suc: 0 = Suc m \implies R
by (ruleSuc-neq-Zero) (erule sym)

lemma Suc-inject: Suc x = Suc y \implies x = y
by (rule inj-Suc [THEN injD])

lemma n-not-Suc-n: n \neq Suc n
by (induct n) simp-all

lemma Suc-n-not-n: Suc n \neq n
by (rule not-sym) (erule n-not-Suc-n)

A special form of induction for reasoning about m < n and m − n.

lemma diff-induct:
assumes \( \forall x. P x 0 \)
and \( \forall y. P 0 (Suc y) \)
and \( \forall x y. P x y \implies P (Suc x) (Suc y) \)
supports P m n
proof (induct n arbitrary: m)
case 0
show ?case by (rule assms(1))
next
case (Suc n)
show ?case
proof (induct m)
case 0
show ?case by (rule assms(2))
next
case (Suc m)
from \( P m n \) show ?case by (rule assms(3))
qed
qed

17.3 Arithmetic operators

instantiation nat :: comm-monoid-diff
begin

primrec plus-nat
where
  add-0: 0 + n = (n::nat)
| add-Suc: Suc m + n = Suc (m + n)

lemma add-0-right [simp]: m + 0 = m
for \( m :: \text{nat} \)
by \((\text{induct } m)\) simp-all

lemma add-Suc-right [simp]: \( m + \text{Suc} \ n = \text{Suc} \ (m + n) \)
by \((\text{induct } m)\) simp-all

declare add-0 [code]

lemma add-Suc-shift [code]: \( \text{Suc} \ m + n = m + \text{Suc} \ n \)
by simp

primrec minus-nat
where
diff-0 [code]: \( m - 0 = (m :: \text{nat}) \)
| diff-Suc: \( m - \text{Suc} \ n = (\text{case } m - n \text{ of } 0 \Rightarrow 0 | \text{Suc } k \Rightarrow k) \)

declare diff-Suc [simp del]

lemma diff-0-eq-0 [simp, code]: \( 0 - n = 0 \)
for \( n :: \text{nat} \)
by \((\text{induct } n)\) (simp-all add: diff-Suc)

lemma diff-Suc-Suc [simp, code]: \( \text{Suc} \ m - \text{Suc} \ n = m - n \)
by \((\text{induct } n)\) (simp-all add: diff-Suc)

instance proof
fix \( n \ m \ q :: \text{nat} \)
show \((n + m) + q = n + (m + q)\) by \((\text{induct } n)\) simp-all
show \( n + m = m + n \) by \((\text{induct } n)\) simp-all
show \( m + n - m = n \) by \((\text{induct } m)\) simp-all
show \( n - m - q = n - (m + q) \) by \((\text{induct } q)\) (simp-all add: diff-Suc)
show \( 0 + n = n \) by simp
show \( 0 - n = 0 \) by simp
qed

end

hide-fact \( \text{(open)} \) add-0 add-0-right diff-0

instantiation \( \text{nat} :: \text{comm-semiring-1-cancel} \)
begin

definition One-nat-def [simp]: \( 1 = \text{Suc } 0 \)

primrec times-nat
where
mult-0: \( 0 \times n = (0 :: \text{nat}) \)
| mult-Suc: \( \text{Suc} \ m \times n = n + (m \times n) \)
lemma mult-0-right [simp]: \( m \times 0 = 0 \)
  for \( m :: \text{nat} \)
  by (induct m) simp-all

lemma mult-Suc-right [simp]: \( m \times \text{Suc } n = m + (m \times n) \)
  by (induct m) (simp-all add: add.left-commute)

lemma add-mult-distrib: \((m + n) \times k = (m \times k) + (n \times k)\)
  for \( m n k :: \text{nat} \)
  by (induct m) (simp-all add: add.assoc)

instance
proof
  fix \( k n m q :: \text{nat} \)
  show \( 0 \neq (1::\text{nat}) \)
    by simp
  show \( 1 \times n = n \)
    by simp
  show \( n \times m = m \times n \)
    by (induct n) simp-all
  show \((n \times m) \times q = n \times (m \times q)\)
    by (induct n) (simp-all add: add-mult-distrib)
  show \( (n + m) \times q = n \times q + m \times q \)
    by (rule add-mult-distrib)
  show \( k \times (m - n) = (k \times m) - (k \times n) \)
    by (induct m n rule: diff-induct) simp-all
qed

end

17.3.1 Addition

Reasoning about \( m + 0 = 0 \), etc.

lemma add-is-0 [iff]: \( m + n = 0 \iff m = 0 \land n = 0 \)
  for \( m n :: \text{nat} \)
  by (cases m) simp-all

lemma add-is-1: \( m + n = \text{Suc } 0 \iff m = \text{Suc } 0 \land n = 0 \lor m = 0 \land n = \text{Suc } 0 \)
  by (cases m) simp-all

lemma one-is-add: \( \text{Suc } 0 = m + n \iff m = \text{Suc } 0 \land n = 0 \lor m = 0 \land n = \text{Suc } 0 \)
  by (rule trans, rule eq-commute, rule add-is-1)

lemma add-eq-self-zero: \( m + n = m \implies n = 0 \)
  for \( m n :: \text{nat} \)
  by (induct m) simp-all
lemma plus-1-eq-Suc:  
plus 1 = Suc  
by (simp add: fun-eq-iff)

lemma Suc-eq-plus1: Suc n = n + 1  
by simp

lemma Suc-eq-plus1-left: Suc n = 1 + n  
by simp

17.3.2 Difference

lemma Suc-diff-diff [simp]: (Suc m - n) - Suc k = m - n - k  
by (simp add: diff-diff-add)

lemma diff-Suc-1: Suc n - 1 = n  
by simp

lemma diff-Suc-1': Suc n - Suc 0 = n  
by simp

17.3.3 Multiplication

lemma mult-is-0 [simp]: m * n = 0 <-> m = 0 ∨ n = 0 for m n :: nat  
by (induct m) auto

lemma mult-eq-1-iff [simp]: m * n = Suc 0 <-> m = Suc 0 ∧ n = Suc 0  
proof (induct m)  
  case 0  
  then show ?case by simp
  next  
  case (Suc m)  
  then show ?case by (induct n) auto
qed

lemma one-eq-mult-iff [simp]: Suc 0 = m * n <-> m = Suc 0 ∧ n = Suc 0  
by (simp add: eq-commute flip: mult-eq-1-iff)

lemma nat-mult-eq-1-iff [simp]: m * n = 1 <-> m = 1 ∧ n = 1  
and nat-1-eq-mult-iff [simp]: 1 = m * n <-> m = 1 ∧ n = 1 for m n :: nat  
by auto

lemma mult-cancel1 [simp]: k * m = k * n <-> m = n ∨ k = 0  
for k m n :: nat
proof  
  have k #= 0 --> k * m = k * n --> m = n  
  proof (induct n arbitrary: m)  
    case 0  
    then show m = 0 by simp
  next
case (Suc n)
then show m = Suc n
  by (cases m) (simp-all add: eq-commute [of 0])
qed
then show ?thesis by auto
qed

lemma mult-cancel2 [simp]; m * k = n * k ↔ m = n ∨ k = 0
for k m n :: nat
by (simp add: mult.commute)

lemma Suc-mult-cancel1: Suc k * m = Suc k * n ↔ m = n
by (subst mult-cancel1) simp

17.4 Orders on nat
17.4.1 Operation definition
instantiation nat :: linorder
begin

primrec less-eq-nat
where
  (0::nat) ≤ n ↔ True
| Suc m ≤ n ↔ (case n of 0 ⇒ False | Suc n ⇒ m ≤ n)

declare less-eq-nat.simps [simp del]

lemma le0 [iff]: 0 ≤ n for
  n :: nat
by (simp add: less-eq-nat.simps)

lemma [code]: 0 ≤ n ↔ True
for n :: nat
by simp

definition less-nat
where less-eq-Suc-le: n < m ↔ Suc n ≤ m

lemma Suc-le-mono [iff]: Suc n ≤ Suc m ↔ n ≤ m
by (simp add: less-eq-nat.simps(2))

lemma Suc-le-eq [code]: Suc m ≤ n ↔ m < n
unfolding less-eq-Suc-le ..

lemma le-0-eq [iff]: n ≤ 0 ↔ n = 0
for n :: nat
by (induct n) (simp-all add: less-eq-nat.simps(2))

lemma not-less0 [iff]: ¬ n < 0
for \( n :: \text{nat} \)
by (simp add: less-eq-Suc-le)

lemma less-nat-zero-code [code]: \( n < 0 \longleftrightarrow False \)
for \( n :: \text{nat} \)
by simp

lemma Suc-less-eq [iff]: \( \text{Suc} \ m < \text{Suc} \ n \longleftrightarrow m < n \)
by (simp add: less-eq-Suc-le)

lemma less-Suc-eq-le [code]: \( m < \text{Suc} \ n \longleftrightarrow m \leq n \)
by (simp add: less-eq-Suc-le)

lemma Suc-less-eq2: \( \text{Suc} \ n < m \longleftrightarrow (\exists m'. m = \text{Suc} \ m' \land n < m') \)
by (cases m) auto

lemma le-SucI: \( m \leq n \Longrightarrow m \leq \text{Suc} \ n \)
by (induct m arbitrary: n) (simp-all add: less-eq-nat.simps(2) split: nat.splits)

lemma Suc-leD: \( \text{Suc} \ m \leq n \Longrightarrow m \leq n \)
by (cases n) (auto intro: le-SucI)

lemma less-SucI: \( m < n \Longrightarrow m < \text{Suc} \ n \)
by (simp add: less-eq-Suc-le) (erule Suc-leD)

lemma Suc-lessD: \( \text{Suc} \ m < n \Longrightarrow m < n \)
by (simp add: less-eq-Suc-le) (erule Suc-leD)

instance
proof
fix \( n \) \( m \) \( q :: \text{nat} \)
show \( n < m \longleftrightarrow n \leq m \land \neg m \leq n \)
proof (induct n arbitrary: m)
case 0
then show \(?case\)
  by (cases m) (simp-all add: less-eq-Suc-le)
next
case (Suc n)
then show \(?case\)
  by (cases m) (simp-all add: less-eq-Suc-le)
qed

show \( n \leq n \)
by (induct n) simp-all
then show \( n = m \ \text{if} \ n \leq m \ \text{and} \ m \leq n \)
  using that by (induct n arbitrary: m)
  (simp-all add: less-eq-nat.simps(2) split: nat.splits)
show \( n \leq q \ \text{if} \ n \leq m \ \text{and} \ m \leq q \)
  using that
proof (induct n arbitrary: m q)
case 0
  show ?case by simp
next
  case (Suc n)
  then show ?case
    by (simp-all (no-asn-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
    simp-all (no-asn-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
    simp-all (no-asn-use) add: less-eq-nat.simps(2) split: nat.splits)
qed
show n ≤ m ∨ m ≤ n
  by (induct n arbitrary: m)
  (simp-all add: less-eq-nat.simps(2) split: nat.splits)
qed
end

instantiation nat :: order-bot
begin

definition bot-nat :: nat
  where bot-nat = 0

instance
  by standard (simp add: bot-nat-def)
end

instance nat :: no-top
  by standard (auto intro: lessSuc-eq-le [THEN iffD2])

17.4.2 Introduction properties

lemma lessI [iff]: n < Suc n
  by (simp add: less-Suc-eq-le)

lemma zero-less-Suc [iff]: 0 < Suc n
  by (simp add: less-Suc-eq-le)

17.4.3 Elimination properties

lemma less-not-refl: ¬ n < n
  for n :: nat
  by (rule order-less-irrefl)

lemma less-not-refl2: n < m ⇒ m ≠ n
  for m n :: nat
  by (rule not-sym) (rule less-imp-neq)

lemma less-not-refl3: s < t ⇒ s ≠ t
  for s t :: nat
by (rule less-imp-neq)

**lemma** less-irrefl-nat: \( n < n \Rightarrow R \)
for \( n :: \text{nat} \)
by (rule notE, rule less-not-refl)

**lemma** less-zeroE: \( n < 0 \Rightarrow R \)
for \( n :: \text{nat} \)
by (rule notE) (rule not-less0)

**lemma** less-Suc-eq: \( m < \text{Suc} n \leftrightarrow m < n \lor m = n \)
unfolding less-Suc-eq le-less ..

**lemma** less-Suc0 [iff]: \( (n < \text{Suc} 0) = (n = 0) \)
by (simp add: less-Suc-eq)

**lemma** less-one [iff]: \( n < 1 \leftrightarrow n = 0 \)
for \( n :: \text{nat} \)
unfolding One-nat-def by (rule less-Suc0)

**lemma** Suc-mono: \( m < n \Rightarrow \text{Suc} m < \text{Suc} n \)
by simp

"Less than" is antisymmetric, sort of.

**lemma** less-antisym: \( \neg n < m \Rightarrow n < \text{Suc} m \Rightarrow m = n \)
unfolding not-less less-Suc-eq-le by (rule antisym)

**lemma** nat-neq-iff: \( m \neq n \leftrightarrow m < n \lor n < m \)
for \( m n :: \text{nat} \)
by (rule linorder-neq-iff)

17.4.4 Inductive (?) properties

**lemma** Suc-less1: \( m < n \Rightarrow \text{Suc} m \neq n \Rightarrow \text{Suc} m < n \)
unfolding less-eq-Suc-le [of \( m \)] le-less by simp

**lemma** lessE:
assumes major: \( i < k \)
and 1: \( k = \text{Suc} i \Rightarrow P \)
and 2: \( \forall j. i < j \Rightarrow k = \text{Suc} j \Rightarrow P \)
shows \( P \)
proof –
from major have \( \exists j. i \leq j \land k = \text{Suc} j \)
unfolding less-eq-Suc-le by (induct \( k \)) simp-all
then have \( (\exists j. i < j \land k = \text{Suc} j) \lor k = \text{Suc} i \)
by (auto simp add: less-le)
with 1 2 show \( P \) by auto
qed
lemma less-SucE:
assumes major: \( m < \text{Suc} \ n \)
and less: \( m < n \implies P \)
and eq: \( m = n \implies P \)
shows \( P \)
proof (rule major [THEN lessE])
show \( \text{Suc} \ n = \text{Suc} \ m \implies P \)
  using eq by blast
show \( \bigwedge j. \left[ m < j ; \text{Suc} \ n = \text{Suc} \ j \right] \implies P \)
  by (blast intro: less)
qed

lemma Suc-lessE:
assumes major: \( \text{Suc} \ i < k \)
and minor: \( \bigwedge j. i < j \implies k = \text{Suc} \ j \implies P \)
shows \( P \)
proof (rule major [THEN lessE])
show \( k = \text{Suc} \ (\text{Suc} \ i) \implies P \)
  using lessI minor by iprover
show \( \bigwedge j. \left[ \text{Suc} \ i < j ; k = \text{Suc} \ j \right] \implies P \)
  using Suc-lessD minor by iprover
qed

lemma Suc-less-SucD: \( \text{Suc} \ m < \text{Suc} \ n \implies m < n \)
by simp

lemma less-trans-Suc:
assumes le: \( i < j \)
shows \( j < k \implies \text{Suc} \ i < k \)
proof (induct k)
case 0
then show ?case by simp
next
case (Suc k)
with le show ?case
  by simp (auto simp add: less-Suc-eq dest: Suc-lessD)
qed

Can be used with less-Suc-eq to get \( n = m \lor n < m \).

lemma not-less-eq: \( \neg m < n \iff n < \text{Suc} \ m \)
by (simp only: not-less less-Suc-eq-le)

lemma not-less-eq-eq: \( \neg m \leq n \iff \text{Suc} \ n \leq m \)
by (simp only: not-le Suc-le-eq)

Properties of 'less than or equal'.

lemma le-imp-less-Suc: \( m \leq n \implies m < \text{Suc} \ n \)
by (simp only: less-Suc-eq-le)
lemma Suc-n-not-le-n: ¬ Suc n ≤ n
  by (simp add: not-le less-Suc-eq-le)

lemma le-Suc-eq: m ≤ Suc n ⟷ m ≤ n ∨ m = Suc n
  by (simp add: less-Suc-eq-le [symmetric] less-Suc-eq)

lemma le-SucE: m ≤ Suc n ⟹ (m ≤ n ⟹ R) ⟹ (m = Suc n ⟹ R) ⟹ R
  by (drule le-Suc-eq [THEN iffD1], iprover+)

lemma Suc-leI: m < n ⟹ Suc m ≤ n
  by (simp only: Suc-le-eq)

Stronger version of Suc-leD.

lemma Suc-le-lessD: Suc m ≤ n ⟹ m < n
  by (simp only: Suc-le-eq)

lemma less-imp-le-nat: m < n ⟹ m ≤ n for m n :: nat
  unfolding less-eq-Suc-le by (rule Suc-leD)

For instance, (Suc m < Suc n) = (Suc m ≤ n) = (m < n)

lemmas le-simps = less-imp-le-nat less-Suc-eq-le Suc-le-eq

Equivalence of m ≤ n and m < n ∨ m = n

lemma less-or-eq-imp-le: m < n ∨ m = n ⟹ m ≤ n for m n :: nat
  unfolding le-less .

lemma le-eq-less-or-eq: m ≤ n ⟷ m < n ∨ m = n
  for m n :: nat
  by (rule le-less)

Useful with blast.

lemma eq-imp-le: m = n ⟹ m ≤ n for m n :: nat
  by auto

lemma le-refl: n ≤ n for n :: nat
  by simp

lemma le-trans: i ≤ j ⟹ j ≤ k ⟹ i ≤ k
  for i j k :: nat
  by (rule order-trans)

lemma le-antisym: m ≤ n ⟹ n ≤ m ⟹ m = n
  for m n :: nat
  by (rule antisym)
lemma `nat-less-le`: $m < n \iff m \leq n \land m \neq n$
  for $m, n :: \text{nat}$
  by (rule `less-le`)

lemma `le-neq-implies-less`: $m \leq n \implies m \neq n \implies m < n$
  for $m, n :: \text{nat}$
  unfolding `less-le` ..

lemma `nat-le-linear`: $m \leq n \lor n \leq m$
  for $m, n :: \text{nat}$
  by (rule `linear`)

lemmas `linorder-neqE-nat` = `linorder-neqE` [where `'a = nat`]

lemma `le-less-Suc-eq`: $m \leq n \implies n < \text{Suc} m \iff n = m$
  unfolding `less-Suc-eq-le` by auto

lemma `not-less-less-Suc-eq`: $\neg n < m \implies n < \text{Suc} m \iff n = m$
  unfolding `not-less` by (rule `le-less-Suc-eq`)

lemmas `not-less-simps` = `not-less-less-Suc-eq` le-less-Suc-eq

lemma `not0-implies-Suc`: $n \neq 0 \implies \exists m. n = \text{Suc} m$
  by (cases $n$) simp-all

lemma `gr0-implies-Suc`: $n > 0 \implies \exists m. n = \text{Suc} m$
  by (cases $n$) simp-all

lemma `gr-implies-not0`: $m < n \implies n \neq 0$
  for $m, n :: \text{nat}$
  by (cases $n$) simp-all

lemma `neq0-conv[iff]`: $n \neq 0 \iff 0 < n$
  for $n :: \text{nat}$
  by (cases $n$) simp-all

This theorem is useful with `blast`

lemma `gr0I`: $(n = 0 \implies \text{False}) \implies 0 < n$
  for $n :: \text{nat}$
  by (rule `neq0-conv[THEN iffD1]`) iprover

lemma `gr0-conv-Suc`: $0 < n \iff (\exists m. n = \text{Suc} m)$
  by (fast intro: not0-implies-Suc)

lemma `not-gr0 [iff]`: $\neg 0 < n \iff n = 0$
  for $n :: \text{nat}$
  using `neq0-conv` by blast

lemma `Suc-le-D`: $\text{Suc} n \leq m' \implies \exists m. m' = \text{Suc} m$
by (induct m') simp-all

Useful in certain inductive arguments

**Lemma** less-Suc-eq-0-disj: \( m < \text{Suc} \ n \iff m = 0 \lor (\exists j. m = \text{Suc} \ j \land j < n) \)

by (cases m) simp-all

**Lemma** All-less-Suc: \((\forall i < \text{Suc} \ n. P \ i) = (P \ n \land (\forall i < n. P \ i))\)

by (auto simp: less-Suc-eq)

**Lemma** All-less-Suc2: \((\forall i < \text{Suc} \ n. P \ i) = (P 0 \land (\forall i < n. P (\text{Suc} \ i)))\)

by (auto simp: less-Suc-eq-0-disj)

**Lemma** Ex-less-Suc: \((\exists i < \text{Suc} \ n. P \ i) = (P n \lor (\exists i < n. P \ i))\)

by (auto simp: less-Suc-eq)

**Lemma** Ex-less-Suc2: \((\exists i < \text{Suc} \ n. P \ i) = (P 0 \lor (\exists i < n. P (\text{Suc} \ i)))\)

by (auto simp: less-Suc-eq-0-disj)

**mono** (non-strict) doesn’t imply increasing, as the function could be constant

**Lemma** strict-mono-imp-increasing:

fixes \( n :: \text{nat} \)

assumes **strict-mono** \( f \) shows \( f n \geq n \)

**proof** (induction \( n \))

case \( 0 \)
then show ?case
by auto

next
case \( (\text{Suc} \ n) \)
then show ?case

unfolding not-less-eq-eq [symmetric]

using Suc-n-not-le-n assms order-trans strict-mono-less-eq by blast

qed

17.4.5 Monotonicity of Addition

**Lemma** Suc-pred [simp]: \( n > 0 \Rightarrow \text{Suc} \ (n - \text{Suc} \ 0) = n \)

by (simp add: diff-Suc split: nat.split)

**Lemma** Suc-diff-1 [simp]: \( 0 < n \Rightarrow \text{Suc} \ (n - 1) = n \)

unfolding One-nat-def by (rule Suc-pred)

**Lemma** nat-add-left-cancel-le [simp]: \( k + m \leq k + n \iff m \leq n \)

for \( k \ m \ n :: \text{nat} \)

by (induct \( k \)) simp-all

**Lemma** nat-add-left-cancel-less [simp]: \( k + m < k + n \iff m < n \)

for \( k \ m \ n :: \text{nat} \)

by (induct \( k \)) simp-all
lemma add-gr-0 [iff]: \( m + n > 0 \iff m > 0 \lor n > 0 \)
  for \( m n :: \text{nat} \)
  by (auto dest: gr0-implies-Suc)

strict, in 1st argument

lemma add-less-mono1: \( i < j \implies i + k < j + k \)
  for \( i j k :: \text{nat} \)
  by (induct k) simp-all

strict, in both arguments

lemma add-less-mono:
  fixes \( i j k l :: \text{nat} \)
  assumes \( i < j \land k < l \)
  shows \( i + k < j + l \)

proof
  have \( i + k < j + k \)
    by (simp add: add-less-mono1 assms)
  also have \( \ldots < j + l \)
    using \( i < j \) by (induction j) (auto simp: assms)
  finally show \(?thesis\).
qed

lemma less-imp-Suc-add: \( m < n \implies \exists k. n = \text{Suc}(m + k) \)
proof (induct n)
  case 0
  then show \(?case\) by simp
next
case Suc
  then show \(?case\)
    by (simp add: order-le-less)
      (blast elim!: less-SucE intro: Nat.add-0-right [symmetric] add-Suc-right [symmetric])
qed

lemma le-Suc-ex: \( k \leq l \implies \exists n. l = k + n \)
  for \( k l :: \text{nat} \)
  by (auto simp: less-Suc-le[symmetric] dest: less-imp-Suc-add)

lemma less-natE:
  assumes \( m < n \)
  obtains \( q \) where \( n = \text{Suc}(m + q) \)
  using assms by (auto dest: less-imp-Suc-add intro: that)

strict, in 1st argument; proof is by induction on \( k > 0 \)

lemma mult-less-mono2:
  fixes \( i j :: \text{nat} \)
  assumes \( i < j \) and \( 0 < k \)
  shows \( k * i < k * j \)
  using \( 0 < k \)
  proof (induct k)
    case 0
then show \(\?\text{case}\) by simp

next

case (Suc \(k\))

with \((i < j)\) show \(\?\text{case}\)
  by (cases \(k\)) (simp-all add: add-less-mono)

qed

Addition is the inverse of subtraction: if \(n \leq m\) then \(n + (m - n) = m\).

lemma add-diff-inverse-nat: \(\neg m < n \Longrightarrow n + (m - n) = m\)

for \(m\ n ::\ \text{nat}\)

by (induct \(m\ n\) rule: diff-induct) simp-all

lemma nat-le-iff-add: \(m \leq n \iff (\exists k. n = m + k)\)

for \(m\ n ::\ \text{nat}\)

using nat-add-left-cancel-le [of \(m\ 0\)] by (auto dest: le-Suc-ex)

The naturals form an ordered \textit{semidom} and a \textit{dioid}.

instance \(\text{nat :: discrete-linordered-semidom}\)

proof

fix \(m\ n\ q ::\ \text{nat}\)

show \((0 < (1::nat))\)
  by simp

show \((m \leq n \Longrightarrow q + m \leq q + n)\)
  by simp

show \((m < n \Longrightarrow 0 < q \Longrightarrow q * m < q * n)\)
  by (simp add: mult-less-mono2)

show \((m \neq 0 \Longrightarrow n \neq 0 \Longrightarrow m * n \neq 0)\)
  by simp

show \((n \leq m \Longrightarrow (m - n) + n = m)\)
  by (simp add: add-diff-inverse-nat add.commute linorder-not-less)

show \((m < n \iff m + 1 \leq n)\)
  by (simp add: Suc-le-eq)

qed

instance \(\text{nat :: dioid}\)

by standard (rule nat-le-iff-add)

declare le0[simp del] — This is now \((0::\?\text{a}) \leq \?x\)

declare le-0-eq[simp del] — This is now \((\?n \leq (0::\?\text{a})) = (\?n = (0::\?\text{a}))\)

declare not-less0[simp del] — This is now \(\neg \?n < (0::\?\text{a})\)

declare not-gr0[simp del] — This is now \((\neg (0::\?\text{a}) < \?n) = (\?n = (0::\?\text{a}))\)

instance \(\text{nat :: ordered-cancel-comm-monoid-add ..}\)
instance \(\text{nat :: ordered-cancel-comm-monoid-diff ..}\)

17.4.6 \textit{min and max}

global-interpretation bot-nat-0: ordering-top \((\geq)\) \((>)\) \((0::nat)\)

by standard simp
global-interpretation max-nat: semilattice-neutr-order max {θ::nat} ⟨(≥)⟩ ⟨(>)⟩ by standard (simp add: max-def)

lemma mono-Suc: mono Suc
by (rule monoI) simp

lemma min-0L [simp]; min 0 n = 0
for n :: nat
by (rule min-absorb1) simp

lemma min-0R [simp]; min n 0 = 0
for n :: nat
by (rule min-absorb2) simp

lemma min-Suc-Suc [simp]; min (Suc m) (Suc n) = Suc (min m n)
by (simp add: mono-Suc min-of-mono)

lemma min-Suc1: min (Suc n) m = (case m of 0 ⇒ 0 | Suc m′ ⇒ Suc(min n m'))
by (simp split: nat.split)

lemma min-Suc2: min m (Suc n) = (case m of 0 ⇒ 0 | Suc m′ ⇒ Suc(min m’ n))
by (simp split: nat.split)

lemma max-0L [simp]; max 0 n = n
for n :: nat
by (fact max-nat.left-neutral)

lemma max-0R [simp]; max n 0 = n
for n :: nat
by (fact max-nat.right-neutral)

lemma max-Suc-Suc [simp]; max (Suc m) (Suc n) = Suc (max m n)
by (simp add: mono-Suc max-of-mono)

lemma max-Suc1: max (Suc n) m = (case m of 0 ⇒ Suc n | Suc m′ ⇒ Suc (max n m'))
by (simp split: nat.split)

lemma max-Suc2: max m (Suc n) = (case m of 0 ⇒ Suc n | Suc m′ ⇒ Suc (max m’ n))
by (simp split: nat.split)

lemma nat-mult-min-left: min m n * q = min (m * q) (n * q)
for m n q :: nat
by (simp add: min-def not-le)
(auto dest: mult-right-le-imp-le mult-right-less-imp-less le-less-trans)
lemma nat-mult-min-right: \( m \times \min n q = \min (m \times n) (m \times q) \)
  for \( m n q :: \text{nat} \)
  by (simp add: min-def not-le
       (auto dest: mult-left-le-imp-le mult-left-less-imp-less le-less-trans)

lemma nat-add-max-left: \( \max m n + q = \max (m + q) (n + q) \)
  for \( m n q :: \text{nat} \)
  by (simp add: max-def)

lemma nat-add-max-right: \( m + \max n q = \max (m + n) (m + q) \)
  for \( m n q :: \text{nat} \)
  by (simp add: max-def)

lemma nat-mult-max-left: \( \max m n \times q = \max (m \times q) (n \times q) \)
  for \( m n q :: \text{nat} \)
  by (simp add: max-def not-le
       (auto dest: mult-right-le-imp-le mult-right-less-imp-less le-less-trans)

lemma nat-mult-max-right: \( m \times \max n q = \max (m \times n) (m \times q) \)
  for \( m n q :: \text{nat} \)
  by (simp add: max-def not-le
       (auto dest: mult-left-le-imp-le mult-left-less-imp-less le-less-trans)

17.4.7 Additional theorems about \( (\leq) \)

Complete induction, aka course-of-values induction

instance nat :: wellorder
proof
  fix \( P \) and \( n :: \text{nat} \)
  assume step: \( (\forall m. m < n \Rightarrow P m) \Rightarrow P n \) for \( n :: \text{nat} \)
  have \( \forall q. q \leq n \Rightarrow P q \)
  proof (induct n)
    case (0 n)
    have \( P 0 \) by (rule step) auto
    with 0 show ?case by auto
  next
    case (Suc m n)
    then have \( n \leq m \lor n = \text{Suc} m \)
    by (simp add: le-Suc-eq)
    then show ?case proof
      assume \( n \leq m \)
      then show \( P n \) by (rule Suc(1))
    next
      assume \( n = \text{Suc} m \)
      show \( P n \) by (rule step) (rule Suc(1), simp add: n le-simps)
  qed
  qed
then show \( P \cdot n \) by auto
qed

lemma Least-eq-0[simp]: \( P \, 0 \implies \text{Least} \, P = 0 \)
  for \( P :: \text{nat} \Rightarrow \text{bool} \)
  by (rule Least-equality[OF - le0])

lemma Least-Suc:
  assumes \( P \, n \sim \neg P \, 0 \)
  shows \((\text{LEAST} \, n. \, P \, n) = \text{Suc} \, (\text{LEAST} \, m. \, P \, (\text{Suc} \, m))\)
proof (cases \( n \))
  case (Suc \( m \))
  show \(?thesis\)
  proof
    using assms Suc
    by (force intro: LeastI Least-le)
  qed
  qed
  qed (use assms in auto)

lemma Least-Suc2:
  \( P \, n \Rightarrow Q \, m \Rightarrow \neg P \, 0 \Rightarrow \forall k. \, P \, (\text{Suc} \, k) = Q \, k \Rightarrow \text{Least} \, P = \text{Suc} \, (\text{Least} \, Q) \)
  by (erule (1) Least-Suc [THEN ss subst]) simp

lemma ex-least-nat-le:
  fixes \( P :: \text{nat} \Rightarrow \text{bool} \)
  assumes \( P \, n \sim \neg P \, 0 \)
  shows \( \exists k \leq n. \, (\forall i<k. \, \neg P \, i) \land P \, k \)
proof (cases \( n \))
  case (Suc \( m \))
  with assms show \(?thesis\)
  by (blast intro: Least-le LeastI-ex dest: not-less-Least)
  qed (use assms in auto)

lemma ex-least-nat-less:
  fixes \( P :: \text{nat} \Rightarrow \text{bool} \)
  assumes \( P \, n \sim \neg P \, 0 \)
shows $\exists k < n. (\forall i \leq k. \neg P i) \land P (Suc k)$

proof (cases n)
  case (Suc m)
  then obtain k where $k \leq n \land \forall i < k. \neg P i\ k$
    using ex-least-nat-le [OF assms] by blast
  show ?thesis
    by (cases k) (use assms k less-eq-Suc-le in auto)
qed (use assms in auto)

lemma nat-less-induct:
  fixes P :: nat \Rightarrow bool
  assumes $\forall n. \forall m. m < n \rightarrow P m \rightarrow P n$
  shows $P n$
  using assms less-induct by blast

old style induction rules:

lemma measure-induct-rule [case-names less]:
  fixes f :: 'a \Rightarrow 'b:wellorder
  assumes step: $\forall x. (\forall y. f y < f x \rightarrow P y) \rightarrow P x$
  shows $P a$
  by (induct m \equiv f a arbitrary; a rule: less-induct) (auto intro: step)

lemma full-nat-induct:
  assumes step: $\forall n. (\forall m. Suc m \leq n \rightarrow P m) \rightarrow P n$
  shows $P n$
  by (rule less-induct) (auto intro: step simp: le-simps)

An induction rule for establishing binary relations

lemma less-Suc-induct [consumes 1]:
  assumes less: $i < j$
    and step: $\forall i. P i (Suc i)$
    and trans: $\forall i k. i < j \Rightarrow j < k \Rightarrow P i j \Rightarrow P j k \Rightarrow P i k$
  shows $P i j$
  proof
    from less obtain j where $j = Suc (i + k)$
      by (auto dest: less-imp-Suc-add)
    have $P i (Suc (i + k))$
      proof (induct k)
        case 0
        show ?case by (simp add: step)
      next
        case (Suc k)
        have $0 + i < Suc k + i$ by (rule add-less-mono1) simp
The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis. P n is true for all natural numbers if

- case “0”: given n = 0 prove P n
- case “smaller”: given n > 0 and ¬ P n prove there exists a smaller natural number m such that ¬ P m.

**lemma** infinite-descent: (∀ n. ¬ P n → ∃ m<n. ¬ P m) → P n for P :: nat ⇒ bool

— compact version without explicit base case

*by (induct n rule: less-induct) auto*

**lemma** infinite-descent0 (case-names 0 smaller):

fixes P :: nat ⇒ bool

assumes 1: ∀ x. V x = 0 ⇒ P x

and 2: ∀ x. V x > 0 ⇒ ¬ P x ⇒ ∃ y. V y < V x ∧ ¬ P y

shows P x

*proof* (rule infinite-descent)

fix n

show ¬ P n ⇒ ∃ m<n. ¬ P m

*using* assms by (cases n > 0) auto

qed

Infinite descent using a mapping to nat: P x is true for all x ∈ D if there exists a V ∈ D ⇒ nat and

- case “0”: given V x = 0 prove P x
- “smaller”: given V x > 0 and ¬ P x prove there exists a y ∈ D such that V y < V x and ¬ P y.

**corollary** infinite-descent0-measure (case-names 0 smaller):

fixes V :: 'a ⇒ nat

assumes 1: ∀ x. V x = 0 ⇒ P x

and 2: ∀ x. V x > 0 ⇒ ¬ P x ⇒ ∃ y. V y < V x ∧ ¬ P y

shows P x

*proof* –

*obtain n where* n = V x by auto

*moreover have* ∃ x. V x = n ⇒ P x

*proof* (induct n rule: infinite-descent0)
case \( 0 \)
with \( 1 \) show \( P \, x \) by \textit{auto}
next
case (smaller \( n \))
then obtain \( x \) where \(*: V \, x = n \) and \( V \, x > 0 \land \neg P \, x \) by \textit{auto}
with \( 2 \) obtain \( y \) where \( V \, y < V \, x \land \neg P \, y \) by \textit{auto}
with \( * \) obtain \( m \) where \( m = V \, y \land m < n \land \neg P \, y \) by \textit{auto}
then show \( \tilde{\text{case}} \) by \textit{auto}
qed
ultimately show \( P \, x \) by \textit{auto}
qed

Again, without explicit base case:

\textbf{lemma \textit{infinite-descent-measure}}:
fixes \( V :: 'a \Rightarrow \text{nat} \)
assumes \( \forall \, x. \neg P \, x = \Rightarrow \exists \, y. \, V \, y < V \, x \land \neg P \, y \)
shows \( P \, x \)
\textbf{proof} –
from \textit{assms} obtain \( n \) where \( n = V \, x \) by \textit{auto}
moreover have \( \forall \, x. \, V \, x = n = \Rightarrow P \, x \)
\textbf{proof} –
\quad have \( \exists \, m < V \, x. \exists \, y. \, V \, y = m \land \neg P \, y \) if \( \neg P \, x \) for \( x \)
\quad using \textit{assms} \textbf{and} \textit{that} by \textit{auto}
\quad then show \( \forall \, x. \, V \, x = n = \Rightarrow P \, x \)
\quad by (induct \( n \) rule: \textit{infinite-descent}, \textit{auto})
qed
ultimately show \( P \, x \) by \textit{auto}
qed

\textbf{A (clumsy) way of lifting \textit{<} monotonicity to \textit{\leq} monotonicity

\textbf{lemma \textit{less-mono-imp-le-mono}}:
fixes \( f :: \text{nat} \Rightarrow \text{nat} \)
and \( i \, j :: \text{nat} \)
assumes \( \forall \, i \, j :: \text{nat}. \, i < j = \Rightarrow f \, i < f \, j \)
and \( i \leq j \)
shows \( f \, i \leq f \, j \)
using \textit{assms} \textbf{by} (\textit{auto simp add: order-le-less})

\textbf{non-strict, in 1st argument

\textbf{lemma \textit{add-le-mono1}}: \( i \leq j = \Rightarrow i + k \leq j + k \)
for \( i \, j \, k :: \text{nat} \)
by (rule \textit{add-right-mono})

\textbf{non-strict, in both arguments

\textbf{lemma \textit{add-le-mono}}: \( i \leq j = \Rightarrow k \leq l = \Rightarrow i + k \leq j + l \)
for \( i \, j \, k \, l :: \text{nat} \)
by (rule \textit{add-mono})
lemma le-add2: \( n \leq m + n \)
for \( m, n : \text{nat} \)
by simp

lemma le-add1: \( n \leq n + m \)
for \( m, n : \text{nat} \)
by simp

lemma less-add-Suc1: \( i < \text{Suc} (i + m) \)
by (rule le-less-trans, rule le-add1, rule lessI)

lemma less-add-Suc2: \( i < \text{Suc} (m + i) \)
by (rule le-less-trans, rule le-add2, rule lessI)

lemma less_iff_Suc_add: \( m < n \iff (\exists k. n = \text{Suc} (m + k)) \)
by (iprover intro: less-add-Suc1 less_imp_Suc_add)

lemma trans-le-add1: \( i \leq j =\Rightarrow i \leq j + m \)
for \( i, j, m : \text{nat} \)
by (rule le-trans, assumption, rule le-add1)

lemma trans-le-add2: \( i \leq j =\Rightarrow i \leq m + j \)
for \( i, j, m : \text{nat} \)
by (rule le-trans, assumption, rule le-add2)

lemma trans-less-add1: \( i < j =\Rightarrow i < j + m \)
for \( i, j, m : \text{nat} \)
by (rule less-le-trans, assumption, rule le-add1)

lemma trans-less-add2: \( i < j =\Rightarrow i < m + j \)
for \( i, j, m : \text{nat} \)
by (rule less-le-trans, assumption, rule le-add2)

lemma add-lessD1: \( i + j < k =\Rightarrow i < k \)
for \( i, j, k : \text{nat} \)
by (rule le-less-trans \([of - i + j]\)) (simp-all add: le-add1)

lemma not-add-less1 [iff]: \( \neg i + j < i \)
for \( i, j : \text{nat} \)
by simp

lemma not-add-less2 [iff]: \( \neg j + i < i \)
for \( i, j : \text{nat} \)
by simp

lemma add-leD1: \( m + k \leq n =\Rightarrow m \leq n \)
for \( k, m, n : \text{nat} \)
by (rule order-trans \([of - m + k]\)) (simp-all add: le-add1)
THEORY “Nat”

lemma add-leD2: \(m + k \leq n \Rightarrow k \leq n\)
for \(k \ m \ n :: \text{nat}\)
by (force simp add: add.commute dest: add-leD1)

lemma add-leE: \(m + k \leq n \Rightarrow (m \leq n \Rightarrow k \leq n \Rightarrow R) \Rightarrow R\)
for \(k \ m \ n :: \text{nat}\)
by (blast dest: add-leD1 add-leD2)

needs \(\forall k\) for \(ac\text{-}simps\) to work

lemma less-add-eq-less: \(\forall k. \ k < l \Rightarrow m + l = k + n \Rightarrow m < n\)
for \(l \ m \ n :: \text{nat}\)
by (force simp del: add-Suc-right simp add: less_iff_Suc_add add-Suc-right [symmetric] ac-simps)

17.4.8 More results about difference

lemma Suc-diff-le: \(n \leq m \Rightarrow \text{Suc} m - n = \text{Suc} (m - n)\)
by (induct \(m \ n\) rule: diff-induct) simp-all

lemma diff-less-Suc: \(m - n < \text{Suc} m\)
by (induct \(m \ n\) rule: diff-induct) (auto simp: less-Suc-eq)

lemma diff-le-self [simp]: \(m - n \leq m\)
for \(m \ n :: \text{nat}\)
by (induct \(m \ n\) rule: diff-induct) (simp-all add: le-SucI)

lemma less-imp-diff-less: \(j < k \Rightarrow j - n < k\)
for \(j \ k \ n :: \text{nat}\)
by (rule le-less-trans, rule diff-le-self)

lemma diff-Suc-less [simp]: \(0 < n \Rightarrow n - \text{Suc} i < n\)
by (cases \(n\)) (auto simp add: le-simps)

lemma diff-add-assoc: \(k \leq j \Rightarrow (i + j) - k = i + (j - k)\)
for \(i \ j \ k :: \text{nat}\)
by (fact ordered-cancel-comm-monoid-diff-class.diff-add-assoc)

lemma add-diff-assoc [simp]: \(k \leq j \Rightarrow i + (j - k) = i + j - k\)
for \(i \ j \ k :: \text{nat}\)
by (fact ordered-cancel-comm-monoid-diff-class.add-diff-assoc)

lemma diff-add-assoc2: \(k \leq j \Rightarrow (j + i) - k = (j - k) + i\)
for \(i \ j \ k :: \text{nat}\)
by (fact ordered-cancel-comm-monoid-diff-class.diff-add-assoc2)

lemma add-diff-assoc2 [simp]: \(k \leq j \Rightarrow j - k + i = j + i - k\)
for \(i \ j \ k :: \text{nat}\)
by (fact ordered-cancel-comm-monoid-diff-class.add-diff-assoc2)
lemma le-imp-diff-is-add: \( i \leq j \implies (j - i = k) = (j = k + i) \)
for \( i j k :: \text{nat} \)
by auto

lemma diff-is-0-eq [simp]: \( m - n = 0 \iff m \leq n \)
for \( m n :: \text{nat} \)
by (induct \( m n \) rule: diff-induct) simp-all

lemma diff-is-0-eq' [simp]: \( m \leq n \implies m - n = 0 \)
for \( m n :: \text{nat} \)
by (rule iffD2, rule diff-is-0-eq)

lemma zero-less-diff [simp]: \( 0 < n - m \iff m < n \)
for \( m n :: \text{nat} \)
by (induct \( m n \) rule: diff-induct) simp-all

lemma less-imp-add-positive:
assumes \( i < j \)
shows \( \exists k :: \text{nat}.\ 0 < k \land i + k = j \)
proof
from assms show \( 0 < j - i \land i + (j - i) = j \)
  by (simp add: order-less-imp-le)
qed

a nice rewrite for bounded subtraction

lemma nat-minus-add-max: \( n - m + m = \max n m \)
for \( m n :: \text{nat} \)
by (simp add: max-def not-le order-less-imp-le)

lemma nat-diff-split: \( P (a - b) \iff (a < b \implies P 0) \land (\forall d.\ a = b + d \implies P d) \)
for \( a b :: \text{nat} \)
— elimination of \( - \) on \( \text{nat} \)
by (cases \( a < b \) (auto simp add: not-less le-less dest: add-eq-selz-zero [OF sym]))

lemma nat-diff-split-asm: \( P (a - b) \iff \neg (a < b \land \neg P 0 \lor (\exists d.\ a = b + d \land \neg P d)) \)
for \( a b :: \text{nat} \)
— elimination of \( - \) on \( \text{nat} \) in assumptions
by (auto split: nat-diff-split)

lemma Suc-pred': \( 0 < n \implies n = \text{Suc}(n - 1) \)
by simp

lemma add-eq-if: \( m + n = (\text{if } m = 0 \text{ then } n \text{ else } \text{Suc } ((m - 1) + n)) \)
unfolding One-nat-def by (cases \( m \)) simp-all

lemma mult-eq-if: \( m \ast n = (\text{if } m = 0 \text{ then } 0 \text{ else } n + ((m - 1) \ast n)) \)
for \( m n :: \text{nat} \)
by (cases \( m \)) simp-all
lemma Suc-diff-eq-diff-pred: $0 < n \implies \text{Suc } m - n = m - (n - 1)$
by (cases n) simp-all

lemma diff-Suc-eq-diff-pred: $m - \text{Suc } n = (m - 1) - n$
by (cases m) simp-all

lemma Let-Suc [simp]: $\text{Let } (\text{Suc } n) f \equiv f (\text{Suc } n)$
by (fact Let-def)

17.4.9 Monotonicity of multiplication

lemma mult-le-mono1: $i \leq j \implies i \times k \leq j \times k$
for $i j k :: \text{nat}$
by (simp add: mult-right-mono)

lemma mult-le-mono2: $i \leq j \implies k \times i \leq k \times j$
for $i j k :: \text{nat}$
by (simp add: mult-left-mono)

≤ monotonicity, BOTH arguments

lemma mult-le-mono: $i \leq j \implies k \leq l \implies i \times k \leq j \times l$
for $i j k l :: \text{nat}$
by (simp add: mult-mono)

lemma mult-less-mono1: $i < j \implies 0 < k \implies i \times k < j \times k$
for $i j k :: \text{nat}$
by (simp add: mult-strict-right-mono)

Diagrams from the standard zero-less-mult-iff in that there are no negative numbers.

lemma nat-0-less-mult-iff [simp]: $0 < m \times n \iff 0 < m \land 0 < n$
for $m n :: \text{nat}$
proof (induct m)
case 0
then show ?case by simp
next
case (Suc m)
then show ?case by (cases n) simp-all
qed

lemma one-le-mult-iff [simp]: $\text{Suc } 0 \leq m \times n \iff \text{Suc } 0 \leq m \land \text{Suc } 0 \leq n$
proof (induct m)
case 0
then show ?case by simp
next
case (Suc m)
then show ?case by (cases n) simp-all
qed
lemma mult-less-cancel2 [simp]: \( m \cdot k < n \cdot k \iff 0 < k \land m < n \)
for \( k \, m \, n :: \text{nat} \)
proof (intro iffI conjI)
  assume \( m: m \cdot k < n \cdot k \)
  then show \( 0 < k \)
    by (cases k) auto
show \( m < n \)
proof (cases k)
  case \( \theta \)
  then show ?thesis
    using \( m \) by auto
next
  case \( \text{Suc } k' \)
  then show ?thesis
    using \( m \) by (simp flip: linorder-not-le) (blast intro: add-mono mult-le-mono1)
qed
next
assume \( 0 < k \land m < n \)
then show \( m \cdot k < n \cdot k \)
  by (blast intro: mult-less-mono1)
qed

lemma mult-less-cancel1 [simp]: \( k \cdot m < k \cdot n \iff 0 < k \land m < n \)
for \( k \, m \, n :: \text{nat} \)
by (simp add: mult.commute [of k])

lemma mult-le-cancel1 [simp]: \( k \cdot m \leq k \cdot n \iff (0 < k \rightarrow m \leq n) \)
for \( k \, m \, n :: \text{nat} \)
by (simp add: linorder-not-less [symmetric], auto)

lemma mult-le-cancel2 [simp]: \( m \cdot k \leq n \cdot k \iff (0 < k \rightarrow m \leq n) \)
for \( k \, m \, n :: \text{nat} \)
by (simp add: linorder-not-less [symmetric], auto)

lemma Suc-mult-less-cancel1: \( \text{Suc } k \cdot m < \text{Suc } k \cdot n \iff m < n \)
by (subst mult-less-cancel1) simp

lemma Suc-mult-le-cancel1: \( \text{Suc } k \cdot m \leq \text{Suc } k \cdot n \iff m \leq n \)
by (subst mult-le-cancel1) simp

lemma le-square: \( m \leq m \cdot m \)
for \( m :: \text{nat} \)
by (cases m) (auto intro: le-add1)

lemma le-cube: \( m \leq m \cdot (m \cdot m) \)
for \( m :: \text{nat} \)
by (cases m) (auto intro: le-add1)
Lemma for $\text{gcd}$

lemma mult-eq-self-implies-0:
  fixes $m\ n :: \text{nat}$
  assumes $m = m \ast n$ shows $n = 1 \lor m = 0$
proof (rule disjCI)
  assume $m \neq 0$
  show $n = 1$
proof (cases $n \ 1 :: \text{nat}$ rule: linorder-cases)
  case greater
  show ?thesis
    using assms mult-less-mono2 [OF greater, of $m$] by auto
qed (use assms $\langle m \neq 0 \rangle$ in auto)
qed

lemma mono-times-nat:
  fixes $n :: \text{nat}$
  assumes $n > 0$
  shows mono $(\times \ n)$
proof
  fix $m\ q :: \text{nat}$
  assume $m \leq q$
  with assms show $n \ast m \leq n \ast q$ by simp
qed

The lattice order on $\text{nat}$.

instantiation $\text{nat} :: \text{distrib-lattice}$
begin

definition $(\text{inf} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}) = \text{min}$
definition $(\text{sup} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}) = \text{max}$
instance
  by intro-classes
    (auto simp add: inf-nat-def sup-nat-def max-def not-le min-def
     intro: order-less-imp-le antisym elim!: order-trans order-less-trans)
end

17.5 Natural operation of natural numbers on functions

We use the same logical constant for the power operations on functions and
relations, in order to share the same syntax.

consts $\text{compow} :: \text{nat} \Rightarrow 'a \Rightarrow 'a$

abbreviation $\text{compower} :: 'a \Rightarrow \text{nat} \Rightarrow 'a$ $(\text{infixr} \ ^{\sim} 80)$
  where $f \ ^{\sim} n \equiv \text{compow} \ n \ f$
notation (latex output)
  \text{compower} ((\sim) [1000] 1000)

f \sim n = f \circ \ldots \circ f, \text{ the } n\text{-fold composition of } f

overloading
\text{funpow} \equiv \text{compow} :: \text{nat} \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)
begin

primrec \text{funpow} :: \text{nat} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a
  where
  \text{funpow} 0 f = \text{id}
  | \text{funpow} (Suc n) f = f \circ \text{funpow} n f
end

lemma \text{funpow-0} [simp]: (f \sim 0) x = x
  by simp

lemma \text{funpow-Suc-right}: f \sim \text{Suc } n = f \sim n \circ f
proof (induct n)
  case 0
  then show ?case by simp
next
  fix n
  assume \text{funpow-Suc-right}: f \sim \text{Suc } n = f \sim n \circ f
  then show f \sim (\text{Suc } n) = f \sim n \circ f
    by (simp add: o-assoc)
qed

lemmas \text{funpow-simps-right} = funpow.simps(1) \text{funpow-Suc-right}

For code generation.
context
begin

qualified definition \text{funpow} :: \text{nat} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a
  where \text{funpow-code-def} [code-abbrev]: \text{funpow} = \text{compow}

lemma [code]:
  \text{funpow} (Suc n) f = f \circ \text{funpow} n f
  \text{funpow} 0 f = \text{id}
  by (simp-all add: funpow-code-def)
end

lemma \text{funpow-add}: f \sim (m + n) = f \sim m \circ f \sim n
  by (induct m) simp-all

lemma \text{funpow-mult}: (f \sim m) \sim n = f \sim (m \ast n)
for f :: 'a ⇒ 'a
by (induct n) simp-all add: funpow-add

lemma funpow-swap1: f ((f ^^ n) x) = (f ^^ n) (f x)
proof -
  have f ((f ^^ n) x) = (f ^^ (n + 1)) x by simp
  also have ... = (f ^^ n o f ^^ 1) x by (simp only: funpow-add)
  also have ... = (f ^^ n) (f x) by simp
  finally show ?thesis .
qed

lemma comp-funpow: comp f ^^ n = comp (f ^^ n)
  for f :: 'a ⇒ 'a
  by (induct n) simp-all

lemma Suc-funpow[simp]: Suc ^^ n = ((+) n)
by (induct n) simp-all

lemma id-funpow[simp]: id ^^ n = id
by (induct n) simp-all

lemma funpow-mono: mono f \implies A \leq B \implies (f ^^ n) A \leq (f ^^ n) B
  for f :: 'a ⇒ ('a::order)
by (induct n arbitrary: A B)
  (auto simp del: funpow.simps(2) simp add: funpow-Suc-right mono-def)

lemma funpow-mono2:
  assumes mono f
  and i ≤ j
  and x ≤ y
  and x ≤ f x
  shows (f ^^ i) x ≤ (f ^^ j) y
using assms(2,3)
proof (induct j arbitrary: y)
  case 0
  then show ?case by simp
next
  case (Suc j)
  show ?case
proof (cases i = Suc j)
    case True
    with assms(1) Suc show ?thesis
      by (simp del: funpow.simps funpow-simps-right monoD funpow-mono)
  next
    case False
    with assms(1,4) Suc show ?thesis
      by (simp del: funpow.simps funpow-simps-right le-eq-less-or-eq less-Suc-eq-le)
  qed
  qed
lemma inj-fn[simp]:
  fixes $f : 'a 	o 'a$
  assumes inj $f$
  shows inj $(f\sim n)$
proof (induction $n$
  case Suc thus ?case using inj-compose[OF assms Suc.IH] by (simp del: comp-apply)
qed simp

lemma surj-fn[simp]:
  fixes $f : 'a 	o 'a$
  assumes surj $f$
  shows surj $(f\sim n)$
proof (induction $n$
qed simp

lemma bij-fn[simp]:
  fixes $f : 'a 	o 'a$
  assumes bij $f$
  shows bij $(f\sim n)$
by (rule bijI[OF inj-fn[OF bij-is-inj[OF assms]] surj-fn[OF bij-is-surj[OF assms]]])

lemma bij-betw-funpow:
  assumes bij-betw $f$ $S$ $S$
  shows bij-betw $(f \sim n)$ $S$ $S$
proof (induct $n$
  case 0 then show ?case by (auto simp: id-def[symmetric])
next
case (Suc $n$)
  then show ?case unfolding funpow.simps using assms by (rule bij-betw-trans)
qed

17.6 Kleene iteration

lemma Kleene-iter-lpfp:
  fixes $f : 'a::order-bot 	o 'a$
  assumes mono $f$
  and $f \bot \leq p$
  shows $(f \sim n) \bot \leq p$
proof (induct $k$
  case 0
  show ?case by simp
next
case Suc
  show ?case using monoD[OF assms(1) Suc] assms(2) by simp
qed
lemma lfp-Kleene-iter:
  assumes mono f
  and (f ^^ Suc k) bot = (f ^^ k) bot
  shows lfp f = (f ^^ k) bot
proof (rule antisym)
  show lfp f ≤ (f ^^ k) bot
    proof (rule lfp-lowerbound)
      have f (lfp (f ^^ Suc n)) = lfp (λx. f ((f ^^ n) x))
        unfolding funpow-Suc-right by (simp add: lfp-rolling f mono-pow comp-def)
      then show f (lfp (f ^^ Suc n)) ≤ lfp (f ^^ Suc n)
        by (simp add: comp-def)
    qed
  have (f ^^ n) (lfp f) = lfp f for n
    by (induct n) (auto simp: mono-def)
  then show lfp (f ^^Suc n) ≤ lfp f
    by (intro lfp-lowerbound) (simp del: funpow.simps)
qed

lemma mono-pow: mono f → mono (f ^^ n)
  for f :: 'a ⇒ 'a:complete-lattice
  by (induct n) (auto simp: mono-def)

lemma lfp-funpow:
  assumes f: mono f
  shows lfp (f ^^ Suc n) = lfp f
proof (rule antisym)
  show lfp f ≤ lfp (f ^^ Suc n)
    proof (rule lfp-lowerbound)
      have f (lfp (f ^^ Suc n)) = lfp (λx. f ((f ^^ n) x))
        unfolding funpow-Suc-right by (simp add: lfp-rolling f mono-pow comp-def)
      then show f (lfp (f ^^ Suc n)) ≤ lfp (f ^^ Suc n)
        by (simp add: comp-def)
    qed
  have (f ^^ n) (lfp f) = lfp f for n
    by (induct n) (auto intro: f lfp-fixpoint)
  then show lfp (f ^^ Suc n) ≤ lfp f
    by (intro lfp-lowerbound) (simp del: funpow.simps)
qed

lemma gfp-funpow:
  assumes f: mono f
  shows gfp (f ^^ Suc n) = gfp f
proof (rule antisym)
  show gfp f ≥ gfp (f ^^ Suc n)
    proof (rule gfp-upperbound)
      have f (gfp (f ^^ Suc n)) = gfp (λx. f ((f ^^ n) x))
        unfolding funpow-Suc-right by (simp add: gfp-rolling f mono-pow comp-def)
      then show f (gfp (f ^^ Suc n)) ≥ gfp (f ^^ Suc n)
        by (simp add: comp-def)
    qed
  have (f ^^ n) (gfp f) = gfp f for n
    by (induct n) (auto intro: f gfp-fixpoint)
  then show gfp (f ^^ Suc n) ≥ gfp f

by (intro gfp-upperbound) (simp del: funpow.simps)
qed

lemma Kleene-iter-gpfp:
  fixes f :: 'a::order-top ⇒ 'a
  assumes mono f
  and p ≤ f p
  shows p ≤ (f ^^ k) top
proof (induct k)
  case 0
  show ?case by simp
next
  case Suc
  show ?case
  using monoD [OF assms (1)]
  by (simp)
qed

lemma gfp-Kleene-iter:
  assumes mono f
  and (f ^^ Suc k) top = (f ^^ k) top
  shows gfp f = (f ^^ k) top
proof (rule antisym)
  have ?rhs ≤ f ?rhs
  using assms (2) by simp
  then show ?rhs ≤ ?lhs
  by (rule gfp-upperbound)
  show ?lhs ≤ ?rhs
  using Kleene-iter-gpfp [OF assms (1)]
  gfp-unfold [OF assms (1)] by simp
qed

17.7 Embedding of the naturals into any \textit{semiring-1}: of-nat

context semiring-1
begin

definition of-nat :: nat ⇒ 'a
where of-nat n = (plus 1 ^^ n) 0

lemma of-nat-simps [simp]:
  shows of-nat-0: of-nat 0 = 0
  and of-nat-Suc: of-nat (Suc m) = 1 + of-nat m
  by (simp-all add: of-nat-def)

lemma of-nat-1 [simp]: of-nat 1 = 1
  by (simp add: of-nat-def)

lemma of-nat-add [simp]: of-nat (m + n) = of-nat m + of-nat n
  by (induct m) (simp-all add: ac-simps)
lemma of-nat-mult [simp]: of-nat \( (m * n) = \) of-nat \( m + \) of-nat \( n \)
  by (induct \( m \)) (simp-all add: ac-simps distrib-right)

lemma mult-of-nat-commute: of-nat \( x * y = y * \) of-nat \( x \)
  by (induct \( x \)) (simp-all add: algebra-simps)

primrec of-nat-aux :: ('a ⇒ 'a) ⇒ nat ⇒ 'a
where
  of-nat-aux inc 0 i = i
| of-nat-aux inc (Suc n) i = of-nat-aux inc n (inc i) — tail recursive

lemma of-nat-code: of-nat \( n = \) of-nat-aux \( \lambda i. i + 1 ) \) \( n 0 \)
proof (induct \( n \))
  case 0
  then show \( \) case by simp
next
  case (Suc n)
  have \( \forall i. \) of-nat-aux \( \lambda i. i + 1 ) \) \( n (i + 1) = \) of-nat-aux \( \lambda i. i + 1 ) \) \( n i + 1 \)
  by (induct \( n \)) simp-all
  from this \( \) of 0 \( \) have of-nat-aux \( \lambda i. i + 1 ) \) \( n 1 = \) of-nat-aux \( \lambda i. i + 1 ) \) \( n 0 + 1 \)
  by simp
  with Suc show ?case
  by (simp add: add.commute)
qed

lemma of-nat-of-bool [simp]:
  of-nat \( (\) of-bool \( P \) = \) of-bool \( P \)
  by auto
end

declare of-nat-code [code]

context semiring-1-cancel
begin

lemma of-nat-diff:
  \( \) of-nat \( (m - n) = \) of-nat \( m - \) of-nat \( n \) if \( n \leq m \)
proof -
  from that obtain q where \( m = n + q \)
  by (blast dest: le-Suc-ex)
  then show \( \) thesis
  by simp
qed

end

Class for unital semirings with characteristic zero. Includes non-ordered
rings like the complex numbers.

```plaintext
class semiring-char-0 = semiring-1 +
  assumes inj-of-nat: inj of-nat
begin

  lemma of-nat-eq-iff [simp]: of-nat m = of-nat n ↔ m = n
    by (auto intro: inj-of-nat injD)

  Special cases where either operand is zero
  lemma of-nat-0-eq-iff [simp]: 0 = of-nat n ↔ 0 = n
    by (fact of-nat-eq-iff [of 0 n, unfolded of-nat-0])

  lemma of-nat-0-iff [simp]: of-nat m = 0 ↔ m = 0
    by (fact of-nat-eq-iff [of m 0, unfolded of-nat-0])

  lemma of-nat-1-eq-iff [simp]: 1 = of-nat n ↔ n=1
    using of-nat-eq-iff by fastforce

  lemma of-nat-eq-1-iff [simp]: of-nat n = 1 ↔ n=1
    using of-nat-eq-iff by fastforce

  lemma of-nat-neq-0 [simp]: of-nat (Suc n) ≠ 0
    unfolding of-nat-eq-0-iff by simp

  lemma of-nat-0-neq [simp]: 0 ≠ of-nat (Suc n)
    unfolding of-nat-0-eq-iff by simp

end
```

```plaintext
class ring-char-0 = ring-1 + semiring-char-0
```

```plaintext
context linordered-nonzero-semiring
begin

  lemma of-nat-0-le-iff [simp]: 0 ≤ of-nat n
    by (induct n) simp-all

  lemma of-nat-less-0-iff [simp]: ¬ of-nat m < 0
    by (simp add: not-less)

  lemma of-nat-mono [simp]: i ≤ j → of-nat i ≤ of-nat j
    by (auto simp: le-iff-add intro!: add-increasing2)

  lemma of-nat-less-iff [simp]: of-nat m < of-nat n ↔ m < n
    proof (induct m n rule: diff-induct)
      case (1 m) then show ?case
        by auto
    next
      case (2 n) then show ?case
```
by (simp add: add-pos-nonneg)
next
case (3 m n)
then show ?case
  by (auto simp: add-commute [of 1] add-mono1 not-less add-right-mono leD)
qed

lemma of-nat-le-iff [simp]: of-nat m ≤ of-nat n ←→ m ≤ n
  by (simp add: not-less [symmetric] linorder-not-less [symmetric])

lemma less-imp-of-nat-less: m < n ⇒ of-nat m < of-nat n
  by simp

lemma of-nat-less-imp-less: of-nat m < of-nat n ⇒ m < n
  by simp

Every linordered-nonzero-semiring has characteristic zero.
subclass semiring-char-0
  by standard (auto intro: injI simp add: order.eq_iff)

Special cases where either operand is zero

lemma (of-nat-le-0-iff [simp]: of-nat m ≤ 0 ←→ m = 0
  by (rule of-nat-le-0-iff [of 0, simplified])

lemma (of-nat-0-less-iff [simp]: 0 < of-nat n ←→ 0 < n
  by (rule of-nat-less-iff [of 0, simplified])

end

class linordered-nonzero-semiring
begin

lemma of-nat-max: of-nat (max x y) = max (of-nat x) (of-nat y)
  by (auto simp: max-def ord-class.max-def)

lemma of-nat-min: of-nat (min x y) = min (of-nat x) (of-nat y)
  by (auto simp: min-def ord-class.min-def)

end

class linordered-semidom
begin

subclass linordered-nonzero-semiring ..

subclass semiring-char-0 ..

end
THEORY "Nat"

context linordered-idom
begin

lemma abs-of-nat [simp]:
|of-nat n| = of-nat n
by (simp add: abs-if)

lemma sgn-of-nat [simp]:
sgn (of-nat n) = of-bool (n > 0)
by simp

end

lemma of-nat-id [simp]: of-nat n = n
by (induct n) simp-all

lemma of-nat-eq-id [simp]: of-nat = id
by (auto simp add: fun-eq-iff)

17.8 The set of natural numbers

context semiring-1
begin

definition Nats :: 'a set (N)
  where N = range of-nat

lemma of-nat-in-Nats [simp]: of-nat n ∈ N
  by (simp add: Nats-def)

lemma Nats-0 [simp]: 0 ∈ N
  using of-nat-0 [symmetric] unfolding Nats-def
  by (rule range-eqI)

lemma Nats-1 [simp]: 1 ∈ N
  using of-nat-1 [symmetric] unfolding Nats-def
  by (rule range-eqI)

lemma Nats-add [simp]: a ∈ N ⇒ b ∈ N ⇒ a + b ∈ N
  unfolding Nats-def using of-nat-add [symmetric]
  by (blast intro: range-eqI)

lemma Nats-mult [simp]: a ∈ N ⇒ b ∈ N ⇒ a * b ∈ N
  unfolding Nats-def using of-nat-mult [symmetric]
  by (blast intro: range-eqI)

lemma Nats-cases [cases set: Nats]:
  assumes x ∈ N
  obtains (of-nat) n where x = of-nat n
unfolding Nats-def

proof –
from \( \langle x \in \mathbb{N} \rangle \) have \( x \in \text{range of-nat} \) unfolding Nats-def .
then obtain \( n \) where \( x = \text{of-nat} \ n \).
then show thesis ..
qed

lemma Nats-induct [case-names of-nat, induct set: Nats]: \( x \in \mathbb{N} \Rightarrow (\forall n. \ P \ (\text{of-nat} \ n)) \Rightarrow P \ x \)
  by (rule Nats-cases) auto

lemma Nats-nonempty [simp]: \( \mathbb{N} \neq \{\} \)
  unfolding Nats-def by auto

end

lemma Nats-diff [simp]:
  fixes \( \cdot a::\cdot'a::\cdot\text{linordered-idom} \)
  assumes \( a \in \mathbb{N} \; b \in \mathbb{N} \; b \leq a \) shows \( a - b \in \mathbb{N} \)
proof –
  obtain \( i \) where \( i: a = \text{of-nat} \ i \)
  using Nats-cases assms by blast
  obtain \( j \) where \( j: b = \text{of-nat} \ j \)
  using Nats-cases assms by blast
  have \( j \leq i \)
    using \( \langle b \leq a \rangle \; i \; j \; \text{of-nat-le-iff} \) by blast
  then have \( \ast: \text{of-nat} \ i - \text{of-nat} \ j = (\text{of-nat} \ (i - j)) :: \cdot'a \)
    by (simp add: of-nat-diff)
  then show \( \ast\text{thesis} \)
    by (simp add: \( \ast \) i j)
qed

17.9 Further arithmetic facts concerning the natural numbers

lemma subst-equals:
  assumes \( t = s \) and \( u = t \)
  shows \( u = s \)
  using assms(2,1) by (rule trans)

locale nat-arith
begin

lemma add1: \( (A::\cdot'a::\text{comm-monoid-add}) \equiv k + a \Rightarrow A + b \equiv k + (a + b) \)
  by (simp only: ac-simps)

lemma add2: \( (B::\cdot'a::\text{comm-monoid-add}) \equiv k + b \Rightarrow a + B \equiv k + (a + b) \)
  by (simp only: ac-simps)
THEORY "Nat"

lemma suc1: \( A == k + a \implies \text{Suc} A \equiv k + \text{Suc} a \)
  by (simp only: add-Suc-right)

lemma rule0: (\( a ::'a::\text{comm-monoid-add} \)) \equiv a + 0
  by (simp only: add-0-right)

end

ML-file 〈Tools/nat-arith.ML〉

simproc-setup nateq-cancel-sums
  \((\( l ::\text{nat} \)) + m = n \mid (\( l ::\text{nat} \)) = m + n \mid \text{Suc} m = n \mid m = \text{Suc} n \) =
  \( K \text{ (try o Nat-Arith.cancel-eq-conv} \))

simproc-setup natless-cancel-sums
  \((\( l ::\text{nat} \)) + m < n \mid (\( l ::\text{nat} \)) < m + n \mid \text{Suc} m < n \mid m < \text{Suc} n \) =
  \( K \text{ (try o Nat-Arith.cancel-less-conv} \))

simproc-setup natle-cancel-sums
  \((\( l ::\text{nat} \)) + m \leq n \mid (\( l ::\text{nat} \)) \leq m + n \mid \text{Suc} m \leq n \mid m \leq \text{Suc} n \) =
  \( K \text{ (try o Nat-Arith.cancel-le-conv} \))

simproc-setup natdiff-cancel-sums
  \((\( l ::\text{nat} \)) + m - n \mid (\( l ::\text{nat} \)) - (m + n) \mid \text{Suc} m - n \mid m - \text{Suc} n \) =
  \( K \text{ (try o Nat-Arith.cancel-diff-conv} \))

context order
begin

lemma lift-Suc-mono-le:
  assumes mono: \( \forall n. f n \leq f (\text{Suc} n) \)
  and \( n \leq n' \)
  shows \( f n \leq f n' \)
proof (cases \( n < n' \))
  case True
  then show \( \text{thesis} \)
    by (induct \( n \) \( n' \) rule: less-Suc-induct) (auto intro: mono)
next
  case False
  with \( \langle n \leq n' \rangle \) show \( \text{thesis} \) by auto
qed

lemma lift-Suc-antimono-le:
  assumes mono: \( \forall n. f n \geq f (\text{Suc} n) \)
  and \( n \leq n' \)
  shows \( f n \geq f n' \)
proof (cases \( n < n' \))
  case True
  then show \( \text{thesis} \)
by (induct \( n \) \( n' \) rule: less-Suc-induct) (auto intro: mono)

next

  case False

with \( \alpha \leq n' \) show \(?thesis \) by auto

qed

lemma lift-Suc-mono-less:
  assumes mono: \( \forall n. f n < f (Suc n) \)
            and \( n < n' \)
  shows \( f n < f n' \)
  using \( \alpha < n' \) by (induct \( n \) \( n' \) rule: less-Suc-induct) (auto intro: mono)

lemma lift-Suc-mono-less-iff: \( \forall n. f n < f (Suc n) \) \( \iff \) \( f n < f m \iff n < m \)
by (blast intro: less-asm' lift-Suc-mono-less \[ of f \]
dest: linorder-not-less\[THEN iffD1 \] le-eq-less-or-eq \[THEN iffD1 \])

end

lemma mono-iff-le-Suc: \( mono f \iff (\forall n. f n \leq f (Suc n)) \)
unfolding mono-def by (auto intro: lift-Suc-mono-le \[ of f \])

lemma antimono-iff-le-Suc: \( antimono f \iff (\forall n. f (Suc n) \leq f n) \)
unfolding antimono-def by (auto intro: lift-Suc-antimono-le \[ of f \])

lemma strict-mono-Suc-iff: \( strict-mono f \iff (\forall n. f n < f (Suc n)) \)
proof (intro iffI strict-monoI)
  assume \(*\): \( \forall n. f n < f (Suc n) \)
  fix \( m n :: nat \) assume \( m < n \)
  thus \( f m < f n \)
    by (induction rule: less-Suc-induct) (use \* in auto)
qed (auto simp: strict-mono-def)

lemma strict-mono-add: \( strict-mono (\lambda n ::'a::linordered-semidom. n + k) \)
by (auto simp: strict-mono-def)

lemma mono-nat-linear-lb:
  fixes \( f :: nat \Rightarrow nat \)
  assumes \( \forall m n. m < n \Rightarrow f m < f n \)
  shows \( f m + k \leq f (m + k) \)
proof (induct \( k \))
  case 0
  then show \(?case \) by simp

next

  case (Suc \( k \))
  then have \( Suc (f m + k) \leq Suc (f (m + k)) \) by simp
also from \( \text{assms} \) \[ of m + k Suc (m + k) \] have \( Suc (f (m + k)) \leq f (Suc (m + k)) \)
  by (simp add: Suc-le-eq)
finally show \(?case \) by simp
qed

Subtraction laws, mostly by Clemens Ballarin

lemma \textit{diff-less-mono}:
  \begin{itemize}
    \item \textbf{fixes} $a \ b \ c :: \text{nat}$
    \item \textbf{assumes} $a < b$ \textbf{and} $c \leq a$
    \item \textbf{shows} $a - c < b - c$
  \end{itemize}
proof –
  \begin{itemize}
    \item \textbf{from} \textit{assms} \textbf{obtain} $d \ e$ \textbf{where} $b = c + (d + e)$ \textbf{and} $a = c + e$ \textbf{and} $d > 0$
    \item \textbf{by} \textbf{(auto dest!: le-Suc-ex less-imp-Suc-add simp add: ac-simps)}
    \item \textbf{then} \textbf{show} ?thesis \textbf{by} simp
  \end{itemize}
qed

lemma \textit{less-diff-conv}:
  \begin{itemize}
    \item $i < j - k \longleftrightarrow i + k < j$
    \item \textbf{for} $i \ j \ k :: \text{nat}$
    \item \textbf{by} \textbf{(cases $k \leq j$)} \textbf{(auto simp add: not-le dest: less-imp-Suc-add le-Suc-ex)}
  \end{itemize}

lemma \textit{less-diff-conv2}:
  \begin{itemize}
    \item $k \leq j \Longrightarrow j - k < i \longleftrightarrow j < i + k$
    \item \textbf{for} $j \ i \ k :: \text{nat}$
    \item \textbf{by} \textbf{(auto dest: le-Suc-ex)}
  \end{itemize}

lemma \textit{le-diff-conv}:
  \begin{itemize}
    \item $j - k \leq i \longleftrightarrow j \leq i + k$
    \item \textbf{for} $j \ i \ k :: \text{nat}$
    \item \textbf{by} \textbf{(cases $k \leq j$)} \textbf{(auto simp add: not-le dest: less-imp-Suc-add le-Suc-ex)}
  \end{itemize}

lemma \textit{diff-diff-cancel} [simp]:
  \begin{itemize}
    \item $i \leq n \Longrightarrow n - (n - i) = i$
    \item \textbf{for} $i \ n :: \text{nat}$
    \item \textbf{by} \textbf{(auto dest: le-Suc-ex)}
  \end{itemize}

lemma \textit{diff-less} [simp]:
  \begin{itemize}
    \item $0 < n \Longrightarrow 0 < m \Longrightarrow m - n < m$
    \item \textbf{for} $i \ n :: \text{nat}$
    \item \textbf{by} \textbf{(auto dest: less-imp-Suc-add)}
  \end{itemize}

Simplification of relational expressions involving subtraction

lemma \textit{diff-diff-eq}:
  \begin{itemize}
    \item $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k - (n - k) = m - n$
    \item \textbf{for} $m \ n \ k :: \text{nat}$
    \item \textbf{by} \textbf{(auto dest!: le-Suc-ex)}
  \end{itemize}

hide-fact \textbf{(open)} \textit{diff-diff-eq}

lemma \textit{eq-diff-iff}:
  \begin{itemize}
    \item $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k = n - k \longleftrightarrow m = n$
    \item \textbf{for} $m \ n \ k :: \text{nat}$
    \item \textbf{by} \textbf{(auto dest: le-Suc-ex)}
  \end{itemize}

lemma \textit{less-diff-iff}:
  \begin{itemize}
    \item $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k < n - k \longleftrightarrow m < n$
    \item \textbf{for} $m \ n \ k :: \text{nat}$
    \item \textbf{by} \textbf{(auto dest!: le-Suc-ex)}
  \end{itemize}

lemma \textit{le-diff-iff}:
  \begin{itemize}
    \item $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k \leq n - k \longleftrightarrow m \leq n$
  \end{itemize}
for m n k :: nat
by (auto dest: le-Suc-ex)

lemma le-diff-iff': a ≤ c ⇒ b ≤ c ⇒ c - a ≤ b ↔ b ≤ a
for a b c :: nat
by (force dest: le-Suc-ex)

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma diff-le-mono: m ≤ n ⇒ m - l ≤ n - l
for m n l :: nat
by (auto dest: less-imp-le less-imp-Suc-add split: nat-split)

lemma diff-le-mono2: m ≤ n ⇒ l - n ≤ l - m
for m n l :: nat
by (auto dest: less-imp-le Suc-ex less-imp-Suc-add less-le-trans split: nat-split)

lemma diff-less-mono2: m < n ⇒ m < l ⇒ l - n < l - m
for m n l :: nat
by (auto dest: less-imp-Suc-add split: nat-diff-split)

lemma diffs0-imp-equal: m - n = 0 ⇒ n - m = 0 ⇒ m = n
for m n :: nat
by (simp split: nat-diff-split)

lemma min-diff: min (m - i) (n - i) = min m n - i
for m n i :: nat
by (cases m n rule: le-cases)
(auto simp add: not-le min.absorb1 min.absorb2 min.absorb-iff1 [symmetric] diff-le-mono)

lemma inj-on-diff-nat:
fixes k :: nat
assumes ∀ n. n ∈ N ⇒ k ≤ n
shows inj-on (λn. n - k) N
proof (rule inj-onI)
fix x y
assume a: x ∈ N y ∈ N x - k = y - k
with assms have x - k + k = y - k + k by auto
with a assms show x = y by (auto simp add: eq-diff-iff)
qed

Rewriting to pull differences out
lemma diff-diff-right [simp]: k ≤ j ⇒ i - (j - k) = i + k - j
for i j k :: nat
by (fact diff-diff-right)

lemma diff-Suc-diff-eq1 [simp]:
assumes k ≤ j
shows i - Suc (j - k) = i + k - Suc j
proof –
  from assms have 
    \*: Suc (j - k) = Suc j - k
    by (simp add: Suc-diff-le)
  from assms have 
    \[Suc j \leq Suc j\]
    by (rule order-trans) simp
with diff-diff-right [of k Suc j i] * show \?thesis
  by simp
qed

lemma diff-Suc-diff-eq2 [simp]:
  assumes \[k \leq j\]
  shows \[Suc (j - k) - i = Suc j - (k + i)\]
proof –
  from assms obtain \[n\] where \[j = k + n\]
    by (auto dest: le-Suc-ex)
moreover have \[Suc n - i = (k + Suc n) - (k + i)\]
  using add-diff-cancel-left [of k Suc n i] by simp
ultimately show \?thesis by simp
qed

lemma Suc-diff-Suc:
  assumes \[n < m\]
  shows \[Suc (m - Suc n) = m - n\]
proof –
  from assms obtain \[q\] where \[m = n + Suc q\]
    by (auto dest: less-imp-Suc-add)
moreover define \[r\] where \[r = Suc q\]
ultimately have \[Suc (m - Suc n) = r \text{ and } m = n + r\]
  by simp-all
then show \?thesis by simp
qed

lemma one-less-mult: \[Suc 0 < n \Rightarrow Suc 0 < m \Rightarrow Suc 0 < m * n\]
  using less-1-mult [of n m] by (simp add: ac-simps)

lemma n-less-m-mult-n: \[0 < n \Rightarrow Suc 0 < m \Rightarrow n < m * n\]
  using mult-strict-right-mono [of 1 m n] by simp

lemma n-less-n-mult-m: \[0 < n \Rightarrow Suc 0 < m \Rightarrow n < n * m\]
  using mult-strict-left-mono [of 1 m n] by simp

Induction starting beyond zero

lemma nat-induct-at-least [consumes 1, case-names base Suc]:
  \[P n \text{ if } n \geq m \text{ P m } \land n \geq m \Rightarrow P n \Rightarrow P (Suc n)\]
proof –
  define \[q\] where \[q = n - m\]
with \[\langle n \geq m \rangle\] have \[n = m + q\]
  by simp
moreover have \[P (m + q)\]
by (induction q) (use that in simp-all)
ultimately show P n
  by simp
qed

lemma nat-induct-non-zero [consumes 1, case-names 1 Suc]:
P n if n > 0 P 1 \( \land \) n > 0 \( \Rightarrow \) P n \( \Rightarrow \) P (Suc n)
proof –
  from \( \langle n > 0 \rangle \) have n ≥ 1
    by (cases n) simp-all
  moreover note \( \langle P 1 \rangle \)
  moreover have \( \land \) n, n ≥ 1 \( \Rightarrow \) P n \( \Rightarrow \) P (Suc n)
    using \( \langle \land \) n, n > 0 \( \Rightarrow \) P n \( \Rightarrow \) P (Suc n)\rangle
    by (simp add: Suc-le-eq)
ultimately show P n
  by (rule nat-induct-at-least)
qed

Specialized induction principles that work 'backwards':

lemma inc-induct [consumes 1, case-names base step]:
  assumes less: i ≤ j
  and base: P j
  and step: \( \land \) n, i ≤ n \( \Rightarrow \) n < j \( \Rightarrow \) P (Suc n) \( \Rightarrow \) P n
  shows P i
  using less step
proof (induct j – i arbitrary: i)
  case (0 i)
  then have i = j by simp
  with base show ?case by simp
next
  case (Suc d n)
  from Suc.hyps have n ≠ j by auto
  with Suc have n < j by (simp add: less-le)
  from \( \langle Suc d = j - n \rangle \) have d + 1 = j - n by simp
  then have d + 1 - 1 = j - n - 1 by simp
  then have d = j - n - 1 by simp
  then have d = j - (n + 1) by (simp add: diff-diff-eq)
  then have d = j - Suc n by simp
  moreover from \( \langle n < j \rangle \) have Suc n ≤ j by (simp add: Suc-le-eq)
  ultimately have P (Suc n)
 proof (rule Suc.hyps)
    fix q
    assume Suc n ≤ q
    then have n ≤ q by (simp add: Suc-le-eq less-imp-le)
    moreover assume q < j
    moreover assume P (Suc q)
    ultimately show P q by (rule Suc.prems)
  qed
  with order-refl \( \langle n < j \rangle \) show P n by (rule Suc.prems)
lemma strict-inc-induct [consumes 1, case_names base step]:

assumes less: \( i < j \)
and base: \( \forall i. j = \text{Suc} \ i \Rightarrow P \ i \)
and step: \( \forall i. i < j \Rightarrow P (\text{Suc} \ i) \Rightarrow P \ i \)

shows \( P \ i \)

using less proof (induct \( j - i - 1 \) arbitrary: \( i \))

case (0 \( i \))

from \( i < j \) obtain \( n \) where \( j = i + n \) and \( n > 0 \)
by (auto dest!: less-imp-Suc-add)

with 0 have \( j = \text{Suc} \ i \)
by (auto intro: order-antisym simp add: Suc-le-eq)

with base show ?case by simp

next
case (Suc \( d \ i \))

from Suc.prems consider \( i \leq j \mid i = \text{Suc} \ j \)
by (auto simp add: le-Suc-eq)

then show ?case by simp

qed

lemma zero-induct-lemma: \( P \ k \Rightarrow (\forall n. P (\text{Suc} \ n) \Rightarrow P \ n) \Rightarrow P (k - i) \)
using inc-induct[of \( k - i \) \( k \) \( P \)] by blast

lemma zero-induct: \( P \ k \Rightarrow (\forall n. P (\text{Suc} \ n) \Rightarrow P \ n) \Rightarrow P \ 0 \)
using inc-induct[of 0 \( k \) \( P \)] by blast

Further induction rule similar to \( \exists i \leq j. \forall P \ j. \forall n. n < j \Rightarrow P \ n \Rightarrow P \ (\text{Suc} \ n) \Rightarrow P \ j \)

lemma dec-induct [consumes 1, case_names base step]:

\( i \leq j \Rightarrow P \ i \Rightarrow (\forall n. i \leq n \Rightarrow n < j \Rightarrow P \ n \Rightarrow P \ (\text{Suc} \ n)) \Rightarrow P \ j \)

proof (induct \( j \) arbitrary: \( i \))

case 0
then show ?case by simp

next
case (Suc \( j \))

from Suc.prems consider \( i \leq j \mid i = \text{Suc} \ j \)
by (auto simp add: le-Suc-eq)

then show ?case
proof cases

case 1

moreover have \( j < \text{Suc} \ j \) by simp

moreover have \( P \ j \) using \( \langle i \leq j, P \ i \rangle \)

proof (rule Suc.hyps)
fix $q$
assume $i \leq q$
moreover assume $q < j$ then have $q < \text{Suc } j$
  by (simp add: less-Suc-eq)
moreoverassume $P q$
ultimately show $P (\text{Suc } q)$ by (rule Suc.prems)
qed
ultimately show $P (\text{Suc } j)$ by (rule Suc.prems)
next
  case 2
  with $\langle P i \rangle$ show $P (\text{Suc } j)$ by simp
qed

lemma transitive-stepwise-le:
  assumes $m \leq n$ and $\forall x. R x x \land x y \land y z \implies R x y \implies R x z$ and $\land n. R n$
  (Suc $n$)
  shows $R m n$
using $(m \leq n)$
  by (induction rule: dec-induct) (use assms in blast)

17.9.1 Greatest operator

lemma ex-has-greatest-nat:
  $P (k::nat) \implies \forall y. P y \implies y \leq b \implies \exists x. P x \land (\forall y. P y \implies y \leq x)$
proof (induction $b \leftarrow k$ arbitrary: $b \ k$ rule: less-induct)
  case less
  show ?case
dooring cases
    assume $\exists n > k. P n$
    then obtain $n$ where $n > k$ $P n$ by blast
    have $n \leq b$ using $\langle P n \rangle$ less.prems(2) by auto
    hence $b - n < b - k$
      by (rule diff.less-monot2 [OF $\langle k < n \rangle$ less-less-trans [OF $\langle k < n \rangle$]])
    from less.prems [OF this $\langle P n \rangle$ less.prems(2)]
    show ?thesis .
next
  assume $\neg (\exists n > k. P n)$
  hence $\forall y. P y \implies y \leq k$ by (auto simp: not-less)
  thus ?thesis using less.prems(1) by auto
qed
qed

lemma
  fixes $k :: \text{nat}$
  assumes $P k$ and minor: $\land y. P y \implies y \leq b$
  shows GreatestI-nat: $P (\text{Greatest } P)$
  and Greatest-le-nat: $k \leq \text{Greatest } P$
proof –
obtain $x$ where $P x \land \forall y. P y \Rightarrow y \leq x$
using assms ex-has-greatest-nat by blast
with $\langle P k \rangle$ show $P (\text{Greatest } P) k \leq \text{Greatest } P$
using GreatestI2-order by blast+

**Qed**

**Lemma** GreatestI-ex-nat:
\[ \exists k::\text{nat}. P k ; \land \forall y. P y = \Rightarrow y \leq b \] = \Rightarrow P (\text{Greatest } P)
by (blast intro: GreatestI-nat)

**17.10 Monotonicity of funpow**

**Lemma** funpow-increasing: $m \leq n \Rightarrow \text{mono } f \Rightarrow (f ^^ n) \top \leq (f ^^ m) \top$
for $f :: 'a::\{\text{lattice,order-top}\} \Rightarrow 'a$
by (induct rule: inc-induct)
(auto simp del: funpow.simps(2) simp add: funpow-Suc-right
intro: order-trans[OF - funpow-mono])

**Lemma** funpow-decreasing: $m \leq n \Rightarrow \text{mono } f \Rightarrow (f ^^ m) \bot \leq (f ^^ n) \bot$
for $f :: 'a::\{\text{lattice,order-bot}\} \Rightarrow 'a$
by (induct rule: dec-induct)
(auto simp del: funpow.simps(2) simp add: funpow-Suc-right
intro: order-trans[OF - funpow-mono])

**Lemma** mono-funpow: mono $Q \Rightarrow \text{mono } (\lambda i. (Q ^^ i) \bot)$
for $Q :: 'a::\{\text{lattice,order-bot}\} \Rightarrow 'a$
by (auto intro!: funpow-decreasing simp: mono-def)

**Lemma** antimono-funpow: mono $Q \Rightarrow \text{antimono } (\lambda i. (Q ^^ i) \top)$
for $Q :: 'a::\{\text{lattice,order-top}\} \Rightarrow 'a$
by (auto intro!: funpow-increasing simp: antimono-def)

**17.11 The divides relation on nat**

**Lemma** dvd-1-left [iff]: Suc 0 dvd k
by (simp add: dvd-def)

**Lemma** dvd-1-iff-1 [simp]: $m \text{ dvd } Suc 0 \leftrightarrow m = Suc 0$
by (simp add: dvd-def)

**Lemma** nat-dvd-1-iff-1 [simp]: $m \text{ dvd } 1 \leftrightarrow m = 1$
for $m :: \text{nat}$
by (simp add: dvd-def)

**Lemma** dvd-antisym: $m \text{ dvd } n \Rightarrow n \text{ dvd } m \Rightarrow m = n$
for $m n :: \text{nat}$
unfolding dvd-def by (force dest: mult-eq-self-implies-10 simp add: mult.assoc)

**Lemma** dvd-diff-nat [simp]: $k \text{ dvd } m \Rightarrow k \text{ dvd } n \Rightarrow k \text{ dvd } (m - n)$
for $k m n :: \text{nat}$
unfolding dvd-def by (blast intro: right-diff-distrib' [symmetric])

lemma dvd-diffD:
  fixes k m n :: nat
  assumes k dvd m − n k dvd n n ≤ m
  shows k dvd m
proof −
  have k dvd n + (m − n)
    using assms by (blast intro: dvd-add)
  with assms show ?thesis
    by simp
qed

lemma dvd-diffD1:
  k dvd m − n =⇒ k dvd m =⇒ n ≤ m =⇒ k dvd n
for k m n :: nat
by (drule-tac m = m in dvd-diff-nat) auto

lemma dvd-mult-cancel:
  fixes m n k :: nat
  assumes k ∗ m dvd k ∗ n and 0 < k
  shows m dvd n
proof −
  from assms(1) obtain q where k ∗ n = (k ∗ m) ∗ q ..
  then have k ∗ n = k ∗ (m ∗ q) by (simp add: ac-simps)
  with ‹0 < k› have n = m ∗ q by (auto simp add: mult-left-cancel)
  then show ?thesis ..
qed

lemma dvd-mult-cancel1:
  fixes m n :: nat
  assumes 0 < m
  shows m ∗ n dvd m −→ n = 1
proof
  assume m ∗ n dvd m
  then have m ∗ n dvd m ∗ 1
    by simp
  then have n dvd 1
    by (iprover intro: assms dvd-mult-cancel)
  then show n = 1
    by auto
qed auto

lemma dvd-mult-cancel2: 0 < m −→ n ∗ m dvd m −→ n = 1
for m n :: nat
using dvd-mult-cancel1 [of m n] by (simp add: ac-simps)

lemma dvd-imp-le: k dvd n −→ 0 < n −→ k ≤ n
for k n :: nat
by (auto elim!: dvdE) (auto simp add: gr0-conv-Suc)
theory "Nat"

lemma nat-dvd-not-less: \( 0 < m \implies m < n \implies \neg n \dvd m \)
  for \( m \, n :: \text{nat} \)
  by (auto elim!: dvdE) (auto simp add: gr0_conv_Suc)

lemma less-eq-dvd-minus:
  fixes \( m \, n :: \text{nat} \)
  assumes \( m \leq n \)
  shows \( m \dvd n \iff m \dvd n - m \)
proof
  from assms have \( n = m + (n - m) \) by simp
  then obtain \( q \) where \( n = m + q \) ..
  then show \(?thesis\) by (simp add: add.commute [of \( m \)])
qed

lemma dvd-minus-self:
  \( m \dvd n - m \implies n < m \lor m \dvd n \)
  for \( m \, n :: \text{nat} \)
  by (cases \( n < m \)) (auto elim!: dvdE simp add: not_less le_imp_diff_is_add dest: less_imp_le)

lemma dvd-minus-add:
  fixes \( m \, n \, q \, r :: \text{nat} \)
  assumes \( q \leq n \) \& \( q \leq r \cdot m \)
  shows \( m \dvd n - q \iff m \dvd n + (r \cdot m - q) \)
proof
  have \( m \dvd n - q \iff m \dvd r \cdot m + (n - q) \)
    using dvd-add-times-triv-left-iff [of \( m \, r \)] by simp
  also from assms have \( \ldots \iff m \dvd r \cdot m + n - q \) by simp
  also from assms have \( \ldots \iff m \dvd (r \cdot m - q) + n \) by simp
  also have \( \ldots \iff m \dvd n + (r \cdot m - q) \) by (simp add: add.commute)
  finally show \(?thesis\) .
qed

17.12 Aliasses

lemma nat-mult-1: \( 1 \cdot n = n \)
  for \( n :: \text{nat} \)
  by (fact mult-1-left)

lemma nat-mult-1-right: \( n \cdot 1 = n \)
  for \( n :: \text{nat} \)
  by (fact mult-1-right)

lemma diff-mult-distrib: \( (m - n) \cdot k = (m \cdot k) - (n \cdot k) \)
  for \( k \, m \, n :: \text{nat} \)
  by (fact left-diff-distrib')

lemma diff-mult-distrib2: \( k \cdot (m - n) = (k \cdot m) - (k \cdot n) \)
  for \( k \, m \, n :: \text{nat} \)
by (fact right-diff-distrib')

lemma le-diff-conv2: $k \leq j \Rightarrow (i \leq j - k) = (i + k \leq j)$
for $i j k :: \text{nat}$
by (fact le-diff-conv2)

lemma diff-self-eq-0 [simp]: $m - m = 0$
for $m :: \text{nat}$
by (fact diff-cancel)

lemma diff-diff-left [simp]: $i - j - k = i - (j + k)$
for $i j k :: \text{nat}$
by (fact diff-diff-add)

lemma diff-commute: $i - j - k = i - k - j$
for $i j k :: \text{nat}$
by (fact diff-right-commute)

lemma diff-add-inverse: $(n + m) - n = m$
for $m n :: \text{nat}$
by (fact add-diff-cancel-left')

lemma diff-add-inverse2: $(m + n) - n = m$
for $m n :: \text{nat}$
by (fact add-diff-cancel-right')

lemma diff-cancel: $(k + m) - (k + n) = m - n$
for $k m n :: \text{nat}$
by (fact add-diff-cancel-left)

lemma diff-cancel2: $(m + k) - (n + k) = m - n$
for $k m n :: \text{nat}$
by (fact add-diff-cancel-right)

lemma diff-add-0: $n - (n + m) = 0$
for $m n :: \text{nat}$
by (fact diff-add-zero)

lemma add-mult-distrib2: $k \ast (m + n) = (k \ast m) + (k \ast n)$
for $k m n :: \text{nat}$
by (fact distrib-left)

lemmas nat-distrib =
add-mult-distrib distrib-left diff-mult-distrib diff-mult-distrib2

17.13 Size of a datatype value

class size =
fixes size :: 'a ⇒ nat — see further theory Wellfounded

instantiation nat :: size
begin

definition size-nat where [simp, code]: size (n::nat) = n
instance ..
end

lemmas size-nat = size-nat-def

lemma size-neq-size-imp-neq: size x ≠ size y ⇒ x ≠ y
  by (erule contrapos-nn) (rule arg-cong)

17.14 Code module namespace

code-identifier
code-module Nat ⇀ (SML) Arith and (OCaml) Arith and (Haskell) Arith

hide-const (open) of-nat-aux

end

18 Fields

theory Fields
imports Nat
begin

18.1 Division rings

A division ring is like a field, but without the commutativity requirement.

class inverse = divide +
  fixes inverse :: 'a ⇒ 'a
begin

abbreviation inverse-divide :: 'a ⇒ 'a ⇒ 'a (infixl '/ 70)
where
  inverse-divide ≡ divide

end

Setup for linear arithmetic prover

ML-file ⟨~/src/Provers/Arith/fast-lin-arith.ML⟩
ML-file ⟨Tools/lin-arith.ML⟩
setup ⟨Lin-Arith.global-setup⟩
THEORY “Fields”

declaration  • K {  
  Lin-Arith.init-arith-data  
  #> Lin-Arith.add-discrete-type type-name (nat)  
  #> Lin-Arith.add-lessD (thm Suc-leI)  
  #> Lin-Arith.add-simps (thms simp-thms ring-distribs if-True if-False  
    minus-diff-eq  
    add-0-left add-0-right order-less-irrefl  
    zero-neq-one zero-less-one zero-le-one  
    zero-neq-one [THEN not-sym] not-one-le-zero not-one-less-zero  
    add-Suc add-Suc-right nat.inject  
    Suc-le-mono Suc-less-ep Zero-not-Suc  
    Suc-not-Zero le-0-eq One-not-def}  
  #> Lin-Arith.add-simprocs [simp proc ‹ group-cancel-add›, simp proc ‹ group-cancel-diff›,  
    simp proc ‹ group-cancel-eq›, simp proc ‹ group-cancel-le›,  
    simp proc ‹ group-cancel-less›,  
    simp proc ‹ natle-cancel-sums›, simp proc ‹ nateq-cancel-sums›,  
    simp proc ‹ natless-cancel-sums›)]

  simproc-setup fast-arith-nat (m::nat) < n | (m::nat) ≤ n | (m::nat) = n) =  
  K Lin-Arith.simproc — Because of this simproc, the arithmetic solver is really  
  only useful to detect inconsistencies among the premises for subgoals which are not  
  themselves (in)equalities, because the latter activate fast-nat-arith-simproc anyway.  
  However, it seems cheaper to activate the solver all the time rather than add the  
  additional check.

lemmas [linarith-split] = nat-diff-split split-min split-max abs-split

Lemmas divide-simps move division to the outside and eliminates them on (in)equalities.

named-theorems divide-simps rewrite rules to eliminate divisions

class division-ring = ring-1 + inverse +  
  assumes left-inverse [simp]: a ≠ 0 ⇒ inverse a * a = 1  
  assumes right-inverse [simp]: a ≠ 0 ⇒ a * inverse a = 1  
  assumes divide-inverse: a / b = a * inverse b  
  assumes inverse-zero [simp]: inverse 0 = 0
begin

subclass ring-1-no-zero-divisors
proof
  fix a b :: 'a  
  assume a: a ≠ 0 and b: b ≠ 0  
  show a * b ≠ 0  
  proof
    assume ab: a * b = 0  
    hence 0 = inverse a * (a * b) * inverse b by simp  
    also have ... = (inverse a * a) * (b * inverse b)  
      by (simp only: mult.assoc)  
    also have ... = 1 using a b by simp
final show False by simp
qed

lemma nonzero-imp-inverse-nonzero:
a \neq 0 \implies \text{inverse } a \neq 0
proof
assume ianz: \text{inverse } a = 0
assume a \neq 0
hence 1 = a * \text{inverse } a by simp
also have \ldots = 0 by (simp add: ianz)
finally have 1 = 0 .
thus False by (simp add: eq-commute)
qed

lemma inverse-zero-imp-zero:
assumes \text{inverse } a = 0
shows a = 0
proof (rule ccontr)
assume a \neq 0
then have \text{inverse } a \neq 0
  by (simp add: nonzero-imp-inverse-nonzero)
with assms show False
  by auto
qed

lemma inverse-unique:
assumes ab: a * b = 1
shows \text{inverse } a = b
proof
have a \neq 0 using ab by (cases a = 0) simp-all
moreover have \text{inverse } a * (a * b) = \text{inverse } a by (simp add: ab)
ultimately show \?thesis by (simp add: mult.assoc [symmetric])
qed

lemma nonzero-inverse-minus-eq:
a \neq 0 \implies \text{inverse } (-a) = -\text{inverse } a
by (rule inverse-unique) simp

lemma nonzero-inverse-inverse-eq:
a \neq 0 \implies \text{inverse } (\text{inverse } a) = a
by (rule inverse-unique) simp

lemma nonzero-inverse-eq-imp-eq:
assumes \text{inverse } a = \text{inverse } b and a \neq 0 and b \neq 0
shows a = b
proof
from \text{inverse } a = \text{inverse } b;
have \text{inverse } (\text{inverse } a) = \text{inverse } (\text{inverse } b) by (rule arg-cong)
with \langle a \neq 0, and \langle b \neq 0 \rangle show a = b
by (simp add: nonzero-inverse-inverse-eq)

qed

lemma inverse-1 [simp]: inverse 1 = 1
  by (rule inverse-unique) simp

subclass divide-trivial
  by standard (simp-all add: divide-inverse)

lemma nonzero-inverse-mult-distrib:
  assumes a ≠ 0 and b ≠ 0
  shows inverse (a * b) = inverse b * inverse a
proof –
  have a * (b * inverse b) * inverse a = 1 using assms by simp
  hence a * b * (inverse b * inverse a) = 1 by (simp only: mult.assoc)
  thus ?thesis by (rule inverse-unique)

qed

lemma division-ring-inverse-add:
  a ≠ 0 ⇒ b ≠ 0 ⇒ inverse a + inverse b = inverse a * (a + b) * inverse b
by (simp add: algebra-simps)

lemma division-ring-inverse-diff:
  a ≠ 0 ⇒ b ≠ 0 ⇒ inverse a - inverse b = inverse a * (b - a) * inverse b
by (simp add: algebra-simps)

lemma right-inverse-eq: b ≠ 0 ⇒ a / b = 1 ↔ a = b
proof
  assume neq: b ≠ 0
  {
    hence a = (a / b) * b by (simp add: divide-inverse mult.assoc)
    also assume a / b = 1
    finally show a = b by simp
  next
    assume a = b
    with neq show a / b = 1 by (simp add: divide-inverse)
  }

qed

lemma nonzero-inverse-eq-divide: a ≠ 0 ⇒ inverse a = 1 / a
by (simp add: divide-inverse)

lemma divide-self [simp]: a ≠ 0 ⇒ a / a = 1
by (simp add: divide-inverse)

lemma inverse-eq-divide [field-simps, field-split-simps, divide-simps]: inverse a = 1 / a
by (simp add: divide-inverse)
lemma add-divide-distrib: \((a + b) / c = a/c + b/c\)
by (simp add: divide-inverse algebra-simps)

lemma times-divide-eq-right [simp]: \(a \ast (b / c) = (a \ast b) / c\)
by (simp add: divide-inverse mult.assoc)

lemma minus-divide-left: \(-(a / b) = (-a) / b\)
by (simp add: divide-inverse)

lemma nonzero-minus-divide-right: \(b \neq 0 \implies -(a / b) = a / (-b)\)
by (simp add: divide-inverse nonzero-inverse-minus-eq)

lemma nonzero-minus-divide-divide: \(b \neq 0 \implies (-a) / (-b) = a / b\)
by (simp add: divide-inverse nonzero-inverse-minus-eq)

lemma divide-minus-left [simp]: \((\neg a) / b = \neg (a / b)\)
by (simp add: divide-inverse)

lemma diff-divide-distrib: \((a - b) / c = a / c - b / c\)
using add-divide-distrib[of a - b c] by simp

lemma nonzero-eq-divide-eq [field-simps]: \(c \neq 0 \implies a = b / c \iff a = c * b\)
proof -
  assume [simp]: \(c \neq 0\)
  have \(a = b / c \iff a * c = (b / c) * c\) by simp
  also have ... \(\iff a * c = b\) by (simp add: divide-inverse mult.assoc)
  finally show ?thesis .
qed

lemma nonzero-eq-divide-eq [field-simps]: \(c \neq 0 \implies b / c = a \iff b = c * a\)
proof -
  assume [simp]: \(c \neq 0\)
  have \(b / c = a \iff (b / c) * c = a * c\) by simp
  also have ... \(\iff b = a * c\) by (simp add: divide-inverse mult.assoc)
  finally show ?thesis .
qed

lemma nonzero-neg-divide-eq-eq [field-simps]: \(b \neq 0 \implies -(a / b) = c \iff -a = c * b\)
using nonzero-divide-eq-eq[of b -a c] by simp

lemma nonzero-neg-divide-eq-eq2 [field-simps]: \(b \neq 0 \implies c = - (a / b) \iff c * b = -a\)
using nonzero-neg-divide-eq-eq[of b a c] by auto

lemma divide-eq-imp: \(c \neq 0 \implies b = a \ast c \iff b / c = a\)
by (simp add: divide-inverse mult.assoc)

lemma eq-divide-imp: \(c \neq 0 \implies a \ast c = b \iff a = b / c\)
by (drule sym) (simp add: divide-inverse mult.assoc)

lemma add-divide-eq-iff [field-simps]:
z \neq 0 \Longrightarrow x + y / z = (x * z + y) / z
by (simp add: add-divide-distrib nonzero-eq-divide-eq)

lemma divide-add-eq-iff [field-simps]:
z \neq 0 \Longrightarrow x / z + y = (x + y * z) / z
by (simp add: add-divide-distrib nonzero-eq-divide-eq)

lemma diff-divide-eq-iff [field-simps]:
z \neq 0 \Rightarrow x - y / z = (x * z - y) / z
by (simp add: diff-divide-distrib nonzero-eq-divide-eq eq-diff-eq)

lemma minus-divide-add-eq-iff [field-simps]:
z \neq 0 \Rightarrow - (x / z) + y = (- x + y * z) / z
by (simp add: add-divide-distrib diff-divide-eq-iff)

lemma divide-diff-eq-iff [field-simps]:
z \neq 0 \Rightarrow x / z - y = (x - y * z) / z
by (simp add: field-simps)

lemma minus-divide-diff-eq-iff [field-simps]:
z \neq 0 \Rightarrow - (x / z) - y = (- x - y * z) / z
by (simp add: divide-diff-eq-iff [symmetric])

lemma division-ring-divide-zero:
a / 0 = 0
by (fact div-by-0)

lemma divide-self-if [simp]:
a / a = (if a = 0 then 0 else 1)
by simp

lemma inverse-nonzero-iff-nonzero [simp]:
inverse a = 0 \iff a = 0
by (rule iffI) (fact inverse-zero-imp-zero, simp)

lemma inverse-minus-eq [simp]:
inverse (- a) = - inverse a

proof cases
assume a=0 thus \thesis by simp
next
assume a\neq0
thus \thesis by (simp add: nonzero-inverse-minus-eq)
qed

lemma inverse-inverse-eq [simp]:
inverse (inverse a) = a
proof cases
  assume a=0 thus ?thesis by simp
next
  assume a\not=0
  thus ?thesis by (simp add: nonzero-inverse-inverse-eq)
qed

lemma inverse-eq-imp-eq:
inverse a = inverse b \implies a = b
by (drule arg-cong [where f=inverse], simp)

lemma inverse-eq-iff-eq [simp]:
inverse a = inverse b \iff a = b
by (force dest!: inverse-eq-imp-eq)

lemma mult-commute-imp-mult-inverse-commute:
assumes y * x = x * y
shows inverse y * x = x * inverse y
proof (cases y=0)
  case False
  hence x * inverse y = inverse y * y * x * inverse y
  by simp
  also have \ldots = inverse y * (x * y * inverse y)
  by (simp add: mult.assoc assms)
  finally show ?thesis by (simp add: mult.assoc False)
qed simp

lemmas mult-inverse-of-nat-commute =
   mult-commute-imp-mult-inverse-commute[OF mult-of-nat-commute]

lemma divide-divide-eq-left':
(a / b) / c = a / (c * b)
by (cases b = 0 \or c = 0)
  (auto simp: divide-inverse mult.assoc nonzero-inverse-mult-distrib)

lemma add-divide-eq-if-simps [field-split-simps, divide-simps]:
a + b / z = (if z = 0 then a else (a * z + b) / z)
a / z + b = (if z = 0 then b else (a + b * z) / z)
- (a / z) + b = (if z = 0 then b else (-a + b * z) / z)
a - b / z = (if z = 0 then a else (a * z - b) / z)
a / z - b = (if z = 0 then -b else (a - b * z) / z)
- (a / z) - b = (if z = 0 then -b else (-a - b * z) / z)
by (simp-all add: add-divide-eq-iff divide-add-eq-iff diff-divide-eq-iff divide-diff-eq-iff
   minus-divide-diff-eq-iff)

lemma [field-split-simps, divide-simps]:
shows divide-eq-eq: b / c = a \iff (if c \not= 0 then b = a * c else a = 0)
  and eq-divide-eq: a = b / c \iff (if c \not= 0 then a * c = b else a = 0)
  and minus-divide-eq-eq: \not(-b / c) = a \iff (if c \not= 0 then -b = a * c else a


and eq-minus-divide-eq: \( a = -(b / c) \iff (if \ c \neq 0 \ then \ a \cdot c = -b \ else \ a = 0) \) by (auto simp add: field-simps)

end

18.2 Fields

class field = comm-ring-1 + inverse +
assumes field-inverse: \( a \neq 0 \implies inverse \ a \cdot a = 1 \)
assumes field-divide-inverse: \( a / b = a \cdot inverse b \)
assumes field-inverse-zero: \( inverse 0 = 0 \)
begin
subclass division-ring
proof
  fix \( a :: 'a \)
  assume \( a \neq 0 \)
  thus \( inverse \ a \cdot a = 1 \) by (rule field-inverse)
  thus \( a \cdot inverse \ a = 1 \) by (simp only: mult.commute)
next
  fix \( a \ b :: 'a \)
  show \( a / b = a \cdot inverse b \) by (rule field-divide-inverse)
next
  show \( inverse 0 = 0 \)
  by (fact field-inverse-zero)
qed

subclass idom-divide
proof
  fix \( b \ a \)
  assume \( b \neq 0 \)
  then show \( a \cdot b / b = a \)
  by (simp add: divide-inverse ac-simps)
next
  fix \( a \)
  show \( a / 0 = 0 \)
  by (simp add: divide-inverse)
qed

There is no slick version using division by zero.

lemma inverse-add:
\( a \neq 0 \implies b \neq 0 \implies inverse \ a + inverse \ b = (a + b) \cdot inverse \ b \)
by (simp add: division-ring-inverse-add ac-simps)

lemma nonzero-mult-divide-mult-cancel-left [simp]:
assumes [simp]: \( c \neq 0 \)
shows \( (c \cdot a) / (c \cdot b) = a / b \)
proof (cases b = 0)
  case True then show ?thesis by simp
next
  case False then have \((c\cdot a)/(c\cdot b) = c \cdot a \cdot (\text{inverse} \ b \cdot \text{inverse} \ c)\)
    by (simp add: divide-inverse nonzero-inverse-mult-distrib)
  also have \(\ldots = a \cdot \text{inverse} \ b \cdot (\text{inverse} \ c \cdot c)\)
    by (simp only: ac-simps)
  finally show \(?thesis\) by (simp add: divide-inverse)
qed

lemma nonzero-mult-divide-mult-cancel-right [simp]:
  \(c \neq 0 \Rightarrow (a \cdot c) / (b \cdot c) = a / b\)
using nonzero-mult-divide-mult-cancel-left [of c a b] by (simp add: ac-simps)

lemma times-divide-eq-left [simp]: \((b / c) \cdot a = (b \cdot a) / c\)
by (simp add: divide-inverse ac-simps)

lemma divide-inverse-commute: \(a / b = \text{inverse} \ b \cdot a\)
by (simp add: divide-inverse mult.commute)

lemma add-frac-eq:
  assumes \(y \neq 0\) and \(z \neq 0\)
  shows \(x / y + w / z = (x \cdot z + w \cdot y) / (y \cdot z)\)
proof -
  have \(x / y + w / z = (x \cdot z) / (y \cdot z) + (y \cdot w) / (y \cdot z)\)
    using assms by simp
  also have \(\ldots = (x \cdot z + y \cdot w) / (y \cdot z)\)
    by (simp only: add-distribute)
  finally show \(?thesis\)
    by (simp only: mult.commute)
qed

Special Cancellation Simprules for Division

lemma nonzero-divide-mult-cancel-right [simp]:
  \(b \neq 0 \Rightarrow b / (a \cdot b) = 1 / a\)
using nonzero-mult-divide-mult-cancel-right [of b 1 a] by simp

lemma nonzero-divide-mult-cancel-left [simp]:
  \(a \neq 0 \Rightarrow a / (a \cdot b) = 1 / b\)
using nonzero-mult-divide-mult-cancel-left [of a 1 b] by simp

lemma nonzero-mult-divide-mult-cancel-left2 [simp]:
  \(c \neq 0 \Rightarrow (c \cdot a) / (b \cdot c) = a / b\)
using nonzero-mult-divide-mult-cancel-left [of c a b] by (simp add: ac-simps)

lemma nonzero-mult-divide-mult-cancel-right2 [simp]:
  \(c \neq 0 \Rightarrow (a \cdot c) / (c \cdot b) = a / b\)
using nonzero-mult-divide-mult-cancel-right [of b c a] by (simp add: ac-simps)

lemma diff-frac-eq:
y ≠ 0 ⇒ z ≠ 0 ⇒ x / y − w / z = (x * z − w * y) / (y * z)
by (simp add: field-simps)

lemma frac-eq-eq:
y ≠ 0 ⇒ z ≠ 0 ⇒ (x / y = w / z) = (x * z = w * y)
by (simp add: field-simps)

lemma divide-minus1 [simp]: x / − 1 = − x
using nonzero-minus-divide-right [of 1 x] by simp

This version builds in division by zero while also re-orienting the right-hand side.

lemma inverse-mult-distrib [simp]:
inverse (a * b) = inverse a * inverse b
proof cases
  assume a ≠ 0 ∧ b ≠ 0
  thus ?thesis by (simp add: nonzero-inverse-mult-distrib ac-simps)
next
  assume ¬ (a ≠ 0 ∧ b ≠ 0)
  thus ?thesis by force
qed

lemma inverse-divide [simp]:
inverse (a / b) = b / a
by (simp add: divide-inverse mult.commute)

Calculations with fractions

There is a whole bunch of simp-rules just for class field but none for class field and nonzero-divides because the latter are covered by a simproc.

lemmas mult-divide-mult-cancel-left = nonzero-mult-divide-mult-cancel-left

lemmas mult-divide-mult-cancel-right = nonzero-mult-divide-mult-cancel-right

lemma divide-divide-eq-right [simp]:
a / (b / c) = (a * c) / b
by (simp add: divide-inverse ac-simps)

lemma divide-divide-eq-left [simp]:
(a / b) / c = a / (b * c)
by (simp add: divide-inverse mult.assoc)

lemma divide-divide-times-eq:
(x / y) / (z / w) = (x * w) / (y * z)
by simp

Special Cancellation Simprules for Division
lemma mult-divide-mult-cancel-left-if [simp]:
  shows \((c \ast a) / (c \ast b) = (if c = 0 then 0 else a / b)\)
  by simp

Division and Unary Minus

lemma minus-divide-right:
  \(- (a / b) = a / - b\)
  by (simp add: divide-inverse)

lemma divide-minus-right [simp]:
  \(a / - b = - (a / b)\)
  by (simp add: divide-inverse)

lemma minus-divide-divide:
  \((- a) / (- b) = a / b\)
  by (cases b=0) (simp-all add: nonzero-minus-divide-divide)

lemma inverse-eq-1-iff [simp]:
  inverse x = 1 \iff x = 1
  using inverse-eq-iff-eq [of x 1] by simp

lemma divide-eq-0-iff [simp]:
  a / b = 0 \iff a = 0 \or b = 0
  by (simp add: divide-inverse)

lemma divide-cancel-right [simp]:
  c / a = c / b \iff c = 0 \or a = b
  by (cases c=0) (simp-all add: divide-inverse)

lemma divide-cancel-left [simp]:
  a / c = a / b \iff c = 0 \or a = b
  by (cases c=0) (simp-all add: divide-inverse)

lemma divide-eq-1-iff [simp]:
  a / b = 1 \iff b \neq 0 \and a = b
  by (cases b=0) (simp-all add: divide-eq-1-iff)

lemma one-eq-divide-iff [simp]:
  1 = a / b \iff b \neq 0 \and a = b
  by (simp add: eq-commute [of 1])

lemma divide-eq-minus-1-iff:
  \((a / b = - 1) \iff b \neq 0 \and a = - b\)
  using divide-eq-1-iff by fastforce

lemma times-divide-times-eq:
  \((x / y) * (z / w) = (x * z) / (y * w)\)
  by simp
lemma add-frac-num:
y \neq 0 \implies x / y + z = (x + z \cdot y) / y
by (simp add: add-divide-distrib)

lemma add-num-frac:
y \neq 0 \implies z + x / y = (x + z \cdot y) / y
by (simp add: add-divide-distrib add.commute)

lemma dvd-field-iff:
a dvd b \iff (a = 0 \implies b = 0)
proof (cases a = 0)
  case False
  then have b = a * (b / a)
    by (simp add: field-simps)
  then have a dvd b ..
  with False show ?thesis
    by simp
qed simp

lemma inj-divide-right [simp]:
inj (\lambda b. b / a) \iff a \neq 0
proof
  have (\lambda b. b / a) = (*) (inverse a)
    by (simp add: field-simps fun-eq_iff)
  then have inj (\lambda y. y / a) \iff inj ((*) (inverse a))
    by simp
  also have \ldots \iff inverse a \neq 0
    by simp
  also have \ldots \iff a \neq 0
    by simp
  finally show ?thesis
    by simp
qed

end

class field-char-0 = field + ring-char-0

18.3 Ordered fields

class field-abs-sgn = field + idom-abs-sgn
begin

lemma sgn-inverse [simp]:
  sgn (inverse a) = inverse (sgn a)
proof (cases a = 0)
  case True then show ?thesis by simp
next
  case False
then have $a \cdot \text{inverse} a = 1$
  by simp
then have $\text{sgn} (a \cdot \text{inverse} a) = \text{sgn} 1$
  by simp
then have $\text{sgn} a \cdot \text{sgn} (\text{inverse} a) = 1$
  by (simp add: sgn-mult)
then have $(\text{inverse} (\text{sgn} a) \cdot \text{sgn} a) \cdot \text{sgn} (\text{inverse} a) = \text{inverse} (\text{sgn} a)$
  by (simp add: ac-simps)
with False show \text{?thesis}
  by (simp add: sgn-eq-0-iff)
qed

lemma \text{abs-inverse} [simp]:
  $|\text{inverse} a| = \text{inverse} |a|$
proof -
  from sgn-mult-abs [of inverse a] sgn-mult-abs [of a]
  have $(\text{inverse} (\text{sgn} a) \cdot |\text{inverse} a|) = \text{inverse} (\text{sgn} a \cdot |a|)$
    by simp
  then show \text{?thesis} by (auto simp add: sgn-eq-0-iff)
qed

lemma sgn-divide [simp]:
  $\text{sgn} (a / b) = \text{sgn} a / \text{sgn} b$
unfolding divide-inverse sgn-mult by simp

lemma abs-divide [simp]:
  $|a / b| = |a| / |b|$
unfolding divide-inverse abs-mult by simp
end

class linordered-field = field + linordered-idom
begin

lemma positive-imp-inverse-positive:
  assumes a-gt-0: $0 < a$
  shows $0 < \text{inverse} a$
proof -
  have $0 < a \cdot \text{inverse} a$
    by (simp add: a-gt-0 [THEN less-imp-not-eq2])
  thus $0 < \text{inverse} a$
    by (simp add: a-gt-0 [THEN less-not-sym] zero-less-mult-iff)
qed

lemma negative-imp-inverse-negative:
  $a < 0 \implies \text{inverse} a < 0$
  using positive-imp-inverse-positive [of $-a$]
by (simp add: nonzero-inverse-minus-eq less-imp-not-eq)

lemma inverse-le-imp-le:
  assumes invle: inverse a ≤ inverse b and apos: 0 < a
  shows b ≤ a
proof (rule classical)
  assume ¬ b ≤ a
  hence a < b by (simp add: linorder-not-le)
  hence bpos: 0 < b by (blast intro: apos less-trans)
  hence a * inverse a ≤ a * inverse b
      by (simp add: apos invle less-imp-le mult-left-mono)
  hence (a * inverse a) * b ≤ (a * inverse b) * b
      by (simp add: bpos less-imp-le mult-right-mono)
  thus b ≤ a by (simp add: mult.assoc apos bpos less-imp-not-eq2)
qed

lemma inverse-positive-imp-positive:
  assumes inv-gt-0: 0 < inverse a and nz: a ≠ 0
  shows 0 < a
proof
  have 0 < inverse (inverse a)
      using inv-gt-0 by (rule positive-imp-inverse-positive)
  thus 0 < a
      using nz by (simp add: nonzero-inverse-inverse-eq)
qed

lemma inverse-negative-imp-negative:
  assumes inv-less-0: inverse a < 0 and nz: a ≠ 0
  shows a < 0
proof
  have inverse (inverse a) < 0
      using inv-less-0 by (rule negative-imp-inverse-negative)
  thus a < 0 using nz by (simp add: nonzero-inverse-inverse-eq)
qed

lemma linordered-field-no-lb:
  ∀ x. ∃ y. y < x
proof
  fix x::'a
  have m1: − (1::'a) < 0 by simp
  from add-strict-right-mono[OF m1, where c=x]
  have (− 1) + x < x by simp
  thus ∃ y. y < x by blast
qed

lemma linordered-field-no-ub:
  ∀ x. ∃ y. y > x
proof
  fix x::'a
have \( m1 : (1 :: a) > 0 \) by simp
from add-strict-right-mono[OF \( m1 \), where \( c=x \)]
have \( 1 + x > x \) by simp
thus \( \exists y. y > x \) by blast
qed

lemma \textit{less-imp-inverse-less}:
assumes \( \text{less: } a < b \) and \( \text{apos: } 0 < a \)
shows \( \text{inverse } b < \text{inverse } a \)
proof (rule ccontr)
  assume \( \neg \text{inverse } b < \text{inverse } a \)
  hence \( \text{inverse } a \leq \text{inverse } b \) by simp
  hence \( \neg (a < b) \)
    by (simp add: not-less inverse-le-imp-le[OF - apos])
  thus \( \text{False} \) by (rule notE[OF - less])
qed

lemma \textit{inverse-less-imp-less}:
assumes \( \text{inverse } a < \text{inverse } b \)
shows \( b < a \)
proof -
  have \( a \neq b \)
    using assms by (simp add: less-le)
  moreover have \( b \leq a \)
    using assms by (force simp: less-le dest: inverse-le-imp-le)
  ultimately show \( \text{thesis} \)
    by (simp add: less-le)
qed

Both premises are essential. Consider -1 and 1.

lemma \textit{inverse-less-iff-less} [simp]:
\( 0 < a \implies 0 < b \implies \text{inverse } a < \text{inverse } b \iff b < a \)
by (blast intro: less-imp-inverse-less dest: inverse-less-imp-less)

lemma \textit{le-imp-inverse-le}:
\( a \leq b \implies 0 < a \implies \text{inverse } b \leq \text{inverse } a \)
by (force simp add: le-less less-imp-inverse-less)

lemma \textit{inverse-le-iff-le} [simp]:
\( 0 < a \implies 0 < b \implies \text{inverse } a \leq \text{inverse } b \iff b \leq a \)
by (blast intro: le-imp-inverse-le dest: inverse-le-imp-le)

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma \textit{inverse-le-imp-le-neg}:
assumes \( \text{inverse } a \leq \text{inverse } b \)
shows \( b \leq a \)
proof (rule classical)
  assume \( \neg b \leq a \)
with \( b < 0 \), have \( a < 0 \)
   by force
with assms show \( b \leq a \)
   using inverse-le-imp-le [of \(-b - a\)] by (simp add: nonzero-inverse-minus-eq)
qed

lemma less-imp-inverse-less-neg:
  assumes \( a < b \) \( b < 0 \)
  shows \( \text{inverse } b < \text{inverse } a \)
proof --
  have \( a < 0 \)
    using assms by (blast intro: less-trans)
  with less-imp-inverse-less [of \(-b - a\)] show ?thesis
  by (simp add: nonzero-inverse-minus-eq assms)
qed

lemma inverse-less-imp-less-neg:
  assumes \( \text{inverse } a < \text{inverse } b \) \( \text{b < 0} \)
  shows \( \text{b < a} \)
proof (rule classical)
  assume \( \neg b < a \)
  with \( b < 0 \)
  have \( a < 0 \)
    by force
  with inverse-less-imp-less [of \(-b - a\)] show ?thesis
  by (simp add: nonzero-inverse-minus-eq assms)
qed

lemma inverse-less-iff-less-neg [simp]:
  \( a < 0 \implies b < 0 \implies \text{inverse } a < \text{inverse } b \iff b < a \)
  using inverse-less-iff-less [of \(-b - a\)]
  by (simp del: inverse-less-iff-less add: nonzero-inverse-minus-eq)
lemma le-imp-inverse-le-neg:
  \( a \leq b \implies b < 0 \implies \text{inverse } b \leq \text{inverse } a \)
  by (force simp add: le-less less-imp-inverse-less-neg)
lemma inverse-le-iff-le-neg [simp]:
  \( a < 0 \implies b < 0 \implies \text{inverse } a \leq \text{inverse } b \iff b \leq a \)
  by (blast intro: le-imp-inverse-le-neg dest: inverse-le-imp-le-neg)
lemma one-less-inverse:
  \( 0 < a \implies a < 1 \implies 1 < \text{inverse } a \)
  using less-imp-inverse-less [of \(a\), unfolded inverse-1]

lemma one-le-inverse:
  \( 0 < a \implies a \leq 1 \implies 1 \leq \text{inverse } a \)
  using le-imp-inverse-le [of \(a\), unfolded inverse-1]

lemma pos-le-divide-eq [field-simps]:
assumes $0 < c$
shows $a \leq b / c \iff a \cdot c \leq b$
proof
  from assms have $a \leq b / c \iff a \cdot c \leq (b / c) \cdot c$
    using mult-le-cancel-right [of $a \cdot c$ $b$ $\text{inverse } c$] by (auto simp add: field-simps)
  also have ... $\iff a \cdot c \leq b$
    by (simp add: less_imp_not_eq2 [OF assms] divide-inverse mult.assoc)
finally show ?thesis .
qed

lemma pos-less-divide-eq [field-simps]:
assumes $0 < c$
shows $a < b / c \iff a \cdot c < b$
proof
  from assms have $a < b / c \iff a \cdot c < (b / c) \cdot c$
    using mult-less-cancel-right [of $a \cdot c$ $b$ $\text{inverse } c$] by auto
  also have ... $= (a \cdot c < b)$
    by (simp add: less_imp_not_eq2 [OF assms] divide-inverse mult.assoc)
finally show ?thesis .
qed

lemma neg-less-divide-eq [field-simps]:
assumes $c < 0$
shows $a < b / c \iff b < a \cdot c$
proof
  from assms have $a < b / c \iff (b / c) \cdot c < a \cdot c$
    using mult-less-cancel-right [of $b$ $\text{inverse } c$ $c a$] by auto
  also have ... $\iff b < a \cdot c$
    by (simp add: less_imp_not_eq [OF assms] divide-inverse mult.assoc)
finally show ?thesis .
qed

lemma neg-le-divide-eq [field-simps]:
assumes $c < 0$
shows $a \leq b / c \iff b \leq a \cdot c$
proof
  from assms have $a \leq b / c \iff (b / c) \cdot c \leq a \cdot c$
    using mult-le-cancel-right [of $b \cdot \text{inverse } c$ $c a$] by (auto simp add: field-simps)
  also have ... $\iff b \leq a \cdot c$
    by (simp add: less_imp_not_eq [OF assms] divide-inverse mult.assoc)
finally show ?thesis .
qed

lemma pos-divide-le-eq [field-simps]:
assumes $0 < c$
shows $b / c \leq a \iff b \leq a \cdot c$
proof
  from assms have $b / c \leq a \iff (b / c) \cdot c \leq a \cdot c$
    using mult-le-cancel-right [of $b / c$ $\text{inverse } c$ $a$] by auto
also have ... $\iff b \leq a \ast c$

by (simp add: less-imp-not-eq2 [OF assms] divide-inverse mult.assoc)

finally show ?thesis.

qed

lemma pos-divide-less-eq [field-simps]:

assumes $0 < c$

shows $b / c < a \iff b < a \ast c$

proof -

from assms have $b / c < a \iff (b / c) \ast c < a \ast c$

using mult-less-cancel-right [of $b / c c a$] by auto

also have ... $\iff b < a \ast c$

by (simp add: less-imp-not-eq2 [OF assms] divide-inverse mult.assoc)

finally show ?thesis.

qed

lemma neg-divide-le-eq [field-simps]:

assumes $c < 0$

shows $b / c \leq a \iff a \ast c \leq b$

proof -

from assms have $b / c \leq a \iff a \ast c \leq (b / c) \ast c$

using mult-le-cancel-right [of $a c b / c$] by auto

also have ... $\iff a \ast c \leq b$

by (simp add: less-imp-not-eq [OF assms] divide-inverse mult.assoc)

finally show ?thesis.

qed

lemma neg-divide-less-eq [field-simps]:

assumes $c < 0$

shows $b / c < a \iff a \ast c < b$

proof -

from assms have $b / c < a \iff a \ast c < b / c \ast c$

using mult-less-cancel-right [of $a c b / c$] by auto

also have ... $\iff a \ast c < b$

by (simp add: less-imp-not-eq [OF assms] divide-inverse mult.assoc)

finally show ?thesis.

qed

The following field-simps rules are necessary, as minus is always moved atop of division but we want to get rid of division.

lemma pos-le-minus-divide-eq [field-simps]: $0 < c \iff a \leq - (b / c) \iff a \ast c \leq - b$

unfolding minus-divide-left by (rule pos-le-divide-eq)

lemma neg-le-minus-divide-eq [field-simps]: $c < 0 \iff a \leq - (b / c) \iff - b \leq a \ast c$

unfolding minus-divide-left by (rule neg-le-divide-eq)

lemma pos-less-minus-divide-eq [field-simps]: $0 < c \iff a < - (b / c) \iff a \ast c$

unfolding minus-divide-left by (rule pos-le-divide-eq)
\[
< - b \\
\text{unfolding} \ \text{minus-divide-left by (rule pos-less-divide-eq)}
\]

\text{lemma} \ \text{neg-less-minus-divide-eq [field-simps]}: \ c < 0 \implies a < (b / c) \iff - b < a * c \\
\text{unfolding} \ \text{minus-divide-left by (rule neg-less-divide-eq)}

\text{lemma} \ \text{pos-minus-divide-less-eq [field-simps]}: \ 0 < c \implies (b / c) < a \iff - b < a \\
\text{unfolding} \ \text{minus-divide-left by (rule pos-divide-less-eq)}

\text{lemma} \ \text{neg-minus-divide-less-eq [field-simps]}: \ c < 0 \implies (b / c) < a \iff a * c < - b \\
\text{unfolding} \ \text{minus-divide-left by (rule neg-divide-less-eq)}

\text{lemma} \ \text{pos-minus-divide-le-eq [field-simps]}: \ 0 < c \implies (b / c) \leq a \iff b \leq a * c \\
\text{unfolding} \ \text{minus-divide-left by (rule pos-divide-le-eq)}

\text{lemma} \ \text{neg-minus-divide-le-eq [field-simps]}: \ c < 0 \implies (b / c) \leq a \iff a * c \leq - b \\
\text{unfolding} \ \text{minus-divide-left by (rule neg-divide-le-eq)}

\text{lemma} \ \text{frac-less-eq:} \\
y \neq 0 \implies z \neq 0 \implies (x / y < w / z \iff (x * z - w * y) / (y * z) < 0) \\
\text{by} (\text{subst less-iff-diff-less-0}) (\text{simp add: diff-frac-eq})

\text{lemma} \ \text{frac-le-eq:} \\
y \neq 0 \implies z \neq 0 \implies (x / y \leq w / z \iff (x * z - w * y) / (y * z) \leq 0) \\
\text{by} (\text{subst le-iff-diff-le-0}) (\text{simp add: diff-frac-eq})

\text{lemma} \ \text{divide-pos-pos[simp]:} \\
0 < x \implies 0 < y \implies 0 < x / y \\
\text{by(simp add:field-simps)}

\text{lemma} \ \text{divide-nonneg-pos:} \\
0 \leq x \implies 0 < y \implies 0 \leq x / y \\
\text{by(simp add:field-simps)}

\text{lemma} \ \text{divide-neg-pos:} \\
x < 0 \implies 0 < y \implies x / y < 0 \\
\text{by(simp add:field-simps)}

\text{lemma} \ \text{divide-nonpos-pos:} \\
x \leq 0 \implies 0 < y \implies x / y \leq 0 \\
\text{by(simp add:field-simps)}

\text{lemma} \ \text{divide-pos-neg:} \\
0 < x \implies y < 0 \implies x / y < 0
by (simp add: field-simps)

lemma divide-nonneg-neg:
\[ 0 \leq x \Rightarrow y < 0 \Rightarrow x / y \leq 0 \]
by (simp add: field-simps)

lemma divide-neg-neg:
\[ x < 0 \Rightarrow y < 0 \Rightarrow 0 < x / y \]
by (simp add: field-simps)

lemma divide-nonpos-neg:
\[ x \leq 0 \Rightarrow y < 0 \Rightarrow 0 \leq x / y \]
by (simp add: field-simps)

lemma divide-strict-right-mono:
\[
\begin{array}{l}
[a < b; 0 < c] \Rightarrow a / c < b / c \\
\end{array}
\]
by (simp add: less_imp_not_eq divide_inverse mult_strict_right_mono positive_imp_inverse_positive)

lemma divide-strict-right-mono-neg:
assumes \[ b < a \]
shows \[ a / c < b / c \]
proof
  have \[ b / -c < a / -c \]
  by (rule divide-strict-right-mono) (use assms in auto)
  then show ?thesis by (simp add: less_imp_not_eq)
qed

The last premise ensures that \( a \) and \( b \) have the same sign

lemma divide-strict-left-mono:
\[
\begin{array}{l}
[b < a; 0 < c; 0 < a*b] \Rightarrow c / a < c / b \\
\end{array}
\]
by (auto simp: field_simps zero_less_mult_iff mult_strict_right_mono)

lemma divide-left-mono:
\[
\begin{array}{l}
[b \leq a; 0 \leq c; 0 < a*b] \Rightarrow c / a \leq c / b \\
\end{array}
\]
by (auto simp: field_simps zero_less_mult_iff mult_right_mono)

lemma divide-strict-left-mono-neg:
\[
\begin{array}{l}
[a < b; c < 0; 0 < a*b] \Rightarrow c / a < c / b \\
\end{array}
\]
by (auto simp: field_simps zero_less_mult_iff mult_strict_right_mono_neg)

lemma mult_imp_div_pos_le:
\[ 0 < y \Rightarrow x \leq z \Rightarrow y \Rightarrow x / y \leq z \]
by (subst pos_divide_le_eq, assumption+)

lemma mult_imp_le_div_pos:
\[ 0 < y \Rightarrow z * y \leq x \Rightarrow z \leq x / y \]
by (simp add: field_simps)

lemma mult_imp_div_pos_less:
\[ 0 < y \Rightarrow x < z \Rightarrow y \Rightarrow x / y < z \]
by (simp add: field-simps)

lemma mult-imp-less-div-pos: \(0 < y \implies z \cdot y < x \implies z < x / y\)
by (simp add: field-simps)

lemma frac-le:
  assumes \(0 \leq y \cdot x \leq y \cdot 0 < w \cdot w \leq z\)
  shows \(x / z \leq y / w\)
proof (rule mult-imp-div-pos-le)
  show \(z > 0\)
    using assms by simp
  have \(x \leq y \cdot z / w\)
  proof (rule mult-imp-le-div-pos [OF \(0 < w\)])
    show \(x \cdot w \leq y \cdot z\)
      using assms by (auto intro: mult-mono)
  qed
  also have \(\ldots = y / w \cdot z\)
    by simp
  finally show \(x \leq y / w \cdot z\).
qed

lemma frac-less:
  assumes \(0 \leq x \cdot x < y \cdot 0 < w \cdot w \leq z\)
  shows \(x / z < y / w\)
proof (rule mult-imp-div-pos-less)
  show \(z > 0\)
    using assms by simp
  have \(x < y \cdot z / w\)
  proof (rule mult-imp-less-div-pos [OF \(0 < w\)])
    show \(x \cdot w < y \cdot z\)
      using assms by (auto intro: mult-less-imp-less)
  qed
  also have \(\ldots = y / w \cdot z\)
    by simp
  finally show \(x < y / w \cdot z\).
qed

lemma frac-less2:
  assumes \(0 < x \cdot x \leq y \cdot 0 < w \cdot w < z\)
  shows \(x / z < y / w\)
proof (rule mult-imp-div-pos-less)
  show \(z > 0\)
    using assms by simp
  have \(x < y / w \cdot z\)
  proof (rule mult-imp-less-div-pos [OF \(0 < w\)])
    show \(x \cdot w < y \cdot z\)
      using assms by (force intro: mult-imp-less-div-pos mult-le-imp-less)
  qed
qed

lemma less-half-sum: \(a < b \implies a < (a+b) / (1+1)\)
by (simp add: field-simps zero-less-two)
lemma gt-half-sum: \( a < b \implies (a+b)/(1+1) < b \)
  by (simp add: field-simps zero-less-two)

subclass unbounded-dense-linorder
proof
  fix x y :: 'a
  from less-add-one show \( \exists y. x < y \) ..
  from less-add-one have \( x + (-1) < (x + 1) + (-1) \) by (rule add-strict-right-mono)
  then have \( x - 1 < x + 1 - 1 \) by simp
  then have \( x - 1 < x \) by (simp add: algebra-simps)
  then show \( \exists y. y < x \) ..
  show \( x < y \implies \exists z>x. z < y \) by (blast intro: less-half-sum gt-half-sum)
qed

subclass field-abs-sgn ..

lemma inverse-sgn [simp]:
  inverse (sgn a) = sgn a
by (cases a 0 rule: linorder-cases) simp-all

lemma divide-sgn [simp]:
  a / sgn b = a * sgn b
by (cases b 0 rule: linorder-cases) simp-all

lemma nonzero-abs-inverse:
  \( a \neq 0 \implies |inverse a| = inverse |a| \)
by (rule abs-inverse)

lemma nonzero-abs-divide:
  \( b \neq 0 \implies |a / b| = |a| / |b| \)
by (rule abs-divide)

lemma field-le-epsilon:
  assumes e: \( \forall e. 0 < e \implies x \leq y + e \)
  shows \( x \leq y \)
proof (rule dense-le)
  fix t assume \( t < x \)
  hence \( 0 < x - t \) by (simp add: less-diff-eq)
  from e [OF this] have \( x + 0 \leq x + (y - t) \) by (simp add: algebra-simps)
  then have \( 0 \leq y - t \) by (simp only: add-le-cancel-left)
  then show \( t \leq y \) by (simp add: algebra-simps)
qed

lemma inverse-positive-iff-positive [simp]: \( 0 < inverse a \) = \( 0 < a \)
proof (cases a = 0)
  case False
  then show \?thesis
by (blast intro: inverse-positive-imp-positive positive-imp-inverse-positive)
qed auto

lemma inverse-negative-iff-negative [simp]: (inverse a < 0) = (a < 0)
proof (cases a = 0)
  case False
  then show ?thesis
    by (blast intro: inverse-negative-imp-negative negative-imp-inverse-negative)
qed auto

lemma inverse-nonnegative-iff-nonnegative [simp]: 0 ≤ inverse a <-> 0 ≤ a
by (simp add: not-less [symmetric])

lemma inverse-nonpositive-iff-nonpositive [simp]: inverse a ≤ 0 <-> a ≤ 0
by (simp add: not-less [symmetric])

lemma one-less-inverse-iff: 1 < inverse x <-> 0 < x ∧ x < 1
  using less-trans[of 1 x 0 for x]
  by (cases x 0 rule: linorder-cases) (auto simp add: field-simps)

lemma one-le-inverse-iff: 1 ≤ inverse x <-> 0 < x ∧ x ≤ 1
proof (cases x = 1)
  case True
  then show ?thesis
    by simp
next
  case False
  then have inverse x ≠ 1 by simp
  then have 1 ≠ inverse x by blast
  then have 1 ≤ inverse x <-> 1 < inverse x by (simp add: le-less)
  with False show ?thesis
    by (auto simp add: one-less-inverse-iff)
qed

lemma inverse-less-1-iff: inverse x < 1 <-> x ≤ 0 ∨ 1 < x
  by (simp add: not-le [symmetric] one-le-inverse-iff)

lemma inverse-le-1-iff: inverse x ≤ 1 <-> x ≤ 0 ∨ 1 ≤ x
  by (simp add: not-less [symmetric] one-le-inverse-iff)

lemma [field-split-simps, divide-simps]:
  shows le-divide-eq: a ≤ b / c <-> (if 0 < c then a * c ≤ b else if c < 0 then b ≤ a * c else a ≤ 0)
  and divide-le-eq: b / c ≤ a <-> (if 0 < c then b ≤ a * c else if c < 0 then a * c ≤ b else 0 ≤ a)
  and less-divide-eq: a < b / c <-> (if 0 < c then a * c < b else if c < 0 then b < a * c else a < 0)
  and divide-less-eq: b / c < a <-> (if 0 < c then b < a * c else if c < 0 then a * c < b else 0 < a)
  and le-minus-divide-eq: a ≤ - (b / c) <-> (if 0 < c then a * c ≤ - b else if c < 0 then - b ≤ a * c else a ≤ 0)
  and minus-divide-le-eq: - (b / c) ≤ a <-> (if 0 < c then - b ≤ a * c else if c < 0 then a * c ≤ - b else 0 ≤ a)
  and less-minus-divide-eq: a < - (b / c) <-> (if 0 < c then a * c < - b else
if \( c < 0 \) then \(-b < a \cdot c \) else \( a < 0 \)

and minus-divide-less-\( q \): \(-b / c < a \) \( \iff \) (if \( 0 < c \) then \(-b < a \cdot c \)) else \( a < 0 \)

by (auto simp: field-simps not-less dest: order.antisym)

Division and Signs

lemma shows zero-less-divide-iff: \( 0 < a / b \) \( \iff \) \( 0 < a \land 0 < b \lor a < 0 \land b < 0 \)

and divide-less-0-iff: \( a / b < 0 \) \( \iff \) \( 0 < a \land b < 0 \lor a < 0 \land 0 < b \)

and zero-le-divide-iff: \( 0 \leq a / b \) \( \iff \) \( 0 \leq a \land 0 \leq b \lor a \leq 0 \land b \leq 0 \)

and divide-le-0-iff: \( a / b \leq 0 \) \( \iff \) \( 0 \leq a \land b \leq 0 \lor a \leq 0 \land 0 \leq b \)

by (auto simp add: field-split-simps)

Division and the Number One

Simplify expressions equated with 1

lemma zero-eq-1-divide-iff \[ simp \]: \( 0 = 1 / a \) \( \iff \) \( a = 0 \)

by (cases \( a = 0 \)) (auto simp: field-simps)

lemma one-divide-eq-0-iff \[ simp \]: \( 1 / a = 0 \) \( \iff \) \( a = 0 \)

using zero-eq-1-divide-iff \[ of \( a \) \] by simp

Simplify expressions such as \( 0 < 1 / x \) to \( 0 < x \)

lemma zero-le-divide-1-iff \[ simp \]: \( 0 \leq 1 / a \) \( \iff \) \( 0 \leq a \)

by (simp add: zero-le-divide-iff)

lemma zero-less-divide-1-iff \[ simp \]: \( 0 < 1 / a \) \( \iff \) \( 0 < a \)

by (simp add: zero-less-divide-iff)

lemma divide-le-0-1-iff \[ simp \]: \( 1 / a \leq 0 \) \( \iff \) \( a \leq 0 \)

by (simp add: divide-le-0-iff)

lemma divide-less-0-1-iff \[ simp \]: \( 1 / a < 0 \) \( \iff \) \( a < 0 \)

by (simp add: divide-less-0-iff)

lemma divide-right-mono:
\[ [a \leq b; 0 \leq c] \implies a/c \leq b/c \]

by (force simp add: divide-strict-right-mono le-less)

lemma divide-right-mono-neg: \( a \leq b \implies c \leq 0 \implies b / c \leq a / c \)

by (auto dest: divide-right-mono [of \( - - c \)])

lemma divide-left-mono-neg: \( a \leq b \implies c \leq 0 \implies 0 < a \cdot b \implies c / a \leq c / b \)

by (auto simp add: mult.commute dest: divide-left-mono [of \( - - c \)])
**THEORY “Fields”**

**lemma** inverse-le-iff: inverse $a \leq inverse b \iff (0 < a \ast b \rightarrow b \leq a) \land (a \ast b \leq 0 \rightarrow a \leq b)$

by (cases a 0 b 0 rule: linorder-cases[case-product linorder-cases])
(auto simp add: field-simps zero-less-mult-iff mult-le-0-iff)

**lemma** inverse-less-iff: inverse $a < inverse b \iff (0 < a \ast b \rightarrow b < a) \land (a \ast b \leq 0 \rightarrow a < b)$

by (subst less-le) (auto simp add: inverse-le-iff)

**lemma** divide-le-cancel: $a / c \leq b / c \iff (0 < c \rightarrow a \leq b) \land (c < 0 \rightarrow b \leq a)$

by (simp add: divide-inverse mult-le-cancel-right)

**lemma** divide-less-cancel: $a / c < b / c \iff (0 < c \rightarrow a < b) \land (c < 0 \rightarrow b < a) \land c \neq 0$

by (auto simp add: divide-inverse mult-less-cancel-right)

Simplify quotients that are compared with the value 1.

**lemma** le-divide-eq-1:
$(1 \leq b / a) = ((0 < a \land a \leq b) \lor (a < 0 \land b \leq a))$

by (auto simp add: le-divide-eq)

**lemma** divide-le-eq-1:
$(b / a \leq 1) = ((0 < a \land b \leq a) \lor (a < 0 \land a \leq b) \lor a=0)$

by (auto simp add: divide-le-eq)

**lemma** less-divide-eq-1:
$(1 < b / a) = ((0 < a \land a < b) \lor (a < 0 \land b < a))$

by (auto simp add: less-divide-eq)

**lemma** divide-less-eq-1:
$(b / a < 1) = ((0 < a \land b < a) \lor (a < 0 \land a < b) \lor a=0)$

by (auto simp add: divide-less-eq)

**lemma** divide-nonneg-nonneg [simp]:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow 0 \leq x / y$

by (auto simp add: field-split-simps)

**lemma** divide-nonpos-nonpos:
$x \leq 0 \Longrightarrow y \leq 0 \Longrightarrow 0 \leq x / y$

by (auto simp add: field-split-simps)

**lemma** divide-nonneg-nonpos:
$0 \leq x \Longrightarrow y \leq 0 \Longrightarrow x / y \leq 0$

by (auto simp add: field-split-simps)

**lemma** divide-nonpos-nonneg:
$x \leq 0 \Longrightarrow 0 \leq y \Longrightarrow x / y \leq 0$

by (auto simp add: field-split-simps)
Conditional Simplification Rules: No Case Splits

**lemma** le-divide-eq-1-pos [simp]:
\[ 0 < a \Rightarrow (1 \leq b/a) = (a \leq b) \]
by (auto simp add: le-divide-eq)

**lemma** le-divide-eq-1-neg [simp]:
\[ a < 0 \Rightarrow (1 \leq b/a) = (b \leq a) \]
by (auto simp add: le-divide-eq)

**lemma** divide-le-eq-1-pos [simp]:
\[ 0 < a \Rightarrow (b/a \leq 1) = (b \leq a) \]
by (auto simp add: divide-le-eq)

**lemma** divide-le-eq-1-neg [simp]:
\[ a < 0 \Rightarrow (b/a \leq 1) = (b \leq a) \]
by (auto simp add: divide-le-eq)

**lemma** less-divide-eq-1-pos [simp]:
\[ 0 < a \Rightarrow (1 < b/a) = (a < b) \]
by (auto simp add: less-divide-eq)

**lemma** less-divide-eq-1-neg [simp]:
\[ a < 0 \Rightarrow (1 < b/a) = (a < b) \]
by (auto simp add: less-divide-eq)

**lemma** divide-less-eq-1-pos [simp]:
\[ 0 < a \Rightarrow (b/a < 1) = (b < a) \]
by (auto simp add: divide-less-eq)

**lemma** divide-less-eq-1-neg [simp]:
\[ a < 0 \Rightarrow (b/a < 1) = (b < a) \]
by (auto simp add: divide-less-eq)

**lemma** eq-divide-eq-1 [simp]:
\[ b/a = 1 \Rightarrow (1 = b/a) = (a \neq 0 \land a = b) \]
by (auto simp add: eq-divide-eq)

**lemma** divide-eq-eq-1 [simp]:
\[ b/a = 1 \Rightarrow (1 = b/a) = (a \neq 0 \land a = b) \]
by (auto simp add: divide-eq-eq)

**lemma** abs-div-pos: \( 0 < y \Rightarrow |x| / y = |x / y| \)
by (simp add: order-less-imp-le)

**lemma** zero-le-divide-abs-iff [simp]: \( 0 \leq a / |b| = (0 \leq a \lor b = 0) \)
by (auto simp: zero-le-divide-iff)

**lemma** divide-le-0-abs-iff [simp]: \( a / |b| \leq 0 = (a \leq 0 \lor b = 0) \)
by (auto simp: divide-le-0-iff)
lemma field-le-mult-one-interval:
  assumes *: \[ \forall z. [0 < z ; z < 1] \implies z \ast x \leq y \]
  shows x \leq y
proof
  (cases 0 < x)
  assume 0 < x
  thus thesis
  using dense-le-bounded[of 0 1 y/x] *
  unfolding le-divide-eq if-P[OF \[0 < x\]] by simp
next
  assume \neg 0 < x hence x \leq 0 by simp
  obtain s::'a where s: 0 < s s < 1 using dense[of 0 1 :\{a\}] by auto
  hence x \leq s * x using mult-le-cancel-right[of 1 x s \[x \leq 0\]] by auto
  also note *\[OF s\]
  finally show thesis .
qed

For creating values between u and v.

lemma scaling-mono:
  assumes u \leq v 0 \leq r r \leq s
  shows u + r * (v - u) \leq v
proof
  have r/s \leq 1 using assms
  using divide-le-eq-1 by fastforce
  moreover have 0 \leq v - u
    using assms by simp
  ultimately have (r/s) * (v - u) \leq 1 * (v - u)
    by (rule mult-right-mono)
  then show thesis
    by (simp add: field-simps)
qed

end

Min/max Simplification Rules

lemma min-mult-distrib-left:
  fixes x::'a::linordered-idom
  shows p \ast \min x y = (if 0 \leq p then \min (p\ast x) (p\ast y) else \max (p\ast x) (p\ast y))
  by (auto simp add: min-def max-def mult-le-cancel-left)

lemma min-mult-distrib-right:
  fixes x::'a::linordered-idom
  shows \min x y \ast p = (if 0 \leq p then \min (x\ast p) (y\ast p) else \max (x\ast p) (y\ast p))
  by (auto simp add: min-def max-def mult-le-cancel-right)

lemma min-divide-distrib-right:
  fixes x::'a::linordered-field
  shows \min x y / p = (if 0 \leq p then \min (x/p) (y/p) else \max (x/p) (y/p))
  by (simp add: min-mult-distrib-right divide-inverse)
lemma max-mult-distrib-left:
  fixes x::'a::linordered-idom
  shows p * max x y = (if 0 ≤ p then max (p*x) (p*y) else min (p*x) (p*y))
by (auto simp add: min-def max-def mult-le-cancel-left)

lemma max-mult-distrib-right:
  fixes x::'a::linordered-idom
  shows max x y * p = (if 0 ≤ p then max (x*p) (y*p) else min (x*p) (y*p))
by (auto simp add: min-def max-def mult-le-cancel-right)

lemma max-divide-distrib-right:
  fixes x::'a::linordered-field
  shows max x y / p = (if 0 ≤ p then max (x/p) (y/p) else min (x/p) (y/p))
by (simp add: max-mult-distrib-right divide_inverse)

hide-fact (open) field-inverse field-divide-inverse field-inverse-zero

code-identifier
  code-module Fields ↦ (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

19 Relations – as sets of pairs, and binary predic-ates

theory Relation
  imports Product-Type Sum-Type Fields
begin

A preliminary: classical rules for reasoning on predicates

declare predicate1I [Pure.intro, intro]
declare predicate1D [Pure.dest, dest]
declare predicate2I [Pure.intro, intro]
declare predicate2D [Pure.dest, dest]
declare bot1E [elim!]
declare bot2E [elim!]
declare top1I [intro!]
declare top2I [intro!]
declare inf1I [intro!]
declare inf2I [intro!]
declare inf1E [elim!]
declare inf2E [elim!]
declare sup1I1 [intro?]
declare sup2I1 [intro?]
declare sup1I2 [intro?]
declare sup2I2 [intro?]
declare sup1E [elim!]

declare sup2E [elim!]
declare sup1CI [intro]
declare sup2CI [intro]
declare Inf1-I [intro]
declare INF1-I [intro]
declare Inf2-I [intro]
declare INF2-I [intro]
declare Inf1-D [elim]
declare INF1-D [elim]
declare Inf2-D [elim]
declare INF2-D [elim]
declare Inf1-E [elim]
declare INF1-E [elim]
declare Inf2-E [elim]
declare INF2-E [elim]
declare Sup1-I [intro]
declare SUP1-I [intro]
declare Sup2-I [intro]
declare SUP2-I [intro]
declare Sup1-E [elim!]
declare SUP1-E [elim!]
declare Sup2-E [elim!]
declare SUP2-E [elim!]

19.1 Fundamental

19.1.1 Relations as sets of pairs

type-synonym 'a rel = ('a × 'a) set

lemma subrelI: (∀x y. (x, y) ∈ r → (x, y) ∈ s) → r ⊆ s
— Version of subsetI for binary relations
by auto

lemma lfp-induct2:
(a, b) ∈ lfp f → mono f →
(∀a b. (a, b) ∈ f (lfp f ∩ {(x, y). P x y}) → P a b) → P a b
— Version of lfp-induct for binary relations
using lfp-induct-set [of (a, b) f case-prod P] by auto

19.1.2 Conversions between set and predicate relations

lemma pred-equals-eq [pred-set-cons]: (λx. x ∈ R) = (λx. x ∈ S) ↔ R = S
by (simp add: set-eq-iff fun-eq-iff)

lemma pred-equals-eq2 [pred-set-cone]: (λx y. (x, y) ∈ R) = (λx y. (x, y) ∈ S)
↔ R = S
by (simp add: set-eq-iff fun-eq-iff)

lemma pred-subset-eq [pred-set-cons]: (λx. x ∈ R) ≤ (λx. x ∈ S) ↔ R ⊆ S
by (simp add: subset-iff le-fun-def)

lemma pred-subset-eq2 [pred-set-conv]: \((\lambda x\ y. (x, y) \in R) \leq (\lambda x\ y. (x, y) \in S)\) 
\(\iff R \subseteq S\)
by (simp add: subset-iff le-fun-def)

lemma bot-empty-eq [pred-set-conv]: \(\bot = (\lambda x. x \in \{\})\)
by (auto simp add: fun-eq-iff)

lemma bot-empty-eq2 [pred-set-conv]: \(\bot = (\lambda x\ y. (x, y) \in \{\})\)
by (auto simp add: fun-eq-iff)

lemma top-empty-eq: \(\top = (\lambda x. x \in UNIV)\)
by (auto simp add: fun-eq-iff)

lemma top-empty-eq2: \(\top = (\lambda x\ y. (x, y) \in UNIV)\)
by (auto simp add: fun-eq-iff)

lemma inf-Int-eq [pred-set-conv]: \((\lambda x. x \in R) \cap (\lambda x. x \in S) = (\lambda x. x \in R \cap S)\)
by (simp add: inf-fun-def)

lemma inf-Int-eq2 [pred-set-conv]: \((\lambda x\ y. (x, y) \in R) \cap (\lambda x\ y. (x, y) \in S) = (\lambda x\ y. (x, y) \in R \cap S)\)
by (simp add: inf-fun-def)

lemma sup-Un-eq [pred-set-conv]: \((\lambda x. x \in R) \cup (\lambda x. x \in S) = (\lambda x. x \in R \cup S)\)
by (simp add: sup-fun-def)

lemma sup-Un-eq2 [pred-set-conv]: \((\lambda x\ y. (x, y) \in R) \cup (\lambda x\ y. (x, y) \in S) = (\lambda x\ y. (x, y) \in R \cup S)\)
by (simp add: sup-fun-def)

lemma INF-INT-eq [pred-set-conv]: \((\prod i \in S. (\lambda x. x \in r\ i)) = (\lambda x. x \in (\prod i \in S. r\ i))\)
by (simp add: fun-eq-iff)

lemma INF-INT-eq2 [pred-set-conv]: \((\prod i \in S. (\lambda x\ y. (x, y) \in r\ i)) = (\lambda x\ y. (x, y) \in (\prod i \in S. r\ i))\)
by (simp add: fun-eq-iff)

lemma SUP-UN-eq [pred-set-conv]: \((\bigcup i \in S. (\lambda x. x \in r\ i)) = (\lambda x. x \in (\bigcup i \in S. r\ i))\)
by (simp add: fun-eq-iff)

lemma SUP-UN-eq2 [pred-set-conv]: \((\bigcup i \in S. (\lambda x\ y. (x, y) \in r\ i)) = (\lambda x\ y. (x, y) \in (\bigcup i \in S. r\ i))\)
by (simp add: fun-eq-iff)

lemma Inf-INT-eq [pred-set-conv]: \(\prod S = (\lambda x. x \in (\prod (Collect i S)))\)
by (simp add: fun-eq-iff)

lemma INF-Int-eq [pred-set-conv]: \( (\prod_{i \in S} (\lambda x. x \in i)) = (\lambda x. x \in \bigcap S) \)
by (simp add: fun-eq-iff)

lemma Inf-INT-eq2 [pred-set-conv]: \( d \in S \Rightarrow (\lambda x y. (x, y) \in i) = (\lambda x y. (x, y) \in \bigcap S) \)
by (simp add: fun-eq-iff)

lemma INF-Int-eq2 [pred-set-conv]: \( (\prod_{i \in S} (\lambda x y. (x, y) \in i)) = (\lambda x y. (x, y) \in \bigcap S) \)
by (simp add: fun-eq-iff)

lemma Sup-SUP-eq [pred-set-conv]: \( \bigsqcup S = (\lambda x. x \in \bigcup (\text{Collect ' case-prod ' } S)) \)
by (simp add: fun-eq-iff)

lemma SUP-Sup-eq [pred-set-conv]: \( \bigsqcup i \in S. (\lambda x. x \in i) = (\lambda x. x \in \bigcup S) \)
by (simp add: fun-eq-iff)

lemma Sup-SUP-eq2 [pred-set-conv]: \( \bigsqcup S = (\lambda x y. (x, y) \in \bigcup (\text{Collect ' case-prod ' } S)) \)
by (simp add: fun-eq-iff)

lemma SUP-Sup-eq2 [pred-set-conv]: \( \bigsqcup i \in S. (\lambda x y. (x, y) \in i) = (\lambda x y. (x, y) \in \bigcup S) \)
by (simp add: fun-eq-iff)

19.2 Properties of relations

19.2.1 Reflexivity

definition refl-on :: 'a set \Rightarrow 'a rel \Rightarrow bool
where refl-on A r \iff r \subseteq A \times A \land (\forall x \in A. (x, x) \in r)

abbreviation refl :: 'a rel \Rightarrow bool — reflexivity over a type
where refl \equiv refl-on UNIV

definition reflp-on :: 'a set \Rightarrow ('a \Rightarrow bool) \Rightarrow bool
where reflp-on A R \iff (\forall x \in A. R x x)

abbreviation reflp :: ('a \Rightarrow bool) \Rightarrow bool
where reflp \equiv reflp-on UNIV

lemma reflp-def [no-atp]: reflp R \iff (\forall x. R x x)
by (simp add: reflp-on-def)

reflp-def is for backward compatibility.

lemma reflp-refl-eq [pred-set-conv]: reflp (\lambda x y. (x, y) \in i) \iff refl r
by (simp add: refl-on-def reflp-def)
lemma refl-onI [intro?]: \( r \subseteq A \times A \implies (\forall x. x \in A \implies (x, x) \in r) \implies \text{refl-on } A \ r \)
  unfolding refl-on-def by (iprover intro: ballI)

lemma reflI: \( (\forall x. (x, x) \in r) \implies \text{refl } r \)
  by (auto intro: refl-onI)

lemma reflp-onI:
  \( (\forall x. x \in A \implies R x x) \implies \text{reflp-on } A \ R \)
  by (simp add: reflp-on-def)

lemma reflpI [intro?]: \( (\forall x. R x x) \implies \text{reflp } R \)
  by (rule reflp-onI)

lemma refl-onD: \( \text{refl-on } A \ r \implies a \in A \implies (a, a) \in r \)
  unfolding refl-on-def by blast

lemma refl-onD1: \( \text{refl-on } A \ r \implies (x, y) \in r \implies x \in A \)
  unfolding refl-on-def by blast

lemma refl-onD2: \( \text{refl-on } A \ r \implies (x, y) \in r \implies y \in A \)
  unfolding refl-on-def by blast

lemma reflD: \( \text{refl } r \implies (a, a) \in r \)
  unfolding refl-on-def by blast

lemma reflp-onD:
  \( \text{reflp-on } A \ R \implies x \in A \implies R x x \)
  by (simp add: reflp-on-def)

lemma reflpD[dest?]: \( \text{reflp } R \implies R x x \)
  by (simp add: reflp-onD)

lemma reflpE:
  assumes \( \text{reflp } r \)
  obtains \( r x x \)
  using assms by (auto dest: refl-onD simp add: reflp-def)

lemma reflp-on-subset: \( \text{reflp-on } A \ R \implies B \subseteq A \implies \text{reflp-on } B \ R \)
  by (auto intro: reflp-onI dest: reflp-onD)

lemma reflp-on-image: \( \text{reflp-on } (f \cdot A) \ R \leftarrow\rightarrow \text{reflp-on } A \ (\lambda a. R (f a) (f b)) \)
  by (simp add: reflp-on-def)

lemma refl-on-Int: \( \text{refl-on } A \ r \implies \text{refl-on } B \ s \implies \text{refl-on } (A \cap B) \ (r \cap s) \)
  unfolding refl-on-def by blast

lemma reflp-on-inf: \( \text{reflp-on } A \ R \implies \text{reflp-on } B \ S \implies \text{reflp-on } (A \cap B) \ (R \cap S) \)
  by (auto intro: reflp-onI dest: reflp-onD)
lemma reflp-inf: reflp r \implies reflp s \implies reflp (r \sqcap s)
  by (rule reflp-on-inf[of UNIV - UNIV, unfolded Int-absorb])

lemma refl-on-Un: refl-on A r \implies refl-on B s \implies refl-on (A \cup B) (r \cup s)
  unfolding refl-on-def by blast

lemma reflp-on-sup: reflp-on A R \implies reflp-on B S \implies reflp-on (A \cup B) (R \sqcup S)
  by (auto intro: reflp-onI dest: reflp-onD)

lemma reflp-on-empty [simp]: reflp-on {} R
  by (simp add: reflp-on-def)

lemma refl-on-singleton [simp]: refl-on {x} \{x, x\}
  by (blast intro: refl-onI)

lemma reflp-on-equality [simp]: reflp-on A (=)
  by (simp add: reflp-on-def)

lemma reflp-on-mono:
  reflp-on A R \implies (\forall x. y \in r. x \in A \land y \in A) \land (\forall x \in A. (x, x) \in r)
  by (auto intro: refl-onI dest: reflp-onD reflp-onD1 reflp-onD2)

lemma reflp-on-equality [simp]: reflp-on A (=)
  by (simp add: reflp-on-def)

lemma reflp-on-monotone:
  reflp-on A R \implies (\forall x. y \in r. x \in A \implies y \in A \implies R x y \implies Q x y) \implies reflp-on A Q
  by (auto intro: reflp-onI dest: reflp-onD)
lemma reflp-mono: \( \text{reflp } R \implies (\forall x y. R x y \implies Q x y) \implies \text{reflp } Q \)
by (rule reflp-on-mono[of UNIV \( R \) \( Q \)]) simp-all

lemma (in preorder) reflp-on-le[simp]: \( \text{reflp-on } A \ (\leq) \)
by (simp add: reflp-onI)

lemma (in preorder) reflp-on-ge[simp]: \( \text{reflp-on } A \ (\geq) \)
by (simp add: reflp-onI)

19.2.2 Irreflexivity

definition irrefl-on :: \( 'a \) set \( \Rightarrow \) \( 'a \) rel \( \Rightarrow \) bool where
\( \text{irrefl-on } A \ r \iff (\forall a \in A. (a, a) \not\in r) \)

abbreviation irrefl :: \( 'a \) rel \( \Rightarrow \) bool where
\( \text{irrefl} \equiv \text{irrefl-on } \text{UNIV} \)

definition irreflp-on :: \( 'a \) set \( \Rightarrow \) \((\ 'a \Rightarrow \ 'a \Rightarrow \) bool) \( \Rightarrow \) bool where
\( \text{irreflp-on } A \ R \iff (\forall a \in A. \neg R a a) \)

abbreviation irreflp :: \( 'a \Rightarrow \ 'a \Rightarrow \) bool where
\( \text{irreflp} \equiv \text{irreflp-on } \text{UNIV} \)

lemma irrefl-def[no-atp]: \( \text{irrefl } r \iff (\forall a. (a, a) \not\in r) \)
by (simp add: irrefl-on-def)

lemma irreflp-def[no-atp]: \( \text{irreflp } R \iff (\forall a. \neg R a a) \)
by (simp add: irreflp-on-def)

irrefl-def and irreflp-def are for backward compatibility.

lemma irreflp-on-irrefl-on-eq [pred-set-conv]: \( \text{irreflp-on } A \ (\lambda a b. (a, b) \in r) \iff \text{irrefl-on } A \ r \)
by (simp add: irrefl-on-def irreflp-on-def)

lemmas irreflp-irrefl-eq = irreflp-on-irrefl-on-eq[of UNIV]

lemma irrefl-onI: \( (\lambda a. a \in A \implies (a, a) \not\in r) \implies \text{irrefl-on } A \ r \)
by (simp add: irrefl-on-def)

lemma irreflI[intro?]: \( (\lambda a. (a, a) \not\in r) \implies \text{irrefl } r \)
by (rule irrefl-onI[of UNIV , simplified])

lemma irreflp-onI: \( (\lambda a. a \in A \implies \neg R a a) \implies \text{irreflp-on } A \ R \)
by (rule irreflp-onI[to-pred])

lemma irreflpI[intro?]: \( (\lambda a. \neg R a a) \implies \text{irreflp } R \)
by (rule irreflpI[to-pred])
lemma irrefl-onD: irrefl-on A r \Longrightarrow a \in A \Longrightarrow (a, a) \notin r
by (simp add: irrefl-on-def)

lemma irreflD: irrefl r \Longrightarrow (x, x) \notin r
by (rule irrefl-onD[of UNIV, simplified])

lemma irreflp-onD: irreflp-on A R \Longrightarrow a \in A \Longrightarrow \neg R a a
by (rule irrefl-onD[to-pred])

lemma irreflpD: irreflp R \Longrightarrow \neg R x x
by (rule irreflD[to-pred])

lemma irrefl-on-distinct[code]:
irrefl-on A r \iff (\forall (a, b) \in r. a \in A \rightarrow b \in A \rightarrow a \neq b)
by (auto simp add: irrefl-on-def)

lemmas irrefl-distinct = irrefl-distinct — For backward compatibility

lemma irrefl-on-subset: irrefl-on A r \Longrightarrow B \subseteq A \Longrightarrow irrefl-on B r
by (auto simp: irrefl-on-def)

lemma irreflp-on-subset: irreflp-on A R \Longrightarrow B \subseteq A \Longrightarrow irreflp-on B R
by (auto simp: irreflp-on-def)

lemma irreflp-on-image: irreflp-on (f ' A) R \iff irreflp-on A (\lambda a b. R (f a) (f b))
by (simp add: irreflp-on-def)

lemma (in preorder) irreflp-on-less[simp]: irreflp-on A (<)
by (simp add: irreflp-onI)

lemma (in preorder) irreflp-on-greater[simp]: irreflp-on A (>)
by (simp add: irreflp-onI)

19.2.3 Asymmetry

definition asym-on :: 'a set \Rightarrow 'a rel \Rightarrow bool where
asym-on A r \iff (\forall x \in A. \forall y \in A. (x, y) \in r \rightarrow (y, x) \notin r)

abbreviation asym :: 'a rel \Rightarrow bool where
asym \equiv asym-on UNIV

definition asymp-on :: 'a set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool where
asymp-on A R \iff (\forall x \in A. \forall y \in A. R x y \rightarrow \neg R y x)

abbreviation asymp :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool where
asymp \equiv asymp-on UNIV

lemma asymp-on-asym-on-eq[pred-set-cone]: asymp-on A (\lambda x y. (x, y) \in r) \iff asym-on A r
lemmas asymp-asym-eq = asymp-on-asym-on-eq[of UNIV] — For backward compatibility

lemma asym-onI[intro]: \( (\forall x y. x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \notin r) \implies \text{asymp-on } A \ R \)  
by (simp add: asym-on-def)

lemma asymI[intro]: \( (\forall x y. (x, y) \in r \implies (y, x) \notin r) \implies \text{asym } r \)  
by (simp add: asym-onI)

lemma asymp-onI[intro]: \( (\forall x y. x \in A \implies y \in A \implies R x y \implies \neg R y x) \implies \text{asymp-on } A \ R \)  
by (rule asym-onI[to-pred])

lemma asym-onD: \( \text{asymp-on } A \ R \implies x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \notin r \)  
by (simp add: asym-on-def)

lemma asymD: \( \text{asym } r \implies (x, y) \in r \implies (y, x) \notin r \)  
by (simp add: asym-onD)

lemma asymp-onD: \( \text{asymp-on } A \ R \implies x \in A \implies y \in A \implies R x y \implies \neg R y x \)  
by (rule asym-onD[to-pred])

lemma asym-if: \( \text{asym } r \iff (\forall x y. (x, y) \in r \implies (y, x) \notin r) \)  
by (blast dest: asymD)

lemma asym-on-subset: \( \text{asymp-on } A \ R \implies B \subseteq A \implies \text{asymp-on } B \ R \)  
by (auto simp: asym-on-def)

lemma asym-on-subset: \( \text{asymp-on } A \ R \implies B \subseteq A \implies \text{asymp-on } B \ R \)  
by (auto simp: asym-on-def)

lemma asym-on-image: \( \text{asymp-on } (f \ A) \ R \iff \text{asymp-on } A \ (\lambda a b. R (f a) (f b)) \)  
by (simp add: asym-on-def)

lemma irrefl-on-if-asymp-on[simp]: \( \text{asymp-on } A \ R \implies \text{irrefl-on } A \ R \)  
by (auto intro: irrefl-onI dest: asym-onD)

lemma irrefl-on-if-asymp-on[simp]: \( \text{asymp-on } A \ R \implies \text{irrefl-on } A \ R \)
by (rule irrefl-on-if-asym-on[to-pred])

lemma (in preorder) asymp-on-less[simp]: asymp-on A (<)
  by (auto intro: dual-order.asym)

lemma (in preorder) asymp-on-greater[simp]: asymp-on A (>)
  by (auto intro: dual-order.asym)

19.2.4 Symmetry

definition sym-on :: 'a set ⇒ 'a rel ⇒ bool where
  sym-on A r ⟷ (∀ x ∈ A. ∀ y ∈ A. (x, y) ∈ r ⟷ (y, x) ∈ r)

abbreviation sym :: 'a rel ⇒ bool where
  sym ≡ sym-on UNIV

definition symp-on :: 'a set ⇒ ('a ⇒ 'a ⇒ bool) ⇒ bool where
  symp-on A R ⟷ (∀ x ∈ A. ∀ y ∈ A. R x y ⟷ R y x)

abbreviation symp :: ('a ⇒ 'a ⇒ bool) ⇒ bool where
  symp ≡ symp-on UNIV

lemma sym-def [no-atp]: sym r ⟷ (∀ x y. (x, y) ∈ r ⟷ (y, x) ∈ r)
  by (simp add: sym-on-def)

lemma symp-def [no-atp]: symp R ⟷ (∀ x y. R x y ⟷ R y x)
  by (simp add: symp-on-def)

sym-def and symp-def are for backward compatibility.

lemma symp-on-sym-on-eq[pred-set-conv]: symp-on A (λx y. (x, y) ∈ r) ⟷ symp-on A r
  by (simp add: sym-on-def symp-on-def)

lemmas symp-sym-eq = symp-on-sym-on-eq[of UNIV] — For backward compatibility

lemma sym-on-subset: sym-on A r ⟷ B ⊆ A ⟹ sym-on B r
  by (auto simp: sym-on-def)

lemma symp-on-subset: symp-on A R ⟷ B ⊆ A ⟹ symp-on B R
  by (auto simp: sym-on-def)

lemma symp-on-image: symp-on (f ' A) R ⟷ symp-on A (λa b. R (f a) (f b))
  by (simp add: sym-on-def)

lemma sym-onI: (∀ x y. x ∈ A ⟹ y ∈ A ⟹ (x, y) ∈ r ⟹ (y, x) ∈ r) ⟹ symp-on A r
  by (simp add: sym-on-def)
lemma symI [intro?]: \( \forall x y. (x, y) \in r \Rightarrow (y, x) \in r \Rightarrow \text{sym } r \)
by (simp add: sym-onI)

lemma symp-onI: \( \forall x y. x \in A \Rightarrow y \in A \Rightarrow R x y \Rightarrow R y x \) \Rightarrow symp-on A R
by (rule sym-onI[to-pred])

lemma symP [intro?]: \( \forall x y. R x y \Rightarrow R y x \) \Rightarrow symp R
by (rule symI[to-pred])

lemma symE:
assumes \( \text{sym } r \) and \( (b, a) \in r \)
obtains \( (a, b) \in r \)
using assms by (simp add: sym-def)

lemma sympE:
assumes \( \text{symp } r \) and \( r b a \)
obtains \( r a b \)
using assms by (rule symE[to-pred])

lemma sym-onD: sym-on A r \Rightarrow x \in A \Rightarrow y \in A \Rightarrow (x, y) \in r \Rightarrow (y, x) \in r
by (simp add: sym-on-def)

lemma symD [dest?]: sym r \Rightarrow (x, y) \in r \Rightarrow (y, x) \in r
by (simp add: sym-onD)

lemma symp-onD: symp-on A R \Rightarrow x \in A \Rightarrow y \in A \Rightarrow R x y \Rightarrow R y x
by (rule sym-onD[to-pred])

lemma sympD [dest?]: symp R \Rightarrow R x y \Rightarrow R y x
by (rule symD[to-pred])

lemma sym-Int: sym r \Rightarrow sym s \Rightarrow sym (r \cap s)
by (fast intro: symI elim: symE)

lemma symp-inf: symp r \Rightarrow symp s \Rightarrow symp (r \cap s)
by (fact sym-Int[to-pred])

lemma symp-Un: symp r \Rightarrow symp s \Rightarrow symp (r \cup s)
by (fast intro: symI elim: symE)

lemma symp-sup: symp r \Rightarrow symp s \Rightarrow symp (r \sqcup s)
by (fact sym-Un[to-pred])

lemma sym-INTER: \( \forall x \in S. \text{sym } (r x) \Rightarrow \text{sym } (\bigcap (r ' S)) \)
by (fast intro: symI elim: symE)

lemma symp-INF: \( \forall x \in S. \text{symp } (r x) \Rightarrow \text{symp } (\bigcap (r ' S)) \)
by (fact sym-INTER[to-pred])
lemma sym-UNION: \( \forall x \in S. \; \text{sym} \; (r \; x) \implies \text{sym} \; (\bigcup (r \; ' \; S)) \)
by (fast intro: symI elim: symE)

lemma symp-SUP: \( \forall x \in S. \; \text{symp} \; (r \; x) \implies \text{symp} \; (\bigcup (r \; ' \; S)) \)
by (fact sym-UNION [to-pred])

19.2.5 Antisymmetry

definition antisym-on :: 'a set \Rightarrow 'a rel \Rightarrow bool where
antisym-on A r \iff \( \forall x \in A. \forall y \in A. \; (x, y) \in r \implies (y, x) \in r \implies x = y \)

abbreviation antisym :: 'a rel \Rightarrow bool where
antisym \equiv antisym-on UNIV

definition antisymp-on :: 'a set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool where
antisymp-on A R \iff \( \forall x \in A. \forall y \in A. \; R \; x \; y \implies R \; y \; x \implies x = y \)

abbreviation antisymp :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool where
antisymp \equiv antisymp-on UNIV

lemma antisym-def [no-atp]: antisym r \iff \( \forall x \; y. \; (x, y) \in r \implies (y, x) \in r \implies x = y \)
by (simp add: antisym-on-def)

lemma antisymp-def [no-atp]: antisymp R \iff \( \forall x \; y. \; R \; x \; y \implies R \; y \; x \implies x = y \)
by (simp add: antisymp-on-def)

antisym-def and antisymp-def are for backward compatibility.

lemma antisym-on-antisym-on-eq[pred-set-conv]:
antisymp-on A (\( \lambda x \; y. (x, y) \in r \)) \iff antisym-on A r
by (simp add: antisym-on-def antisymp-on-def)

lemmas antisym-antisym-eq = antisym-on-antisym-on-eq[of UNIV] — For backward compatibility

lemma antisym-on-subset: antisym-on A r \Longrightarrow B \subseteq A \Longrightarrow antisym-on B r
by (auto simp: antisym-on-def)

lemma antisym-on-subset: antisym-on A R \Longrightarrow B \subseteq A \Longrightarrow antisym-on B R
by (auto simp: antisym-on-def)

lemma antisymp-on-image:
assumes inj-on f A
shows antisym-on (f \; ' \; A) R \iff antisym-on A (\( \lambda a \; b. \; R \; (f \; a) \; (f \; b) \))
using assms by (auto simp: antisym-on-def inj-on-def)

lemma antisym-onI:
\[(\forall x \in A \rightarrow y \in A \rightarrow (x, y) \in r \rightarrow (y, x) \in r \rightarrow x = y) \rightarrow \text{antisym-on } A r\]

unfolding \textit{antisym-on-def} by simp

\textbf{lemma} \textit{antisymI} [intro]:
\[(\forall x y. (x, y) \in r \rightarrow (y, x) \in r \rightarrow x = y) \rightarrow \text{antisym } r\]

by (simp add: antisym-onI)

\textbf{lemma} \textit{antisymp-onI}:
\[(\forall x y. x \in A \rightarrow y \in A \rightarrow R x y \rightarrow R y x \rightarrow x = y) \rightarrow \text{antisym-on } A R\]

by (rule antisym-onI[to-pred])

\textbf{lemma} \textit{antisymp-onI} [intro]:
\[(\forall x y. R x y \rightarrow R y x \rightarrow x = y) \rightarrow \text{antisymp } R\]

by (rule antisymI[to-pred])

\textbf{lemma} \textit{antisym-onD}:
\[\text{antisym-on } A r \rightarrow x \in A \rightarrow y \in A \rightarrow (x, y) \in r \rightarrow (y, x) \in r \rightarrow x = y\]

by (simp add: antisym-on-def)

\textbf{lemma} \textit{antisymD} [dest]:
\[\text{antisym } r \rightarrow (x, y) \in r \rightarrow (y, x) \in r \rightarrow x = y\]

by (simp add: antisym-onD)

\textbf{lemma} \textit{antisymp-onD}:
\[\text{antisym-on } A R \rightarrow x \in A \rightarrow y \in A \rightarrow R x y \rightarrow R y x \rightarrow x = y\]

by (rule antisym-onD[to-pred])

\textbf{lemma} \textit{antisymp-onD} [dest]:
\[\text{antisymp } R \rightarrow R x y \rightarrow R y x \rightarrow x = y\]

by (rule antisymD[to-pred])

\textbf{lemma} \textit{antisym-subset}:
\[r \subseteq s \rightarrow \text{antisym } s \rightarrow \text{antisym } r\]

unfolding \textit{antisym-def} by blast

\textbf{lemma} \textit{antisymp-less-eq}:
\[r \leq s \rightarrow \text{antisymp } s \rightarrow \text{antisymp } r\]

by (fact antisym-subset [to-pred])

\textbf{lemma} \textit{antisym-empty} [simp]:
\[\text{antisym } \{}\]

unfolding \textit{antisym-def} by blast

\textbf{lemma} \textit{antisymp-bot} [simp]:
\[\text{antisymp } \bot\]

by (fact antisym-empty [to-pred])

\textbf{lemma} \textit{antisymp-equality} [simp]:

antisym 
  by (auto intro: antisymI)

lemma antisym-singleton [simp]:
  antisym \{x\}
  by (blast intro: antisymI)

lemma antisym-on-if-asym-on: asymp-on A r \implies antisym-on A r
  by (rule antisym-on-if-asym-on[to-pred])

lemma (in preorder) antisym-on-less[simp]: antisym-on A (<)
  by (rule antisym-on-if-asym-on[OF asymp-on-less])

lemma (in preorder) antisym-on-greater[simp]: antisym-on A (>)
  by (rule antisym-on-if-asym-on[OF asymp-on-greater])

lemma (in order) antisym-on-le[simp]: antisym-on A (\leq)
  by (simp add: antisym-onI)

lemma (in order) antisym-on-ge[simp]: antisym-on A (\geq)
  by (simp add: antisym-onI)

19.2.6 Transitivity

definition trans-on :: 'a set \Rightarrow 'a rel \Rightarrow bool where
  trans-on A r \iff (\forall x \in A. \forall y \in A. \forall z \in A. (x, y) \in r \implies (y, z) \in r \implies (x, z) \in r)

abbreviation trans :: 'a rel \Rightarrow bool where
  trans \equiv trans-on UNIV

definition transp-on :: 'a set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool where
  transp-on A R \iff (\forall x \in A. \forall y \in A. \forall z \in A. R x y \implies R y z \implies R x z)

abbreviation transp :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool where
  transp \equiv transp-on UNIV

lemma trans-def[no-atp]: trans r \iff (\forall x y z. (x, y) \in r \implies (y, z) \in r \implies (x, z) \in r)
  by (simp add: trans-on-def)

lemma transp-def: transp R \iff (\forall x y z. R x y \implies R y z \implies R x z)
  by (simp add: transp-on-def)

trans-def and transp-def are for backward compatibility.

lemma transp-on-trans-on-eq[pred-set-cone]: transp-on A (\lambda x y. (x, y) \in r) \iff
trans-on A r
  by (simp add: trans-on-def transp-on-def)

lemmas transp-trans-eq = transp-on-trans-on-eq[of UNIV] — For backward compatibility

lemma trans-onI:
  \( (\forall x y z. x \in A \implies y \in A \implies z \in A \implies (x, y) \in r \implies (y, z) \in r \implies (x, z) \in r) \implies \)
  trans-on A r
  unfolding trans-on-def
  by (intro ballI) iprover

lemma transI [intro?: (\forall x y z. (x, y) \in r \implies (y, z) \in r \implies (x, z) \in r) \implies \]
  trans r
  by (rule trans-onI)

lemma transp-onI:
  \( (\forall x y z. x \in A \implies y \in A \implies z \in A \implies R x y \implies R y z \implies R x z) \implies \)
  transp-on A R
  by (rule trans-onI[to-pred])

lemma transpI [intro?: (\forall x y z. R x y \implies R y z \implies R x z) \implies transp R]
  by (rule transI[to-pred])

lemma transE:
  assumes trans r and (x, y) \in r and (y, z) \in r
  obtains (x, z) \in r
  using assms by (unfold trans-def) iprover

lemma transpE:
  assumes transp r and r x y and r y z
  obtains r x z
  using assms by (rule transE[to-pred])

lemma trans-onD:
  trans-on A r \implies x \in A \implies y \in A \implies z \in A \implies (x, y) \in r \implies (y, z) \in r \implies
  (x, z) \in r
  unfolding trans-on-def
  by (elim ballE) iprover+

lemma transD[dest?: trans r \implies (x, y) \in r \implies (y, z) \in r \implies (x, z) \in r]
  by (simp add: trans-onD[of UNIV r x y z])

lemma transp-onD: transp-on A R \implies x \in A \implies y \in A \implies z \in A \implies R x y
  \implies R y z \implies R x z
  by (rule trans-onD[to-pred])

lemma transpD[dest?: transp R \implies R x y \implies R y z \implies R x z]
by (rule transD[to-pred])

**lemma** trans-on-subset: trans-on A r \(\Rightarrow\) B \(\subseteq\) A \(\Rightarrow\) trans-on B r
by (auto simp: trans-on-def)

**lemma** transp-on-subset: transp-on A R \(\Rightarrow\) B \(\subseteq\) A \(\Rightarrow\) transp-on B R
by (auto simp: transp-on-def)

**lemma** transp-on-image: transp-on (f ' A) R \(\iff\) transp-on A (\(\lambda a b. R (f a) (f b)\))
by (simp add: transp-on-def)

**lemma** trans-Int: trans r \(\Rightarrow\) trans s \(\Rightarrow\) trans (r \(\cap\) s)
by (fast intro: transI elim: transE)

**lemma** transp-inf: transp r \(\Rightarrow\) transp s \(\Rightarrow\) transp (r \(\sqcap\) s)
by (fact trans-Int [to-pred])

**lemma** transp-INTER: \(\forall x \in S.\) trans (r x) \(\Rightarrow\) trans (\(\bigcap\) (r ' S))
by (fast intro: transI elim: transD)

**lemma** transp-INF: \(\forall x \in S.\) transp (r x) \(\Rightarrow\) transp (\(\bigcap\) (r ' S))
by (fact trans-INTER [to-pred])

**lemma** trans-on-join [code]:
trans-on A r \(\iff\) \(\forall (x, y1) \in r.\) \(x \in A \Rightarrow y1 \in A \Rightarrow\)
(\(\forall (y2, z) \in r.\) \(y1 = y2 \Rightarrow z \in A \Rightarrow (x, z) \in r\))
by (auto simp: trans-on-def)

**lemma** trans-join: trans r \(\iff\) \(\forall (x, y1) \in r.\) \(\forall (y2, z) \in r.\) \(y1 = y2 \Rightarrow (x, z) \in r\)
by (auto simp add: trans-def)

**lemma** transp-trans: transp r \(\iff\) trans (\(\{ (x, y). \; r x y \}\))
by (simp add: trans-def transp-def)

**lemma** transp-equality [simp]: transp (=)
by (auto intro: transpI)

**lemma** trans-empty [simp]: trans \(\{\}\)
by (blast intro: transI)

**lemma** transp-empty [simp]: transp (\(\lambda x y. False\))
using trans-empty[to-pred] by (simp add: bot-fun-def)

**lemma** trans-singleton [simp]: trans \(\{ (a, a)\}\)
by (blast intro: transI)

**lemma** transp-singleton [simp]: transp (\(\lambda x y. x = a \land y = a\))
THEORY “Relation” 461

by (simp add: transp-def)

lemma asym-on-iff-irrefl-on-if-trans-on: trans-on A r → asym-on A r ←→ irrefl-on A r
  by (auto intro: irrefl-on-if-asym-on dest: trans-onD irrefl-onD)

lemma asym-on-iff-irreflp-on-if-transp-on: transp-on A R → asymp-on A R ←→ irreflp-on A R
  by (rule asym-on-iff-irrefl-on-if-trans-on[to-pred])

lemma (in preorder) transp-on-le[simp]: transp-on A (≤)
  by (auto intro: transp-onI order-trans)

lemma (in preorder) transp-on-less[simp]: transp-on A (<)
  by (auto intro: transp-onI less-trans)

lemma (in preorder) transp-on-ge[simp]: transp-on A (≥)
  by (auto intro: transp-onI order-trans)

lemma (in preorder) transp-on-greater[simp]: transp-on A (>)
  by (auto intro: transp-onI less-trans)

19.2.7 Totality

definition total-on :: 'a set ⇒ 'a rel ⇒ bool where
  total-on A r ←→ (∀x ∈ A. ∀y ∈ A. x ≠ y → (x, y) ∈ r ∨ (y, x) ∈ r)

abbreviation total :: 'a rel ⇒ bool where
  total ≡ total-on UNIV

definition totalp-on :: 'a set ⇒ (‘a ⇒ ‘a ⇒ bool) ⇒ bool where
  totalp-on A R ←→ (∀x ∈ A. ∀y ∈ A. x ≠ y → R x y ∨ R y x)

abbreviation totalp :: (‘a ⇒ ‘a ⇒ bool) ⇒ bool where
  totalp ≡ totalp-on UNIV

lemma totalp-on-total-on-eq[pred-set-conv]: totalp-on A (λx y. (x, y) ∈ r) ←→ total-on A r
  by (simp add: totalp-on-def total-on-def)

lemma total-onI [intro?]:
  (∀x y. x ∈ A → y ∈ A → x ≠ y → (x, y) ∈ r ∨ (y, x) ∈ r) → total-on A r
  unfolding total-on-def by blast

lemma totalI: (∀x y. x ≠ y → (x, y) ∈ r ∨ (y, x) ∈ r) → total r
  by (rule total-onI)

lemma totalp-onI: (∀x y. x ∈ A → y ∈ A → x ≠ y → R x y ∨ R y x) → totalp-on A R
THEORY “Relation”

by (rule total-onI[to-pred])

lemma totalpI: \(\forall x y. x \neq y \Rightarrow R x y \lor R y x\) \(\Rightarrow\) totalp R
by (rule totalI[to-pred])

lemma totalp-onD:
\(\text{totalp-on } A R \Rightarrow x \in A \Rightarrow y \in A \Rightarrow x \neq y \Rightarrow R x y \lor R y x\)
by (simp add: totalp-on-def)

lemma totalpD: totalp R \(\Rightarrow\) x \(\neq\) y \(\Rightarrow\) R x y \lor R y x
by (simp add: totalp-onD)

lemma totalp-on-subset: totalp-on A R \(\Rightarrow\) B \(\subseteq\) A \(\Rightarrow\) totalp-on B R
by (auto intro: totalp-onI dest: totalp-onD)

lemma totalp-on-image:
assumes inj-on f A
shows totalp-on (f' A) R \(\iff\) totalp-on A (\(\lambda a b. R(f a)(f b)\))
using assms by (auto simp: totalp-on-def inj-on-def)

lemma totalp-on-empty [simp]: totalp-on \{\} R
by (simp add: totalp-on-def)

lemma totalp-on-empty [simp]: totalp-on \{\} R
by (simp add: totalp-on-def)

lemma totalp-on-singleton [simp]: totalp-on \{x\} R
by (simp add: totalp-on-def)

lemma totalp-on-singleton [simp]: totalp-on \{x\} R
by (simp add: totalp-on-def)

lemma (in linorder) totalp-on-less [simp]: totalp-on A (<)
by (auto intro: totalp-onI)

lemma (in linorder) totalp-on-greater [simp]: totalp-on A (>)
by (auto intro: totalp-onI)

lemma (in linorder) totalp-on-le [simp]: totalp-on A (\(\leq\))
by (rule totalp-onI, rule linear)

lemma (in linorder) totalp-on-ge [simp]: totalp-on A (\(\geq\))
by (rule totalp-onI, rule linear)
19.2.8 Single valued relations

definition single-valued :: ('a × 'b) set ⇒ bool
  where single-valued r ⇔ (∀ x y. (x, y) ∈ r → (∀ z. (x, z) ∈ r → y = z))

definition single-valuedp :: ('a ⇒ 'b ⇒ bool) ⇒ bool
  where single-valuedp r ⇔ (∀ x y z. r x y → r x z → y = z)

lemma single-valuedp-single-valued-eq [pred-set-conv]:
  single-valuedp (λ x y. (x, y) ∈ r) ⇔ single-valued r
  by (simp add: single-valued-def single-valuedp-def)

lemma single-valuedp-iff-Uniq:
  single-valuedp r ⇔ (∀ x y. r x y → (∃ y. r x y = z) = y)
  unfolding Uniq-def single-valuedp-def by auto

lemma single-valuedI:
  (∀ x y. (x, y) ∈ r =⇒ (∃ z. (x, z) ∈ r =⇒ y = z)) =⇒ single-valued r
  unfolding single-valued-def by blast

lemma single-valuedpI:
  (∀ x y. r x y =⇒ (∃ z. r x z =⇒ y = z)) =⇒ single-valuedp r
  by (fact single-valuedI [to-pred])

lemma single-valuedD:
  single-valued r =⇒ (x, y) ∈ r =⇒ (x, z) ∈ r =⇒ y = z
  by (simp add: single-valued-def)

lemma single-valuedpD:
  single-valuedp r =⇒ r x y =⇒ r x z =⇒ y = z
  by (fact single-valuedD [to-pred])

lemma single-valued-empty [simp]:
  single-valued { }
  by (simp add: single-valued-def)

lemma single-valuedp-bot [simp]:
  single-valuedp ⊥
  by (fact single-valued-empty [simp])

lemma single-valued-subset:
  r ⊆ s =⇒ single-valued s =⇒ single-valued r
  unfolding single-valued-def by blast

lemma single-valuedp-less-eq:
  r ⊆ s =⇒ single-valuedp s =⇒ single-valuedp r
  by (fact single-valued-subset [simp])
19.3 Relation operations

19.3.1 The identity relation

definition Id :: 'a rel
  where \( \text{Id} = \{ p. \exists x. p = (x, x) \} \)

lemma IdI [intro]: \((a, a) \in \text{Id}\) by (simp add: Id-def)

lemma IdE [elim!]: \(p \in \text{Id} \Rightarrow (\forall x. p = (x, x) \Rightarrow P) \Rightarrow P\)
  unfolding Id-def by (iprover elim: CollectE)

lemma pair-in-Id-conv [iff]: \((a, b) \in \text{Id} \iff a = b\)
  unfolding Id-def by blast

lemma refl-Id: refl Id by (simp add: refl-on-def)

lemma antisym-Id: antisym Id
  — A strange result, since \(\text{Id}\) is also symmetric.
  by (simp add: antisym-def)

lemma sym-Id: sym Id by (simp add: sym-def)

lemma trans-Id: trans Id by (simp add: trans-def)

lemma single-valued-Id [simp]: single-valued Id by (unfold single-valued-def) blast

lemma irrefl-diff-Id [simp]: irrefl \((r - \text{Id})\)
  by (simp add: irrefl-def)

lemma trans-diff-Id: trans\(r \Rightarrow \text{antisym} r \Rightarrow \text{trans} (r - \text{Id})\)
  unfolding antisym-def trans-def by blast

lemma total-on-diff-Id [simp]: total-on A \((r - \text{Id})\) = total-on A r
  by (simp add: total-on-def)

lemma Id-fstsnd-eq: \(\text{Id} = \{x. \text{fst} x = \text{snd} x\}\)
  by force

19.3.2 Diagonal: identity over a set

definition Id-on :: 'a set \Rightarrow 'a rel
  where \(\text{Id-on} A = (\bigcup x \in A. \{ (x, x) \})\)

lemma Id-on-empty [simp]: \(\text{Id-on} \{\} = \{\}\)
THEORY “Relation”

by (simp add: Id-on-def)

lemma Id-on-eqI: a = b ⟹ a ∈ A ⟹ (a, b) ∈ Id-on A
  by (simp add: Id-on-def)

lemma Id-onI [intro!]: a ∈ A ⟹ (a, a) ∈ Id-on A
  by (rule Id-on-eqI) (rule refl)

lemma Id-onE [elim!]: c ∈ Id-on A ⟹ (∀x. x ∈ A ⟹ c = (x, x) ⟹ P) ⟹ P
  — The general elimination rule.
  unfolding Id-on-def by (iprover elim! "UN-E singletonE")

lemma Id-on-iff: (x, y) ∈ Id-on A ⟷ x = y ∧ x ∈ A
  by blast

lemma Id-on-def' [nitpick-unfold]: Id-on {x. A x} = Collect (λ(x, y). x = y ∧ A x)
  by auto

lemma Id-on-subset-Times: Id-on A ⊆ A × A
  by blast

lemma refl-on-Id-on: refl-on A (Id-on A)
  by (rule refl-onI [OF Id-on-subset-Times Id-onI])

lemma antisym-Id-on [simp]: antisym (Id-on A)
  unfolding antisym-def by blast

lemma sym-Id-on [simp]: sym (Id-on A)
  by (rule symI) clarify

lemma trans-Id-on [simp]: trans (Id-on A)
  by (fast intro: transI elim: transD)

lemma single-valued-Id-on [simp]: single-valued (Id-on A)
  unfolding single-valued-def by blast

19.3.3 Composition

inductive-set relcomp :: ('a × 'b) set ⇒ ('b × 'c) set ⇒ ('a × 'c) set (infixr O 75)
  for r :: ('a × 'b) set and s :: ('b × 'c) set
  where relcompI [intro]: (a, b) ∈ r ⟹ (b, c) ∈ s ⟹ (a, c) ∈ r O s

notation relcompp (infixr OO 75)

lemmas relcomppI = relcompp.intros

For historic reasons, the elimination rules are not wholly corresponding. Feel free to consolidate this.
THEORY “Relation”

inductive-cases relcompEpair: (a, c) ∈ r O s
inductive-cases relcompp [elim!]: (r OO s) a c

lemma relcompE [elim!]: xz ∈ r O s
  (Πxyz. xz = (x, z) ⟹ (x, y) ∈ r ⟹ (y, z) ∈ s ⟹ P) ⟹ P
apply (cases xz)
apply simp
apply (erule relcompEpair)
apply iprover
done

lemma R-O-Id [simp]: R O Id = R
  by fast

lemma Id-O-R [simp]: Id O R = R
  by fast

lemma relcomp-empty1 [simp]: {} O R = {}
  by blast

lemma relcompp-bot1 [simp]: ⊥ OO R = ⊥
  by (fact relcomp-empty1 [to-pred])

lemma relcomp-empty2 [simp]: R O {} = {}
  by blast

lemma relcompp-bot2 [simp]: R OO ⊥ = ⊥
  by (fact relcomp-empty2 [to-pred])

lemma O-assoc: (R O S) O T = R O (S O T)
  by blast

lemma relcompp-assoc: (r OO s) OO t = r OO (s OO t)
  by (fact O-assoc [to-pred])

lemma trans-O-subset: trans r ⟹ r O r ⊆ r
  by (unfold trans-def) blast

lemma transp-relcompp-less-eq: transp r ⟹ r OO r ⊆ r
  by (fact trans-O-subset [to-pred])

lemma relcomp-mono: r' ⊆ r ⟹ s' ⊆ s ⟹ r' O s' ⊆ r O s
  by blast

lemma relcompp-mono: r' ≤ r ⟹ s' ≤ s ⟹ r' OO s' ≤ r OO s
  by (fact relcomp-mono [to-pred])

lemma relcomp-subset-Sigma: r ⊆ A × B ⟹ s ⊆ B × C ⟹ r O s ⊆ A × C
  by blast
lemma relcomp-distrib [simp]: \( R \circ (S \cup T) = (R \circ S) \cup (R \circ T) \)
by auto

lemma relcomp-distrb [simp]: \( R \circ\circ (S \cup T) = R \circ \circ S \cup R \circ\circ T \)
by (fact relcomp-distrib [to-pred])

lemma relcomp-distrib2 [simp]: \( (S \cup T) \circ R = (S \circ R) \cup (T \circ R) \)
by auto

lemma relcompp-distrib2 [simp]: \( (S \cup\cup T) \circ R = S \cup\cup R \circ T \)
by (fact relcomp-distrib2 [to-pred])

lemma relcomp-UNION-distrib: \( s \circ \bigcup (r \cdot I) = \bigcup i \in I. s \circ r i \)
by auto

lemma relcompp-SUP-distrib: \( s \cup\cup (r \cdot I) = \bigcup i \in I. s \cup\cup r i \)
by (fact relcomp-UNION-distrib [to-pred])

lemma relcomp-UNION-distrib2: \( \bigcup (r \cdot I) \circ s = \bigcup i \in I. r i \circ s \)
by auto

lemma relcompp-SUP-distrib2: \( \bigcup (r \cdot I) \circ s = \bigcup i \in I. r i \circ s \)
by (fact relcomp-UNION-distrib2 [to-pred])

lemma single-valued-relcomp: \( \text{single-valued } r \Rightarrow \text{single-valued } s \Rightarrow \text{single-valued } (r \circ s) \)
unfolding single-valued-def by blast

lemma relcomp-unfold: \( r \circ s = \{(x, z). \exists y. (x, y) \in r \land (y, z) \in s\} \)
by (auto simp add: set-eq-iff)

lemma relcomp-apply: \( (R \circ\circ S) a c \iff (\exists b. R a b \land S b c) \)
unfolding relcomp-unfold [to-pred] ..

lemma eq-OO: \( (=) \circ\circ R = R \)
by blast

lemma OO-eq: \( R \circ\circ (=) = R \)
by blast

19.3.4 Converse

inductive-set \( \text{converse} :: (\cdot a \times \cdot b) \text{ set } \Rightarrow (\cdot b \times \cdot a) \text{ set } ((-^1) [1000] 999) \)
for \( r :: (\cdot a \times \cdot b) \text{ set } \)
where \( (a, b) \in r \Rightarrow (b, a) \in r^{-1} \)

notation \( \text{conversep } ((-^1) [1000] 1000) \)
notation (ASCII)
  converse (_−1_) [1000] 999) and
  conversep (_−−1_) [1000] 1000)

lemma converseI [sym]: (a, b) ∈ r ⇒ (b, a) ∈ r⁻¹
  by (fact converse.intros)

lemma conversepI : r a b ⇒ r⁻¹⁻¹ b a
  by (fact conversep.intros)

lemma converseD [sym]: (a, b) ∈ r⁻¹ ⇒ (b, a) ∈ r
  by (erule converse.cases) iprover

lemma conversepD : r⁻¹⁻¹⁻¹ b a ⇒ r a b
  by (fact converseD [to-pred])

lemma converseE [elim!]: yx ∈ r⁻¹ ⇒ (∀x y. yx = (y, x) ⇒ (x, y) ∈ r ⇒ P) ⇒ P
  — More general than converseD, as it “splits” the member of the relation.
  apply (cases yx)
  apply simp
  apply (erule converse.cases)
  apply iprover
  done

lemmas conversepE [elim!] = conversep.cases

lemma converse-iff [iff]: (a, b) ∈ r⁻¹ ↔ (b, a) ∈ r
  by (auto intro: converseI)

lemma conversep-iff [iff]: r⁻¹⁻¹ a b = r a b
  by (fact converse-iff [to-pred])

lemma converse-converse [simp]: (r⁻¹)⁻¹ = r
  by (simp add: set-eq-iff)

lemma conversep-conversep [simp]: (r⁻¹⁻¹)⁻¹⁻¹ = r
  by (fact converse-conversep [to-pred])

lemma converse-empty [simp]: {}⁻¹ = {}
  by auto

lemma converse-UNIV [simp]: UNIV⁻¹ = UNIV
  by auto

lemma converse-relcomp: (r O s)⁻¹ = s⁻¹ O r⁻¹
  by blast

lemma converse-relcompp: (r OO s)⁻¹⁻¹⁻¹ = s⁻¹⁻¹ O O r⁻¹⁻¹
THEORY “Relation”

by (iprover intro: order-antisym conversepI relcomppI elim: relcomppE dest: conversepD)

lemma converse-Int: \((r \cap s)^{-1} = r^{-1} \cap s^{-1}\)
  by blast

lemma converse-meet: \((r \cap s)^{-1^{-1}} = r^{-1^{-1}} \cap s^{-1^{-1}}\)
  by (simp add: inf-fun-def) (iprover intro: conversepI ext dest: conversepD)

lemma converse-Un: \((r \cup s)^{-1} = r^{-1} \cup s^{-1}\)
  by blast

lemma converse-join: \((r \sqcup s)^{-1^{-1}} = r^{-1^{-1}} \sqcup s^{-1^{-1}}\)
  by (simp add: sup-fun-def) (iprover intro: conversepI ext dest: conversepD)

lemma converse-INTER: \(\bigcap (r \cdot S)^{-1} = \bigcap x \in S. \ (r \ x)^{-1}\)
  by fast

lemma converse-UNION: \(\bigcup (r \cdot S)^{-1} = \bigcup x \in S. \ (r \ x)^{-1}\)
  by blast

lemma converse-mono[simp]: \(r^{-1} \subseteq s^{-1} \iff r \subseteq s\)
  by auto

lemma conversep-mono[simp]: \(r^{-1^{-1}} \leq s^{-1^{-1}} \iff r \leq s\)
  by (fact converse-mono[to-pred])

lemma converse-inject[simp]: \(r^{-1} = s^{-1} \iff r = s\)
  by auto

lemma conversep-inject[simp]: \(r^{-1^{-1}} = s^{-1^{-1}} \iff r = s\)
  by (fact converse-inject[to-pred])

lemma converse-subset-swap: \(r \subseteq s^{-1} \iff r^{-1} \subseteq s\)
  by auto

lemma conversep-le-swap: \(r \leq s^{-1^{-1}} \iff r^{-1^{-1}} \leq s\)
  by (fact converse-subset-swap[to-pred])

lemma converse-Id [simp]: \(Id^{-1} = Id\)
  by blast

lemma converse-Id-on [simp]: \((Id-on \ A)^{-1} = Id-on \ A\)
  by blast

lemma refl-on-converse [simp]: \(refl-on \ A \ (r^{-1}) = refl-on \ A \ r\)
  by (auto simp: refl-on-def)

lemma reflp-on-conversp [simp]: \(reflp-on \ A \ R^{-1^{-1}} \iff reflp-on \ A \ R\)
THEORY “Relation”

by (auto simp: reflp-on-def)

lemma irrefl-on-converse [simp]: irrefl-on A \((r^{-1}) = irrefl-on A r
by (simp add: irrefl-on-def)

lemma irreflp-on-converse [simp]: irreflp-on A \((r^{-1} = irreflp-on A r
by (rule irrefl-on-converse[to-pred])

lemma sym-on-converse [simp]: sym-on A \((r^{-1} = sym-on A r
by (auto intro: sym-onI dest: sym-onD)

lemma symp-on-conversep [simp]: symp-on A R \((r^{-1} = symp-on A R
by (rule sym-on-converse[to-pred])

lemma asym-on-converse [simp]: asym-on A \((r^{-1} = asym-on A r
by (auto dest: asym-onD)

lemma asymp-on-conversep [simp]: asymp-on A R \((r^{-1} = asymp-on A R
by (rule asym-on-converse[to-pred])

lemma antisym-on-converse [simp]: antisym-on A \((r^{-1} = antisym-on A r
by (auto intro: antisym-onI dest: antisym-onD)

lemma antisymp-on-conversep [simp]: antisymp-on A R \((r^{-1} = antisymp-on A R
by (rule antisym-on-converse[to-pred])

lemma trans-on-converse [simp]: trans-on A \((r^{-1} = trans-on A r
by (auto intro: trans-onI dest: trans-onD)

lemma transp-on-conversep [simp]: transp-on A R \((r^{-1} = transp-on A R
by (rule trans-on-converse[to-pred])

lemma sym-conv-converse-eq: sym \(r \iff r^{-1} = r
unfolding sym-def by fast

lemma sym-Un-converse: sym \((r \cup r^{-1})
unfolding sym-def by blast

lemma sym-Int-converse: sym \((r \cap r^{-1})
unfolding sym-def by blast

lemma total-on-converse [simp]: total-on A \((r^{-1} = total-on A r
by (auto simp: total-on-def)

lemma totalp-on-converse [simp]: totalp-on A R \((r^{-1} = totalp-on A R
by (rule total-on-converse[to-pred])

lemma conversep-noteq [simp]: \((\neq)^{-1} = (\neq)
by (auto simp add: fun-eq-iff)
THEORY “Relation”

lemma converse-eq [simp]: \((=)^{-1}^{-1} = (=)\)
by (auto simp add: fun-eq-iff)

lemma converse-unfold [code]: \(r^{-1} = \{(y, x). (x, y) \in r\}\)
by (simp add: set-eq-iff)

19.3.5 Domain, range and field

inductive-set Domain :: \(\{a \times b\} set\Rightarrow \{a\} set\) for \(r :: \{a \times b\} set\)
where DomainI [intro]: \((a,b) \in r \Longrightarrow a \in \text{Domain } r\)

lemmas DomainPI = Domainp.DomainI

inductive-cases DomainE [elim!]: \(a \in \text{Domain } r\)
inductive-cases DomainpE [elim!]: Domainp \(r a\)

inductive-set Range :: \(\{a \times b\} set\Rightarrow \{b\} set\) for \(r :: \{a \times b\} set\)
where RangeI [intro]: \((a,b) \in r \Longrightarrow b \in \text{Range } r\)

lemmas RangePI = Rangep.RangeI

inductive-cases RangeE [elim!]: \(b \in \text{Range } r\)
inductive-cases RangepE [elim!]: Rangep \(r b\)

definition Field :: \({a} rel \Rightarrow {a} set\)
where Field \(r = \text{Domain } r \cup \text{Range } r\)

lemma Field-iff: \(x \in \text{Field } r \iff (\exists y. (x,y) \in r \lor (y,x) \in r)\)
by (auto simp: Field-def)

lemma FieldI1: \((i,j) \in R \Longrightarrow i \in \text{Field } R\)
unfolding Field-def by blast

lemma FieldI2: \((i,j) \in R \Longrightarrow j \in \text{Field } R\)
unfolding Field-def by auto

lemma Domain-fst [code]: Domain \(r = \text{fst } r\)
by force

lemma Range-snd [code]: Range \(r = \text{snd } r\)
by force

lemma fst-eq-Domain: \(\text{fst } R = \text{Domain } R\)
by force

lemma snd-eq-Range: \(\text{snd } R = \text{Range } R\)
by force
lemma range-fst [simp]: range fst = UNIV  
  by (auto simp: fst_eq_Domain)

lemma range-snd [simp]: range snd = UNIV  
  by (auto simp: snd_eq_Range)

lemma Domain-empty [simp]: Domain {} = {}  
  by auto

lemma Range-empty [simp]: Range {} = {}  
  by auto

lemma Field-empty [simp]: Field {} = {}  
  by (simp add: Field_def)

lemma Domain-empty-iff: Domain r = {} ←→ r = {}  
  by auto

lemma Range-empty-iff: Range r = {} ←→ r = {}  
  by auto

lemma Domain-insert [simp]: Domain (insert (a, b) r) = insert a (Domain r)  
  by blast

lemma Range-insert [simp]: Range (insert (a, b) r) = insert b (Range r)  
  by blast

lemma Field-insert [simp]: Field (insert (a, b) r) = {a, b} ∪ Field r  
  by (auto simp add: Field_def)

lemma Domain-iff: a ∈ Domain r ←→ (∃y. (a, y) ∈ r)  
  by blast

lemma Range-iff: a ∈ Range r ←→ (∃y. (y, a) ∈ r)  
  by blast

lemma Domain-Id [simp]: Domain Id = UNIV  
  by blast

lemma Range-Id [simp]: Range Id = UNIV  
  by blast

lemma Domain-Id-on [simp]: Domain (Id-on A) = A  
  by blast

lemma Range-Id-on [simp]: Range (Id-on A) = A  
  by blast

lemma Domain-Un-eq: Domain (A ∪ B) = Domain A ∪ Domain B
by blast

lemma Range-Un-eq: Range \((A \cup B)\) = Range A \cup Range B
  by blast

lemma Field-Un [simp]: Field \((r \cup s)\) = Field r \cup Field s
  by (auto simp: Field-def)

lemma Domain-Int-subset: Domain \((A \cap B)\) \subseteq Domain A \cap Domain B
  by blast

lemma Range-Int-subset: Range \((A \cap B)\) \subseteq Range A \cap Range B
  by blast

lemma Domain-Diff-subset: Domain A \setminus Domain B \subseteq Domain \((A \setminus B)\)
  by blast

lemma Range-Diff-subset: Range A \setminus Range B \subseteq Range \((A \setminus B)\)
  by blast

lemma Domain-Union: Domain \(\bigcup S\) = \(\bigcup A \in S.\ Domain A\)
  by blast

lemma Range-Union: Range \(\bigcup S\) = \(\bigcup A \in S.\ Range A\)
  by blast

lemma Field-Union [simp]: Field \(\bigcup R\) = \(\bigcup (Field \cdot R)\)
  by (auto simp: Field-def)

lemma Domain-converse [simp]: Domain \((r^{-1})\) = Range r
  by auto

lemma Range-converse [simp]: Range \((r^{-1})\) = Domain r
  by blast

lemma Field-converse [simp]: Field \((r^{-1})\) = Field r
  by (auto simp: Field-def)

lemma Domain-Collect-case-prod [simp]: Domain \(\{(x, y).\ P x y\}\) = \(\{x.\ \exists y.\ P x y\}\)
  by auto

lemma Range-Collect-case-prod [simp]: Range \(\{(x, y).\ P x y\}\) = \(\{y.\ \exists x.\ P x y\}\)
  by auto

lemma Domain-mono: \(r \subseteq s\) \implies Domain r \subseteq Domain s
  by blast

lemma Range-mono: \(r \subseteq s\) \implies Range r \subseteq Range s
THEORY "Relation"

by blast

lemma mono-Field: \( r \subseteq s \implies \text{Field } r \subseteq \text{Field } s \)
by (auto simp: Field-def Domain-def Range-def)

lemma Domain-unfold: \( \text{Domain } r = \{ x. \exists y. (x, y) \in r \} \)
by blast

lemma Field-square [simp]: \( \text{Field } (x \times x) = x \)
unfolding Field-def by blast

19.3.6 Image of a set under a relation

definition \( \text{Image} :: (\lambda a \times b. \text{set } a \implies \text{set } b) \) (infixr "\"" 90)
where \( r \"" s = \{ y. \exists x \in s. (x, y) \in r \} \)

lemma Image-iff: \( b \in r""A \iff (\exists x \in A. (x, b) \in r) \)
by (simp add: Image-def)

lemma Image-singleton: \( r""\{a\} = \{b. (a, b) \in r\} \)
by (simp add: Image-def)

lemma Image-singleton-iff [iff]: \( b \in r""\{a\} \iff (a, b) \in r \)
by (rule Image-iff [THEN trans]) simp

lemma ImageI [intro]: \( (a, b) \in r \implies a \in A \implies b \in r""A \)
unfolding Image-def by blast

lemma ImageE [elim!]: \( b \in r \"" A \implies (\forall x. (x, b) \in r \implies x \in A \implies P) \implies P \)
unfolding Image-def by (iprover elim!: CollectE bexE)

lemma rev-ImageI: \( a \in A \implies (a, b) \in r \implies b \in r \"" A \)
— This version’s more effective when we already have the required \( a \)
by blast

lemma Image-empty1 [simp]: \( \{\} \"" X = \{\} \)
by auto

lemma Image-empty2 [simp]: \( R""\{\} = \{\} \)
by blast

lemma Image-Id [simp]: \( \text{Id } "" A = A \)
by blast

lemma Image-Id-on [simp]: \( \text{Id-on } A \"" B = A \cap B \)
by blast

lemma Image-Int-subset: \( R \"" (A \cap B) \subseteq R \"" A \cap R \"" B \)
by blast
lemma Image-Int-eq: single-valued (converse R) \( \Rightarrow \) R \( ^{\sim} \) (A \( \cap \) B) = R \( ^{\sim} \) A \( \cap \) R \( ^{\sim} \) B
  by (auto simp: single-valued-def)

lemma Image-Un: R \( ^{\sim} \) (A \( \cup \) B) = R \( ^{\sim} \) A \( \cup \) R \( ^{\sim} \) B
  by blast

lemma Un-Image: (R \( \cup \) S) \( ^{\sim} \) A = R \( ^{\sim} \) A \( \cup \) S \( ^{\sim} \) A
  by blast

lemma Image-subset: r \( \subseteq \) A \( \times \) B \( \Rightarrow \) r - C \( \subseteq \) B
  by (iprover intro!: subsetI elim!: ImageE dest: subsetD SigmaD2)

lemma Image-eq-UN: \( r^{\sim} \) = (\( \bigcup \) y\( \in \) B. \( r^{\sim}\) \{y\})
  — NOT suitable for rewriting
  by blast

lemma Image-mono: r' \( \subseteq \) r \( \Rightarrow \) A' \( \subseteq \) A \( \Rightarrow \) (r' \( ^{\sim} \) A') \( \subseteq \) (r \( ^{\sim} \) A)
  by blast

lemma Image-UN: r \( ^{\sim} \) (\( \bigcup \) (B \( \backslash \) A)) = (\( \bigcup \) x\( \in \) A. r \( ^{\sim} \) (B x))
  by blast

lemma UN-Image: (\( \bigcup \) i\( \in \) I. X i) \( ^{\sim} \) S = (\( \bigcup \) i\( \in \) I. X i \( ^{\sim} \) S)
  by auto

lemma Image-INT-subset: r \( ^{\sim} \) (\( \bigcap \) (B \( \backslash \) A)) \( \subseteq \) (\( \bigcap \) x\( \in \) A. r \( ^{\sim} \) (B x))
  by blast

Converse inclusion requires some assumptions

lemma Image-INT-eq:
  assumes single-valued (r^{-1})
  and A \( \neq \) \{\}
  shows r \( ^{\sim} \) (\( \bigcap \) (B \( \backslash \) A)) = (\( \bigcap \) x\( \in \) A. r \( ^{\sim} \) B x)
proof (rule equalityI, rule Image-INT-subset)
  show (\( \bigcap \) x\( \in \) A. r \( ^{\sim} \) B x) \( \subseteq \) r \( ^{\sim} \) (\( \bigcap \) (B \( \backslash \) A))
proof
    fix x
    assume x \( \in \) (\( \bigcap \) x\( \in \) A. r \( ^{\sim} \) B x)
    then show x \( \in \) r \( ^{\sim} \) (\( \bigcap \) (B \( \backslash \) A))
      using assms unfolding single-valued-def by simp blast
  qed
qed

lemma Image-subset-eq: r'\( ^{\sim} \)A \( \subseteq \) B \( \iff \) A \( \subseteq \) \( \sim \)(r^{-1} \( ^{\sim} \) \( \sim \) B)
  by blast

lemma Image-Collect-case-prod [simp]: \( \{ (x, y). P x y \} \) \( ^{\sim} \) A = \{ y. \( \exists \) x\( \in \) A. P x y\}
THEORY “Relation” 476

by auto

lemma Sigma-Image: (SIGMA x:A. B x) "= (\bigcup x \in X \cap A. B x)
by auto

lemma recomp-Image: (X O Y) " = Y " (X " Z)
by auto

19.3.7 Inverse image
definition inv-image :: 'b rel \Rightarrow (\forall 'a. f 'a \Rightarrow 'b) \Rightarrow 'a rel
where inv-image r f = \{(x, y). (f x, f y) \in r\}
definition inv-imagep :: (\forall 'b. f 'b \Rightarrow bool) \Rightarrow (\forall 'a. f 'a \Rightarrow 'a \Rightarrow bool
where inv-imagep r f = (\lambda x y. r (f x) (f y))

lemma [pred-set-conv]: inv-imagep (\lambda x y. (x, y) \in r) f = (\lambda x y. (x, y) \in inv-image r f)
by (simp add: inv-image-def inv-imagep-def)

lemma sym-inv-image: sym r \Rightarrow sym (inv-image r f)
unfolding sym-def inv-image-def by blast

lemma trans-inv-image: trans r \Rightarrow trans (inv-image r f)
unfolding trans-def inv-image-def by (simp (no_asm)) blast

lemma total-inv-image: [\inj f; total r] \Rightarrow total (inv-image r f)
unfolding inv-image-def total-on-def by (auto simp: inj-eq)

lemma asym-inv-image: asym R \Rightarrow asym (inv-image R f)
by (simp add: inv-image-def asym-iff)

lemma in-inv-image[simp]: (x, y) \in inv-image r f \longleftrightarrow (f x, f y) \in r
by (auto simp: inv-image-def)

lemma converse-inv-image[simp]: (inv-image R f)^{-1} = inv-image (R^{-1}) f
unfolding inv-image-def converse-unfold by auto

lemma in-inv-imagep [simp]: inv-imagep r f x y = r (f x) (f y)
by (simp add: inv-imagep-def)

19.3.8 Powerset
definition Powp :: (\forall 'a. f 'a \Rightarrow bool) \Rightarrow 'a set \Rightarrow bool
where Powp A = (\lambda B. \forall x \in B. A x)

lemma Powp-Pow-eq [pred-set-conv]: Powp (\lambda x. x \in A) = (\lambda x. x \in Pow A)
by (auto simp add: Powp-def fun-eq-iff)
lemmas Powp-mono [mono] = Pow-mono [to-pred]

end

20 Finite sets

theory Finite-Set
  imports Product-Type Sum-Type Fields Relation
begin

20.1 Predicate for finite sets

context notes [[inductive-internals]]
begin

inductive finite :: 'a set ⇒ bool
where
  emptyI [simp, intro!]: finite {}
| insertI [simp, intro!]: finite A =⇒ finite (insert a A)

end

simproc-setup finite-Collect (finite (Collect P)) = "K Set-Comprehension-Pointfree.proc"

declare [[simproc del: finite-Collect]]

lemma finite-induct [case-names empty insert, induct set: finite]:
  — Discharging x /∈ F entails extra work.
  assumes finite F
  assumes P {}
  and insert: ∃ x F. finite F =⇒ x /∈ F =⇒ P F =⇒ P (insert x F)
  shows P F
  using (finite F)
proof induct
  show P {} by fact
next
  fix x F
  assume F: finite F and P: P F
  show P (insert x F)
  proof cases
    assume x ∈ F
    then have insert x F = F by (rule insert-absorb)
    with P show ?thesis by (simp only:)
  next
    assume x /∈ F
    from F this P show ?thesis by (rule insert)
  qed
qed
lemma infinite-finite-induct [case-names infinite empty insert]:
assumes infinite: \( \forall A. \neg \text{finite } A \implies P A \)
and empty: \( P \{\} \)
and insert: \( \forall x F. \text{finite } F \implies x \notin F \implies P F \implies P (\text{insert } x F) \)
shows \( P A \)
proof (cases finite A)
case False
with infinite show \( \text{?thesis} \).
next
case True
then show \( \text{?thesis} \) by (induct A) (fact empty insert)+
qed

20.1.1 Choice principles

lemma ex-new-if-finite: — does not depend on def of finite at all
assumes \( \neg \text{finite } (\text{UNIV :: 'a set}) \) and \( \text{finite } A \)
shows \( \exists a :: 'a. a \notin A \)
proof –
  from assms have \( A \neq \text{UNIV} \) by blast
  then show \( \text{?thesis} \) by blast
qed

A finite choice principle. Does not need the SOME choice operator.

lemma finite-set-choice: \( \text{finite } A \implies \forall x \in A. \exists y. P x y \implies \exists f. \forall x \in A. P x (f x) \)
proof (induct rule: finite-induct)
case empty
  then show \( \text{?case} \) by simp
next
case (insert a A)
  then obtain f b where f: \( \forall x \in A. P x (f x) \) and ab: \( P a b \)
  by auto
  show \( \text{?case} \) (is \( \exists f. \ ?P f \))
  proof
    show \( ?P (\lambda x. \text{if } x = a \text{ then } b \text{ else } f x) \)
    using f ab by auto
  qed
qed

20.1.2 Finite sets are the images of initial segments of natural numbers

lemma finite-imp-nat-seg-image-inj-on:
assumes \( \text{finite } A \)
shows \( \exists (n :: \text{nat}) f. A = f \{i. i < n\} \land \text{inj-on } f \{i. i < n\} \)
using assms
proof induct
case empty
  show \( \text{?case} \)
proof
  show \( \exists f. \{\} = f \cdot \{i :: \text{nat}. \ i < 0\} \land \text{inj-on } f \{i. \ i < 0\} \)
  by simp
qed

next
  case (insert a A)
  have \( \text{notinA}: a \notin A \) by fact
  from insert.hyps obtain n f where \( A = f \cdot \{i :: \text{nat}. \ i < n\} \land \text{inj-on } f \{i. \ i < n\} \)
  by blast
  then have \( \text{insert a A} = f(n := a) \cdot \{i. \ i < \text{Suc } n\} \land \text{inj-on } (f(n := a)) \{i. \ i < \text{Suc } n\} \)
  using \( \text{notinA} \) by (auto simp add: image-def Ball-def inj-on-def less-Suc-eq)
  then show ?case by blast
qed

lemma nat-seg-image-imp-finite: \( A = f \cdot \{i :: \text{nat}. \ i < n\} \implies \text{finite } A \)
proof (induct n arbitrary: A)
  case 0
  then show ?case by simp
next
  case (Suc n)
  let \( ?B = f \cdot \{i. \ i < n\} \)
  have \( \text{finB}: \text{finite } ?B \) by (rule Suc.hyps[OF refl])
  show ?case
  proof (cases \( \exists k < n. \ f \cdot n = f \cdot k \))
    case True
    using \( \text{Suc.prems} \) by (auto simp: less-Suc-eq)
    then show ?thesis using \( \text{finB} \) by simp
  next
    case False
    then have \( A = \text{insert } (f \cdot n) \cdot ?B \)
    using \( \text{Suc.prems} \) by (auto simp: less-Suc-eq)
    then show ?thesis using \( \text{finB} \) by simp
  qed
qed

lemma finite-conv-nat-seg-image: \( \text{finite } A \iff (\\exists n. \ A = f \cdot \{i :: \text{nat}. \ i < n\}) \)
by (blast intro: nat-seg-image-imp-finite dest: finite-imp-nat-seg-image-inj-on)

lemma finite-imp-inj-to-nat-seg:
  assumes \( \text{finite } A \)
  shows \( \exists f n. \ f \cdot A = \{i :: \text{nat}. \ i < n\} \land \text{inj-on } f A \)
proof -
  from finite-imp-nat-seg-image-inj-on [OF \( \text{finite } A \)]
  obtain f and n :: \( \text{nat} \) where \( \text{bij}: \text{bij-between } f \{i. \ i < n\} \ A \)
  by (auto simp: bij-between-def)
  let \( ?f = \text{the-inv-into } \{i. \ i < n\} \ f \)
have inj-on \( ?f \ A \land \neg f \ A = \{i. i<n\} \)
by (fold bij-betw-def) (rule bij-betw-the-inv-into[OF bij])
then show \(?thesis\) by blast
qed

lemma finite-Collect-less-nat [iff]: finite \( \{n::nat. n < k\} \)
by (fastforce simp: finite-conv-nat-seg-image)

lemma finite-Collect-le-nat [iff]: finite \( \{n::nat. n \leq k\} \)
by (simp add: le-eq-less-or-eq Collect-disj-eq)

20.2 Finiteness and common set operations

lemma rev-finite-subset: finite \( B \implies A \subseteq B \implies \) finite \( A \)
proof (induct arbitrary: \( A \) rule: finite-induct)
case empty
then show \(?case\) by simp
next
case (insert \( x \) \( F \) \( A \))
have \( A \subseteq insert \( x \) \( F \) \) and \( r \): \( A \setminus \{x\} \subseteq F \implies \) finite \( (A \setminus \{x\}) \)
by fact+
show finite \( A \)
proof cases
assume \( x: x \in A \)
with \( A \) have \( A \setminus \{x\} \subseteq F \) by (simp add: subset-insert-iff)
with \( r \) have finite \( (A \setminus \{x\}) \)
then have \( insert \( x \) \( (A \setminus \{x\}) \) = A \)
using \( x \) by (rule insert-Diff)
finally show \(?thesis\)
next
show \(?thesis\) when \( A \subseteq F \)
using that by fact
assume \( x \notin A \)
with \( A \) show \( A \subseteq F \)
by (simp add: subset-insert-iff)
qed
qed

lemma finite-subset: \( A \subseteq B \implies \) finite \( B \implies \) finite \( A \)
by (rule rev-finite-subset)

simproc-setup finite (finite \( A \)) = :
let
val finite-subset = @{thm finite-subset}
val Eq-TrueI = @{thm Eq-TrueI}
fun is-subset \( A \) th = case Thm.prop_of th of
\(- \$ (Const (const-name\{less-eq\}, Type (type-name\{fan\}), [Type (type-name\{set\},
\)
fun is-finite th = case Thm.prop_of th of
    (- $(Const (const-name finite', - $ A)) => SOME(A,th)
    | - => NONE;

fun comb (A,sub-th) (A',fin-th) ths = if A aconv A' then (sub-th,fin-th) :: ths else ths
fun proc ctxt ct =
  (let
    val - $ A = Thm.term_of ct
    val prems = Simplifier.prems_of ctxt
    val fins = map-filter is-finite prems
    val subsets = map-filter (is-subset A) prems
    in case fold-product comb subsets fins [] of
      (sub-th,fin-th) :: - => SOME((fin-th RS (sub-th RS finite-subset)) RS
        Eq-TrueI)
      | - => NONE
    end)
in K proc end

declare [[simproc del: finite]]

lemma finite-UnI:
  assumes finite F and finite G
  shows finite (F ∪ G)
  using assms by induct simp-all

lemma finite-Un [iff]: finite (F ∪ G) ⟷ finite F ∧ finite G
  by (blast intro: finite-UnI finite-subset [of - F ∪ G])

lemma finite-insert [simpl]: finite (insert a A) ⟷ finite A
  proof
    have finite {a} ∧ finite A ⟷ finite A by simp
    then have finite ({a} ∪ A) ⟷ finite A by (simp only: finite-Un)
    then show ?thesis by simp
  qed

lemma finite-Int [simpl, intro]: finite F ∨ finite G ⟹ finite (F ∩ G)
  by (blast intro: finite-subset)

lemma finite-Collect-conjI [simpl, intro]:
  finite {x. P x} ∨ finite {x. Q x} ⟹ finite {x. P x ∧ Q x}
  by (simp add: Collect-conj-eq)
lemma finite-Collect-disjI [simp]:
  finite \{ x. P x \land Q x \} \iff finite \{ x. P x \} \land finite \{ x. Q x \}
by (simp add: Collect-disj-eq)

lemma finite-Diff [simp, intro]: finite A \implies finite (A - B)
by (rule finite-subset, rule Diff-subset)

lemma finite-Diff2 [simp]:
  assumes finite B
  shows finite (A - B) \iff finite A
proof -
  have finite A \iff finite ((A - B) \cup (A \cap B))
  by (simp add: Un-Diff-Int)
  also have \ldots \iff finite (A - B)
  using ‹finite B› by simp
  finally show ?thesis ..
qed

lemma finite-Diff-insert [iff]: finite (A - insert a B) \iff finite (A - B)
proof -
  have finite (A - B) \iff finite (A - B - \{a\}) by simp
  moreover have A - insert a B = A - B - \{a\} by auto
  ultimately show ?thesis by simp
qed

lemma finite-compl [simp]:
  finite (A :: 'a set) \implies finite (- A) \iff finite (UNIV :: 'a set)
by (simp add: Compl-eq-Diff-UNIV)

lemma finite-Collect-not [simp]:
  finite \{ x :: 'a. P x \} \implies finite \{ x. \neg P x \} \iff finite (UNIV :: 'a set)
by (simp add: Collect-neg-eq)

lemma finite-Union [simp, intro]:
  finite A \implies (\forall M. M \in A \implies finite M) \implies finite (\bigcup A)
by (induct rule: finite-induct) simp-all

lemma finite-UN-I [intro]:
  finite A \implies (\forall a. a \in A \implies finite (B a)) \implies finite (\bigcup a\in A. B a)
by (induct rule: finite-induct) simp-all

lemma finite-UN [simp]: finite A \implies finite (\bigcup (B :: 'a)) \iff (\forall x\in A. finite (B x))
by (blast intro: finite-subset)

lemma finite-Inter [intro]: \exists A\in M. finite A \implies finite (\bigcap M)
by (blast intro: Inter-lower finite-subset)

lemma finite-INT [intro]: \exists x\in I. finite (A x) \implies finite (\bigcap x\in I. A x)
THEORY “Finite-Set” 483

by (blast intro: INT-lower finite-subset)

lemma finite-imageI [simp, intro]: finite F \implies finite (h ' F)
  by (induct rule: finite-induct) simp-all

lemma finite-image-set [simp]: finite \{x. P x\} \implies finite \{f x | x. P x\}
  by (simp add: image-Collect [symmetric])

lemma finite-image-set2:
  finite \{x. P x\} \implies finite \{y. Q y\} \implies finite \{f x | x. P x \wedge Q y\}
  by (rule finite-subset [where B = \bigcup x \in \{x. P x\}. \bigcup y \in \{y. Q y\}. \{f x y\}]) auto

lemma finite-imageD:
  assumes finite (f ' A) and inj-on f A
  shows finite A
  using assms
proof (induct f ' A arbitrary: A)
case empty
then show ?case by simp
next
case (insert x B)
then have B-A: insert x B = f ' A by simp
then obtain y where x = f y and y \in A by blast
from B-A \(x \notin B\) have B = f ' (A - {x}) by blast
with B-A \(x \notin B\) \(x = f y\) \(\langle\text{inj-on f A}\rangle\) \(\langle y \in A\rangle\) have B = f ' (A - {y}) by (simp add: inj-on-image-set-diff)
moreover from \(\langle\text{inj-on f A}\rangle\) have inj-on f (A - {y}) by (rule inj-on-diff)
ultimately have finite (A - {y}) by (rule insert.hyps)
then show finite A by simp
qed

lemma finite-image-iff: inj-on f A \implies finite (f ' A) \iff finite A
  using finite-imageD by blast

lemma finite-surj: finite A \implies B \subseteq f ' A \implies finite B
  by (erule finite-subset) (rule finite-imageI)

lemma finite-range-imageI: finite (range g) \implies finite (range (\lambda x. f (g x)))
  by (drule finite-imageI) (simp add: range-composition)

lemma finite-subset-image:
  assumes finite B
  shows B \subseteq f ' A \implies \exists C \subseteq A. finite C \land B = f ' C
using assms

proof induct
  case empty
  then show ?case by simp

next
  case insert
  then show ?case
    by (clarsimp simp del: image-insert simp add: image-insert [symmetric]) blast
qed

lemma all-subset-image: \( (\forall B. B \subseteq f \cdot A \rightarrow P B) \leftrightarrow (\forall B. B \subseteq A \rightarrow P (f \cdot B)) \)
  by (safe elim!: subset-imageE) (use image-mono in ‹ blast+++ ‹)

lemma all-finite-subset-image: \( (\forall B. \text{finite } B \land B \subseteq f \cdot A \rightarrow P B) \leftrightarrow (\forall B. \text{finite } B \land B \subseteq A \rightarrow P (f \cdot B)) \)
  proof safe
    fix \( B :: 'a \text{ set} \)
    assume \( B: \text{finite } B \land B \subseteq f \cdot A \land P B \)
    show \( \forall B. \text{finite } B \land B \subseteq A \land P (f \cdot B) \)
      using finite-subset-image [OF B] P by blast
  qed blast

lemma ex-finite-subset-image: \( (\exists B. \text{finite } B \land B \subseteq f \cdot A \land P B) \leftrightarrow (\exists B. \text{finite } B \land B \subseteq A \land P (f \cdot B)) \)
  proof safe
    fix \( B :: 'a \text{ set} \)
    assume \( B: \text{finite } B \land B \subseteq f \cdot A \land P B \)
    show \( \exists B. \text{finite } B \land B \subseteq A \land P (f \cdot B) \)
      using finite-subset-image [OF B] P by blast
  qed blast

lemma finite-vimage-IntI: finite \( F \Longrightarrow \text{inj-on } h A \Longrightarrow \text{finite } (h \cdot \cdot F \cap A) \)
  proof (induct rule: finite-induct)
    case (insert \( x \) \( F \))
    then show ?case
      by (simp add: vimage-insert [of h x F] finite-subset [OF inj-on-vimage-singleton] Int-Un-distrib2)
  qed simp

lemma finite-finite-vimage-IntI:
  assumes \( \text{finite } F \)
  and \( \forall y. y \in F \Longrightarrow \text{finite } (h \cdot \cdot \{y\} \cap A) \)
  shows \( \text{finite } (h \cdot \cdot F \cap A) \)
  proof
    have \( *: h \cdot \cdot F \cap A = \bigcup y \in F. (h \cdot \cdot \{y\} \cap A) \)
      by blast
    show \( \text{thesis} \)
      by (simp only: * assms finite-UN-I)
qed

lemma finite-vimageI: finite F \implies inj h \implies finite (h⁻¹ F)
  using finite-vimage-IntI[of F h UNIV] by auto

lemma finite-vimageD': finite (f⁻¹ A) \implies A \subseteq range f \implies finite A
  by (auto simp add: subset_image_iff intro: finite_subset[rotated])

lemma finite-vimageD: finite (h⁻¹ F) \implies surj h \implies finite F
  by (auto dest: finite-vimageD')

lemma finite-vimage-iff: bij h \implies finite (h⁻¹ F) \iff finite F
  unfolding bij-def by (auto elim: finite-vimageD finite-vimageI)

lemma finite-inverse-image-gen:
assumes finite A inj-on f D
shows finite \{ j \in D. f j \in A \}
using finite-vimage-IntI[OF assms]
by (simp add: Collect_conj_eq inf_commute vimage_def)

lemma finite-inverse-image:
assumes finite A inj f
shows finite \{ j. f j \in A \}
using finite-inverse-image-gen[OF assms]
by simp

lemma finite-Collect-bex [simp]:
assumes finite A
shows finite \{ x. \exists y \in A. Q x y \} \iff (\forall y \in A. finite \{ x. Q x y \})
proof
  have \{ x. \exists y \in A. Q x y \} = (\bigcup y \in A. \{ x. Q x y \})
  by auto
with assms show ?thesis by simp
qed

lemma finite-Collect-bounded-ex [simp]:
assumes finite \{ y. P y \}
shows finite \{ x. \exists y. P y \land Q x y \} \iff \(\forall y. P y \implies finite \{ x. Q x y \})
proof
  have \{ x. \exists y. P y \land Q x y \} = (\bigcup y \in \{ y. P y \}. \{ x. Q x y \})
  by auto
with assms show ?thesis
  by simp
qed

lemma finite-Plus: finite A \implies finite B \implies finite (A <+> B)
  by (simp add: Plus_def)

lemma finite-PlusD:
fixes A :: 'a set and B :: 'b set
assumes fin: finite (A <+> B)
shows finite A finite B
proof –
  have Inl A ⊆ A <+> B
    by auto
  then have finite (Inl A :: ('a + 'b) set)
    using fin by (rule finite-subset)
  then show finite A
    by (rule finite-imageD) (auto intro: inj-onI)
next
  have Inr B ⊆ A <+> B
    by auto
  then have finite (Inr B :: ('a + 'b) set)
    using fin by (rule finite-subset)
  then show finite B
    by (rule finite-imageD) (auto intro: inj-onI)
qed

lemma finite-Plus-iff [simp]: finite (A <+> B) ←→ finite A ∧ finite B
  by (auto intro: finite-PlusD finite-Plus)

lemma finite-Plus-UNIV-iff [simp]:
  finite (UNIV :: ('a + 'b) set) ←→ finite (UNIV :: 'a set) ∧ finite (UNIV :: 'b set)
  by (subst UNIV-Plus-UNIV [symmetric]) (rule finite-Plus-iff)

lemma finite-SigmaI [simp, intro]:
  finite A → (∀a. a ∈ A → finite (B a)) → finite (SIGMA a:A. B a)
  unfolding Sigma-def by blast

lemma finite-SigmaI2:
  assumes finite {x:A. B x ≠ {}}
  and ∃a. a ∈ A → finite (B a)
  shows finite (Sigma A B)
proof –
  from assms have finite (Sigma {x:A. B x ≠ {}} B)
    by auto
  also have Sigma {x:A. B x ≠ {}} B = Sigma A B
    by auto
  finally show ?thesis .
qed

lemma finite-cartesian-product: finite A → finite B → finite (A × B)
  by (rule finite-SigmaI)

lemma finite-Prod-UNIV:
  finite (UNIV :: 'a set) ⇒ finite (UNIV :: 'b set) ⇒ finite (UNIV :: ('a × 'b) set)
  by (simp only: UNIV-Times-UNIV [symmetric] finite-cartesian-product)
lemma finite-cartesian-productD1: 
assumes finite \((A \times B)\) and \(B \neq \{\}\) 
shows finite \(A\)
proof -
  from assms obtain \(n f\) where \(A \times B = f ' \{i::nat. i < n\}\) 
    by (auto simp add: finite-conv-nat-seg-image)
  then have \(\text{fst} ' (A \times B) = \text{fst} ' f ' \{i::nat. i < n\}\) 
    by simp
  with \(B \neq \{\}\) have \(A = (\text{fst} \circ f) ' \{i::nat. i < n\}\) 
    by blast
  then show \(?thesis\)
    by (auto simp add: finite-conv-nat-seg-image)
qed

lemma finite-cartesian-productD2: 
assumes finite \((A \times B)\) and \(A \neq \{\}\)
shows finite \(B\)
proof -
  from assms obtain \(n f\) where \(A \times B = f ' \{i::nat. i < n\}\) 
    by (auto simp add: finite-conv-nat-seg-image)
  then have \(\text{snd} ' (A \times B) = \text{snd} ' f ' \{i::nat. i < n\}\) 
    by simp
  with \(A \neq \{\}\) have \(B = (\text{snd} \circ f) ' \{i::nat. i < n\}\) 
    by simp
  then have \(?thesis\)
    by (auto simp add: finite-conv-nat-seg-image)
qed

lemma finite-cartesian-product-iff: 
finite \((A \times B)\) \(
\iff\)
\((A = \{\} \lor B = \{\} \lor (\text{finite } A \land \text{finite } B))\)
by (auto dest: finite-cartesian-productD1 finite-cartesian-productD2 finite-cartesian-product)

lemma finite-prod: 
finite \((\text{UNIV :: } ('a \times 'b) \text{ set})\) \(
\iff\)
finite \((\text{UNIV :: } 'a \text{ set}) \land \text{finite } (\text{UNIV :: } 'b \text{ set})\)
using finite-cartesian-product-iff[of UNIV UNIV] by simp

lemma finite-Pow-iff [iff]: finite \((\text{Pow } A)\) \(
\iff\)
finite \(A\)
proof
  assume finite \((\text{Pow } A)\)
  then have finite \(((\lambda x. \{x\}) ' A)\) 
    by (blast intro: finite-subset)
  then show finite \(A\)
    by (rule finite-imageD [unfolded inj-on-def]) simp
next
assume finite A
then show finite (Pow A)
  by induct (simp-all add: Pow-insert)
qed

corollary finite-Collect-subsets [simp, intro]: finite A ⇒ finite {B. B ⊆ A}
by (simp add: Pow-def [symmetric])

lemma finite-set: finite (UNIV :: 'a set) ↔ finite (UNIV :: 'a set)
by (simp only: finite-Pow-iff Pow-UNIV [symmetric])

lemma finite-UnionD: finite (⋃ A) =⇒ finite A
by (blast intro: finite-subset [OF subset-Pow-Union])

lemma finite-bind:
  assumes finite S
  assumes ∀x ∈ S. finite (f x)
  shows finite (Set.bind S f)
using assms by (simp add: bind-UNION)

lemma finite-filter [simp]: finite S =⇒ finite (Set.filter P S)
unfolding Set.filter-def by simp

lemma finite-set-of-finite-funs:
  assumes finite A finite B
  shows finite {f. ∀x. (x ∈ A −→ f x ∈ B) ∧ (x /∈ A −→ f x = d)} (is finite ?S)
proof –
  let ?F = λf. {(a,b). a ∈ A ∧ b = f a}
  have ?F ′ ?S ⊆ Pow(A × B)
    by auto
  from finite-subset[OF this] assms have 1: finite (?F ′ ?S)
    by simp
  have 2: inj-on ?F ?S
    by (fastforce simp add: inj-on-def set-eq-iff fun-eq-iff)
  show ?thesis
    by (rule finite-imageD [OF 1 2])
qed

lemma not-finite-existsD:
  assumes ¬ finite {a. P a}
  shows ∃a. P a
proof (rule classical)
  assume ¬ ?thesis
  with assms show ?thesis by auto
qed

lemma finite-converse [iff]: finite (r⁻¹) =⇒ finite r
unfolding converse-def conversep-iff
using [[simpnode add: finite-Collect]]
by (auto elim: finite-imageD simp: inj-on-def)

**lemma finite-Domain**: finite \( r \implies \text{finite (Domain } r) \)
by (induct set: finite) auto

**lemma finite-Range**: finite \( r \implies \text{finite (Range } r) \)
by (induct set: finite) auto

**lemma finite-Field**: finite \( r \implies \text{finite (Field } r) \)
by (simp add: Field-def finite-Domain finite-Range)

**lemma finite-Image**: finite \( R \implies \text{finite (R '' A)} \)
by (rule finite-subset[OF - finite-Range]) auto

### 20.3 Further induction rules on finite sets

**lemma finite-ne-induct** [case-names singleton insert, consumes 2]:
assumes finite \( F \) and \( F \neq \{\} \)
assumes \( \forall x. P \{x\} \)
and \( \forall F. \text{finite } F \implies F \neq \{\} \implies x \notin F \implies P F \implies P \text{ (insert } x F) \)
shows \( P F \)
using assms
proof induct
  case empty
  then show ?case by simp
next
  case (insert \( x F \))
  then show ?case by cases auto
qed

**lemma finite-subset-induct** [consumes 2, case-names empty insert]:
assumes finite \( F \) and \( F \subseteq A \)
and empty: \( P \{\} \)
and insert: \( \forall a. \text{finite } F \implies a \in A \implies a \notin F \implies P F \implies P \text{ (insert } a F) \)
shows \( P F \)
using \( \{\text{finite } F\} \) \( \{F \subseteq A\} \)
proof induct
  show \( P \{\} \) by fact
next
  fix \( x F \)
  assume finite \( F \) and \( x \notin F \) and \( P: F \subseteq A \implies P F \) and \( i: \text{insert } x F \subseteq A \)
  show \( P \text{ (insert } x F) \)
  proof (rule insert)
    from \( i \) show \( x \in A \) by blast
    from \( i \) have \( F \subseteq A \) by blast
    with \( P \) show \( P F \).
    show finite \( F \) by fact
    show \( x \notin F \) by fact
  qed

qed
qed

lemma finite-empty-induct:
  assumes finite A
  and P A
  and remove: \( \forall a. \text{finite } A \implies a \in A \implies P A \implies P (A \setminus \{a\}) \)
  shows P {}
proof –
  have P (A - B) if B \subseteq A for B :: 'a set
proof –
  from (finite A) that have finite B
  by (rule rev-finite-subset)
  from this (B \subseteq A) show P (A - B)
  proof
    case empty
    from (P A) show ?case by simp
  next
    case (insert b B)
    have P (A - B - \{b\})
    proof (rule remove)
      from \(\text{finite A}\) show finite (A - B)
      by (rule auto)
      from insert show b \in A - B
      by simp
      from insert show P (A - B)
      by simp
    qed
    also have A - B - \{b\} = A - insert b B
    by (rule Diff-insert [symmetric])
    finally show ?case .
  qed
  qed
  then have P (A - A) by blast
  then show ?thesis by simp
  qed

lemma finite-update-induct [consumes 1, case-names const update]:
  assumes finite: \(\text{finite } \{a. f a \neq c\}\)
  and const: P (\lambda a. c)
  and update: \(\forall a. f \text{ finite } \{a. f a \neq c\} \implies f a = c \implies b \neq c \implies P f \implies P (f(a := b))\)
  shows P f
using finite
proof (induct \(\{a. f a \neq c\}\) arbitrary: f)
  case empty
  with const show ?case by simp
next
  case (insert a A)
  then have A = \{a'. (f(a := c)) a' \neq c\} and f a \neq c
by auto

with \{finite A\} have finite \{a'. (f(a := c)) a' \neq c\}
  by simp

have (f(a := c)) a = c
  by simp

from insert \{A = \{a'. (f(a := c)) a' \neq c\}\} have P (f(a := c))
  by simp

with \{finite \{a'. (f(a := c)) a' \neq c\}\} \{f(a := c)) a = c\ \{f \neq c\}
have P ((f(a := c))(a := f a))
  by (rule update)
then show ?case by simp

qed

lemma finite-subset-induct' [consumes 2, case-names empty insert]:
  assumes finite F and F \subseteq A
  and empty: P \{
  and insert: \forall a. \{finite F; a \in A; F \subseteq A; a \notin F; \} \implies P (insert a F)
  shows P F
  using assms(1,2)
proof induct
  show P \{\} by fact
next
  fix x F
  assume finite F and x \notin F and
  P: F \subseteq A \implies P F and i: insert x F \subseteq A
  show P (insert x F)
proof (rule insert)
  from i show x \in A by blast
  from i have F \subseteq A by blast
  with P show P F .
  show finite F by fact
  show x \notin F by fact
  show F \subseteq A by fact
  qed
qed

20.4 Class finite

class finite =
  assumes finite-UNIV: finite (UNIV :: 'a set)
begin

lemma finite [simp]: finite (A :: 'a set)
  by (rule subset-UNIV finite-UNIV finite-subset)+

lemma finite-code [code]: finite (A :: 'a set) \iff True
  by simp

end
instance `prod :: (finite, finite) finite`
  by standard (simp only: UNIV-Times-UNIV [symmetric] finite-cartesian-product finite)

lemma `inj-graph`: inj (λf. {(x, y). y = f x})
  by (rule inj-onI) (auto simp add: set-eq-iff fun-eq-iff)

instance `fun :: (finite, finite) finite`
proof
  show finite (UNIV :: ('a ⇒ 'b) set)
proof (rule finite-imageD)
    let ?graph = λf::'a ⇒ 'b. {(x, y). y = f x}
    have range ?graph ⊆ Pow UNIV
      by simp
    moreover have finite (Pow (UNIV :: ('a * 'b) set))
      by (simp only: finite-Pow-iff finite)
    ultimately show finite (range ?graph)
      by (rule finite-subset)
    show inj ?graph
      by (rule inj-graph)
  qed
qed

instance `bool :: finite`
  by standard (simp add: UNIV-bool)

instance `set :: (finite) finite`
  by standard (simp only: Pow-UNIV [symmetric] finite-Pow-iff finite)

instance `unit :: finite`
  by standard (simp add: UNIV-unit)

instance `sum :: (finite, finite) finite`
  by standard (simp only: UNIV-Plus-UNIV [symmetric] finite-Plus finite)

20.5 A basic fold functional for finite sets

The intended behaviour is fold f z {x₁, ..., xₙ} = f x₁ (... (f xₙ z)...) if f
is “left-commutative”. The commutativity requirement is relativised to the
carrier set S:

locale comp-fun-commute-on =
  fixes S :: 'a set
  fixes f :: 'a ⇒ 'b ⇒ 'b
  assumes comp-fun-commute-on: x ∈ S ⟹ y ∈ S ⟹ f y ∘ f x = f x ∘ f y
begin

lemma `fun-left-comm`: x ∈ S ⟹ y ∈ S ⟹ f y (f x z) = f x (f y z)
  using comp-fun-commute-on by (simp add: fun-eq-iff)
THEORY "Finite-Set"

lemma commute-left-comp: \( x \in S \Rightarrow y \in S \Rightarrow f y \circ (f x \circ g) = f x \circ (f y \circ g) \) by (simp add: o-assoc comp-fun-commute-on)

end

inductive fold-graph :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b ⇒ bool
  for \( f :: 'a ⇒ 'b ⇒ 'b \) and \( z :: 'b \)
  where
    emptyI [intro]: fold-graph f z \{\} z
  | insertI [intro]: \( x / \notin A \Rightarrow \) fold-graph f z A y ⇒ fold-graph f z (insert x A) (f x y)

inductive-cases empty-fold-graphE [elim!]: fold-graph f z \{\} x

lemma fold-graph-closed-lemma:
  fold-graph f z A x ∧ x ∈ B
  if fold-graph g z A x
  then \( \forall a b. a \in A \Rightarrow b \in B \Rightarrow f a b = g a b \)
  \( \forall a b. a \in A \Rightarrow b \in B \Rightarrow g a b \in B \)
  z ∈ B
  using that (1-3)

proof (induction rule: fold-graph.induct)
  case (insertI x A y)
  have fold-graph f z A y y ∈ B
    unfolding atomize-conj
    by (rule insertI.1H) (auto intro: insertI.prems)
  then have g x y ∈ B and f-eq: \( f x y = g x y \)
    by (auto simp: insertI.prems)
  moreover have fold-graph f z (insert x A) (f x y)
    by (rule fold-graph.insertI; fact)
  ultimately
  show \(?case \)
    by (simp add: f-eq)
qed (auto intro!; that)

lemma fold-graph-closed-eq:
  fold-graph f z A = fold-graph g z A
  if \( \forall a b. a \in A \Rightarrow b \in B \Rightarrow f a b = g a b \)
  \( \forall a b. a \in A \Rightarrow b \in B \Rightarrow g a b \in B \)
  z ∈ B
  using fold-graph-closed-lemma[of f z A - B g] fold-graph-closed-lemma[of g z A - B f] that
  by auto

definition fold :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b
  where fold f z A = (if finite A then (THE y. fold-graph f z A y) else z)

lemma fold-closed-eq: fold f z A = fold g z A
if $\forall a \in A \Rightarrow b \in B \Rightarrow f a b = g a b$
$\forall a \in A \Rightarrow b \in B \Rightarrow g a b \in B$

unfolding Finite-Set.fold-def
by (subst fold-graph-closed-eq[where $B=B$ and $g=g$]) (auto simp: that)

A tempting alternative for the definition is if finite $A$ then THE $y$. fold-graph $f z A y$ else $e$. It allows the removal of finiteness assumptions from the theorems fold-comm, fold-reindex and fold-distrib. The proofs become ugly. It is not worth the effort. (???)

lemma finite-imp-fold-graph:
finite $A$ $\Rightarrow$ $\exists x$. fold-graph $f z A x$
by (induct rule: finite-induct) auto

20.5.1 From fold-graph to fold
context comp-fun-commute-on
begin

lemma fold-graph-finite:
assumes fold-graph $f z A y$
shows finite $A$
using assms by induct simp-all

lemma fold-graph-insertE-aux:
assumes $A \subseteq S$
assumes fold-graph $f z A y a \in A$
shows $\exists y'. y = f a y' \wedge$ fold-graph $f z (A - \{a\}) y'$
using assms(2-1)
proof (induct set: fold-graph)
case emptyI
then showthesis by simp
next
case (insertI $x A y$)
showthesis by simp
proof (cases $x = a$)
case True
with insertI showthesis by auto
next
case False
then obtain $y'$ where $y: y = f a y'$ and $y':$ fold-graph $f z (A - \{a\}) y'$
using insertI by auto
from insertI have $x \in S$ $a \in S$ by auto
then have $f x y = f a (f x y')$
unfolding $y$ by (intro fun-left-comm; simp)
moreover have fold-graph $f z (insert x A - \{a\}) (f x y')$
using $y'$ and $cx \neq a$ and $cx \notin A$
by (simp add: insert-Diff-if fold-graph.insertI)
ultimately showthesis
by fast
THEORY "Finite-Set"

qed

lemma fold-graph-insertE:
  assumes insert x A ⊆ S
  assumes fold-graph f z (insert x A) v and x /∈ A
  obtains y where v = f x y and fold-graph f z A y
  using assms by (auto dest: fold-graph-insertE-aux[OF insert x A ⊆ S - insertI1])

lemma fold-graph-determ:
  assumes A ⊆ S
  assumes fold-graph f z A x fold-graph f z A y
  shows y = x
  using assms(2−,1)
  proof (induct arbitrary: y set: fold-graph)
    case emptyI
    then show ?case by fast
  next
    case (insertI x A y v)
    from ‹insert x A ⊆ S› and ‹fold-graph f z (insert x A) v› and ‹x /∈ A›
    obtain y' where v = f x y' and fold-graph f z A y'
      by (rule fold-graph-insertE)
    from ‹fold-graph f z A y'› insertI have y' = y
      by simp
    with ‹v = f x y'› show v = f x y
      by simp
  qed

lemma fold-equality: A ⊆ S =⇒ fold-graph f z A y =⇒ fold f z A = y
by (cases finite A) (auto simp add: fold-def intro: fold-graph-determ dest: fold-graph-finite)

lemma fold-graph-fold:
  assumes A ⊆ S
  assumes finite A
  shows fold-graph f z A (fold f z A)
proof −
  from ‹finite A› have ∃x. fold-graph f z A x
    by (rule finite-imp-fold-graph)
  moreover note fold-graph-determ[OF ‹A ⊆ S›]
  ultimately have ∃!x. fold-graph f z A x
    by (rule ex-ex1I)
  then have fold-graph f z A (The (fold-graph f z A))
    by (rule theI')
  with assms show ?thesis
    by (simp add: fold-def)
  qed

The base case for fold:
lemma (in −) fold-infinite [simp]: \( \neg \text{finite } A \implies \text{fold } f \ z \ A = z \)
  by (auto simp: fold-def)

lemma (in −) fold-empty [simp]: \( \text{fold } f \ z \ {} = z \)
  by (auto simp: fold-def)

The various recursion equations for fold:

lemma fold-insert [simp]:
  assumes insert x A \( \subseteq \) S finite A and x /\( \notin \) A
  shows \( \text{fold } f \ z \ (\text{insert } x \ A) = f x \ (\text{fold } f \ z \ A) \)
proof (rule fold-equality[OF \( \text{insert } x \ A \subseteq S \)]
  fix z
  from \( \text{insert } x \ A \subseteq S \) finite A have \( \text{fold-graph } f \ z \ A \ (\text{fold } f \ z \ A) \)
  by (blast intro: fold-graph-fold)
with \( x \ /\notin \ A \) have \( \text{fold-graph } f \ z \ (\text{insert } x \ A) \ (f x \ (\text{fold } f \ z \ A)) \)
  by (rule fold-graph-insertI)
then show \( \text{fold-graph } f \ z \ (\text{insert } x \ A) \ (f x \ (\text{fold } f \ z \ A)) \)
  by simp
qed

declare (in −) empty-fold-graphE [rule del] fold-graph-intros [rule del]
— No more proofs involve these.

lemma fold-fun-left-comm:
  assumes insert x A \( \subseteq \) S finite A
  shows \( f x \ (\text{fold } f \ z \ A) = \text{fold } f \ (f x \ z) \ A \)
using assms(2,1) proof (induct rule: finite-induct)
case empty
then show ?case by simp
next
case (insert y F)
then have \( \text{fold } f \ (f x \ z) \ (\text{insert } y \ F) = f y \ (\text{fold } f \ (f x \ z) \ F) \)
  by simp
also have 
\( f x \ (f y \ (\text{fold } f \ z \ F)) \)
  using insert by (simp add: fun-left-comm[where \( ?y=x \)])
also have \( f x \ (\text{fold } f \ z \ (\text{insert } y \ F)) \)
proof =
  from insert have \( \text{insert } y \ F \subseteq S \) by simp
  from fold-insert[OF this] insert show ?thesis by simp
qed
finally show ?case ..

lemma fold-insert2:
  insert x A \( \subseteq \) S \( \Rightarrow \) finite A \( \Rightarrow \) x /\( \notin \) A \( \Rightarrow \) \( \text{fold } f \ z \ (\text{insert } x \ A) = \text{fold } f \ (f x \ z) \ A \)
by (simp add: fold-fun-left-comm)
theory Finite-Set

lemma fold-rec:
  assumes A ⊆ S
  assumes finite A and x ∈ A
  shows fold f z A = f x (fold f z (A - {x}))
proof -
  have A: A = insert x (A - {x})
    using x ∈ A by blast
  then have fold f z A = fold f z (insert x (A - {x}))
    by simp
  also have ... = f x (fold f z (A - {x}))
    by (rule fold-insert) (use assms in auto)
  finally show ?thesis.
qed

lemma fold-insert-remove:
  assumes insert x A ⊆ S
  assumes finite A
  shows fold f z (insert x A) = f x (fold f z (A - {x}))
proof -
  from finite A have finite (insert x A)
    by auto
  moreover have x ∈ insert x A
    by auto
  ultimately have fold f z (insert x A) = f x (fold f z (insert x A - {x}))
    using insert x A ⊆ S by (blast intro: fold-rec)
  then show ?thesis
    by simp
qed

lemma fold-set-union-disj:
  assumes A ⊆ S B ⊆ S
  assumes finite A finite B A ∩ B = {}
  shows Finite-Set.fold f z (A ∪ B) = Finite-Set.fold f (Finite-Set.fold f z A) B
using finite B; assms(1,2,3,5)
proof induct
  case (insert x F)
  have fold f z (A ∪ insert x F) = f x (fold f (fold f z A) F)
    using insert by auto
  also have ... = fold f (fold f z A) (insert x F)
    using insert by (blast intro: fold-insert[ symmetric])
  finally show ?case.
qed simp

end

Other properties of fold:

lemma finite-set-fold-single [simp]: Finite-Set.fold f z {x} = f x z
proof −

have fold-graph f z {x} (f x z)
  by (auto intro: fold-graph.intros)
moreover
{ fix X y
  have fold-graph f z X y \implies (X = \{\} \rightarrow y = z) \land (X = \{x\} \rightarrow y = f x z)
  by (induct rule: fold-graph.induct) auto
}
ultimately have (THE y. fold-graph f z {x} y) = f x z
  by blast
thus ?thesis
  by (simp add: Finite-Set.fold-def)
qed

lemma fold-graph-image:
assumes inj-on g A
shows fold-graph f z (g ` A) = fold-graph (f \circ g) z A
proof
fix w
show fold-graph f z (g ` A) w = fold-graph (f \circ g) z A w
proof
  assume fold-graph f z (g ` A) w
  then show fold-graph (f \circ g) z A w
    using assms
    proof (induct g ` A w arbitrary: A)
      case emptyI
      then show ?case
        by (auto intro: fold-graph.emptyI)
    next
      case (insertI x A r B)
      from inj-on g B \langle x \notin A \rangle insert x A = image g B obtain x’ A’
      where x’ \notin A’ and [simp]: B = insert x’ A’ x = g x’ A = g’ A’
      by (rule inj-img-insertE)
      from insertI.prems have fold-graph (f \circ g) z A’ r
        by (auto intro: insertI.hyps)
      with \langle x’ \notin A’ \rangle have fold-graph (f \circ g) z (insert x’ A’) ((f \circ g) x’ r)
        by (rule fold-graph.insertI)
      then show ?case
        by simp
    qed
    next
      assume fold-graph (f \circ g) z A w
      then show fold-graph f z (g ` A) w
        using assms
        proof (induct)
          case emptyI
          then show ?case
            by (auto intro: fold-graph.emptyI)
        next
case (insertI x A r)
from (x \notin A) insertI.prems have g x \notin g ' A
   by auto
moreover from insertI have fold-graph f z (g ' A) r
   by simp
ultimately have fold-graph f z (insert (g x) (g ' A)) (f (g x) r)
   by (rule fold-graph.insertI)
then show ?case
   by simp
qed

qed

lemma fold-image:
assumes inj-on g A
shows fold f z (g ' A) = fold (f \circ g) z A
proof (cases finite A)
case False
with assms show ?thesis
   by (auto dest: finite-imageD simp add: fold-def)
next
case True
then show ?thesis
   by (auto simp add: fold-def fold-graph-image[OF assms])
qed

lemma fold-cong:
assumes comp-fun-commute-on S f comp-fun-commute-on S g
   and A \subseteq S finite A
   and cong: \(x. \ x \in A \implies f x = g x\)
   and s = t and A = B
shows fold f s A = fold g t B
proof -
have fold f s A = fold g s A
   using (finite A \cdot A \subseteq S \cdot cong
proof (induct A)
case empty
then show ?case by simp
next
case insert
interpret f: comp-fun-commute-on S f by (fact (comp-fun-commute-on S f))
interpret g: comp-fun-commute-on S g by (fact (comp-fun-commute-on S g))
from insert show ?case by simp
qed
with assms show ?thesis by simp
qed

A simplified version for idempotent functions:
locale comp-fun-idem-on = comp-fun-commute-on +
assumes \( \text{comp-fun-idem-on} : x \in S \implies f x \circ f x = f x \)

begin

lemma \( \text{fun-left-idem} : x \in S \implies f x (f x z) = f x z \)
using \( \text{comp-fun-idem-on} \) by (simp add: \text{fun-eq-iff})

lemma \( \text{fold-insert-idem} : \)
assumes \( \text{insert} \ x \ A \subseteq S \)
assumes \( \text{fin} : \text{finite} \ A \)
shows \( \text{fold} \ f \ z (\text{insert} \ x \ A) = f x (\text{fold} \ f \ z \ A) \)
proof cases
assume \( x \in A \)
then obtain \( B \) where \( A = \text{insert} \ x \ B \) and \( x \notin B \)
by (rule \text{set-insert})
then show \( \text{thesis} \)
using \( \text{assms} \) by (simp add: \text{comp-fun-idem-on} \text{fun-left-idem})
next
assume \( x \notin A \)
then show \( \text{thesis} \)
using \( \text{assms} \) by auto
qed

declare \( \text{fold-insert} \ [\text{simp del}] \text{fold-insert-idem} \ [\text{simp}] \)

lemma \( \text{fold-insert-idem2} : \text{insert} \ x \ A \subseteq S \implies \text{finite} \ A \implies \text{fold} \ f \ z (\text{insert} \ x \ A) = \text{fold} \ f (f x z) \ A \)
by (simp add: \text{fold-fun-left-comm})
end

20.5.2 Liftings to \text{comp-fun-commute-on} etc.

lemma \( \text{in} \text{comp-fun-commute-on} \text{comp-comp-fun-commute-on} : \)
\( \text{range} \ g \subseteq S \implies \text{comp-fun-commute-on} \ R (f \circ g) \)
by standard (force intro: \text{comp-fun-commute-on})

lemma \( \text{in} \text{comp-fun-idem-on} \text{comp-comp-fun-idem-on} : \)
assumes \( \text{range} \ g \subseteq S \)
shows \( \text{comp-fun-idem-on} \ R (f \circ g) \)
proof
interpret \( f\! g : \text{comp-fun-commute-on} \ R f \circ g \)
by (fact \text{comp-comp-fun-commute-on}[OF \text{range} \ g \subseteq S])
show \( x \in R \implies y \in R \implies (f \circ g) y \circ (f \circ g) x = (f \circ g) x \circ (f \circ g) y \) for \( x \ y \)
by (fact \text{f-g.comp-fun-commute-on})
qed (use \( \text{range} \ g \subseteq S \) in \( \text{force intro: comp-fun-idem-on} \))

lemma \( \text{in} \text{comp-fun-commute-on} \text{comp-fun-commute-on-funpow} : \)
\( \text{comp-fun-commute-on} \ S (\lambda x. f x \circ g x) \)
proof
fix \( x \) \( y \) assume \( x \in S \) \( y \in S \)
show \( f y \circ f x = f x \circ f y \)
proof (cases \( x = y \))
  case True
  then show ?thesis by simp
next
  case False
  show ?thesis
proof (induct \( g x \) arbitrary: \( g \))
  case 0
  then show ?case by simp
next
  case (Suc \( n \) \( g \))
  have hyp1: \( f y \circ f x = f x \circ f y \)
  proof (induct \( g y \) arbitrary: \( g \))
    case 0
    then show ?case by simp
    next
    case (Suc \( n \) \( g \))
    define \( h \) where \( h z = (\text{Suc } n) \cdot (\text{Suc } h y) \)
    with Suc have \( n = h y \)
      by simp
    with Suc have hyp: \( f y \circ f x = f x \circ f y \)
      by auto
    from Suc h-def have \( g y = \text{Suc } (h y) \)
      by simp
    with \( x \in S; \ y \in S \) show ?case
      by (simp add: comp-assoc hyp) (simp add: o-assoc comp-fun-commute-on)
  qed
  define \( h \) where \( h z = (\text{Suc } x \cdot \text{Suc } y) \) for \( z \)
  with Suc have \( n = h x \)
    by simp
  with Suc have \( f y \circ f x = f x \circ f y \)
    by auto
  with False h-def have hyp2: \( f y \circ f x = f x \circ f y \)
    by simp
  from Suc h-def have \( g x = \text{Suc } (h x) \)
    by simp
  then show ?case
    by (simp del: funpow.simps add: funpow-Suc-right o-assoc hyp2) (simp add: comp-assoc hyp1)
  qed
qed

gdef \( : \cdot a \Rightarrow b \Rightarrow b \)

20.5.3 UNIV as carrier set
locale comp-fun-commute =
fixes \( f :: 'a \Rightarrow 'b \Rightarrow 'b \)
assumes \texttt{comp-fun-commute}: \( f y \circ f x = f x \circ f y \)

begin

lemma \texttt{(in \texttt{\_}) comp-fun-commute-def\': comp-fun-commute} \( f \) = \texttt{comp-fun-commute-on} \( \texttt{UNIV} \ f \)

unfolding \texttt{comp-fun-commute-def} \texttt{comp-fun-commute-on-def} by blast

We abuse the \texttt{rewrites} functionality of locales to remove trivial assumptions that result from instantiating the carrier set to \( \texttt{UNIV} \).

sublocale \texttt{comp-fun-commute-on} \( \texttt{UNIV} \ f \)

rewrites \( \forall X. (X \subseteq \texttt{UNIV}) \equiv \texttt{True} \)

and \( \forall x. x \in \texttt{UNIV} \equiv \texttt{True} \)

and \( \forall P. (\texttt{True} \Rightarrow P) \equiv \texttt{Trueprop} P \)

and \( \forall P Q. (\texttt{True} \Rightarrow \texttt{PROP} P \Rightarrow \texttt{PROP} Q) \equiv (\texttt{PROP} P \Rightarrow \texttt{True} \Rightarrow \texttt{PROP} Q) \)

proof –

show \texttt{comp-fun-commute-on} \( \texttt{UNIV} \ f \)

by standard (simp add: \texttt{comp-fun-commute})

qed simp-all

end

lemma \texttt{(in \texttt{comp-fun-commute}) comp-comp-fun-commute}: \texttt{comp-fun-commute} \( (f \circ g) \)

unfolding \texttt{comp-fun-commute-def}\' by \texttt{(fact comp-comp-fun-commute-on)}

lemma \texttt{(in \texttt{comp-fun-commute}) comp-fun-commute-funpow}: \texttt{comp-fun-commute} \( (\lambda x. f x \ ^^ \ g x) \)

unfolding \texttt{comp-fun-commute-def}\' by \texttt{(fact comp-fun-commute-on-funpow)}

locale \texttt{comp-fun-idem} = \texttt{comp-fun-commute} +

assumes \texttt{comp-fun-idem}: \( f x \circ f x = f x \)

begin

lemma \texttt{(in \texttt{\_}) comp-fun-idem-def\': comp-fun-idem} \( f \) = \texttt{comp-fun-idem-on} \( \texttt{UNIV} \ f \)

unfolding \texttt{comp-fun-idem-def} \texttt{comp-fun-idem-on-def} \texttt{comp-fun-idem-def} \texttt{comp-fun-commute-def}\' by blast

Again, we abuse the \texttt{rewrites} functionality of locales to remove trivial assumptions that result from instantiating the carrier set to \( \texttt{UNIV} \).

sublocale \texttt{comp-fun-idem-on} \( \texttt{UNIV} \ f \)

rewrites \( \forall X. (X \subseteq \texttt{UNIV}) \equiv \texttt{True} \)

and \( \forall x. x \in \texttt{UNIV} \equiv \texttt{True} \)

and \( \forall P. (\texttt{True} \Rightarrow P) \equiv \texttt{Trueprop} P \)

and \( \forall P Q. (\texttt{True} \Rightarrow \texttt{PROP} P \Rightarrow \texttt{PROP} Q) \equiv (\texttt{PROP} P \Rightarrow \texttt{True} \Rightarrow \texttt{PROP} Q) \)

proof –
show comp-fun-idem-on UNIV f
  by standard (simp-all add: comp-fun-idem comp-fun-commute)
qed simp-all

end

lemma (in comp-fun-idem) comp-comp-fun-idem: comp-fun-idem (f o g)
  unfolding comp-fun-idem-def' by (fact comp-comp-fun-idem-on)

20.5.4 Expressing set operations via fold

lemma comp-fun-commute-const: comp-fun-commute (λ- f)
  by standard (rule refl)

lemma comp-fun-idem-insert: comp-fun-idem insert
  by standard auto

lemma comp-fun-idem-remove: comp-fun-idem Set.remove
  by standard auto

lemma (in semilattice-inf) comp-fun-idem-inf: comp-fun-idem inf
  by standard (auto simp add: inf-left-commute)

lemma (in semilattice-sup) comp-fun-idem-sup: comp-fun-idem sup
  by standard (auto simp add: sup-left-commute)

lemma union-fold-insert:
  assumes finite A
  shows A ∪ B = fold insert B A
proof –
  interpret comp-fun-idem insert
  by (fact comp-fun-idem-insert)
  from ‹finite A› show ?thesis
    by (induct A arbitrary: B) simp-all
qed

lemma minus-fold-remove:
  assumes finite A
  shows B − A = fold Set.remove B A
proof –
  interpret comp-fun-idem Set.remove
  by (fact comp-fun-idem-remove)
  from ‹finite A› have fold Set.remove B A = B − A
    by (induct A arbitrary: B) auto
  then show ?thesis ..
qed

lemma comp-fun-commute-filter-fold:
  comp-fun-commute (λx A'. if P x then Set.insert x A' else A')
proof –
interpret comp-fun-idem Set.insert by (fact comp-fun-idem-insert)
show ?thesis by standard (auto simp: fun-eq_iff)
qed

lemma Set-filter-fold:
  assumes finite A
  shows Set.filter P A = fold (λ x A'. if P x then Set.insert x A' else A') {} A
  using assms
proof –
interpret commute-insert: comp-fun-commute (λ x A'. if P x then Set.insert x A' else A')
  by (fact comp-fun-commute-filter-fold)
from ‹finite A› show ?thesis
  by induct (auto simp add: Set.filter_def)
qed

lemma inter-Set-filter:
  assumes finite B
  shows A ∩ B = Set.filter (λ x. x ∈ A) B
  using assms
by induct (auto simp: Set.filter_def)

lemma image-fold-insert:
  assumes finite A
  shows image f A = fold (λ k s. Set.insert (f k) A) {} A
proof –
interpret comp-fun-commute λ k s. Set.insert (f k) A
  by standard auto
show ?thesis
  using assms by (induct A) auto
qed

lemma Ball-fold:
  assumes finite A
  shows Ball A P = fold (λ k s. s ∧ P k) True A
proof –
interpret comp-fun-commute λ k s. s ∧ P k
  by standard auto
show ?thesis
  using assms by (induct A) auto
qed

lemma Bex-fold:
  assumes finite A
  shows Bex A P = fold (λ k s. s ∨ P k) False A
proof –
interpret comp-fun-commute λ k s. s ∨ P k
  by standard auto
show ?thesis
  using assms by (induct A) auto
qed

lemma comp-fun-commute-Pow-fold: comp-fun-commute ($\lambda x \ A. A \cup \text{Set.insert } x \ 'A$)
  by (clarsimp simp: fun-eq-iff comp-fun-commute-def) blast

lemma Pow-fold:
  assumes finite A
  shows Pow A = fold ($\lambda y \ . \ \text{Set.insert } (x, y)$) A
proof –
  interpret comp-fun-commute $\lambda x \ A. A \cup \text{Set.insert } x \ 'A$
  by (rule comp-fun-commute-Pow-fold)
  show ?thesis
    using assms by (induct A) (auto simp: Pow-insert)
qed

lemma fold-union-pair:
  assumes finite B
  shows $\bigcup y \in B. \{(x, y)\} \cup A = fold (\lambda y. \text{Set.insert } (x, y)) B A$
proof –
  interpret comp-fun-commute $\lambda y. \text{Set.insert } (x, y)$
  by standard auto
  show ?thesis
    using assms by (induct arbitrary: A) simp-all
qed

lemma comp-fun-commute-product-fold:
  finite B $\Rightarrow$ comp-fun-commute ($\lambda x z \ . \ \text{fold } (\lambda y. \text{Set.insert } (x, y)) z B$)
  by standard (auto simp: fold-union-pair [symmetric])

lemma product-fold:
  assumes finite A finite B
  shows $A \times B = \text{fold } (\lambda y. \text{Set.insert } (x, y)) z B$
proof –
  interpret commute-product: comp-fun-commute ($\lambda x z. \ \text{fold } (\lambda y. \text{Set.insert } (x, y)) z B$)
  by (fact comp-fun-commute-product-fold[OF finite B])
  from assms show ?thesis unfolding Sigma-def
    by (induct A) (simp-all add: fold-union-pair)
qed

context complete-lattice
begin

lemma inf-Inf-fold-inf:
  assumes finite A
  shows $\text{inf } (\text{Inf } A) \ B = \text{fold } \text{inf } B A$

proof —
interpret comp-fun-idem inf
by (fact comp-fun-idem-inf)
from ‹finite A› fold-fun-left-comm show ?thesis
by (induct A arbitrary: B) (simp-all add: inf-commute fun-eq-iff)
qed

lemma sup-Sup-fold-sup:
  assumes finite A
  shows sup (Sup A) B = fold sup B A
proof —
interpret comp-fun-idem sup
by (fact comp-fun-idem-sup)
from ‹finite A› fold-fun-left-comm show ?thesis
by (induct A arbitrary: B) (simp-all add: sup-commute fun-eq-iff)
qed

lemma Inf-fold-inf: finite A ⇒ Inf A = fold inf top A
  using inf-INF-fold-inf [of A top] by (simp add: inf-absorb2)

lemma Sup-fold-sup: finite A ⇒ Sup A = fold sup bot A
  using sup-SUP-fold-sup [of A bot] by (simp add: sup-absorb2)

lemma inf-INF-fold-inf:
  assumes finite A
  shows inf B (⨅(f ' A)) = fold (inf ∘ f) B A (is ?inf = ?fold)
proof —
interpret comp-fun-idem inf by (fact comp-fun-idem-inf)
interpret comp-fun-idem inf ∘ f by (fact comp-comp-fun-idem)
from ‹finite A› have ?fold = ?inf
  by (induct A arbitrary: B) (simp-all add: inf-left-commute)
then show ?thesis ..
qed

lemma sup-SUP-fold-sup:
  assumes finite A
  shows sup B (⨆(f ' A)) = fold (sup ∘ f) B A (is ?sup = ?fold)
proof —
interpret comp-fun-idem sup by (fact comp-fun-idem-sup)
interpret comp-fun-idem sup ∘ f by (fact comp-comp-fun-idem)
from ‹finite A› have ?fold = ?sup
  by (induct A arbitrary: B) (simp-all add: sup-left-commute)
then show ?thesis ..
qed

lemma INF-fold-inf: finite A ⇒ ⨅(f ' A) = fold (inf ∘ f) top A
  using inf-INF-fold-inf [of A top] by simp

lemma SUP-fold-sup: finite A ⇒ ⨆(f ' A) = fold (sup ∘ f) bot A
THEORY “Finite-Set”

using sup-SUP-fold-sup [of A bot] by simp

lemma finite-Inf-in:
  assumes finite A A ≠ {} and inf: \( \forall x y. [x \in A; y \in A] \implies \inf x y \in A \)
  shows Inf A ∈ A
proof –
  have Inf B ∈ A if B ≤ A B ≠ {} for B
  using finite-subset [OF \( B \subseteq A \). finite A] that
  by (induction B) (use inf in \( \text{force+} \))
  then show ?thesis
  by (simp add: assms)
qed

lemma finite-Sup-in:
  assumes finite A A ≠ {} and sup: \( \forall x y. [x \in A; y \in A] \implies \sup x y \in A \)
  shows Sup A ∈ A
proof –
  have Sup B ∈ A if B ≤ A B ≠ {} for B
  using finite-subset [OF \( B \subseteq A \). finite A] that
  by (induction B) (use sup in \( \text{force+} \))
  then show ?thesis
  by (simp add: assms)
qed

end

20.5.5 Expressing relation operations via fold

lemma Id-on-fold:
  assumes finite A
  shows Id-on A = Finite-Set.fold (λx. Set.insert (Pair x x)) {} A
proof –
  interpret comp-fun-commute λx. Set.insert (Pair x x)
  by standard auto
  from assms show ?thesis
  unfolding Id-on-def by (induct A) simp-all
qed

lemma comp-fun-commute-Image-fold:
  comp-fun-commute (λ(x, y) A. if x ∈ S then Set.insert y A else A)
proof –
  interpret comp-fun-idem Set.insert
  by (fact comp-fun-idem-insert)
  show ?thesis
  by standard (auto simp: fun-eq_iff comp-fun-commute split: prod.split)
qed

lemma Image-fold:
  assumes finite R
shows $R :: S = \text{Finite-Set.fold} \ (\lambda(x,y) \ A. \text{if } x \in S \text{ then } \text{Set.insert} \ y \ A \text{ else } A) \ [] \ R$

proof –
interpret \text{comp-fun-commute} \ (\lambda(x,y) \ A. \text{if } x \in S \text{ then } \text{Set.insert} \ y \ A \text{ else } A)
by (rule \text{comp-fun-commute-Image-fold})
have *: $\forall \ A' \in X. \text{Set.insert} \ x \ F' = (\text{if } \text{fst } x \in S \text{ then } \text{Set.insert} \ (\text{snd } x) \ (F' :: S) \text{ else } (F' :: S))$
by (force intro: \text{rev-ImageI})
show ?thesis
using assms by (induct R) (auto simp: *)
qed

lemma \text{insert-relcomp-union-fold}:
assumes fin: $\text{finite } S$
shows $\{x\} \ O S \cup X = \text{Finite-Set.fold} \ (\lambda(w,z) \ A'. \text{if } \text{snd } x = w \text{ then } \text{Set.insert} \ (\text{fst } x, z) \ A' \text{ else } A') \ X S$
proof –
interpret \text{comp-fun-commute} \ (\lambda(w,z) \ A'. \text{if } \text{snd } x = w \text{ then } \text{Set.insert} \ (\text{fst } x, z) \ A' \text{ else } A')
by (fact \text{comp-fun-idem-insert})
show \text{comp-fun-commute} \ (\lambda(w,z) \ A'. \text{if } \text{snd } x = w \text{ then } \text{Set.insert} \ (\text{fst } x, z) \ A' \text{ else } A')
by standard (auto simp add: \text{fun-eq-iff split: prod.split})
qed

have *: $\{x\} \ O S = \{(x', z), x' = \text{fst } x \land (\text{snd } x, z) \in S\}$
by (auto simp: \text{relcomp-unfold intro!: exI})
show ?thesis
unfolding * using \text{finite } S by (induct S) (auto split: prod.split)
qed

lemma \text{insert-relcomp-fold}:
assumes fin: $\text{finite } S$
shows $\text{Set.insert} \ x \ R \ O S = \text{Finite-Set.fold} \ (\lambda(w,z) \ A'. \text{if } \text{snd } x = w \text{ then } \text{Set.insert} \ (\text{fst } x, z) \ A' \text{ else } A') \ (R \ O S) S$
proof –
have $\text{Set.insert} \ x \ R \ O S = (\{x\} \ O S) \cup (R \ O S)$
by auto
then show ?thesis
by (auto simp: \text{insert-relcomp-union-fold} [OF assms])
qed

lemma \text{comp-fun-commute-relcomp-fold}:
assumes fin: $\text{finite } S$
shows \text{comp-fun-commute} \ (\lambda(x,y) \ A. \text{if } y = w \text{ then } \text{Set.insert} \ (x, z) \ A' \text{ else } A') \ A S$
proof –
have $\exists a b A$.

\[
\text{Finite-Set.fold } (\lambda(w, z) A') \text{ if } b = w \text{ then } \text{Set.insert } (a, z) A' \text{ else } A' = \{(a,b)\} \cup A
\]
by (auto simp: insert-relcomp-union-fold[OF assms] cong: if-cong)
show \text{?thesis}
by standard (auto simp: \*)
qued

lemma relcomp-fold:
assumes finite R finite S
shows $R \circ S = \text{Finite-Set.fold } (\lambda(w, z) A') \text{ if } y = w \text{ then } \text{Set.insert } (x, z) A' \text{ else } A'$
proof –
interpret commute-relcomp-fold: comp-fun-commute
(\lambda(x, y) A. \text{Finite-Set.fold } (\lambda(w, z) A') \text{ if } y = w \text{ then } \text{insert } (x, z) A' \text{ else } A')
A S
by (fact comp-fun-commute-relcomp-fold[OF \text{finite } S])
from assms show \text{?thesis}
by (induct R) (auto simp: comp-fun-commute-relcomp-fold insert-relcomp-fold cong: if-cong)
qued

20.6 Locales as mini-packages for fold operations

20.6.1 The natural case

locale folding-on =
fixes S :: 'a set
fixes f :: 'a \Rightarrow 'b \Rightarrow 'b and z :: 'b
assumes comp-fun-commute-on: $x \in S \Rightarrow y \in S \Rightarrow f y o f x = f x o f y$
begin
interpretation fold?: comp-fun-commute-on S f
by standard (simp add: comp-fun-commute-on)
definition F :: 'a set \Rightarrow 'b
where eq-fold: $F A = \text{Finite-Set.fold } f z A$

lemma empty [simp]: $F \{} = z$
by (simp add: eq-fold)

lemma infinite [simp]: \neg \text{finite } A \Rightarrow F A = z
by (simp add: eq-fold)

lemma insert [simp]:
assumes insert x A \subseteq S and finite A and x \notin A
shows $F (\text{insert } x A) = f x (F A)$
proof –
from fold-insert assms
have Finite-Set.fold f z (insert x A) 
  = f x (Finite-Set.fold f z A) 
  by simp 
with ‹finite A› show ?thesis by (simp add: eq-fold fun-eq-iff)
qed

lemma remove:
  assumes A ⊆ S and finite A and x ∈ A 
  shows F A = f x (F (A - {x})) 
proof - 
  from ‹x ∈ A› obtain B where A = insert x B and x ∉ B 
    by (auto dest: mk-disjoint-insert) 
  moreover from ‹finite A› A have finite B by simp 
  ultimately show ?thesis 
    using ‹A ⊆ S› by auto 
qed

lemma insert-remove:
  assumes insert x A ⊆ S and finite A 
  shows F (insert x A) = f x (F (A - {x})) 
using assms by (cases x ∈ A) (simp-all add: remove insert-absorb)
end

20.6.2 With idempotency
locale folding-idem-on = folding-on + 
  assumes comp-fun-idem-on: x ∈ S ⇒ y ∈ S ⇒ f x o f x = f x 
begin 
declare insert [simp del]
interpretation fold?: comp-fun-idem-on S f 
  by standard (simp-all add: comp-fun-commute-on comp-fun-idem-on)
lemma insert-idem [simp]:
  assumes insert x A ⊆ S and finite A 
  shows F (insert x A) = f x (F A) 
proof - 
  from fold-insert-idem assms 
  have fold f z (insert x A) = f x (fold f z A) by simp 
  with ‹finite A› show ?thesis by (simp add: eq-fold fun-eq-iff)
qed
end

20.6.3 UNIV as the carrier set
locale folding = 
  fixes f :: 'a ⇒ 'b and z :: 'b
assumes \( \text{comp-fun-commute}: f \circ y = y \circ f \) 

begin

lemma (in -) folding-def': folding \( f = \text{folding-on UNIV} \ f \) 
unfolding folding-def folding-on-def by blast 

Again, we abuse the \texttt{rewrites} functionality of locales to remove trivial assumptions that result from instantiating the carrier set to \( \text{UNIV} \).

sublocale folding-on UNIV \( f \)
rewrites \( \bigwedge X. (X \subseteq \text{UNIV}) \equiv \text{True} \)
and \( \bigwedge x. x \in \text{UNIV} \equiv \text{True} \)
and \( \bigwedge P. (\text{True} \implies P) \equiv \text{Trueprop P} \)
and \( \bigwedge P Q. (\text{True} \implies \text{PROP P} \implies \text{PROP Q}) \equiv (\text{PROP P} \implies \text{True} \implies \text{PROP Q}) \)
proof -
show folding-on UNIV \( f \) 
by standard (simp add: comp-fun-commute)
qed simp-all 

end

locale folding-idem = folding + 
assumes \( \text{comp-fun-idem}: f \circ f = f \)
begin

lemma (in -) folding-idem-def': folding-idem \( f = \text{folding-idem-on UNIV} \ f \) 
unfolding folding-idem-def folding-def folding-idem-on-def 
unfolding folding-idem-axioms-def folding-idem-on-axioms-def 
by blast 

Again, we abuse the \texttt{rewrites} functionality of locales to remove trivial assumptions that result from instantiating the carrier set to \( \text{UNIV} \).

sublocale folding-idem-on UNIV \( f \)
rewrites \( \bigwedge X. (X \subseteq \text{UNIV}) \equiv \text{True} \)
and \( \bigwedge x. x \in \text{UNIV} \equiv \text{True} \)
and \( \bigwedge P. (\text{True} \implies P) \equiv \text{Trueprop P} \)
and \( \bigwedge P Q. (\text{True} \implies \text{PROP P} \implies \text{PROP Q}) \equiv (\text{PROP P} \implies \text{True} \implies \text{PROP Q}) \)
proof -
show folding-idem-on UNIV \( f \) 
by standard (simp add: comp-fun-idem)
qed simp-all 

end

20.7 Finite cardinality

The traditional definition \( \text{card} \ A \equiv \text{LEAST} \ n. \ \exists f. \ A = \{ f \ i \ | \ i. \ i < n \} \) is ugly to work with. But now that we have \texttt{fold} things are easy:
global-interpretation card: folding λ. Suc 0
defines card = folding-on.F (λ. Suc) 0
by standard (rule refl)

lemma card-insert-disjoint: finite A → x ∉ A → card (insert x A) = Suc (card A)
by (fact card.insert)

lemma card-insert-if: finite A → card (insert x A) = (if x ∈ A then card A else Suc (card A))
by auto (simp add: card.insert-remove card.remove)

lemma card-ge-0-finite: card A > 0 → finite A
by (rule ccontr) simp

lemma card-0-eq [simp]: finite A → card A = 0 ↔ A = {}
by (auto dest: mk-disjoint-insert)

lemma finite-UNIV-card-ge-0: finite (UNIV :: 'a set) → card (UNIV :: 'a set) > 0
by (rule ccontr) simp

lemma card-eq-0-iff: card A = 0 ↔ A = {} ∨ ¬ finite A
by auto

lemma card-range-greater-zero: finite (range f) → card (range f) > 0
by (rule ccontr) simp add: card-eq-0-iff

lemma card-Suc-Diff1:
assumes finite A x ∈ A shows Suc (card (A - {x})) = card A
proof –
have Suc (card (A - {x})) = card (insert x (A - {x}))
  using assms by (simp add: card.insert-remove)
also have ... = card A
  using assms by (simp add: card-insert-if)
finally show ?thesis .
qed

lemma card-insert-le-m1:
assumes n > 0 card y ≤ n - 1 shows card (insert x y) ≤ n
using assms by (cases finite y) (auto simp: card-insert-if)

lemma card-Diff-singleton:
assumes x ∈ A shows card (A - {x}) = card A - 1
proof (cases finite A)
case True
with assms show \(?thesis
  by (simp add: card-Suc-Diff1 [symmetric])
qed auto

lemma card-Diff-singleton-if:
  card \((A - \{x\})\) = (if \(x \in A\) then card \(A\) - 1 else card \(A\))
by (simp add: card-Diff-singleton)

lemma card-Diff-insert[simp]:
  assumes \(a \in A\) and \(a \notin B\)
  shows \(card \((A - insert a B)\) = card \((A - B)\) - 1\)
proof -
  have \(A - insert a B = (A - B) - \{a\}\)
    using assms by blast
  then show \(?thesis
    using assms by (simp add: card-Diff-singleton)
qed

lemma card-insert-le: card \(A\) \(\leq\) card \((insert x A)\)
proof (cases finite \(A\))
case True
  then show \(?thesis
    by (simp add: card-insert-if)
qed auto

lemma card-Collect-less-nat[simp]: card \(\{i :: nat. i < n\}\) = \(n\)
by (induct \(n\)) (simp-all add: less-Suc-eq Collect-disj-eq)

lemma card-Collect-le-nat[simp]: card \(\{i :: nat. i \leq n\}\) = Suc \(n\)
using card-Collect-less-nat[of Suc \(n\)]
by (simp add: less-Suc-eq-le)

lemma card-mono:
  assumes finite \(B\) and \(A \subseteq B\)
  shows card \(A\) \(\leq\) card \(B\)
proof -
  from assms have finite \(A\)
    by (auto intro: finite-subset)
  then show \(?thesis
    using assms
  proof (induct \(A\) arbitrary: \(B\))
    case empty
    then show \(?case by simp

next
case (insert \(x\) \(A\))
  then have \(x \in B\)
    by simp
  from insert have \(A \subseteq B - \{x\}\) and finite \((B - \{x\})\)
    by auto
  with insert.hyps have card \(A\) \(\leq\) card \((B - \{x\})\)
by auto

with (finite A) (x \notin A) (finite B) (x \in B) show ?case
  by simp (simp only: card.remove)

qed

lemma card-seteq:
  assumes finite B and A: A \subseteq B card B \leq card A
  shows A = B
  using assms
proof (induction arbitrary: A rule: finite-induct)
  case (insert b B)
  then have A: finite A A - {b} \subseteq B
    by force
  then have card B \leq card (A - {b})
    using insert by (auto simp add: card-Diff-singleton-if)
  then have A - {b} = B
    using A insert.IH by auto
  then show ?case
    using insert.hyps insert.prems by auto
qed auto

lemma psubset-card-mono: finite B \Rightarrow A < B \Rightarrow card A < card B
  using card-seteq [of A B] by (auto simp add: psubset-eq)

lemma card-Un-Int:
  assumes finite A finite B
  shows card A + card B = card (A \cup B) + card (A \cap B)
  using assms
proof (induct A)
  case empty
  then show ?case by simp
next
  case insert
  then show ?case
    by (auto simp add: insert-absorb Int-insert-left)
qed

lemma card-Un-disjoint: finite A \Rightarrow finite B \Rightarrow A \cap B = \{\} \Rightarrow card (A \cup B) = card A + card B
  using card-Un-Int [of A B] by simp

lemma card-Un-disjnt: [finite A; finite B; disjnt A B] \Rightarrow card (A \cup B) = card A + card B
  by (simp add: card-Un-disjoint disjnt-def)

lemma card-Un-le: card (A \cup B) \leq card A + card B
proof (cases finite A \land finite B)
  case True
then show ?thesis
  using le_iff_add card-Un-Int [of A B] by auto
qed auto

lemma card-Diff-subset:
  assumes finite B
  and B ⊆ A
  shows card (A - B) = card A - card B
  using assms
proof (cases finite A)
  case False
  with assms show ?thesis
    by simp
next
  case True
  with assms show ?thesis
    by (induct B arbitrary: A) simp-all
qed

lemma card-Diff-subset-Int:
  assumes finite (A ∩ B)
  shows card (A - B) = card A - card (A ∩ B)
proof
  have A - B = A - A ∩ B by auto
  with assms show ?thesis
    by (simp add: card-Diff-subset)
qed

lemma card-Int-Diff:
  assumes finite A
  shows card A = card (A ∩ B) + card (A - B)
  by (simp add: assms card-Diff-subset-Int card-mono)

lemma diff-card-le-card-Diff:
  assumes finite B
  shows card A - card B ≤ card (A - B)
proof
  have card A - card B ≤ card A - card (A ∩ B)
    using card_mono[OF assms Int-lower2, of A] by arith
  also have ... = card (A - B)
    using assms by (simp add: card-Diff-subset-Int)
  finally show ?thesis .
qed

lemma card-le-sym-Diff:
  assumes finite A finite B card A ≤ card B
  shows card(A - B) ≤ card(B - A)
proof
  have card(A - B) = card A - card (A ∩ B) using assms(1,2) by (simp add:
card-Diff-subset-Int
also have \ldots \leq \card{B - (A \cap B)} using assms(3) by linarith
also have \ldots = \card{(B - A)} using assms(1,2) by (simp add: card-Diff-subset-Int Int-commute)
finally show \?thesis.
qed

lemma card-less-sym-Diff:
  assumes finite A finite B \card{A} < \card{B}
  shows {\card{A - B}} < {\card{B - A}}
proof
  have \card{A - B} = \card{A} - \card{A \cap B} using assms(1,2) by (simp add: card-Diff-subset-Int)
  also have \ldots < \card{B - card (A \cap B)} using assms(3) by (simp add: card-mono diff-less-mono)
  also have \ldots = \card{(B - A)} using assms(1,2) by (simp add: card-Diff-subset-Int Int-commute)
  finally show \?thesis.
qed

lemma card-Diff1-less-iff: \card{(A - \{x\})} < \card{A} \iff finite A \land x \in A
proof (cases finite A \land x \in A)
  case True
  then show \?thesis by (auto simp: card-gt-0-iff intro: diff-less)
qed auto

lemma card-Diff1-less: finite A \implies x \in A \implies \card{(A - \{x\})} < \card{A}
unfolding card-Diff1-less-iff by auto

lemma card-Diff2-less:
  assumes finite A x \in A y \in A shows \card{(A - \{x\} - \{y\})} < \card{A}
proof (cases x = y)
  case True
  with assms show \?thesis by (simp add: card-Diff1-less del: card-Diff-insert)
next
  case False
  then have \card{(A - \{x\}) - \{y\}} < \card{(A - \{x\})} \land \card{(A - \{x\})} < \card{A}
    using assms by (intro card-Diff1-less; simp+)
  then show \?thesis by (blast intro: less-trans)
qed

lemma card-Diff1-le: \card{(A - \{x\})} \leq \card{A}
proof (cases finite A)
  case True
  then show \?thesis by (cases x \in A) (simp-all add: card-Diff1-less less-imp-le)
qed auto

lemma card-psubset: finite B \implies A \subseteq B \implies \text{card } A < \text{card } B \implies A < B
by (erule psubsetI) blast

lemma card-le-inj:
assumes fA: finite A
and fB: finite B
and c: \text{card } A \leq \text{card } B
shows \exists f. f : A \subseteq B \land \text{inj-on } f A
using fA fB c
proof (induct arbitrary: B rule: finite-induct)
case empty
then show ?case by simp
next
case (insert x s t)
then show ?case
proof (induct rule: finite-induct [OF insert.prems(1)])
case 1
then show ?case by simp
next
case (2 y t)
from 2.prems(1,2,5) 2.hyps(1,2) have \text{cst}: \text{card } s \leq \text{card } t
by simp
from 2.prems(3) [OF 2.hyps(1) \text{cst}]
obtain f where *: f : s \subseteq t \land \text{inj-on } f s
by blast
let \?g = (\lambda a. \text{if } a = x \text{ then } y \text{ else } f a)
have \?g : \text{insert } x s \subseteq \text{insert } y t \land \text{inj-on } \?g \text{ (insert } x s)
using * 2.prems(2) 2.hyps(2) unfolding \text{inj-on-def}
by auto
then show ?case by (rule exI[where ?x=\?g])
qed

lemma card-subset-eq:
assumes fB: finite B
and AB: A \subseteq B
and c: \text{card } A = \text{card } B
shows A = B
proof
from fB AB have fA: finite A
by (auto intro: finite-subset)
from fA fB have fBA: finite (B - A)
by auto
have e: A \cap (B - A) = {}
by blast
have eq: A \cup (B - A) = B
using AB by blast
from card-Un-disjoint[OF fA fBA e, unfolded eq c] have \text{card } (B - A) = 0
by arith
then have \( B - A = \{\} \)
  unfolding \( \text{card-eq-0-iff} \) using \( fA fB \) by simp
with \( A \cap B \) show \( A = B \)
  by blast
qed

lemma \( \text{insert-partition} \):
  \( x \notin F \implies \forall c1 \in \text{insert} x F. \forall c2 \in \text{insert} x F. c1 \neq c2 \implies c1 \cap c2 = \{\} \implies x \cap \bigcup F = \{\} \)
  by auto

lemma \( \text{finite-psubset-induct} \) [consumes 1, case-names psubset]:
  assumes \( \text{finite} A \)
  and \( \text{major} \): \( \forall A. \text{finite} A \implies (\forall B. B \subset A \implies P B) \implies P A \)
  shows \( P A \)
  using \( \text{finite} \)
  proof (induct \( A \) taking \( \text{card} \) rule: \( \text{measure-induct-rule} \))
    case \( \text{less} A \)
    have \( \text{fin} \): \( \text{finite} A \) by fact
    have \( \text{ih} \): \( \text{card} B < \text{card} A \implies \text{finite} B \implies P B \) for \( B \) by fact
    have \( P B \) if \( B \subset A \) for \( B \)
    proof
      from that have \( \text{card} B < \text{card} A \)
        using \( \text{psubset-card-mono} \) \( \text{fin} \) by blast
      moreover
      from that have \( B \subset A \)
        by auto
      then have \( \text{finite} B \)
        using \( \text{fin \ finite-subset} \) by blast
      ultimately show \( \text{thesis} \) using \( \text{ih} \) by simp
    qed
    with \( \text{fin} \) show \( P A \) using \( \text{major} \) by blast
    qed

lemma \( \text{finite-induct-select} \) [consumes 1, case-names empty select]:
  assumes \( \text{finite} S \)
  and \( P \{\} \)
  and \( \text{select} \): \( \forall T. T \subset S \implies P T \implies \exists s \in S - T. P (\text{insert} s T) \)
  shows \( P S \)
  proof
    have \( 0 \leq \text{card} S \) by simp
    then have \( \exists T \subseteq S. \text{card} T = \text{card} S \land P T \)
    proof (induct rule: \( \text{dec-induct} \))
      case \( \text{base} \) with \( \{P \{\}\} \)
      show \( ?\text{case} \)
        by (intro \( \text{exI} \) [of - \( \{\} \)]) auto
    next
      case \( \text{step} n \)
then obtain \( T \) where \( T \subseteq S \) \( \text{card} \ T = n \ P \ T \)
by auto
with \( \langle n < \text{card} S \rangle \) have \( T \subseteq S \ P \ T \)
by auto
with \( \text{select} \ [\text{of} \ T] \) obtain \( s \) where \( s \in S \ s \notin T \ P \ (\text{insert} \ s \ T) \)
by auto
with \( \langle \text{finite} S \rangle \) show \( \sim \)case
by \( \text{(intro exI [of - \text{insert} \ s \ T])} \) \( \text{(auto dest: finite-subset)} \)
qed

\text{lemma remove-induct} [\text{case-names empty infinite remove}]:
assumes \( \text{empty:} \ P \ \langle \{\} :: 'a \ \text{set} \rangle \)
and \( \text{infinite:} \sim \text{finite} B \implies P B \)
and \( \text{remove:} \ \forall A. \text{finite} A \implies A \neq \{\} \implies A \subseteq B \implies (\forall x. x \in A \implies P (A - \{x\})) \implies P A \)
shows \( P B \)
proof (cases finite B)
  case False
  then show \( \sim \)thesis by (rule infinite)
next
  case True
  define \( A \) where \( A = B \)
  with \( \text{True} \) have \( \text{finite} A \ A \subseteq B \)
  by simp-all
  then show \( P A \)
  proof (induct card A arbitrary: A)
    case 0
    then have \( A = \{\} \) by auto
    with \( \text{empty} \) show \( \sim \)case by simp
  next
    case (Suc \ n \ A)
    from \( \langle A \subseteq B \rangle \) and \( \langle \text{finite} B \rangle \) have \( \text{finite} A \)
    by (rule finite-subset)
    moreover from Suc.hyps have \( A \neq \{\} \) by auto
    moreover note \( \langle A \subseteq B \rangle \)
    moreover have \( P (A - \{x\}) \) if \( x: x \in A \) for \( x \)
    using \( x \text{ Suc.prems} \langle \text{Suc} \ n = \text{card} \ A \rangle \) by (intro Suc) auto
    ultimately show \( \sim \)case by (rule remove)
  qed
qed

\text{lemma finite-remove-induct} [\text{consumes} 1, \text{case-names empty remove}]:
fixes \( P :: 'a \ \text{set} \Rightarrow \text{bool} \)
assumes \( \text{finite} B \)
and \( P \ \{\} \)
and \( \forall A. \text{finite} A \implies A \neq \{\} \implies A \subseteq B \implies (\forall x. x \in A \implies P (A - \{x\})) \)
Main cardinality theorem.

**Lemma** `card-partition` [rule-format]:

- `finite C ⇒ finite (∪C) ⇒ (∀c∈C. card c = k) ⇒ ∃c1 ∈ C. ∀c2 ∈ C. c1 ≠ c2 → c1 ∩ c2 = {}`
- `k * card C = card (∪C)`

**Proof** (induct rule: finite-induct)

- **Case** `empty`
  - then show `?case` by simp

- **Next**
  - **Case** `(insert x F)`
    - then show `?case` by `(simp add: card-Un-disjoint insert-partition finite-subset [of - (∪(insert - -))])`

**Qed**

**Lemma** `card-eq-UNIV-imp-eq-UNIV`:

- `assumes fin: finite (UNIV :: 'a set)`
  - and `card: card A = card (UNIV :: 'a set)`
- shows `A = (UNIV :: 'a set)`

**Proof**

- show `A ⊆ UNIV` by simp
- show `UNIV ⊆ A`
  - show `x ∈ A` for `x`
    - proof (rule ccontr)
      - assume `x /∈ A`
      - then have `A ⊂ UNIV` by auto
        - with `fin` have `card A < card (UNIV :: 'a set)`
          - by (fact psubset-card-mono)
          - with `card` show `False` by simp
    - qed
  - qed
- qed

The form of a finite set of given cardinality

**Lemma** `card-eq-SucD`:

- `assumes card A = Suc k`
- shows `∃b B. A = insert b B ∧ b /∈ B ∧ card B = k ∧ (k = 0 → B = {})`

**Proof**

- have `fin: finite A` using `assms` by (auto intro: ccontr)
- moreover have `card A ≠ 0` using `assms` by auto
- ultimately obtain `b` where `b: b ∈ A`
  - by auto
show \( ?\)thesis
proof (intro exI conjI)
  show \( A = \text{insert} \ b \ (A - \{b\}) \)
    using \( \) by blast
  show \( b \notin A - \{b\} \)
    by blast
  show \( \text{card} \ (A - \{b\}) = k \) and \( k = 0 \longrightarrow A - \{b\} = \{\} \)
    using \( \) by (fastforce dest: mk-disjoint-insert)+
qed

lemma card-Suc-eq:
  \( \text{card} \ A = \text{Suc} \ k \longleftrightarrow \) \( (\exists \ b. \ A = \text{insert} \ b \ B \land b \notin B \land \text{card} \ B = k \land (k = 0 \longrightarrow B = \{\})) \)
by (auto simp: card-insert-if card-gt-0-iff elim: card-eq-SucD)

lemma card-Suc-eq-finite:
  \( \text{card} \ A = \text{Suc} \ k \longleftrightarrow (\exists \ b. \ A = \text{insert} \ b \ B \land b \notin B \land \text{card} \ B = k \land \text{finite} \ B) \)
unfolding card-Suc-eq using card-gt-0-iff by fastforce

lemma card-1-singletonE:
  assumes \( \text{card} \ A = 1 \)
  obtains \( x \) where \( A = \{x\} \)
using \( \) by (auto simp: card-Suc-eq)

lemma is-singleton-altdef: is-singleton \( A \longleftrightarrow \text{card} \ A = 1 \)
unfolding is-singleton-def
by (auto elim!: card-1-singletonE is-singletonE simp del: One-nat-def)

lemma card-1-singleton-iff: card \( A = \text{Suc} \ 0 \longleftrightarrow (\exists x. \ A = \{x\}) \)
by (simp add: card-Suc-eq)

lemma card-le-Suc0-iff-eq:
  assumes \( \text{finite} \ A \)
  shows \( \text{card} \ A \leq \text{Suc} \ 0 \longleftrightarrow (\forall a1 \in A. \forall a2 \in A. \ a1 = a2) \) (is \( ?C = ?A \))
proof
  assume \( ?C \) thus \( ?A \) using \( \) by (auto simp: le-Suc-eq dest: card-eq-SucD)
next
  assume \( ?A \)
  show \( ?C \)
proof cases
  assume \( A = \{\} \) thus \( ?C \) using \( ?A \) by simp
next
  assume \( A \neq \{\} \)
  then obtain \( a \) where \( A = \{a\} \) using \( ?A \) by blast
  thus \( ?C \) by simp
qed

qed
lemma card-le-Suc-iff:
Suc n ≤ card A = (∀ a B. A = insert a B ∧ a ∉ B ∧ n ≤ card B ∧ finite B)
proof (cases finite A)
  case True
  then show ?thesis
  by (fastforce simp: card-Suc-eq less-eq-nat.split nat.splits)
qed auto

lemma finite-fun-UNIVD2:
assumes fin: finite (UNIV :: ('a ⇒ 'b) set)
sows finite (UNIV :: 'b set)
proof –
  from fin have finite (range (λf :: 'a ⇒ 'b. f arbitrary)) for arbitrary
  by (rule finite-imageI)
  moreover have UNIV = range (λf :: 'a ⇒ 'b. f arbitrary) for arbitrary
  by (rule UNIV-eq-I) auto
  ultimately show finite (UNIV :: 'b set)
  by simp
qed

lemma card-UNIV-unit [simp]: card (UNIV :: unit set) = 1
  unfolding UNIV-unit by simp

lemma infinite-arbitrarily-large:
assumes ¬ finite A
shows ∃ B. finite B ∧ card B = n ∧ B ⊆ A
proof (induction n)
  case 0
  show ?case by (intro exI[of _ - {}]) auto
next
  case (Suc n)
  then obtain B where B: finite B ∧ card B = n ∧ B ⊆ A ..
  with (¬ finite A) have A ≠ B by auto
  with B have B ⊆ A by auto
  then have ∃ x. x ∈ A − B
    by (elim psubset-imp-ex-mem)
  then obtain x where x: x ∈ A − B ..
  with B have finite (insert x B) ∧ card (insert x B) = Suc n ∧ insert x B ⊆ A
    by auto
  then show ∃ B. finite B ∧ card B = Suc n ∧ B ⊆ A ..
qed

Sometimes, to prove that a set is finite, it is convenient to work with finite subsets and to show that their cardinalities are uniformly bounded. This possibility is formalized in the next criterion.

lemma finite-if-finite-subsets-card-bdd:
assumes ∃ G. G ⊆ F ⇒ finite G ⇒ card G ≤ C
shows finite F ∧ card F ≤ C
proof (cases finite F)
THEORY "Finite-Set"

case False
obtain n::nat where n: n > max C 0 by auto
obtain G where G: G ⊆ F card G = n using infinite-arbitrarily-large[of False] by auto
hence finite G using {n > max C 0} using card.infinite gr-implies-not0 by blast
hence False using assms G n not-less by auto
thus ?thesis ..
next
case True thus ?thesis using assms[of F] by auto
qed

lemma obtain-subset-with-card-n:
assumes n ≤ card S
obtains T where T ⊆ S card T = n finite T
proof –
obtain n' where card S = n + n'
  using le-Suc-ex[of assms] by blast
with that show thesis
proof (induct n' arbitrary: S)
case 0
  thus ?case by (cases finite S) auto
next
case Suc
  thus ?case by (auto simp add: card-Suc-eq)
qed
qed

lemma exists-subset-between:
assumes
  card A ≤ n
  n ≤ card C
  A ⊆ C
  finite C
shows ∃ B. A ⊆ B ∧ B ⊆ C ∧ card B = n
using assms
proof (induct n arbitrary: A C)
case 0
thus ?case using finite-subset[of A C] by (intro exI[of - {}], auto)
next
case (Suc n A C)
show ?case
proof (cases A = {}) 
case True
  from obtain-subset-with-card-n[of Suc(3)]
obtain B where B ⊆ C card B = Suc n by blast
thus ?thesis unfolding True by blast
next
case False
then obtain \( a \) where \( a \in A \) by auto
let \(?A = A - \{a\}\)
let \(?C = C - \{a\}\)

have 1: \( \text{card } ?A \leq n \) using \( \text{ Suc}(2) - a \)
  using finite-subset by fastforce

have 2: \( \text{card } ?C \geq n \) using \( \text{ Suc}(2) - a \) by auto
from \( \text{ Suc}(1)(\text{OF } 1 2 - \text{ finite-subset}\{\text{OF } - \text{ Suc}(5)\}) \) \( \text{ Suc}(2) - (\cdot) \)
obtain \( B \) where \(?A \subseteq B \subseteq ?C \) card \( B = n \) by blast
thus \(?\text{thesis }\) using \( \text{ a Suc}(2) - (\cdot) \)
by (intro exI[of - insert a B], auto intro: card-insert-disjoint finite-subset[of B C])
qed

20.7.1 Cardinality of image

lemma card-image-le: \( \text{finite } A \Rightarrow \text{ card } (f ' A) \leq \text{ card } A \)
  by (induct rule: finite-induct) (simp-all add: le-SucI card-insert-if)

lemma card-image: \( \text{inj-on } f A \Rightarrow \text{ card } (f ' A) = \text{ card } A \)
proof (induct A rule: infinite-finite-induct)
case (infinite A)
then have \( \neg \text{ finite } (f ' A) \) by (auto dest: finite-imageD)
with infinite show \(?\text{case }\) by simp
qed simp-all

lemma bij-betw-same-card: \( \text{bij-betw } f A B \Rightarrow \text{ card } A = \text{ card } B \)
by (auto simp: card-image bij-betw-def)

lemma endo-inj-surj: \( \text{finite } A \Rightarrow f ' A \subseteq A \Rightarrow \text{ inj-on } f A \Rightarrow f ' A = A \)
by (simp add: card-seteq card-image)

lemma eq-card-imp-inj-on:
  assumes \( \text{finite } A \) \( \text{ card}(f ' A) = \text{ card } A \)
  shows \( \text{ inj-on } f A \)
  using assms
proof (induct rule:finite-induct)
case empty
show \(?\text{case }\) by simp
next
case (insert x A)
then show \(?\text{case }\)
  using card-image-le [of A f] by (simp add: card-insert-if split: if-splits)
qed

lemma inj-on-iff-eq-card: \( \text{finite } A \Rightarrow \text{ inj-on } f A \iff \text{ card } (f ' A) = \text{ card } A \)
by (blast intro: card-image eq-card-imp-inj-on)

lemma card-inj-on-le:
assumes \( \text{inj-on } f \quad A \subseteq B \quad \text{finite } B \)
shows \( \text{card } A \leq \text{card } B \)

proof –
  have finite A
  using assms by (blast intro: finite-imageD dest: finite-subset)
then show \( \text{thesis} \)
  using assms by (force intro: card-mono simp: card-image [symmetric])
qed

lemma \( \text{inj-on-iff-card-le:} \)
  \[ \{ \text{finite } A; \text{finite } B \} \implies (\exists f. \text{inj-on } f \ A \land f \ A \leq B) = (\text{card } A \leq \text{card } B) \]

lemma \( \text{surj-card-le: finite } A \implies B \subseteq f \ A \implies \text{card } B \leq \text{card } A \)
by (blast intro: card-image-le card-mono le-trans)

lemma \( \text{card-bij-eq:} \)
  \( \text{inj-on } f \ A \implies f \ A \subseteq B \implies \text{inj-on } g \ B \implies g \ B \subseteq A \implies \text{finite } A \implies \text{finite } B \)
  \( \implies \text{card } A = \text{card } B \)
by (auto intro: le-antisym card-inj-on-le)

lemma \( \text{bij-betw-finite: bij-betw } f \ A \ B \implies \text{finite } A \longleftrightarrow \text{finite } B \)

unfolding bij-betw-def using finite-imageD[of f A] by auto

lemma \( \text{inj-on-finite: inj-on } f A \implies f \ A \leq B \implies \text{finite } B \implies \text{finite } A \)
using finite-imageD finite-subset by blast

lemma \( \text{card-inverse[simp]:} \)
  \( \text{card } (R^{-1}) = \text{card } R \)
proof –
  have \( \ast : \bigwedge R. \text{prod.swap } f \ R = R^{-1} \) by auto
  \{
    assume \( \neg \text{finite } R \)
    hence \( \text{thesis} \)
    by auto
  \}
  moreover \{
    assume finite R
    with card-image-le[of R prod.swap] card-image-le[of R^{-1} prod.swap]
  \}
qed
20.7.2 Pigeonhole Principles

**lemma pigeonhole:** \( \text{card } A > \text{card } (f \ A) \implies \neg \text{inj-on } f \ A \)

by (auto dest: card-image less-irrefl-nat)

**lemma pigeonhole-infinite:**

assumes \( \neg \text{finite } A \) and \( \text{finite } (f' A) \)

shows \( \exists a0 \in A. \neg \text{finite } \{ a \in A. f a = f a0 \} \)

using assms(2,1)

**proof**

(induct \( f' A \) arbitrary; A rule: finite-induct)

case empty

then show \(?case\) by simp

next

case (insert \( b \) \( F \))

show \(?case\)

proof (cases finite \( \{ a \in A. f a = b \} \))

case True

with \( \neg \text{finite } A \) have \( \neg \text{finite } (A - \{ a \in A. f a = b \}) \)

by simp

also have \( A - \{ a \in A. f a = b \} = \{ a \in A. f a \neq b \} \)

by blast

finally have \( \neg \text{finite } \{ a \in A. f a \neq b \} \).

from insert(3)(OF - this) insert(2,4) show \(?thesis\)

by simp (blast intro: rev-finite-subset)

next

case False

then have \( \{ a \in A. f a = b \} \neq \{ \} \) by force

with False show \(?thesis\) by blast

qed

qed

**lemma pigeonhole-infinite-rel:**

assumes \( \neg \text{finite } A \)

and \( \text{finite } B \)

and \( \forall a \in A. \exists b \in B. R a b \)

shows \( \exists b \in B. \neg \text{finite } \{ a : A. R a b \} \)

**proof**

let \( \lambda a. \{ b \in B. R a b \} \)

from finite-Pow-iff[THEN iffD2, OF (finite B)] have finite \( (?F \cdot A) \)

by (blast intro: rev-finite-subset)

from pigeonhole-infinite [where \( f = ?F \), OF assms(1) this]

obtain \( a0 \) where \( a0 \in A \) and infinite: \( \neg \text{finite } \{ a \in A. ?F a = ?F a0 \} \)

obtain \( b0 \) where \( b0 \in B \) and \( R a0 b0 \)

using \( \langle a0 \in A \rangle \) assms(3) by blast

have finite \( \{ a \in A. ?F a = ?F a0 \} \) if finite \( \{ a \in A. R a b0 \} \)
using \( b_0 \in B \) \( R \ a_0 \ b_0 \) that by \( \text{blast intro: rev-finite-subset} \)

with infinite \( b_0 \in B \) show ?thesis
  by blast

qed

20.7.3 Cardinality of sums

lemma card-Plus:
  assumes finite A finite B
  shows card (A \( <+> \) B) = card A + card B
proof –
  have Inl'A \( \cap \) Inr'B = \{\} by fast
  with assms show ?thesis
    by (simp add: Plus-def card-Un-disjoint card-image)

qed

lemma card-Plus-conv-if:
  card (A \( <+> \) B) = (if finite A \( \land \) finite B then card A + card B else 0)
  by (auto simp add: cardinality_plus)

Relates to equivalence classes. Based on a theorem of F. Kammüller.

lemma dvd-partition:
  assumes f: finite \( \bigcup \) C
  and \( \forall c \in C \). k dvd card \( \forall c1 \in C \). \( \forall c2 \in C \). c1 \( \neq \) c2 \( \rightarrow \) c1 \( \cap \) c2 = \{\}
  shows k dvd card \( \bigcup \) C
proof –
  have finite C
    by \( \text{rule finite-UnionD [OF f]} \)
  then show ?thesis
    using assms
proof (induct rule: finite-induct)
  case empty
  show ?case by simp
next
  case (insert c C)
  then have c \( \cap \) \( \bigcup \) C = \{\}
    by auto
  with insert show ?case
    by (simp add: cardinality_as_sum)
qed

qed

20.8 Minimal and maximal elements of finite sets

context begin

qualified lemma
  assumes finite A and asymp-on A R and transp-on A R and \( \exists x \in A \). P x
  shows
THEORY “Finite-Set”

bex-min-element-with-property: \( \exists x \in A. P x \land (\forall y \in A. R y x \rightarrow \neg P y) \) and
bex-max-element-with-property: \( \exists x \in A. P x \land (\forall y \in A. R x y \rightarrow \neg P y) \)

unfolding atomize-conj
using assms
proof (induction A rule: finite-induct)
case empty
hence False
  by simp-all
thus ?case ..
next
case (insert x F)

from insert.prems have asymp-on F R
  using asymp-on-subset by blast

from insert.prems have transp-on F R
  using transp-on-subset by blast

show ?case
proof (cases P x)
case True
show ?thesis
proof (cases \( \exists a \in F. P a \))
case True
  with insert.IH obtain min max where
  min \( \in F \) and \( P \ min \) and \( \forall z \in F. R \ z \ min \rightarrow \neg P \ z \)
  max \( \in F \) and \( P \ max \) and \( \forall z \in F. R \ max \ z \rightarrow \neg P \ z \)
  using \( \langle \text{asymp-on F R} \rangle \ \langle \text{transp-on F R} \rangle \) by auto

show ?thesis
proof (rule conjI)
  show \( \exists y \in \text{insert x F}. P y \land (\forall z \in \text{insert x F}. R y z \rightarrow \neg P z) \)
  proof (cases R max x)
  case True
  show ?thesis
  proof (intro bexI conjI ballI impI)
    show \( x \in \text{insert x F} \)
      by simp
  next
    show P x
      using \( \langle P x \rangle \) by simp
  next
    fix z assume \( z \in \text{insert x F} \) and \( R \ x \ z \)
    hence \( z = x \lor z \in F \)
      by simp
    thus \( \neg P \ z \)
    proof (rule disjE)
      assume \( z = x \)
      hence \( R \ x \ x \)
using \( \langle R \ x \ z \rangle \) by simp
moreover have \( \neg \ R \ x \ x \)
using \( \langle \text{asymp-on} \ (\text{insert} \ x \ F) \ R \rangle \)[\( \text{THEN irreflp-on-if-asympl-on, \ THEN irreflp-onD} \)]
by simp
ultimately have \( \text{False} \) by simp
thus \( ?\text{thesis} \) ..
next
assume \( z \in F \)
moreover have \( R \ max \ z \)
using \( \langle R \ max \ x \rangle \ \langle R \ x \ z \rangle \)
using \( \langle \text{transp-on} \ (\text{insert} \ x \ F) \ R \rangle \)[\( \text{THEN transp-onD, \ of \ max \ x \ z} \)]
using \( \langle \text{max} \ \in \ F \rangle \ \langle z \ \in \ F \rangle \) by simp
ultimately show \( ?\text{thesis} \)
using \( \forall \ z \in F. \ R \ max \ z \quad \rightarrow \quad \neg \ P \ z \) by simp
qed
qed
next
next
next
case \( \text{False} \)
show \( ?\text{thesis} \)
proof (.intro bexI conjI ballI impl)
show \( \text{max} \ \in \ \text{insert} \ x \ F \)
using \( \langle \text{max} \ \in \ F \rangle \) by simp
next
show \( P \ max \)
using \( \langle P \ max \rangle \) by simp
next
fix \( z \) assume \( z \in \text{insert} \ x \ F \) and \( R \ max \ z \)
hence \( z = x \lor z \in F \)
by simp
thus \( \neg \ P \ z \)
proof (rule disjE)
assume \( z = x \)
hence \( \text{False} \)
using \( \langle \neg \ R \ max \ x \rangle \ \langle R \ max \ z \rangle \) by simp
thus \( ?\text{thesis} \) ..
next
assume \( z \in F \)
thus \( ?\text{thesis} \)
using \( \langle R \ max \ z \rangle \ \forall z \in F. \ R \ max \ z \quad \rightarrow \quad \neg \ P \ z \) by simp
qed
qed
next
next
next
show \( \exists \ y \in \text{insert} \ x \ F. \ P \ y \quad \wedge \quad (\forall z \in \text{insert} \ x \ F. \ R \ z \ y \quad \rightarrow \quad \neg \ P \ z) \)
proof (cases \( R \ x \ \text{min} \))
  case \( \text{True} \)
  show \( ?\text{thesis} \)
proof (intro bexI conjI ballI impI)
  show \( x \in \text{insert } x \ F \)
    by simp
next
  show \( P x \)
    using \( \langle P x \rangle \) by simp
next
  fix \( z \)
  assume \( z \in \text{insert } x \ F \) and \( R z x \)
  hence \( z = x \lor z \in F \)
    by simp
  thus \( \neg P z \)
proof (rule disjE)
  assume \( z = x \)
  hence \( R x x \)
  using \( \langle R z x \rangle \) by simp
  moreover have \( \neg R x x \)
  using \( \langle \text{asymp-on } (\text{insert } x \ F) \ R \rangle \)[THEN irrefl-on-if-asymp-on, THEN irrefl-onD]
    by simp
  ultimately have \( \text{False} \)
    by simp
  thus \?thesis ..
next
  assume \( z \in F \)
  moreover have \( R z \ \text{min} \)
    using \( \langle R z x \rangle , \langle R x \ \text{min} \rangle \) [THEN transp-onD, of \( z \ x \ \text{min} \)]
  using \( \langle \text{min} \in F \rangle , \langle z \in F \rangle \) by simp
  ultimately show \?thesis
    using \( \forall z \in F. \ R z \ \text{min} \longrightarrow \neg P z \rangle \) by simp
qed
qed
next
  case False
  show \?thesis
proof (intro bexI conjI ballI impI)
  show \( \text{min} \in \text{insert } x \ F \)
    using \( \langle \text{min} \in F \rangle \) by simp
next
  show \( P \ \text{min} \)
    using \( \langle P \ \text{min} \rangle \) by simp
next
  fix \( z \)
  assume \( z \in \text{insert } x \ F \) and \( R z \ \text{min} \)
  hence \( z = x \lor z \in F \)
    by simp
  thus \( \neg P z \)
proof (rule disjE)
  assume \( z = x \)
  hence \( \text{False} \)
qualified lemma
assumes finite A and asymp-on A R and transp-on A R and A ≠ {}
shows
  bex-min-element: \exists m \in A. \forall x \in A. x \neq m \longrightarrow \neg R x m and
  bex-max-element: \exists m \in A. \forall x \in A. x \neq m \longrightarrow \neg R m x
using (A ≠ {}),
  bex-min-element-with-property[OF assms(1,2,3), of λ- True, simplified]
  bex-max-element-with-property[OF assms(1,2,3), of λ- True, simplified]
by blast+

end

The following alternative form might sometimes be easier to work with.

lemma is-min-element-in-set-iff:
asymp-on A R \implies (\forall y \in A. y \neq x \longrightarrow \neg R y x) \iff (\forall y. R y x \longrightarrow y \notin A)
by (auto dest: asymp-onD)

lemma is-max-element-in-set-iff:


Proof:

The proof involves showing that for any relation $R$ on a finite set $A$, the following holds:

$$\text{asymp-on } A \ R \iff (\forall y \in A. \ y \neq x \rightarrow R \ x \ y) \iff (\forall y. \ R \ x \ y \rightarrow y \notin A)$$

by (auto dest: asymp-onD)

Before delving into the proof, let's define some terms:

- **asymp-on**: A relation $R$ is asymptotic on $A$ if for all $y \in A$, if $y \neq x$ then $R x y$.
- **∀**: Universal quantifier.
- **⇒**: Implication.
- **∀ y ∈ A. y ≠ x → R x y**: For all $y$ in $A$, if $y$ is not equal to $x$, then $R x y$.
- **∀ y. R x y → y ∉ A**: For all $y$, if $R x y$ then $y$ is not in $A$.

The proof begins with the context:

**context begin**

**qualified lemma**

assumes finite $A$ and $A \neq \{\}$ and transp-on $A$ $R$ and totalp-on $A$ $R$

shows

- **bex-least-element**: $\exists l \in A. \ \forall x \in A. \ x \neq l \rightarrow R \ l \ x$
- **bex-greatest-element**: $\exists g \in A. \ \forall x \in A. \ x \neq g \rightarrow R \ x \ g$

**unfolding atomize-conj**

**proof** (induction $A$ rule: finite-induct)

**case** empty

hence False by simp

thus ?case ..

next

**case** (insert $a$ $A'$)

from insert.prems(2) have transp-on-$A'$: transp-on $A'$ $R$

by (auto intro: transp-onI dest: transp-onD)

from insert.prems(3) have

- totalp-on-a-$A'$-raw: $\forall y \in A'. \ a \neq y \rightarrow R \ a \ y \lor R \ y \ a$
- totalp-on-$A'$: totalp-on $A'$ $R$

by (simp-all add: totalp-on-def)

show ?case

**proof** (cases $A' = \{\})$

**case** True

thus ?thesis by simp

next

**case** False

then obtain least greatest where

- least $\in A'$ and least-of-$A'$: $\forall x \in A'. \ x \neq \text{least} \rightarrow R \text{least} \ x$
- greatest $\in A'$ and greatest-of-$A'$: $\forall x \in A'. \ x \neq \text{greatest} \rightarrow R \text{greatest} \ x$

using insert.IH[OF - transp-on-$A'$ totalp-on-$A'$] by auto

show ?thesis

**proof** (rule conjI)

show $\exists l \in \text{insert} \ a \ A'. \ \forall x \in \text{insert} \ a \ A'. \ x \neq l \rightarrow R \ l \ x$

**proof** (cases $R \ a \ \text{least}$)

**case** True

show ?thesis

**proof** (intro bexI ballI impl)

show $a \in \text{insert} \ a \ A'$

by simp

next

fix $x$
THEORY "Finite-Set"

show \( \forall x. x \in \text{insert } a \ A' \Rightarrow x \neq a \Rightarrow R \ a \ x \)
using True \langle \text{least } \in A' \ \text{least-of-A' } \rangle
using \text{insert.prems(2)}[\text{THEN transp-onD, of a least}] 
by auto
qed

next

case False
show \(?thesis
proof (intro bexI ballI impI)
show least \in \text{insert } a \ A'
using \langle \text{least } \in A' \ \text{by simp} \rangle
next
fix \( x \)
show \( x \in \text{insert } a \ A' \Rightarrow x \neq \ \text{least} \Rightarrow R \ \text{least } x \)
using False \langle \text{least } \in A' \ \text{least-of-A' totalp-on-a-A'-raw} \rangle 
by (cases \( x = a \)) auto
qed
qed

next
show \( \exists \ g \in \text{insert } a \ A'. \ \forall x \in \text{insert } a \ A'. x \neq g \Rightarrow R \ x \ g \)
proof (cases \text{R greatest } a)
case True
show \(?thesis
proof (intro bexI ballI impI)
show a \in \text{insert } a \ A'
by simp
next
fix \( x \)
show \( \forall x. x \in \text{insert } a \ A' \Rightarrow x \neq a \Rightarrow R \ a \ x \)
using True \langle \text{greatest } \in A' \ \text{greatest-of-A' } \rangle
using \text{insert.prems(2)}[\text{THEN transp-onD, of - greatest } a] 
by auto
qed
next
case False
show \(?thesis
proof (intro bexI ballI impI)
show greatest \in \text{insert } a \ A'
using \langle \text{greatest } \in A' \ \text{by simp} \rangle
next
fix \( x \)
show \( x \in \text{insert } a \ A' \Rightarrow x \neq \ \text{greatest} \Rightarrow R \ \text{greatest } x \)
using False \langle \text{greatest } \in A' \ \text{greatest-of-A' totalp-on-a-A'-raw} \rangle 
by (cases \( x = a \)) auto
qed
qed
qed
qed
end

20.8.1 Finite orders

context order
begin

lemma finite-has-maximal:
  assumes finite A and A ≠ {} 
  shows ∃ m ∈ A. ∀ b ∈ A. m ≤ b −→ m = b
proof
  obtain m where m ∈ A and m-is-max: ∀ x∈A. x ≠ m −→ ¬ m < x
    using Finite-Set.bex-max-element[OF finite A - - ⟨A ≠ {}⟩, of (<?)] by auto
  moreover have ∀ b ∈ A. m ≤ b −→ m = b 
    using m-is-max by (auto simp: le_less)
  ultimately show ?thesis
    by auto
qed

lemma finite-has-maximal2:
[ finite A; a ∈ A ] =⇒ ∃ m ∈ A. a ≤ m ∧ (∀ b ∈ A. m ≤ b −→ m = b)
using finite-has-maximal[of {b ∈ A. a ≤ b}] by fastforce

lemma finite-has-minimal:
  assumes finite A and A ≠ {} 
  shows ∃ m ∈ A. ∀ b ∈ A. b ≤ m −→ m = b
proof
  obtain m where m ∈ A and m-is-min: ∀ x∈A. x ≠ m −→ ¬ x < m
    using Finite-Set.bex-min-element[OF finite A - - ⟨A ≠ {}⟩, of (<?)] by auto
  moreover have ∀ b ∈ A. b ≤ m −→ m = b 
    using m-is-min by (auto simp: le_less)
  ultimately show ?thesis
    by auto
qed

lemma finite-has-minimal2:
[ finite A; a ∈ A ] =⇒ ∃ m ∈ A. m ≤ a ∧ (∀ b ∈ A. b ≤ m −→ m = b)
using finite-has-minimal[of {b ∈ A. b ≤ a}] by fastforce

end

20.8.2 Relating injectivity and surjectivity

lemma finite-surj-inj:
  assumes finite A A ⊆ f ` A 
  shows inj-on f A
proof
  have f ` A = A 
    by (rule card-seteq [THEN sym]) (auto simp add: assms card_image_le)
then show \( ?\text{thesis} \) using \( \text{assms} \)
by (simp add: eq-card-imp-inj-on)

qed

\textbf{lemma} \texttt{finite-UNIV-surj-inj}: \( \text{finite}(\text{UNIV}:: 'a \text{ set}) \Rightarrow \text{surj } f \Rightarrow \text{inj } f \)
for \( f :: 'a \Rightarrow 'a \)
by (blast intro: \text{finite-surj-inj} \text{subset-UNIV})

\textbf{lemma} \texttt{finite-UNIV-inj-surj}: \( \text{finite}(\text{UNIV}:: 'a \text{ set}) \Rightarrow \text{inj } f \Rightarrow \text{surj } f \)
for \( f :: 'a \Rightarrow 'a \)
by (fastforce simp \text{surj-def} dest: \text{endo-inj-surj})

\textbf{lemma} \texttt{surjective-iff-injective-gen}:
assumes \( fS: \text{finite } S \)
and \( fT: \text{finite } T \)
and \( c: \text{card } S = \text{card } T \)
and \( ST: f \cdot S \subseteq T \)
shows \( (\forall y \in T. \exists x \in S. f x = y) \iff \text{inj-on } f S \)
(is \( ?\text{lhs} \iff ?\text{rhs} \))

\textbf{proof} \;
\textbf{assume} \( h: ?\text{lhs} \)
\{
\text{fix } x \ y
\text{assume } x: x \in S
\text{assume } y: y \in S
\text{assume } f: f x = f y
\text{from } fS \text{ have } S0: \text{card } S \neq 0
\text{by } auto
\text{have } x = y
\textbf{proof} \text{(rule ccontr)}
\text{assume } xy: \neg ?\text{thesis}
\text{have } th: \text{card } S \leq \text{card } (f \cdot (S - \{y\}))
\text{unfolding } c
\textbf{proof} \text{(rule card-mono)}
\text{show } \text{finite } (f \cdot (S - \{y\}))
\text{by } (simp add: fS)
\text{have } [x \neq y; x \in S; z \in S; f x = f y]
\Rightarrow \exists x \in S. x \neq y \land f z = f x \text{ for } z
\text{by } (cases z = y \rightarrow z = x) \text{ auto}
\text{then show } T \subseteq f \cdot (S - \{y\})
\text{using } h \text{ xy x y f by } fastforce
\textbf{qed}
\textbf{also have } \ldots \leq \text{card } (S - \{y\})
\text{by } (simp add: \text{card-image-le fS})
\textbf{also have } \ldots \leq \text{card } S - 1 \text{ using } fS \text{ by } simp
\textbf{finally show } False \text{ using } S0 \text{ by } arith
\textbf{qed}
\}
then show ?\text{rhs}
unfolding inj-on-def by blast
next
assume h: ?rhs
have f' S = T
  by (simp add: ST c card-image card-subset-eq fT h)
then show ?lhs by blast
qed

hide-const (open) Finite-Set.fold

20.9 Infinite Sets

Some elementary facts about infinite sets, mostly by Stephan Merz. Beware!
Because "infinite" merely abbreviates a negation, these lemmas may not work
well with blast.

abbreviation infinite :: 'a set ⇒ bool
  where infinite S ≡ ¬ finite S

Infinite sets are non-empty, and if we remove some elements from an infinite
set, the result is still infinite.

lemma infinite-UNIV-nat [iff]: infinite (UNIV :: nat set)
proof
  assume finite (UNIV :: nat set)
  with finite-UNIV-inj-surj [of Suc] show False
    by simp (blast dest: Suc-neq-Zero surjD)
qed

lemma infinite-UNIV-char-0:
infinite (UNIV :: 'a::semiring-char-0 set)
proof
  assume finite (UNIV :: 'a set)
  with subset-UNIV have finite (range of-nat :: 'a set)
    by (rule finite-subset)
  moreover have inj (of-nat :: nat ⇒ 'a)
    by (simp add: inj-on-def)
  ultimately have finite (UNIV :: nat set)
    by (rule finite-imageD)
  then show False
    by simp
qed

lemma infinite-imp-nonempty: infinite S ⇒ S ≠ {} 
  by auto

lemma infinite-remove: infinite S ⇒ infinite (S − {a})
  by simp

lemma Diff-infinite-finite:
  assumes finite T infinite S
shows infinite \((S - T)\)

proof induct

from \(\text{infinite } S\) show infinite \((S - \{\})\)

by auto

next

fix \(T x\)

assume \(ih\): infinite \((S - T)\)

have \(S - (\text{insert } x T) = (S - T) - \{x\}\)

by (rule Diff-insert)

with \(ih\) show infinite \((S - (\text{insert } x T))\)

by (simp add: infinite-remove)

qed

lemma Un-infinite: infinite \(S \Rightarrow \text{infinite } (S \cup T)\)

by simp

lemma infinite-Un: infinite \((S \cup T) \iff \text{infinite } S \lor \text{infinite } T\)

by simp

lemma infinite-super:

assumes \(S \subseteq T\)

and infinite \(S\)

shows infinite \(T\)

proof

assume finite \(T\)

with \(S \subseteq T\) have finite \(S\) by (simp add: finite-subset)

with \(\text{infinite } S\) show False by simp

qed

proposition infinite-coinduct [consumes 1, case-names infinite]:

assumes \(X A\)

and step: \(\forall A. X A \Rightarrow \exists x \in A. X (A - \{x\}) \lor \text{infinite } (A - \{x\})\)

shows infinite \(A\)

proof

assume finite \(A\)

then show False

using \(\langle X A \rangle\)

proof ( induction rule: finite-psubset-induct)

case \(\text{psubset } A\)

then obtain \(x\) where \(x \in A X (A - \{x\}) \lor \text{infinite } (A - \{x\})\)

using local.step psubset.prems by blast

then have \(X (A - \{x\})\)

using psubset.hyps by blast

show False

proof ( rule psubset.IH [where \(B = A - \{x\})]\)

show \(A - \{x\} \subseteq A\)

using \(\langle x \in A \rangle\) by blast

qed fact
For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

**lemma** \textit{inf-img-fin-dom}:

\textbf{assumes} img: finite (f ' A) \hspace{1cm} \textbf{and} dom: infinite A

\textbf{shows} \exists y \in f ' A. infinite (f − ' \{y\} \cap A)

\textbf{proof} (rule ccontr)

have A ⊆ (∪y \in f ' A. f − ' \{y\} \cap A) by auto

moreover assume ¬thesis

with img have finite (∪y \in f ' A. f − ' \{y\} \cap A) by blast

ultimately have finite A by (rule finite-subset)

with dom show False by contradiction

**qed**

**lemma** \textit{inf-img-fin-domE}:

\textbf{assumes} finite (f ' A) \textbf{and} infinite A

\textbf{obtains} y where y \in f'A \textbf{and} infinite (f − ' \{y\} \cap A)

\textbf{using} \textit{assms} by (blast dest: \textit{inf-img-fin-dom}')

**lemma** \textit{inf-img-fin-dom}:

\textbf{assumes} img: finite (f' A) \textbf{and} dom: infinite A

\textbf{shows} \exists y \in f' A. infinite (f − ' \{y\})

\textbf{using} \textit{inf-img-fin-dom}'[OF \textit{assms}] by auto

**lemma** \textit{inf-img-fin-domE}:

\textbf{assumes} finite (f' A) \textbf{and} infinite A

\textbf{obtains} y where y \in f' A \textbf{and} infinite (f − ' \{y\})

\textbf{using} \textit{assms} by (blast dest: \textit{inf-img-fin-dom})

**proposition** \textit{finite-image-absD}: finite (abs ' S) \implies finite S

for S :: 'a::linordered-ring set

by (rule ccontr) (auto simp: abs-eq-iff vimage_def dest: \textit{inf-img-fin-dom})

### 20.10 The finite powerset operator

**definition** \textit{Fpow} :: 'a set ⇒ 'a set set

where \textit{Fpow} A ≡ \{X. X ⊆ A ∧ finite X\}

**lemma** \textit{Fpow-mono} : A ⊆ B \implies Fpow A ⊆ Fpow B

\textbf{unfolding} \textit{Fpow-def} by auto

**lemma** \textit{empty-in-Fpow} : \{\} \∈ Fpow A

\textbf{unfolding} \textit{Fpow-def} by auto
lemma Fpow-not-empty: Fpow A ≠ {} using empty-in-Fpow by blast

lemma Fpow-subset-Pow: Fpow A ⊆ Pow A unfolding Fpow-def by auto

lemma Fpow-Pow-finite: Fpow A = Pow A Int {A. finite A} unfolding Fpow-def Pow-def by blast

lemma inj-on-image-Fpow: assumes inj-on f A shows inj-on (image f) (Fpow A) using assms Fpow-subset-Pow[of A] subset-inj-on[of image f Pow A] inj-on-image-Pow by blast

lemma image-Fpow-mono: assumes f ' A ⊆ B shows (image f) ' (Fpow A) ⊆ Fpow B using assms by (unfold Fpow-def, auto)

end

21 Reflexive and Transitive closure of a relation

theory Transitive-Closure imports Finite-Set abbrevs ∗∗ = ∗ ∗∗ and ∗+ = + ++ and ∗∗ = = == begin

ML-file ⟨~∵/src/Provers/trancl.ML⟩

rtrancl is reflexive/transitive closure, trancl is transitive closure, reflcl is reflexive closure.
These postfix operators have maximum priority, forcing their operands to be atomic.
context notes [[inductive-internals]] begin

inductive-set rtrancl :: (′a × ′a) set ⇒ (′a × ′a) set ((∗) [1000] 999) for r :: (′a × ′a) set where
rtrancl-refl [intro!, Pure.intro!, simp]: (a, a) ∈ r∗ | rtrancl-into-rtrancl [Pure.intro]: (a, b) ∈ r∗ ⇒ (b, c) ∈ r ⇒ (a, c) ∈ r

inductive-set trancl :: (′a × ′a) set ⇒ (′a × ′a) set ((∗+) [1000] 999) for r :: (′a × ′a) set
where
\[ r\text{-into-trancl [intro, Pure.intro]} : (a, b) \in r \Rightarrow (a, b) \in r^+ \]
| trancl-into-trancl [Pure.intro] : (a, b) \in r^+ \Rightarrow (b, c) \in r \Rightarrow (a, c) \in r^+ \]

notation
\[ r\text{trancl} ((\ast\ast) [1000] 1000) \text{ and } \]
\[ trancl ((\ast\ast) [1000] 1000) \]

declare
\[ r\text{trancl-def [nitpick-unfold del]} \]
\[ r\text{tranclp-def [nitpick-unfold del]} \]
\[ trancl-def [nitpick-unfold del] \]
\[ tranclp-def [nitpick-unfold del] \]

end

abbreviation reflcl :: ('a x 'a) set ⇒ ('a x 'a) set ((\ast) [1000] 999)
\[ \text{where } r^\equiv \equiv r \cup \text{Id} \]

abbreviation reflclp :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool ((\ast\ast) [1000] 1000)
\[ \text{where } r^\approx \equiv \sup r (\equiv) \]

notation (ASCII)
\[ \text{reflcl ((\ast) [1000] 999) and } \]
\[ \text{reflclp ((\ast\ast) [1000] 1000) and } \]
\[ \text{reflcl ((\ast\ast) [1000] 999) and } \]
\[ \text{reflclp ((\ast\ast) [1000] 1000) and } \]
\[ \text{reflcl ((\ast\ast) [1000] 1000) and } \]
\[ \text{reflclp ((\ast\ast) [1000] 1000) and } \]

21.1 Reflexive closure

lemma reflcl-set-eq [pred-set-cone]: (sup (λx y. (x, y) ∈ r) (\equiv)) = (λx y. (x, y) ∈ r \cup \text{Id})
\[ \text{by (auto simp: fun-eq-iff)} \]

lemma refl-reflcl [simp]: refl (r^\equiv)
\[ \text{by (simp add: refl-on-def)} \]

lemma reflp-on-reflclp [simp]: reflp-on A R^\equiv
\[ \text{by (simp add: reflp-on-def)} \]

lemma antisym-on-reflcl [simp]: antisym-on A (r^\equiv) \iff antisym-on A r
\[ \text{by (simp add: antisym-on-def)} \]

lemma antisymp-on-reflclp [simp]: antisymp-on A R^\equiv \iff antisymp-on A R
\[ \text{by (rule antisym-on-reflclp[to-pred])} \]

lemma trans-on-reflcl [simp]: trans-on A r ⇒ trans-on A (r^\equiv)
by (auto intro: trans-onI dest: trans-onD)

lemma transp-on-refclp[simp]: transp-on A R → transp-on A R^==
by (rule trans-on-refcl[to-pred])

lemma antisym-on-refclp-if-asymp-on:
assumes asymp-on A R
shows antisym-on A R^=
unfolding antisym-on-refclp
using antisym-on-if-asymp-on[OF ‹asymp-on A R›] .

lemma antisym-on-refclp-if-asymp-on: asym-on A R → antisym-on A (R^=)
using antisym-on-refclp-if-asymp-on[to-set] .

lemma reflclp-idemp [simp]: (P^=)= = P^==
by blast

lemma reflclp-ident-if-reflp[simp]: reflp R → R^= = R
by (auto dest: reflpD)

The following are special cases of reflclp-ident-if-reflp, but they appear duplicated in multiple, independent theories, which causes name clashes.

lemma (in preorder) reflclp-less-eq[simp]: (≤)^= = (≤)
using reflp-on-le by (simp only: reflclp-ident-if-reflp)

lemma (in preorder) reflclp-greater-eq[simp]: (≥)^= = (≥)
using reflp-on-ge by (simp only: reflclp-ident-if-reflp)

lemma order-refclp-if-transp-and-asymp:
assumes transp R and asymp R
shows class.order R^= = R
proof unfold-locales
show ∃x y. R x y = (R^= x y ∧ ¬ R^= y x)
using ‹asymp R› asympD by fastforce
next
show ∃x. R^= x x
by simp
next
show ∃x y z. R^= x y → R^= y z → R^= x z
using transp-on-refclp[OF ‹transp R›, THEN transpD] .
next
show ∃x y. R^= x y → R^= y x → x = y
using antisym-on-refclp-if-asymp-on[OF ‹asymp R›, THEN antisymD] .
qed

21.2 Reflexive-transitive closure

lemma r-into-rtrancl [intro]: ∀p. p ∈ r → p ∈ r^=
— rtrancl of r contains r
by (simp add: split-tupled-all rtrancl-refl [THEN rtrancl-into-rtrancl])

lemma r-into-rtranclp [intro]: \( r \times y \Rightarrow r^{**} \times y \)
— rtrancl of \( r \) contains \( r \)
by (erule rtranclp.rtrancl-refl [THEN rtranclp.rtrancl-into-rtrancl])

lemma rtranclp-mono: \( r \leq s \Rightarrow r^{**} \leq s^{**} \)
— monotonicity of rtrancl
proof (rule predicate2I)
  show \( s^{**} \times y \) if \( r \leq s \) \( r^{**} \times y \) for \( x \) \( y \)
  using \( r^{**} \times y \) \( \cdot r \leq s \)
  by (induction rule: rtranclp.induct) (blast intro: rtranclp.rtrancl-into-rtrancl)+
qed

lemma mono-rtranclp[mono]: \( (\forall a b. x a b \Rightarrow y a b) \Rightarrow x^{**} a b \Rightarrow y^{**} a b \)
using rtranclp-mono[of x y]
by auto

lemmas rtrancl-mono = rtranclp-mono [to-set]

theorem rtranclp-induct [consumes 1, case-names base step, induct set: rtranclp]:
  assumes a: \( r^{**} \times a b \)
  and cases: \( P a \\wedge y z. r^{**} \times a y \Rightarrow r y z \Rightarrow P y \Rightarrow P z \)
  shows \( P b \)
  using a by (induct x≡a b) (rule cases)+

lemmas rtranclp-induct2 = rtranclp-induct [induct set: rtrancl] = rtranclp-induct [to-set]

lemmas rtranclp-induct2 =
  rtranclp-induct[of -(ax,ay) (bx,by), split-rule, consumes 1, case-names refl step]

lemmas rtranclp-induct2 =
  rtranclp-induct[of (ax,ay) (bx,by), split-format (complete), consumes 1, case-names refl step]

lemma refl-rtrancl: refl (r*)
  unfolding refl-on-def by fast

Transitivity of transitive closure.

lemma trans-rtrancl: trans (r*)
proof (rule transI)
  fix \( x y z \)
  assume \((x, y) \in r^*\)
  assume \((y, z) \in r^*\)
  then show \((x, z) \in r^*\)
  proof induct
    case base
    show \((x, y) \in r^*\) by fact
  next
    case (step u v)
THEORY “Transitive-Closure” 543

from \( (x, u) \in r^* \) and \( (u, v) \in r \)
show \( (x, v) \in r^* \).
qed

lemmas rtrancl-trans = trans-rtrancl [THEN transD]

lemma rtranclp-trans:
assumes \( r^{**} x y \)
and \( r^{**} y z \)
shows \( r^{**} x z \)
using assms(2,1) by induct iprover+

lemma rtranclE [cases set: rtrancl]:
fixes \( a \) \( b \) :: 'a
assumes major: \( (a, b) \in r^* \)
obtains
(base) \( a = b \)
| (step) \( y \) where \( (a, y) \in r^* \) and \( (y, b) \in r \)
--- elimination of rtrancl -- by induction on a special formula
proof --
have \( a = b \lor (\exists y. \ (a, y) \in r^* \land (y, b) \in r) \)
by (rule major [THEN rtrancl-induct]) blast+
then show \?thesis
by (auto intro: base step)
qed

lemma rtrancl-Int-subset: \( Id \subseteq s \Longrightarrow (r^* \cap s) O r \subseteq s \Rightarrow r^* \subseteq s \)
by (fastforce elim: rtrancl-induct)

lemma converse-rtranclp-into-rtranclp: \( r \ a \ b \Longrightarrow r^{**} \ b \ c \Longrightarrow r^{**} \ a \ c \)
by (rule rtranclp-trans) iprover+

lemmas converse-rtrancl-into-rtrancl = converse-rtranclp-into-rtranclp [to-set]

More \( r^* \) equations and inclusions.

lemma rtranclp-idemp [simp]: \( (r^{**})^{**} = r^{**} \)
proof --
have \( r^{****} x y \Longrightarrow r^{**} x y \) for \( x y \)
by (induction rule: rtranclp-induct) (blast intro: rtranclp-trans)+
then show \?thesis
by (auto intro!: order-antisym)
qed

lemmas rtrancl-idemp [simp] = rtranclp-idemp [to-set]

lemma rtrancl-idemp-self-comp [simp]: \( R^* \ O \ R^* = R^* \)
by (force intro: rtrancl-trans)
THEORY "Transitive-Closure"

lemma rtrancl-subset-rtrancl: \( r \subseteq s^* \implies r^* \subseteq s^* \)
  by (drule rtrancl-mono, simp)

lemma rtranclp-subset: \( R \leq S \implies S \leq R^{**} \implies S^{**} = R^{**} \)
  by (fastforce dest: rtranclp-mono)

lemmas rtrancl-subset = rtranclp-subset [to-set]

lemma rtranclp-sup-rtranclp: \((\operatorname{sup} (R^{**}) (S^{**}))^{**} = (\operatorname{sup} R S)^{**} \)
  by (blast intro!: rtranclp-subset intro: rtranclp-mono [THEN predicate2D])

lemmas rtrancl-Un-rtranclp = rtranclp-sup-rtranclp [to-set]

lemma rtranclp-reflclp [simp]: \( (R \equiv) = R^{**} \)
  by (blast intro!: rtranclp-subset [symmetric])

lemmas rtrancl-refl = rtranclp-reflclp [to-set]

lemma rtrancl-r-diff-Id: \( (r - \operatorname{Id})^{*} = r^{*} \)
  by (rule rtrancl-subset [symmetric]) auto

lemma rtranclp-r-diff-Id: \( (\inf r (\neq))^{**} = r^{**} \)
  by (rule rtranclp-subset [symmetric]) auto

theorem rtranclp-converseD:
  assumes \( (r^{-1})^{*} x y \)
  shows \( r^{**} y x \)
  using assms by induct (iprover intro: rtranclp-trans dest!: conversepD)+

lemmas rtrancl-converseD = rtranclp-converseD [to-set]

theorem rtranclp-converseI:
  assumes \( r^{**} y x \)
  shows \( (r^{-1})^{**} x y \)
  using assms by induct (iprover intro: rtranclp-trans conversepI)+

lemmas rtrancl-converseI = rtranclp-converseI [to-set]

lemma rtrancl-converse: \( (r^{-1})^{*} = (r^{*})^{-1} \)
  by (fast dest!: rtranclp-converseD intro!: rtranclp-converseI)

lemma sym-rtrancl: \( \operatorname{sym} r \implies \operatorname{sym} (r^{*}) \)
  by (simp only: sym-conv-converse-eq rtrancl-converse [symmetric])

theorem converse-rtranclp-induct [consumes 1, case-names base step]:
  assumes major: \( r^{**} a b \)
  and cases: \( P b \land y z \implies r^{**} z b \implies P z \implies P y \)
  shows \( P a \)
  using rtranclp-converseI [OF major]
by induct (iprover intro: cases dest!: conversepD rtranclp-converseD)+

lemmas converse-rtrancl-induct = converse-rtranclp-induct [to-set]

lemmas converse-rtranclp-induct2 =
  converse-rtranclp-induct [of - (ax, ay) (bx, by), split-format, consumes 1, case-names refl step]

lemmas converse-rtranclp-induct2 =
  converse-rtranclp-induct [of (ax, ay) (bx, by), split-format (complete), consumes 1, case-names refl step]

lemma converse-rtranclpE [consumes 1, case-names base step]:
  assumes major: r** x z
  and cases: x = z ⊢ P ∧ y. r x y ⊢ r** y z ⊢ P
  shows P
proof —
  have x = z ∨ (∃ y. r x y ∧ r** y z)
  by (rule major [THEN converse-rtranclp-induct]) iprover+
  then show ?thesis
  by (auto intro: cases)
qed

lemmas converse-rtranclE = converse-rtranclpE [to-set]

lemmas converse-rtranclpE2 = converse-rtranclpE [of - (xa,xb) (za,zb), split-rule]

lemmas converse-rtranclpE2 = converse-rtranclE [of (xa,xb) (za,zb), split-rule]

lemma r-comp-rtrancl-eq: r O r* = r* O r
by (blast elim: rtranclE converse-rtranclE
  intro: rtrancl-into-rtrancl converse-rtrancl-into-rtrancl)

lemma rtrancl-unfold: r* = Id ∪ r* O r
by (auto intro: rtrancl-into-rtrancl elim: rtranclE)

lemma rtrancl-Un-separatorE:
  (a, b) ∈ (P ∪ Q)* ⇒ ∀ x y. (a, x) ∈ P* ⇒ (x, y) ∈ Q ⇒ x = y ⇒ (a, b) ∈ P*
proof (induct rule: rtrancl.induct)
  case rtrancl-refl
  then show ?case by blast
next
  case rtrancl-into-rtrancl
  then show ?case by (blast intro: rtrancl-trans)
qed

lemma rtrancl-Un-separator-converseE:
  (a, b) ∈ (P ∪ Q)* ⇒ ∀ x y. (x, b) ∈ P* ⇒ (y, x) ∈ Q ⇒ y = x ⇒ (a, b)
\[ \in P^* \]

**proof (induct rule: converse-rtrancl-induct)**

- **case base**
  - then show ?case by blast

- **next**
  - **case step**
    - then show ?case by (blast intro: rtrancl-trans)

**qed**

**lemma** Image-closed-trancl:

- assumes \( r ^{"} X \subseteq X \)
- shows \( r ^* ^{"} X = X \)

**proof**

- from assms have 
  \[ \{ y . \exists x \in X . (x, y) \in r \} \subseteq X \]
  - by auto
  - have \( x \in X \) if \( I : (y, x) \in r ^* \) and \( 2 : y \in X \) for \( x y \)

**proof**

- from \( 1 \) show \( x \in X \)
  - **proof induction**
    - **case base**
      - show ?case by (fact 2)
    - **next**
      - **case step**
        - with **show** ?case by auto
    - **qed**
  - **qed**
  - then show ?thesis by auto
  - **qed**

**lemma** rtranclp-ident-if-reflp-and-transp:

- assumes reflp \( R \) and transp \( R \)
- shows \( R ^{**} = R \)

**proof (intro ext ifI)**

- fix \( x y \)
  - show \( R ^{**} x y \implies R x y \)

**proof (induction y rule: rtranclp-induct)**

- **case base**
  - using \( \text{reflp } R : [\text{THEN reflpD}] \)
- **next**
  - **case (step y z)**
    - thus ?case
      - using \( \text{transp } R : [\text{THEN transpD, of } x y \ z] \) by simp
  - **qed**
- **next**
  - fix \( x y \)
  - show \( R x y \implies R ^{**} x y \)
    - using \( r \text{-into-rtranclp} \)

**qed**
The following are special cases of \texttt{rtranclp-ident-if-reflp-and-transp}, but they appear duplicated in multiple, independent theories, which causes name clashes.

\textbf{lemma (in preorder) rtranclp-less-eq[simp]:} \((\leq)^* = (\leq)\)
\begin{itemize}
\item \textbf{using} reflp-on-le transp-on-le \textbf{by} (simp only: rtranclp-ident-if-reflp-and-transp)
\end{itemize}

\textbf{lemma (in preorder) rtranclp-greater-eq[simp]:} \((\geq)^* = (\geq)\)
\begin{itemize}
\item \textbf{using} reflp-on-ge transp-on-ge \textbf{by} (simp only: rtranclp-ident-if-reflp-and-transp)
\end{itemize}

\subsection{21.3 Transitive closure}

\textbf{lemma totalp-on-tranclp:} \(\text{totalp-on } A \ R \Longrightarrow \text{totalp-on } A \ (\text{trancl } R)\)
\begin{itemize}
\item \textbf{by} (auto intro: totalp-onI dest: totalp-onD)
\end{itemize}

\textbf{lemma total-on-trancl:} \(\text{total-on } A \ r \Longrightarrow \text{total-on } A \ (\text{trancl } r)\)
\begin{itemize}
\item \textbf{by} (rule totalp-on-tranclp[to-set])
\end{itemize}

\textbf{lemma trancl-mono:}
\begin{itemize}
\item \textbf{assumes} \(p \in r^+ \ r \subseteq s\)
\item \textbf{shows} \(p \in s^+\)
\end{itemize}
\begin{itemize}
\item \textbf{proof} –
\item \textbf{have} \([(a, b) \in r^+; r \subseteq s] \Longrightarrow (a, b) \in s^+ \text{ for } a \ b\)
\item \textbf{by} (induction rule: trancl.induct) (iprover dest: subsetD)+
\item \textbf{with assms show} \(\text{thesis}\)
\item \textbf{by} (cases p) force
\end{itemize}
\textbf{qed}

\textbf{lemma r-into-trancl':} \(\bigwedge \ p. \ p \in r \Longrightarrow p \in r^+\)
\begin{itemize}
\item \textbf{by} (simp only: split-tupled-all) (erule r-into-trancl)
\end{itemize}

Conversions between \texttt{trancl} and \texttt{rtrancl}.

\textbf{lemma tranclp-into-rtranclp:} \(r^{++} a \ b \Longrightarrow r^{**} a \ b\)
\begin{itemize}
\item \textbf{by} (erule tranclp.induct) iprover+
\end{itemize}

\textbf{lemmas tranclp-into-rtrancl = tranclp-into-rtranclp[to-set]}

\textbf{lemma rtranclp-into-tranclp1:}
\begin{itemize}
\item \textbf{assumes} \(r^{**} a \ b\)
\item \textbf{shows} \(r b c \Longrightarrow r^{++} a \ c\)
\end{itemize}
\begin{itemize}
\item \textbf{using} \texttt{assms} \textbf{by} (induct arbitrary: c) iprover+
\end{itemize}

\textbf{lemmas rtranclp-into-tranclp1 = rtranclp-into-tranclp1[to-set]}

\textbf{lemma rtranclp-into-tranclp2:}
\begin{itemize}
\item \textbf{assumes} \(r \ a \ b \ r^{**} b \ c \text{ shows } r^{++} a \ c\)
\item \textbf{— intro rule from } \texttt{r} \text{ and } \texttt{rtrancl}
\item \textbf{using} \(r^{**} b \ c\)
\end{itemize}
\begin{itemize}
\item \textbf{proof} (cases rule: rtranclp.cases)
THEORY "Transitive-Closure"

case rtrancl-refl
  with assms show \( ?thesis \)
    by (iprover
next
  case rtrancl-into-rtrancl
  with assms show \( ?thesis \)
    by (auto intro: rtranclp-trans [THEN rtranclp-into-tranclp1])
qed

lemmas rtrancl-into-trancl2 = rtranclp-into-tranclp2 [to-set]

Nice induction rule for trancl

lemma tranclp-induct [consumes 1, case-names base step, induct pred: tranclp]:
  assumes \( a: r^+ a b \)
  and cases: \( \land y. r a y \Rightarrow P y \land y z. r^+ a y \Rightarrow r y z \Rightarrow P y \Rightarrow P z \)
  shows \( P b \)
  using \( a \) by (induct \( x\equiv a b \)) (iprover intro: cases)+

lemmas tranclp-induct [induct set: trancl] = tranclp-induct [to-set]

lemmas tranclp-induct2 =
  tranclp-induct [of - (ax, ay) (bx, by), split-rule, consumes 1, case-names base step]

lemmas tranclp-trans-induct:
  assumes major: \( r^+ x y \)
  and cases: \( \land x y. r x y \Rightarrow P x y \land x y z. r^+ x y \Rightarrow P x y \Rightarrow r^+ y z \Rightarrow P y z \Rightarrow P x z \)
  shows \( P x y \)
  — Another induction rule for trancl, incorporating transitivity
  by (iprover intro: major [THEN tranclp-induct] cases)

lemmas trancl-trans-induct = tranclp-trans-induct [to-set]

lemma tranclE [cases set: trancl]:
  assumes \( (a, b) \in r^+ \)
  obtains
    (base) \( (a, b) \in r \)
    | (step) \( c \) where \( (a, c) \in r^+ \) and \( (c, b) \in r \)
  using assms by cases simp-all

lemma trancl-Int-subset: \( r \subseteq s \Rightarrow (r^+ \cap s) O r \subseteq s \Rightarrow r^+ \subseteq s \)
  by (fastforce simp add: elim: tranclp-induct)

lemma trancl-unfold: \( r^+ = r \cup r^+ O r \)
by (auto intro: trancl-into-trancl elim: tranclE)

Transitivity of $r^+$

**Lemma** trans-trancl \[\text{(simp): trans } (r^+)\]

**Proof** (rule transI)
- fix $x$ $y$ $z$
- assume $(x, y) \in r^+$
- assume $(y, z) \in r^+$
- then show $(x, z) \in r^+$

**Proof** induct
- case (base $u$)
  - from $\langle x, y \rangle \in r^+$, and $\langle y, u \rangle \in r$
  - show $(x, u) \in r^+$ ..
- next
  - case (step $u$ $v$)
  - from $\langle x, u \rangle \in r^+$, and $\langle u, v \rangle \in r$
  - show $(x, v) \in r^+$ ..
- qed
- qed

**Lemmas** trancl-trans = trans-trancl \[\text{THEN transD}\]

**Lemma** tranclp-trans:
- assumes $r^{++} x y$
  - and $r^{++} y z$
- shows $r^{++} x z$
- using assms\((2,1)\) by induct iprover+

**Lemma** trancl-id \[\text{[simp]: trans } r \implies r^+ = r\]

- unfolding trans-def by (fastforce simp add: elim: trancl-induct)

**Lemma** rtranclp-tranclp-tranclp:
- assumes $r^{**} x y$
- shows $\forall z. r^{++} y z \implies r^{++} x z$
- using assms by induct (iprover intro: tranclp-trans)+

**Lemmas** rtrancl-trancl-trancl = rtranclp-tranclp-tranclp \[\text{to-set}\]

**Lemma** tranclp-into-tranclp2: $r a b \implies r^{++} b c \implies r^{++} a c$
- by (erule tranclp-trans \[OF tranclp.r-into-trancl\])

**Lemmas** trancl-into-trancl2 = tranclp-into-tranclp2 \[\text{to-set}\]

**Lemma** tranclp-converseI:\n- assumes $(r^{++})^{-1-1} x y$
- shows $(r^{-1-1})^{++} x y$
- using conversepD \[OF assms\]

**Proof** (induction rule: tranclp-induct)
- case (base $y$)
- then show ?case
THEORY “Transitive-Closure” 550

by (iprover intro: conversepI)
next
case (step y z)
then show ?case
  by (iprover intro: conversepI tranclp-trans)
qed

lemmas trancl-converseI = tranclp-converseI [to-set]

lemma tranclp-converseD:
  assumes \((r^{1-1})^{++} x y\) shows \((r^{++})^{-1-1} x y\)
proof –
  have \(r^{++} y x\)
    using assms
    by (induction rule: tranclp-induct) (iprover dest: conversepD intro: tranclp-trans)+
then show ?thesis
  by (rule conversepI)
qed

lemmas trancl-converseD = tranclp-converseD [to-set]

lemma tranclp-converse: \((r^{1-1})^{++} = (r^{++})^{-1-1}\)
by (fastforce simp add: fun-eq-iff intro: tranclp-converseI dest: tranclp-converseD)

lemmas trancl-converse = tranclp-converse [to-set]

lemma sym-trancl: \(\text{sym } r \implies \text{sym } (r^+)\)
  by (simp only: sym-conv-converse-eq trancl-converse [symmetric])

lemma converse-tranclp-induct [consumes 1, case-names base step]:
  assumes major: \(r^{++} a b\)
  and cases: \(\forall y. r y b \implies P y \land y z. r y z \implies r^{++} z b \implies P z \implies P y\)
  shows \(P a\)
proof —
  have \(r^{-1-1}^{++} b a\)
    by (intro tranclp-converseI conversepI major)
then show ?thesis
  by (induction rule: tranclp-induct) (blast intro: cases dest: tranclp-converseD)+
qed

lemmas converse-tranclp-induct = converse-tranclp-induct [to-set]

lemma tranclpD: \(R^{++} x y \implies \exists z. R x z \land R^{**} z y\)
proof (induction rule: converse-tranclp-induct)
case (step u v)
then show ?case
  by (blast intro: tranclp-trans)
qed auto
lemmas tranclD = tranclpD [to-set]

lemma converse-tranclpE:
  assumes major: tranclp r x z
  and base: r x z \Rightarrow P
  and step: \forall y. r x y \Rightarrow tranclp r y z \Rightarrow P
  shows P
proof –
  from tranclpD [OF major] obtain y where r x y and rtranclp r y z
  by iprover
  from this(2) show P
proof (cases rule: rtranclp.cases)
  case rtrancl-refl
  with (r x y) base show P
  by iprover
next
  case rtrancl-into-rtrancl
  then have tranclp r y z
  by (iprover intro: rtranclp-into-tranclp1)
  with (r x y) step show P
  by iprover
qed

lemmas converse-tranclE = converse-tranclpE [to-set]

lemma tranclD2: (x, y) \in R^+ \Rightarrow \exists z. (x, z) \in R^* \land (z, y) \in R
by (blast elim: tranclE intro: trancl-into-rtrancl)

lemma irrefl-tranclI: r^{-1} \cap r^* = \{\} \Rightarrow (x, x) \notin r^+
by (blast elim: tranclE dest: trancl-into-rtrancl)

lemma irrefl-trancl-rD: \forall x. (x, x) \notin r^+ \Rightarrow (x, y) \in r \Rightarrow x \neq y
by (blast dest: r-into-trancl)

lemma trancl-subset-Sigma-aux: (a, b) \in r^* \Rightarrow r \subseteq A \times A \Rightarrow a = b \lor a \in A
by (induct rule: rtrancl-induct) auto

lemma trancl-subset-Sigma:
  assumes r \subseteq A \times A shows r^+ \subseteq A \times A
proof (rule trancl-Int-subset [OF assms])
  show (r^+ \cap A \times A \subseteq A \times A
    using assms by auto
qed

lemma reflclp-tranclp [simp]: (r^{+\+})^{=} = r^{**}
by (fast elim: rtranclp.cases tranclp-into-rtranclp dest: rtranclp-into-tranclp1)

lemmas reflcl-trancl [simp] = reflclp-tranclp [to-set]
lemma trancl-refl [simp]: \((r^=)^+ = r^*
\)
proof
  have \((a, b) \in (r^=)^+ \implies (a, b) \in r^*\) for \(a\ b\)
    by (force dest: trancl-into-rtrancl)
  moreover have \((a, b) \in (r^=)^+\) if \((a, b) \in r^*\) for \(a\ b\)
    using that
  proof (cases \(a\ b\) rule: rtranclE)
    case step
    show ?thesis
      by (rule rtrancl-into-trancl1) (use step in auto)
  qed auto
ultimately show ?thesis
  by auto
qed

lemma rtrancl-trancl-refl [code]: \(r^* = (r^=)^=\)
  by simp

lemma trancl-empty [simp]: \(\{\}\^+ = \{\}\)
  by (auto elim: trancl-induct)

lemma rtrancl-empty [simp]: \(\{\}\^* = Id\)
  by (rule subst [OF reflcl-trancl]) simp

lemma rtranclpD:
  \(R^* a b \implies a = b \lor a \neq b \land R^+ a b\)
  by (force simp [symmetric] simp del: reflclp-tranclp)

lemmas rtranclD = rtranclpD [to-set]

lemma rtrancl-eq-or-trancl:
  \((x,y) \in R^* \iff x = y \lor x \neq y \land (x, y) \in R^+\)
  by (fast elim: trancl-into-rtrancl dest: rtranclD)

lemma trancl-unfold-right: \(r^+ = r^* O r\)
  by (auto dest: tranclD2 intro: rtrancl-into-trancl1)

lemma trancl-unfold-left: \(r^+ = r O r^*\)
  by (auto dest: tranclD intro: rtrancl-into-trancl2)

lemma trancl-insert:
  \(\{\text{insert } (y, x) r\}^+ = r^+ \cup \{(a, b). (a, y) \in r^* \land (x, b) \in r^*\}\)
  — primitive recursion for trancl over finite relations

proof
  have \((a, b) \in (\text{insert } (y, x) r)^+ \implies \)
    \((a, b) \in r^+ \cup \{(a, b). (a, y) \in r^* \land (x, b) \in r^*\}\)
    by (erule trancl-induct) (blast intro: rtrancl-into-trancl1 trancl-into-trancl trancl-trans)+
  moreover have \((r^+ \cup \{(a, b). (a, y) \in r^* \land (x, b) \in r^*\} \subseteq (\text{insert } (y, x) r)^+)\)
qed
ultimately show \(\text{thesis}\)
  by auto
qed

lemma trancl-insert2:
  \[(\text{insert } (a, b) \, r^+) = r^+ \cup \{(x, y). ((x, a) \in r^+ \land x = a) \land ((b, y) \in r^+ \lor y = b)\}\]
  by (auto simp: trancl-insert rtrancl-eq-or-trancl)

lemma rtrancl-insert: \((\text{insert } (a, b) \, r)^* = r^* \cup \{(x, y). (x, a) \in r^* \land (b, y) \in r^*\}\)
using trancl-insert[of a b r]
  by (simp add: rtrancl-trancl-refcl del: reflcl-trancl) blast

Simplifying nested closures

lemma rtrancl-trancl-absorb[simp]: \((R^*)^+ = R^*\)
  by (simp add: trans-rtrancl)

lemma trancl-rtrancl-absorb[simp]: \((R^+)^* = R^*\)
  by (subst reflcl-trancl[symmetric]) simp

lemma rtrancl-refcl-absorb[simp]: \((R^*)^\omega = R^*\)
  by auto

Domain and Range

lemma Domain-rtrancl [simp]: \(\text{Domain } (R^*) = \text{UNIV}\)
  by blast

lemma Range-rtrancl [simp]: \(\text{Range } (R^*) = \text{UNIV}\)
  by blast

lemma rtrancl-Un-subset: \((R^* \cup S^*) \subseteq (R \cup S)^*\)
  by (rule rtrancl-Un-rtrancl [THEN subst]) fast

lemma in-rtrancl-Un: \(x \in R^* \lor x \in S^* \implies x \in (R \cup S)^*\)
  by (blast intro: subsetD [OF rtrancl-Un-subset])

lemma trancl-domain [simp]: \(\text{Domain } (r^+) = \text{Domain } r\)
  by (unfold Domain-unfold) blast

lemma trancl-range [simp]: \(\text{Range } (r^+) = \text{Range } r\)

unfolding Domain-converse [symmetric] by (simp add: trancl-converse [symmetric])

lemma Not-Domain-rtrancl:
  assumes \(x \notin \text{Domain } R\) shows \((x, y) \in R^* \iff x = y\)
proof
  have \((x, y) \in R^* \implies x = y\)
    by (erule trancl-induct) (use assms in auto)
  then show \(\text{thesis}\)
    by auto
qed

lemma trancl-subset-Field2: \( r^+ \subseteq \text{Field } r \times \text{Field } r \)
by (rule trancl-Int-subset) (auto simp: Field-def)

lemma finite-trancl[simp]: finite \( (r^+) \) = finite \( r \)
proof
  show finite \( (r^+) \) = finite \( r \)
    by (blast intro: r-into-trancl' finite-subset)
  qed

lemma finite-rtrancl-Image[simp]: assumes finite \( R \) finite \( A \)
  shows finite \( (R^\ast'' A) \)
proof (rule ccontr)
  assume infinite \( (R^\ast'' A) \)
  with assms show False
  by (simp add: rtrancl-trancl-reflcl Un-Image del: reflcl-trancl)
qed

More about converse \( rtrancl \) and \( trancl \), should be merged with main body.

lemma single-valued-confluent:
  assumes single-valued \( r \) and \( xy: (x, y) \in r^* \) and \( zx: (x, z) \in r^* \)
  shows \( (y, z) \in r^* \) \or \( (z, y) \in r^* \)
using \( xy \)
proof (induction rule: rtrancl-induct)
  case base
  show ?case
    by (simp add: assms)
next
  case (step y z)
  with \( zx \) (single-valued \( r \)) show ?case
  by (auto elim: converse-rtranclE dest: single-valuedD intro: rtrancl-trans)
qed

lemma r-r-into-trancl: \( (a, b) \in R \Longrightarrow (b, c) \in R \Longrightarrow (a, c) \in R^+ \)
by (fast intro: trancl-trans)

lemma trancl-into-trancl: \( (a, b) \in r^+ \Longrightarrow (b, c) \in r \Longrightarrow (a, c) \in r^+ \)
by (induct rule: trancl-induct) (fast intro: r-r-into-trancl trancl-trans)

lemma tranclp-rtranclp-tranclp:
  assumes \( r^{++'} \) \( a \ b \ r^{++'} \ b \ c \) shows \( r^{++} \ a \ c \)
proof –
  obtain \( z \) where \( r \ a \ z \ r^{++} \ z \ c \)
  using assms by (iprover dest: tranclpD rtranclp-trans)
  then show ?thesis
    by (blast dest: rtranclp-into-tranclp2)
qed

**Lemma:** rtranclp-conversep: $r^{-1-1**} = r**-1-1$

by (auto simp add: fun-eq-iff intro: rtranclp-converseI rtranclp-converseD)

**Lemmas:**
- symp-rtranclp = sym-rtrancl[to-pred]
- symp-conv-conversep-eq = sym-conv-converse-eq[to-pred]
- rtranclp-tranclp-absorb [simp] = rtrancl-trancl-absorb[to-pred]
- tranclp-rtranclp-absorb [simp] = trancl-rtrancl-absorb[to-pred]
- rtranclp-reflclp-absorb [simp] = rtrancl-reflcl-absorb[to-pred]

**Lemmas:**
- transitive-closure-trans [trans] = r-r-into-trancl trancl-trans rtrancl-trans
  trancl.trancl-into-trancl trancl-into-trancl2
  rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
  rtrancl-trancl-trancl-trancl
- transitive-closurep-trans' [trans] = tranclp-trans rtranclp-trans
  tranclp.trancl-into-trancl tranclp-into-trancl2
  rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
  rtranclp-rtranclp-tranclp tranclp-rtranclp-tranclp

**Declare:** trancl-into-rtrancl [elim]

**Lemma:** tranclp-ident-if-transp:
- assumes transp R
- shows $R^{++} = R$

**Proof:** (intro ext iffI)
- fix $x \, y$
- show $R^{++} x \, y \Longrightarrow R x \, y$

**Proof:** (induction $y$ rule: tranclp-induct)
- case (base $y$)
  - thus ?case
    - by simp
- next
  - case (step $y \, z$)
  - thus ?case
    - using (transp $R$)[THEN transpD, of $x \, y \, z$] by simp

**Qed**

**Next**
- fix $x \, y$
- show $R x \, y \Longrightarrow R^{++} x \, y$
  - using tranclp.r-into-trancl .

**Qed**
The following are special cases of \textit{tranclp-ident-if-transp}, but they appear duplicated in multiple, independent theories, which causes name clashes.

\begin{enumerate}
\item \textbf{lemma} (in preorder) \textit{tranclp-less}:[simp]: $(\prec)^{++} = (\prec)$
  \textit{using} \textit{transp-on-less} \textit{by} (simp only: \textit{tranclp-ident-if-transp})
\item \textbf{lemma} (in preorder) \textit{tranclp-less-eq}:[simp]: $(\preceq) = (\preceq)$
  \textit{using} \textit{transp-on-le} \textit{by} (simp only: \textit{tranclp-ident-if-transp})
\item \textbf{lemma} (in preorder) \textit{tranclp-greater}:[simp]: $(\succ)^{++} = (\succ)$
  \textit{using} \textit{transp-on-greater} \textit{by} (simp only: \textit{tranclp-ident-if-transp})
\item \textbf{lemma} (in preorder) \textit{tranclp-greater-eq}:[simp]: $(\succeq) = (\succeq)$
  \textit{using} \textit{transp-on-ge} \textit{by} (simp only: \textit{tranclp-ident-if-transp})
\end{enumerate}

21.4 Symmetric closure

\begin{enumerate}
\item \textbf{definition} \textit{symclp} :: $(\forall a \rightarrow \forall a \rightarrow \text{bool}) \rightarrow \forall a \rightarrow \forall a \rightarrow \text{bool}$
  \textit{where} \textit{symclp r x y} \textit{←→} \textit{r x y} \textit{∨} \textit{r y x}
\item \textbf{lemma} \textit{symclpI}:[simp, intro]: shows \textit{symclpI1}: \textit{r x y} = \textit{⇒} \textit{symclp r x y}
  \textit{and} \textit{symclpI2}: \textit{r y x} = \textit{⇒} \textit{symclp r x y}
  \textit{by}(simp-all add: \textit{symclp-def})
\item \textbf{lemma} \textit{symclpE} [consumes 1, cases pred]:
  \textit{assumes} \textit{symclp r x y}
  \textit{obtains} (base) \textit{r x y} \textit{|} (sym) \textit{r y x}
  \textit{using} \textit{assms} \textit{by}(auto simp add: \textit{symclp-def})
\item \textbf{lemma} \textit{symclp-pointfree}: \textit{symclp r} = \textit{sup r} \textit{r}^{-1-1}
  \textit{by}(auto simp add: \textit{symclp-def fan-eq-iff})
\item \textbf{lemma} \textit{symclp-greater}: \textit{r} \textit{≤} \textit{symclp r}
  \textit{by}(simp add: \textit{symclp-pointfree})
\item \textbf{lemma} \textit{symclp-conversep} [simp]: \textit{symclp r}^{-1-1} = \textit{symclp r}
  \textit{by}(simp add: \textit{symclp-pointfree sup.commute})
\item \textbf{lemma} \textit{symp-on-symclp} [simp]: \textit{symp-on} \textit{A} (\textit{symclp} \textit{R})
  \textit{by}(auto simp add: \textit{symclp-pointfree sup.commute})
\item \textbf{lemma} \textit{symp-symclp-eq}: \textit{symp} \textit{r} \textit{⇒} \textit{symclp r} = \textit{r}
  \textit{by}(simp add: \textit{symclp-pointfree symp-cone-conversep-eq})
\item \textbf{lemma} \textit{symp-rtranclp-symclp} [simp]: \textit{symp} (\textit{symclp} \textit{r})^{**}
  \textit{by}(simp add: \textit{symclp-pointfree symp-rtranclp})
\item \textbf{lemma} \textit{rtranclp-symclp-sym} [sym]: (\textit{symclp} \textit{r})^{**} \textit{x} \textit{y} \textit{⇒} (\textit{symclp} \textit{r})^{**} \textit{y} \textit{x}
  \textit{by}(rule \textit{sympD}[OF \textit{symp-rtranclp-symclp}])
\end{enumerate}
**THEORY** “Transitive-Closure”

**lemma** symclp-idem [simp]: symclp (symclp r) = symclp r
  
  by (simp add: symclp-pointfree sup-commute converse-join)

**lemma** reflp-on-rtranclp [simp]: reflp A R**
  
  by (simp add: reflp-on-def)

### 21.5 The power operation on relations

$$R \sim n = R \circ \ldots \circ R$$, the n-fold composition of $$R$$

**overloading**

relpow ≡ compow :: nat ⇒ ('a × 'a) set ⇒ ('a × 'a) set

relpowp ≡ compow :: nat ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool)

begin

primrec relpow :: nat ⇒ ('a × 'a) set ⇒ ('a × 'a) set

where

  relpow 0 R = Id

| relpow (Suc n) R = (relpow n R) O R

primrec relpowp :: nat ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool)

where

  relpowp 0 R = HOL.eq

| relpowp (Suc n) R = (relpow n R) OO R

end

**lemma** relpowp-relpow-eq [pred-set-conv]:

$$(\lambda x. y. (x, y) \in R) \sim n = (\lambda x. y. (x, y) \in R \sim n)$$ for $$R : \text{rel}$$

by (induct n) (simp-all add: relcompp-relcomp-eq)

For code generation:

**definition** relpow :: nat ⇒ ('a × 'a) set ⇒ ('a × 'a) set

where relpow-code-def [code-abbrev]: relpow = compow

**definition** relpowp :: nat ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool)

where relpowp-code-def [code-abbrev]: relpowp = compow

**lemma** [code]:

relpow (Suc n) R = (relpow n R) O R

relpow 0 R = Id

by (simp-all add: relpow-code-def)

**lemma** [code]:

relpowp (Suc n) R = (R \sim n) OO R

relpowp 0 R = HOL.eq

by (simp-all add: relpowp-code-def)

hide-const (open) relpow
THEORY “Transitive-Closure”

hide-const (open) relpowp

lemma relpow-1 [simp]: R ^^ 1 = R
for R :: (′a × ′a) set
by simp

lemma relpowp-1 [simp]: P ^^ 1 = P
for P :: ′a ⇒ ′a ⇒ bool
by (fact relpow-1 [to-pred])

lemma relpow-0-I: (x, x) ∈ R ^^ 0
by simp

lemma relpowp-0-I: (P ^^ 0) x x
by (fact relpow-0-I [to-pred])

lemma relpow-Suc-I: (x, y) ∈ R ^^ n ⇒ (y, z) ∈ R ⇒ (x, z) ∈ R ^^ Suc n
by auto

lemma relpowp-Suc-I: P x y ⇒ (P ^^ n) y z ⇒ (P ^^ Suc n) x z
by (fact relpow-Suc-I [to-pred])

lemma relpow-Suc-E: (x, z) ∈ R ^^ Suc n ⇒ (∀ y. (x, y) ∈ R ⇒ (y, z) ∈ R ⇒ P) ⇒ P
by auto

lemma relpowp-Suc-E: (P ^^ Suc n) x z ⇒ (P ^^ n) x y ⇒ P y z ⇒ Q)
⇒ Q
by (fact relpow-Suc-E [to-pred])

lemma relpow-E:
(x, z) ∈ R ^^ n ⇒
(n = 0 ⇒ x = z ⇒ P) ⇒
(∀ y m. n = Suc m ⇒ (x, y) ∈ R ^ m ⇒ (y, z) ∈ R ⇒ P) ⇒ P
by (cases n) auto

lemma relpowp-E:
(\(P \triangleq n\)) \(x\ z \Rightarrow\n
(n = 0 \Rightarrow x = z \Rightarrow Q) \Rightarrow \)
\((\forall y. m. \ n = \text{Suc}\ m \Rightarrow (P \triangleq m) \ x\ y \Rightarrow P\ y\ z \Rightarrow Q) \Rightarrow Q \n
\text{by (fact relpow-E [to-pred])} \)

\textbf{lemma} \: \text{relpow-Suc-D2}: \((x,\ z) \in R \triangleq \text{Suc}\ n \Rightarrow \exists y. (x,\ y) \in R \land (y,\ z) \in R \triangleq n \Rightarrow P) \Rightarrow P \n
\text{by (induct n arbitrary: } x\ z) \n
\text{ (blast intro: relpow-0-I relpow-Suc-I elim: relpow-0-E relpow-Suc-E)+} \n
\textbf{lemma} \: \text{relpowp-Suc-D2}: \((P \triangleq \text{Suc}\ n) \ x\ z \Rightarrow \exists y. P\ x\ y \land (P \triangleq n)\ y\ z \n
\text{by (fact relpow-Suc-D2 [to-pred])} \n
\textbf{lemma} \: \text{relpow-Suc-E2}: \((x,\ z) \in R \triangleq \text{Suc}\ n \Rightarrow (\forall y. (x,\ y) \in R \Rightarrow (y,\ z) \in R \triangleq n \Rightarrow P) \Rightarrow P \n
\text{by (blast dest: relpow-Suc-D2)} \n
\textbf{lemma} \: \text{relpowp-Suc-E2}: \((P \triangleq \text{Suc}\ n) \ x\ z \Rightarrow (\forall y. P\ x\ y \Rightarrow (P \triangleq n)\ y\ z \Rightarrow Q) \Rightarrow Q \n
\text{by (fact relpow-Suc-E2 [to-pred])} \n
\textbf{lemma} \: \text{relpow-Suc-D2'}: \forall x\ y\ z. (x,\ y) \in R \triangleq \text{Suc}\ n \land (y,\ z) \in R \Rightarrow (\exists w. (x,\ w) \in R \land (w,\ z) \in R \triangleq n) \n
\text{by (induct n) (simp-all, blast)} \n
\textbf{lemma} \: \text{relpowp-Suc-D2'}: \forall x\ y\ z. (P \triangleq n) \ x\ y \land P\ y\ z \Rightarrow (\exists w. P\ x\ w \land (P \triangleq n)\ w\ z) \n
\text{by (fact relpow-Suc-D2' [to-pred])} \n
\textbf{lemma} \: \text{relpow-E2}: \n\text{assumes} (x,\ z) \in R \triangleq n \ n = 0 \Rightarrow x = z \Rightarrow P \n\n(\forall y. m. \ n = \text{Suc}\ m \Rightarrow (x,\ y) \in R \Rightarrow (y,\ z) \in R \triangleq m \Rightarrow P) \n\n\text{shows} P \n\n\textbf{proof} (cases n) \n\n\text{case } 0 \n\n\text{with assms show } ?\text{thesis} \n\n\text{by simp} \n\n\textbf{next} \n\n\text{case } (\text{Suc}\ m) \n\n\text{with assms relpow-Suc-D2' [of } m\ R\text{]} \textbf{show} ?\text{thesis} \n\n\text{by force} \n\n\text{qed} \n\n\textbf{lemma} \: \text{relpowp-E2}: \n(P \triangleq n) \ x\ z \Rightarrow\n(n = 0 \Rightarrow x = z \Rightarrow Q) \Rightarrow \n(\forall y. m. \ n = \text{Suc}\ m \Rightarrow P\ x\ y \Rightarrow (P \triangleq m)\ y\ z \Rightarrow Q) \Rightarrow Q \n
\text{by (fact relpow-E2 [to-pred])} \n
THEORY “Transitive-Closure”
lemma relpowp-trans[trans]: \((R \rightarrow i) x y \Longrightarrow (R \rightarrow j) y z \Longrightarrow (R \rightarrow (i + j)) x z\)

proof (induction \(i\) arbitrary: \(x\))
  case 0
  thus ?case by simp
next
  case (Suc \(i\))
  obtain \(x'\) where \((R \rightarrow i) x x'\) and \((R \rightarrow i) x' y\)
    using \(\langle R \rightarrow Suc \ i \rangle x y\) [THEN relpowp-Suc-D2] by auto

  show \((R \rightarrow (i + j)) x z\)
    unfolding add-Suc
  proof (rule relpowp-Suc-I2)
    show \((R \rightarrow i) x x'\) using \(\langle R \rightarrow Suc \ i \rangle x x'\).
    next
    show \((R \rightarrow j) y z\) using Suc.IH [OF \(\langle R \rightarrow i \rangle x x'\) \(\langle R \rightarrow j \rangle y z\)].
    qed
  qed

  lemma relpow-trans[trans]: \((x, y) \in R \rightarrow i \Longrightarrow (y, z) \in R \rightarrow j \Longrightarrow (x, z) \in R \rightarrow (i + j)\)
    using relpowp-trans[to-set].

lemma relpowp-left-unique:
  fixes \(R::'a \Rightarrow 'a \Rightarrow bool\) and \(n::nat\) and \(x y z::'a\)
  assumes lunique: \(\forall x y z. \ R x z \Longrightarrow R y z \Longrightarrow x = y\)
  shows \((R \rightarrow n) x z \Longrightarrow (R \rightarrow n) y z \Longrightarrow x = y\)
proof (induction \(n\) arbitrary: \(x y z\))
  case 0
  thus ?case by simp
next
  case (Suc \(n'\))
  then obtain \(x' y'::'a\) where
    \((R \rightarrow n') x x'\) and \((R \rightarrow n') x' z\) and
    \((R \rightarrow n') y y'\) and \((R \rightarrow n') y' z\)
    by auto

  have \(x' = y'\)
    using lunique[OF \(\langle R \rightarrow n' \rangle x x'\) \(\langle R \rightarrow n' \rangle y y'\)].

  show \(x = y\)
  proof (rule Suc.IH)
    show \((R \rightarrow n') x x'\)
      using \(\langle R \rightarrow n' \rangle x x'\).
    next
    show \((R \rightarrow n') y x'\)
      using \(\langle R \rightarrow n' \rangle y x'\).
unfolding \langle x' = y' \rangle .

qed

lemma relpow-left-unique:

fixes R :: ('a × 'a) set and n :: nat and x y z :: 'a

shows \( (\forall x y z. (x, z) \in R \implies (y, z) \in R \implies x = y) \implies (x, z) \in R \sp{n} \implies (y, z) \in R \sp{n} \implies x = y \) 

using relpowp-left-unique[to-set].

lemma relpowp-right-unique:

fixes R :: 'a ⇒ 'a ⇒ bool and n :: nat and x y z :: 'a

assumes runique: \( (\forall x y z. R x y \implies R x z \implies y = z) \)

shows \( (R \sp{n}) x y \implies (R \sp{n}) x z \implies y = z \)

proof (induction n arbitrary: x y z)
  case 0
  thus ?case by simp
next
  case (Suc n')
  then obtain x' :: 'a where
    \( (R \sp{n'}) x x' \) and \( R x' y \) and \( R x' z \)
    by auto
  thus y = z
    using runique by simp
qed

lemma relpow-right-unique:

fixes R :: ('a × 'a) set and n :: nat and x y z :: 'a

shows \( (\forall x y z. (x, y) \in R \implies (x, z) \in R \implies y = z) \implies (x, y) \in (R \sp{n}) \implies (x, z) \in (R \sp{n}) \implies y = z \) 

using relpowp-right-unique[to-set].

lemma relpow-add: \( R \sp{\langle m + n \rangle} = R \sp{\langle m \rangle} O R \sp{\langle n \rangle} \)

by (induct n) auto

lemma relpow-add: \( P \sp{\langle m + n \rangle} = P \sp{\langle m \rangle} OO P \sp{\langle n \rangle} \)

by (fact relpow-add[to-pred])

lemma relpow-commute: \( R O R \sp{\langle n \rangle} = R \sp{\langle n \rangle} O R \)

by (induct n) (simp-all add: O-assoc [symmetric])

lemma relpow-commute: \( P OO P \sp{\langle n \rangle} = P \sp{\langle n \rangle} OO P \)

by (fact relpow-commute[to-pred])

lemma relpow-empty: \( 0 < n \implies (\{\} :: ('a × 'a) set) \sp{\langle n \rangle} = \{\} \)

by (cases n) auto

lemma relpow-bot: \( 0 < n \implies (\bot :: 'a ⇒ 'a ⇒ bool) \sp{\langle n \rangle} = \bot \)
by (fact relpow-empty [to-pred])

lemma rtrancl-imp-UN-relpow:
  assumes \( p \in R^* \)
  shows \( p \in (\bigcup n. R \ ^\sim n) \)
proof (cases \( p \))
  case (Pair \( x \ y \))
  with assms have \( (x, y) \in R^* \) by simp
  then have \( (x, y) \in (\bigcup n. R \ ^\sim n) \)
  proof induct
    case base
    show \(?case\) by (blast intro: relpow-0-I)
  next
    case step
    then show \(?case\) by (blast intro: relpow-Suc-I)
  qed
with Pair show \(?thesis\) by simp
qed

lemma rtranclp-imp-Sup-relpowp:
  assumes \( (P^*) \ x \ y \)
  shows \( (\bigcap n. P \ ^\sim n) \ x \ y \)
using assms and rtrancl-imp-UN-relpow [of \( x \ y \), to-pred] by simp

lemma relpow-imp-rtrancl:
  assumes \( p \in R \ ^\sim n \)
  shows \( p \in R^* \)
proof (cases \( p \))
  case (Pair \( x \ y \))
  with assms have \( (x, y) \in R \ ^\sim n \) by simp
  then have \( (x, y) \in R^* \)
  proof (induct \( n \) arbitrary: \( x \ y \))
    case 0
    then show \(?case\) by simp
  next
    case Suc
    then show \(?case\)
      by (blast elim: relpow-Suc-E intro: rtrancl-into-rtrancl)
  qed
with Pair show \(?thesis\) by simp
qed

lemma relpowp-imp-rtranclp: \( (P \ ^\sim n) \ x \ y \implies (P^*) \ x \ y \)
using relpow-imp-rtrancl [of \( x \ y \), to-pred] by simp

lemma rtrancl-is-UN-relpow: \( R^* = (\bigcup n. R \ ^\sim n) \)
by (blast intro: rtrancl-imp-UN-relpow relpow-imp-rtrancl)

lemma rtranclp-is-Sup-relpowp: \( P^* = (\bigcap n. P \ ^\sim n) \)
using \texttt{rtrancl-is-UN-relpow [to-pred, of \textit{P}]} \textbf{by} \textit{auto}

\textbf{lemma \texttt{rtrancl-power}}: \(p \in R^* \iff (\exists n. \ p \in R \bowtie n)\)
\textbf{by (simp add: rtrancl-is-UN-relpow)}

\textbf{lemma \texttt{rtranclp-power}}: \((P^{**}) \ x \ y \iff (\exists n . (P \bowtie n) \ x \ y)\)
\textbf{by (simp add: rtranclp-is-Sup-relpowp)}

\textbf{lemma \texttt{trancl-power}}: \(p \in R^+ \iff (\exists n > 0 . \ p \in R \bowtie n)\)
\textbf{proof –}
\begin{itemize}
  \item have \((a, \ b) \in R^+ \iff (\exists n>0. \ (a, \ b) \in R \bowtie n)\) \textit{for} \(a, \ b\)
  \textbf{proof safe}
  \begin{itemize}
    \item show \((a, \ b) \in R^+ \implies \exists n>0. \ (a, \ b) \in R \bowtie n\)
      \textbf{by (fastforce simp: rtrancl-is-UN-relpow relcomp-unfold dest: tranclD2)}
    \item show \((a, \ b) \in R^+ \text{ if} \ n > 0 \ (a, \ b) \in R \bowtie n \text{ for} \ n\)
      \textbf{proof (cases n)}
      \begin{itemize}
        \item case \((\text{Suc} \ n)\)
        \textbf{with that show \(\exists n>0. \ (a, \ b) \in R \bowtie n\) \textit{for} \(n\)}
        \textbf{by (auto simp: dest: relpow-imp-rtrancl rtrancl-into-trancl1)}
      \end{itemize}
    \end{itemize}
\end{itemize}
\textbf{qed (use that in auto)}
\textbf{qed}
\textbf{then show \(\exists n > 0. \ p \in R \bowtie n\) \textit{by (cases} \textit{p) auto \textit{qed}}

\textbf{lemma \texttt{tranclp-power}}: \((P^{++}) \ x \ y \iff (\exists n > 0 . (P \bowtie n) \ x \ y)\)
\textbf{using trancl-power \texttt{[to-pred, of \textit{P} (x, y)]} \textbf{by simp}

\textbf{lemma \texttt{rtrancl-imp-relpow}}: \(p \in R^* \implies \exists n. \ p \in R \bowtie n\)
\textbf{by (auto dest: rtrancl-imp-UN-relpow)}

\textbf{lemma \texttt{rtranclp-imp-relpowp}}: \((P^{**}) \ x \ y \implies \exists n. \ (P \bowtie n) \ x \ y\)
\textbf{by (auto dest: rtranclp-imp-Sup-relpowp)}

By Sternagel/Thiemann:

\textbf{lemma \texttt{relpow-fun-conv}}: \((a, \ b) \in R \bowtie n \iff (\exists f . \ f \ 0 = a \land f \ n = b \land (\forall i < n. \ (f \ i, f \ (\text{Suc} \ i)) \in R))\)
\textbf{proof (induct} \textit{n arbitrary:} \textit{b)}
\begin{itemize}
  \item case \(0\)
  \textbf{show \(\exists case\) \textit{by auto}}
\end{itemize}
\textbf{next}
\begin{itemize}
  \item case \((\text{Suc} \ n)\)
  \textbf{show \(\exists case\) \textit{by auto}}
\end{itemize}
\textbf{proof –}
\begin{itemize}
  \item have \((\exists y. \ (\exists f . \ f \ 0 = a \land f \ n = y \land (\forall i < n. \ (f \ i, f \ (\text{Suc} \ i)) \in R)) \land (y, b) \in R)\)
    \textbf{is} \(\iff (\exists f . \ f \ 0 = a \land f \ (\text{Suc} \ n) = b \land (\forall i < Suc \ n. \ (f \ i, f \ (\text{Suc} \ i)) \in R))\)
    \textbf{proof}
\end{itemize}
THEORY "Transitive-Closure"

assume ?l
then obtain c f
  where 1: f 0 = a  f n = c  \(\forall i. i < n \implies (f i, f (Suc i)) \in R\)  (c,b) \in R
by auto
let \(\gamma = \lambda m. \) if m = Suc n then b else f m
show ?r by (rule exI[of - ?\gamma]) (simp add: 1)
next
assume ?r
then obtain f where 1: f 0 = a  b = f (Suc n)
  (Suc i) \in R
by auto
show ?l by (rule exI[of - f n], rule conjI, rule exI[of - f], auto simp add: 1)
qed
then show ?thesis by (simp add: relcomp-unfold Suc)
qed

lemma relpowp-fun-conv: 
(P ^^ n) x y \iff \exists f. f 0 = x \land f n = y \land (\forall i<n. P (f i) (f (Suc i)))
by (fact relpow-fun-conv [to-pred])

lemma relpow-finite-bounded1:
  fixes R :: ('a \times 'a) set
  assumes finite R and k > 0
  shows R ^^ k \subseteq (\bigcup n \in \{n. \theta < n \land n \leq \text{card} R\}. R ^^ n)
  (is - \subseteq ?r)
proof -
  have (a, b) \in R ^^ (Suc k) \implies \exists n. \theta < n \land n \leq \text{card} R \land (a, b) \in R ^^ n for a b k
  proof (induct k arbitrary: b)
    case 0
    then have R \neq {} by auto
    with card-0-eq[OF finite R] have card R \geq Suc \theta by auto
    then show ?case using 0 by force
  next
    case (Suc k)
    then obtain a' where (a, a') \in R ^^ (Suc k) and (a', b) \in R
    by auto
    from Suc(1)[OF (a, a') \in R ^^ (Suc k)] obtain n where n \leq \text{card} R and (a, a') \in R ^^ n
    by auto
    have (a, b) \in R ^^ (Suc n)
      using (a, a') \in R ^^ n and (a', b) \in R by auto
    from n \leq \text{card} R consider n < card R | n = card R by force
    then show ?case
    proof cases
      case 1
      then show ?thesis
        using (a, b) \in R ^^ (Suc n) Suc-leI[OF n < card R] by blast
next
case 2
from \langle a, b \rangle \in R \longmapsto (\text{Suc} n) \rangle \ [unfolded \ relpow-fun-conv]
obtain f where f 0 = a and f (\text{Suc} n) = b
and steps: \forall i. i \leq n \implies \langle f i, f (\text{Suc} i) \rangle \in R \ by \ auto
let \ ?p = \lambda i. \langle f i, f (\text{Suc} i) \rangle
let \ ?N = \{ i. i \leq n \}
have \ ?p \ ?N \subseteq R 
using steps by auto
from card-mono[of assms(1) this] have card (\ ?p \ ?N \leq\ card R .
also have \ldots < card ?N
using \ ?n = card R \ by \ simp
finally have \ \neg inj-on \ ?p \ ?N 
by (rule pigeonhole)
then obtain i j where \ i \leq n \ and \ j \leq n \ and \ ij: \ i \neq j \ and \ pij: \ ?p i = ?p j 
by (auto simp: inj-on-def)
let \ ?i = \min i j
let \ ?j = \max i j
have \ ?i \leq n \ and \ ?j \leq n \ and \ pij: \ ?p \ ?i = ?p \ ?j \ and \ ij: \ ?i \neq ?j 
using \ i \ j \ pij \ obtaining \ i \ j \ where \ i \leq n \ and \ j \leq n \ and \ ij: \ i \neq j 
and \ pij: \ ?p i = ?p j 
by blast
let \ ?g = \lambda l. \ if l \leq i \ then f l \ else f (l + (j - i))
let \ ?n = \text{Suc} (n - \ (j - i))
have abl: \langle a, b \rangle \in R \longmapsto ?n 
unfolding relpow-fun-conv proof (rule exI[of - ?g], intro conjI impI allI)
show \ ?g ?n = b 
using \ (f(Suc n) = b \ j \ ij \ by \ auto
next
fix \ k
assume \ k < \ ?n
show \ \ ?g k, ?g (\text{Suc} k) \in R
proof (cases \ k < \ i)
case True
with \ i \ have \ k \leq n 
by auto
from steps[of this] show \ ?thesis 
using True by simp
next
case False
then have \ i \leq k \ by auto
show \ ?thesis
proof (cases \ k = \ i)
case True
then show \ ?thesis 
using \ ij \ pij \ steps[of \ i] \ by \ simp
next
  case False
  with \langle i \leq k \rangle have i < k by auto
  then have small: k + \langle j - i \rangle \leq n
    using \langle k < n \rangle by arith
  show \langle thesis \rangle
    using steps[OF small] \langle i < k \rangle by auto
  qed
qed
qed
qed

lemma relpow-finite-bounded:
  fixes R :: ('a × 'a) set
  assumes finite R
  shows R ^^ k \subseteq (\bigcup n \in \{ n. n \leq \text{card} R \}. R ^^ n)
proof (cases k)
  case (Suc k)
  then show \langle thesis \rangle
    using relpow-finite-bounded1[OF assms, of k] by auto
qed
force

lemma rtrancl-finite-eq-relpow:
  finite R =\implies R^* = (\bigcup n \in \{ n. 0 < n \land n \leq \text{card} R \}. R ^^ n)
by (fastforce simp: rtrancl-power dest: relpow-finite-bounded)

lemma trancl-finite-eq-relpow:
  assumes finite R
  shows R^+ = (\bigcup n \in \{ n. 0 < n \land n \leq \text{card} R \}. R ^^ n)
proof
  have \forall a b. [0 < n; (a, b) \in R ^^ n] \implies \exists \exists x > 0. x \leq \text{card} R \land (a, b) \in R ^^ x
    using assms by (auto dest: relpow-finite-bounded1)
  then show \langle thesis \rangle
    by (auto simp: trancl-power)
qed

lemma finite-relcomp[simp,intro]:
  assumes finite R and finite S
  shows finite (R O S)
proof
  have R O S = (\bigcup (x, y) \in R. \bigcup (u, v) \in S. if u = y then \{(x, v)\} else {})
    by (force simp: split-def image-constant-conv split: if-splits)
then show \textit{thesis} \\
using \textit{assms} by \textit{clarsimp} \\
\texttt{qed}

\textbf{lemma} \textit{finite-relpow} [simp, intro]: \\
\textit{fixes} $R : (\, ^\prime a \times ^\prime a \,)$ \textit{set} \\
\textit{assumes} \textit{finite} $R$ \\
\textit{shows} $n > 0 \implies \textit{finite} (\, ^\prime a \times ^\prime a \,)^n$ \\
\textit{proof} (\textit{induct} $n$) \\
\textit{case} \textit{$\emptyset$} \\
then show \textit{?case} by \textit{simp} \\
next \\
\textit{case} (\textit{Suc} $n$) \\
then show \textit{?case} by (\textit{cases} $n$) (\textit{use \textit{assms} in \textit{simp-all}}) \\
\texttt{qed}

\textbf{lemma} \textit{single-valued-relpow}: \\
\textit{fixes} $R : (\, ^\prime a \times ^\prime a \,)$ \textit{set} \\
\textit{shows} \textit{single-valued} $R \implies \textit{single-valued} (\, ^\prime a \times ^\prime a \,)^n$ \\
\textit{proof} (\textit{induct} $n$ \textit{arbitrary:} $R$) \\
\textit{case} \textit{$\emptyset$} \\
then show \textit{?case} by \textit{simp} \\
next \\
\textit{case} (\textit{Suc} $n$) \\
\textit{show} \textit{?case} \\
by (\textit{rule single-valuedI}) \\
(\textit{use \textit{Suc} in \textit{fast dest: single-valuedD elim: relpow-Suc-E}}) \\
\texttt{qed}

\textbf{21.6 Bounded transitive closure}

\textbf{definition} \textit{ntrancl} :: $nat \Rightarrow (\, ^\prime a \times ^\prime a \,)$ \textit{set} \Rightarrow (\, ^\prime a \times ^\prime a \,)$ \textit{set} \\
\textit{where} \textit{ntrancl} $n$ $R$ \textit{=} \textit{(}\bigcup \textit{i} \in \{i. \, 0 < i \land i \leq \textit{Suc} \, n\}. \, R ^ i\textit{)}$

\textbf{lemma} \textit{ntrancl-Zero} [simp, code]: \textit{ntrancl} $0$ $R$ \textit{=} $R$ \\
\textit{proof} \\
\textit{show} $R \subseteq \textit{ntrancl} \, 0 \, R$ \\
unfolding \textit{ntrancl-def} by \textit{fastforce} \\
\textit{have} $0 < i \land i \leq \textit{Suc} \, 0 \iff i = 1$ \textit{for} $i$ \\
by \textit{auto} \\
\textit{then show} $\textit{ntrancl} \, 0 \, R \subseteq R$ \\
unfolding \textit{ntrancl-def} \textit{by} \textit{auto} \\
\texttt{qed}

\textbf{lemma} \textit{ntrancl-Suc} [simp]: \textit{ntrancl} \textit{(Suc} $n$) $R$ \textit{=} \textit{ntrancl} $n$ $R$ $O \ (\textit{Id} \cup R)$ \\
\textit{proof} \\
\textit{have} $(a, \ b) \in \textit{ntrancl} \, n \, R \ O \ (\textit{Id} \cup R)$ \textit{if} $(a, \ b) \in \textit{ntrancl} \, (\textit{Suc} \ n) \ R \ \textit{for} \ a \ , \ b$ \\
\textit{proof} \\
\textit{from} \textit{that} \textit{obtain} $i$ \textit{where} $0 < i \ i \leq \textit{Suc} \ (\textit{Suc} \ n)$ $(a, \ b) \in R^i$
unfolding \textit{ntrancl-def} by auto

show \textit{?thesis}

proof (cases i = 1)
  case True
  with \((a, b) \in R \rightsquigarrow i\) show \textit{?thesis}
    by (auto simp: \textit{ntrancl-def})

next
  case False
  with \(\langle 0 < i \rangle\) obtain \(j\) where \(j = \ Suc 0 < j\)
    by (cases i) auto
  with \(\langle a, b \rangle \in R \rightsquigarrow j\) obtain \(c\) where \(c1: (a, c) \in R \rightsquigarrow j\) and \(c2: (c, b) \in R\)

  have \(\langle a, c \rangle \in ntrancl \ n \ R\)
    by (fastforce simp: \textit{ntrancl-def})
  with \(c2\) show \textit{?thesis} by fastforce

qed

then show \(ntrancl (\ Suc \ n) \ R \subseteq ntrancl \ n \ R \ O \ (Id \cup \ R)\)
  by auto

show \(ntrancl \ n \ R \ O \ (Id \cup \ R) \subseteq ntrancl (\ Suc \ n) \ R\)
  by (fastforce simp: \textit{ntrancl-def})

qed

lemma [code]: \(ntrancl (\ Suc \ n) \ r = (\ let \ r' = ntrancl \ n \ r \ in \ r' \cup \ r' \ O \ r)\)
  by (auto simp: \textit{Let-def})

lemma \textit{finite-trancl-ntranl}: \(\ finite \ R \Longrightarrow \ trancl \ R = ntrancl (\ \ card \ R - 1 \) \ R\)
  by (cases card R) (auto simp: \textit{trancl-finite-eq-relpow relpow-empty ntrancl-def})

\section{21.7 Acyclic relations}

definition \textit{acyclic} :: \('a \times 'a\) set \Rightarrow bool
  \where \textit{acyclic} \ r \longleftrightarrow (\forall \ a. \ (x,x) \notin r^+)

abbreviation \textit{acyclicP} :: \('a \Rightarrow 'a \Rightarrow bool\) \Rightarrow bool
  \where \textit{acyclicP} \ r \equiv \textit{acyclic} \ \ \{(x, y). \ r \ x \ y\}

lemma \textit{acyclic-irrefl} [code]: \(\textit{acyclic} \ r \longleftrightarrow \textit{irrefl} \ (r^+)\)
  by (simp add: \textit{acyclic-def irrefl-def})

lemma \textit{acyclicI}: \(\forall \ a. \ (x, x) \notin r^+ \Longrightarrow \textit{acyclic} \ r\)
  by (simp add: \textit{acyclic-def})

lemma (in \textit{preorder}) \textit{acyclicI-order}:
  \assumes \textit{*}: \(\forall \ a. \ (a, b) \in r \Longrightarrow f b < f a\)
  \shows \textit{acyclic} \ r

proof
  have \(f b < f a\) if \((a, b) \in r^+\) for \(a, b\)
using that by induct (auto intro: less-trans) 
then show ?thesis
  by (auto intro!: acyclicI)
qed

lemma acyclic-insert [iff]: acyclic (insert (y, x) r) \iff acyclic r \land (x, y) \notin r^*
  by (simp add: acyclic-def trancl-insert) (blast intro: rtrancl-trans)

lemma acyclic-converse [iff]: acyclic (r^-1) \iff acyclic r
  by (simp add: acyclic-def trancl-converse)

lemmas acyclicP-converse [iff] = acyclic-converse [to-pred]

lemma acyclic-impl-antisym-rtrancl: acyclic r \implies antisym (r^*)
  by (simp add: acyclic-def antisym-def)
  (blast elim: rtranclE intro: rtrancl-into-trancl1 rtrancl-trancl-trancl)

lemma acyclic-subset: acyclic s \implies r \subseteq s \implies acyclic r
  unfolding acyclic-def by (blast intro: trancl-mono)

21.8 Setup of transitivity reasoner

ML "structure Trancl-Tac = Trancl-Tac
\{
  val r-into-trancl = @{thm trancl.r-into-trancl};
  val trancl-trans = @{thm trancl-trans};
  val rtrancl-refl = @{thm rtrancl.rtrancl-refl};
  val r-into-rtrancl = @{thm r-into-rtrancl};
  val trancl-into-rtrancl = @{thm trancl-into-rtrancl};
  val rtrancl-trancl-trancl = @{thm rtrancl-trancl-trancl};
  val trancl-rtrancl-trancl = @{thm trancl-rtrancl-trancl};
  val rtrancl-trans = @{thm rtrancl-trans};

  fun decomp Const-Trueprop for t =
    let
      fun dec Const-Trueprop [Set.member - for Const-Pair - for a b] rel =
        let
          fun decomp Const-Pair [rtrancl - for r] = (r, r^*)
          | dec Const-Pair [trancl - for r] = (r, r+)
          | dec r = (r, r);
          val (rel, r) = decomp (Envir.beta-eta-contract rel);
          in SOME (a, b, rel, r) end
        in dec t end
      | decomp - = NONE
    in decomp - = NONE;
  \);
structure Tranclp-Tac = Trancl-Tac

(
  val r-into-trancl = @{thm tranclp.r-into-trancl};
  val trancl-trans = @{thm trancl-trans};
  val rtrancl-refl = @{thm rtranclp.rtrancl-refl};
  val r-into-rtrancl = @{thm r-into-rtranclp};
  val trancl-into-rtrancl = @{thm tranclp-into-rtranclp};
  val rtrancl-trancl-trancl = @{thm rtranclp-tranclp-tranclp};
  val trancl-trans = @{thm trancl-trans};
)

fun decomp Const-Trueprop for t =
  let
    fun dec (rel $ a $ b) =
      let
        fun decr Const-rtranclp - for r = (r,r+);
        | decr Const-tranclp - for r = (r,r);
        in SOME (a, b, rel, r) end
        | dec - = NONE
      in dec t end
    | decomp - = NONE;
  in dec t end

setup (map-theory-simpset (fn ctxt => ctxt
    addSolver (mk-solver Trancl Trancl-Tac.trancl-tac)
    addSolver (mk-solver Rtrancl Trancl-Tac.rtrancl-tac)
    addSolver (mk-solver Tranclp Tranclp-Tac.trancl-tac)
    addSolver (mk-solver Rtranclp Tranclp-Tac.rtranclp-tac))

lemma transp-rtranclp [simp]: transp R**
  by(auto simp add: transp-def)

Optional methods.

method-setup trancl =
  (Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.trancl-tac))
  (simple transitivity reasoner)

method-setup rtrancl =
  (Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.rtrancl-tac))
  (simple transitivity reasoner)

method-setup tranclp =
  (Scan.succeed (SIMPLE-METHOD' o Tranclp-Tac.trancl-tac))
  (simple transitivity reasoner (predicate version))

method-setup rtranclp =

22 Well-founded Recursion

theory Wellfounded
  imports Transitive-Closure
begin

22.1 Basic Definitions

definition wf-on :: `'a set ⇒ 'a rel ⇒ bool where
  wf-on A r ←→ (∀P. (∀x ∈ A. (∀y ∈ A. (y, x) ∈ r → P y) → P x) → (∀x ∈ A. P x))

abbreviation wf :: ('a × 'a) set ⇒ bool where
  wf ≡ wf-on UNIV

definition wfp-on :: 'a set ⇒ ('a ⇒ 'a ⇒ bool) ⇒ bool where
  wfp-on A R ←→ (∀P. (∀x ∈ A. (∀y ∈ A. R y x → P y) → P x) → (∀x ∈ A. P x))

abbreviation wfP :: ('a ⇒ 'a ⇒ bool) ⇒ bool where
  wfP ≡ wfp-on UNIV

alias wfp = wfP

We keep old name wfp for backward compatibility, but offer new name wfp to be consistent with similar predicates, e.g., asymp, transp, totalp.

22.2 Equivalence of Definitions

lemma wfp-on-wf-on-eq[pred-set-conv]: wfp-on A (λx y. (x, y) ∈ r) ←→ wf-on A r
  by (simp add: wfp-on-def wf-on-def)

lemma wf-def: wf r ←→ (∀P. (∀x. (∀y. (y, x) ∈ r → P y) → P x) → (∀x. P x))
  unfolding wf-on-def by simp

lemma wfpP-def: wfp r ←→ wf {x, y}. r x y
  unfolding wf-def wfp-on-def by simp

lemma wfpP-wf-eq: wfpP (λx y. (x, y) ∈ r) = wf r
  using wfp-on-wf-on-eq .
22.3 Induction Principles

**lemma** \(\text{wf-on-induct} [\text{consumes 1, case-names in-set less, induct set: wf-on}]\):

assumes \(\text{wf-on } A \; r \; \text{and } x \in A \; \text{and} \; \forall x. \; x \in A \implies (\forall y. \; y \in A \implies (y, x) \in r \implies P y) \implies P x\)

shows \(P x\)

using assms\((2,3)\) by (auto intro: \(\text{wf-on } A \; r\) [unfolded \(\text{wf-on-def}\), rule-format])

**lemma** \(\text{wfp-on-induct} [\text{consumes 1, case-names in-set less, induct pred: wfp-on}]\):

assumes \(\text{wfp-on } A \; r \; \text{and } x \in A \; \text{and} \; \forall x. \; x \in A \implies (\forall y. \; y \in A \implies r y x \implies P y) \implies P x\)

shows \(P x\)

using assms by (fact \(\text{wf-on-induct[to-pred]}\))

**lemma** \(\text{wf-induct}\):

assumes \(\text{wf } r\)

and \(\forall x. \; \forall y. \; (y, x) \in r \implies P y \implies P x\)

shows \(P a\)

using assms by (auto intro: \(\text{wf-on-induct[of UNIV]}\))

**lemmas** \(\text{wfP-induct} = \text{wf-induct[to-pred]}\)

**lemmas** \(\text{wfP-induct-rule} = \text{wf-induct[rule-format, consumes 1, case-names less, induct set: wf]}\)

**lemmas** \(\text{wfP-induct-rule} = \text{wf-induct-rule[to-pred, induct set: wfp]}\)

**lemma** \(\text{wf-on-iff-wf}: \; \text{wf-on } A \; r \iff \text{wf } \{(x, y) \in r. \; x \in A \land y \in A\}\)

**proof** (rule iffI)

assume \(\text{wf}: \; \text{wf-on } A \; r\)

show \(\text{wf } \{(x, y) \in r. \; x \in A \land y \in A\}\)

unfolding \(\text{wf-def}\)

**proof** (intro allI impI ballI)

fix \(P x\)

assume \(IH: \forall x. \; (\forall y. \; (y, x) \in \{(x, y). \; (x, y) \in r \land x \in A \land y \in A\} \implies P y) \implies P x\)

show \(P x\)

**proof** (cases \(x \in A\))

\begin{itemize}
  \item case \(\text{True}\)
  \begin{itemize}
    \item show \(\text{?thesis}\)
    \begin{itemize}
      \item using \(\text{wf}\)
      \begin{itemize}
        \item proof (induction \(x\) rule: \(\text{wf-on-induct}\))
        \begin{itemize}
          \item case \(\text{in-set}\)
          \begin{itemize}
            \item thus \(\text{?case}\)
            \begin{itemize}
              \item using \(\text{True}\).
            \end{itemize}
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item next
    \begin{itemize}
      \item case \(\text{(less } x)\)
      \begin{itemize}
        \item thus \(\text{?case}\)
        \begin{itemize}
          \item by (auto intro: \(IH[\text{rule-format]}\))
        \end{itemize}
      \end{itemize}
    \end{itemize}
\end{itemize}

qed
next
  case False
  then show ?thesis
    by (auto intro: IH[rule-format])
qed
qed
next
assumption
unfolding
proof
fix
assume
IH:
 affliction
same
next
assume
wf
wf
proof
fix
assume
IH:
 affliction
same
next
assume
wf
wf
proof
fix
assume
IH:
 affliction
same
qed
qed
22.4  Introduction Rules

lemma wfUNIVI: \( (\forall x. (\forall y. (y, x) \in r \Rightarrow P y) \Rightarrow P x) \Rightarrow P x \) \( \Rightarrow \)
wf r
  unfolding wf-def by blast

lemmas wfPUNIVI = wfUNIVI [to-pred]

Restriction to domain A and range B. If r is well-founded over their intersection, then wf r.

lemma wfI: r \subseteq A \times B
  and \( \forall x P. [\forall y. (y, x) \in r \Rightarrow P y] \Rightarrow P x; \ x \in A; \ x \in B] \Rightarrow P x \)
  shows wf r
  using assms unfolding wf-def by blast

22.5  Ordering Properties

lemma wf-not-sym: wf r \( \Rightarrow \) (a, x) \in r \Rightarrow (x, a) \notin r
  by (induct a arbitrary: x set: wf) blast
**THEORY “Wellfounded”**

**lemma** \texttt{wf-asym:}
- \texttt{assumes} \( \text{wf } r \ (a, x) \in r \)
- \texttt{obtains} \( (x, a) \notin r \)
- \texttt{by} (drule \texttt{wf-not-sym}[OF \texttt{assms}])

**lemma** \texttt{wf-imp-asym:} \( \text{wf } r \implies \text{asym } r \)
- \texttt{by} (auto intro: asymI elim: \texttt{wf-asym})

**lemma** \texttt{wfP-imp-asym:} \( \text{wfP } r \implies \text{asymP } r \)
- \texttt{by} (rule \texttt{wf-imp-asym}[to-pred])

**lemma** \texttt{wf-not-refl [simp]:} \( \text{wf } r \implies (a, a) \notin r \)
- \texttt{by} (blast elim: \texttt{wf-asym})

**lemma** \texttt{wf-irrefl:}
- \texttt{assumes} \( \text{wf } r \)
- \texttt{obtains} \( (a, a) \notin r \)
- \texttt{by} (drule \texttt{wf-not-refl}[OF \texttt{assms}])

**lemma** \texttt{wf-imp-irrefl:}
- \texttt{assumes} \( \text{wf } r \)
- \texttt{shows} \( \text{irrefl } r \)
- \texttt{using} \texttt{wf-irrefl}[OF \texttt{assms}]
- \texttt{by} (auto simp add: \texttt{irrefl-def})

**lemma** \texttt{wfP-imp-irreflp:} \( \text{wfP } r \implies \text{irreflp } r \)
- \texttt{by} (rule \texttt{wf-imp-irrefl}[to-pred])

**lemma** \texttt{wf-wellorderI:}
- \texttt{assumes} \( \text{wf } \{ (x::\text{ord}, y). x < y \} \)
- \texttt{and} \( \text{lin: } \text{OFCLASS}(\text{ord, linorder-class}) \)
- \texttt{shows} \( \text{OFCLASS}(\text{ord, wellorder-class}) \)
- \texttt{apply} (rule \texttt{wellorder-class.intro }[OF \texttt{lin}])
- \texttt{apply} (simp add: \texttt{wellorder-class.intro class.wellorder-axioms.intro wf-induct-rule [OF \texttt{wf}])
- \texttt{done}

**lemma** \texttt{(in wellorder) \texttt{wf}:} \( \text{wf } \{ (x, y). x < y \} \)
- \texttt{unfolding} \texttt{wf-def} \texttt{by} (blast intro: less-induct)

**lemma** \texttt{(in wellorder) \texttt{wfP-less[simp]:}} \( \text{wfP } (<) \)
- \texttt{by} (simp add: \texttt{wfP-def})

**lemma** \texttt{(in wellorder) \texttt{wfp-on-less[simp]:}} \( \text{wfp-on } A (<) \)
- \texttt{unfolding} \texttt{wfp-on-def}
- \texttt{proof} (intro allI impI ballI)
- \texttt{fix} \( P \ x \)
- \texttt{assume} \( \text{hyps: } \forall x \in A. (\forall y \in A. y < x \implies P y) \implies P x \)
- \texttt{show} \( x \in A \implies P x \)
- \texttt{proof} (induction \( x \) rule: less-induct)
case (less x)
show ?case
proof (rule hyps[rule-format])
  show \( x \in A \)
    using \( x \in A \).
next
  show \( \forall y. y \in A \implies y < x \implies P y \)
    using less.IH.
qed
qed

22.6 Basic Results

Point-free characterization of well-foundedness

lemma wf-onE-pf:
  assumes \( \text{wf}: \text{wf-on } A \text{ and } B \subseteq A \text{ and } B \subseteq r \comp B \)
  shows \( B = \{\} \)
proof
  have \( x \notin B \) if \( x \in A \) for \( x \)
    using \( \text{wf} \)
proof (induction \( x \) rule: wf-on-induct)
  case in-set
  show ?case
    using \( \text{that} \).
next
  case (less \( x \))
  have \( x \notin r \comp B \)
    using less.IH \( \langle B \subseteq A \rangle \) by blast
  thus ?case
    using \( \langle B \subseteq r \comp B \rangle \) by blast
qed
with \( \langle B \subseteq A \rangle \) show \( \text{?thesis} \)
  by blast
qed

lemma wfE-pf: \( \text{wf } R \implies A \subseteq R \comp A \implies A = \{\} \)
  using \( \text{wf-onE-pf[of UNIV, simplified]} \).

lemma wf-onI-pf:
  assumes \( \forall B. B \subseteq A \implies B \subseteq R \comp B \implies B = \{\} \)
  shows \( \text{wf-on } A \text{ R} \)
unfolding \( \text{wf-on-def} \)
proof (intro allI impl ballI)
  fix \( P :: 'a \Rightarrow \text{bool} \) and \( x :: 'a \)
  let \( ?B = \{x \in A. \neg P x \} \)
  assume \( \forall x \in A. \forall y \in A. (y, x) \in R \implies P y \implies P x \)
  hence \( ?B \subseteq R \comp ?B \) by blast
  hence \( \{x \in A. \neg P x \} = \{\} \)
using assms(1)[of ?B] by simp
moreover assume x ∈ A
ultimately show P x
  by simp
qed

lemma wfl-pf: (∀A. A ⊆ R " A → A = {}) → wf R
  using wfl-onI-pf[of UNIV, simplified].

22.6.1 Minimal-element characterization of well-foundedness

lemma wfl-on-iff-ex-minimal: wfl-on A R ←→ (∀B ⊆ A. B ≠ {} → (∃z ∈ B. ∀y. (y, z) ∈ R → y /∈ B))
  proof (intro iffI allI impI)
  fix B
  assume wfl-on A R and B ⊆ A and B ≠ {} 
  show ∃z ∈ B. ∀y. (y, z) ∈ R → y /∈ B
    using wfl-onE-pf[OF ‹wfl-on A R› ‹B ⊆ A› ‹B ≠ {}›] by blast
  next
  assume ex-min: ∀B ⊆ A. B ≠ {} → (∃z ∈ B. ∀y. (y, z) ∈ R → y /∈ B)
  show wfl-on A R
  proof (rule wfl-onI-pf)
    fix B
    assume B ⊆ A and B ⊆ R " B
    have False if B ≠ {}
    using ex-min[rule-format, OF ‹B ⊆ A› ‹B ≠ {}›]
    using ‹B ⊆ R " B› by blast
    thus B = {} by blast
  qed
  qed

lemma wfl-iff-ex-minimal: wf R ←→ (∀B. B ≠ {} → (∃z ∈ B. ∀y. (y, z) ∈ R → y /∈ B))
  using wfl-on-iff-ex-minimal[of UNIV, simplified].

lemma wfp-on-iff-ex-minimal: wfp-on A R ←→ (∀B ⊆ A. B ≠ {} → (∃z ∈ B. ∀y. R y z → y /∈ B))
  using wfl-on-iff-ex-minimal[of A, to-pred] by simp

lemma wfp-iff-ex-minimal: wfp R ←→ (∀B. B ≠ {} → (∃z ∈ B. ∀y. R y z → y /∈ B))
  using wfp-on-iff-ex-minimal[of UNIV, simplified].

lemma wfE-min:
  assumes w: wf R and Q: x ∈ Q
  obtains z where z ∈ Q \{y. (y, z) ∈ R → y /∈ Q
  using Q wfE-pf[OF w, of Q] by blast
**THEORY “Wellfounded”**

577

**lemma** \(\text{wfE-min}'\):

\[
\text{wf } R \implies Q \neq \{\} \implies (\forall z. z \in Q \implies (\forall y. (y, z) \in R \implies y \notin Q) \implies \text{thesis})
\]

\(\implies \text{thesis}
\]

\text{using } \text{wfE-min[of } R - Q\text{]} \text{ by blast}

**lemma** \(\text{wfI-min}\):

\[
\text{assumes } a: (\forall x. x \in Q \implies (\exists z \in Q. \forall y. (y, z) \in R \implies y \notin Q))
\]

\(\text{shows } \text{wf } R
\]

**proof** (rule \(\text{wfI-pf}\))

\text{fix } A

\text{assume } b: A \subseteq R \text{ " } A

\text{have False if } x \in A \text{ for } x

\text{using } a[\text{OF that}] \text{ by blast}

\text{then show } A = \{\} \text{ by blast}

qed

**lemma** \(\text{wf-eq-minimal}\):

\(\text{wf } r \iff (\forall Q x. x \in Q \implies (\exists z \in Q. \forall y. (y, z) \in r \implies y \notin Q))
\]

\(\text{unfolding } \text{wf-iff-ex-minimal} \text{ by blast}

**lemmas** \(\text{wfP-eq-minimal} = \text{wf-eq-minimal \ [to-pred]}

22.6.2 Antimonotonicity

**lemma** \(\text{wf-on-antimono-strong}\):

\[
\text{assumes } \text{wf-on } B R \text{ and } A \subseteq B \text{ and } (\forall x y. x \in A \implies y \in A \implies (x, y) \in q \implies (x, y) \in r)
\]

\(\text{shows } \text{wf-on } A q
\]

\text{unfolding } \text{wf-on-iff-ex-minimal}

**proof** (intro allI impI)

\text{fix } AA \text{ assume } AA \subseteq A \text{ and } AA \neq \{\}

\text{hence } (\exists z \in AA. \forall y. (y, z) \in r \implies y \notin AA

\text{using } \langle \text{wf-on } B r \rangle \langle A \subseteq B\rangle

\text{by (simp add: } \text{wf-on-iff-ex-minimal})

\text{then show } (\exists z \in AA. \forall y. (y, z) \in q \implies y \notin AA

\text{using } \langle AA \subseteq A\rangle \text{ assms(3) by blast}

qed

**lemma** \(\text{wfp-on-antimono-strong}\):

\[
\text{wfp-on } B R \implies A \subseteq B \implies (\forall x y. x \in A \implies y \in A \implies Q x y \implies R x y) \implies \text{wfp-on } A Q
\]

\(\text{using } \text{wf-on-antimono-strong[of } B - A, \text{ to-pred]} .

**lemma** \(\text{wf-on-antimono}\):

\[
A \subseteq B \implies q \subseteq r \implies \text{wf-on } B r \leq \text{wf-on } A q
\]

\(\text{using } \text{wf-on-antimono-strong[of } B r A q\text{]} \text{ by auto}

**lemma** \(\text{wfp-on-antimono}\):

\[
A \subseteq B \implies Q \leq R \implies \text{wfp-on } B R \leq \text{wfp-on } A Q
\]

\(\text{using } \text{wfp-on-antimono-strong[of } B R A Q\text{]} \text{ by auto}

lemma wf-on-subset: $\text{wf-on } B \text{ r } \Rightarrow A \subseteq B \Rightarrow \text{wf-on } A \text{ r}$
using $\text{wf-on-antimono-strong}$.

lemma wfp-on-subset: $\text{wfp-on } B \text{ R } \Rightarrow A \subseteq B \Rightarrow \text{wfp-on } A \text{ R}$
using $\text{wfp-on-antimono-strong}$.

22.6.3 Well-foundedness of transitive closure

lemma $\text{wf-trancl}$:
assumes $\text{wf r}$
shows $\text{wf } (\text{r } +)$
proof
have $P \ x$ if induct-step: $\forall x. (\forall y. (y, x) \in r^+ \Rightarrow P y) \Rightarrow P x$ for $P x$
proof (rule induct-step)
  show $P y$ if $(y, x) \in r^+$ for $y$
  using $\langle \text{wf } r \rangle$ and that
proof (induct $x$ arbitrary: $y$)
  case (less $x$)
  note hyp = $\langle \forall x'. y'. (x', x) \in r \Rightarrow (y', x') \in r^+ \Rightarrow P y' \rangle$
  from $\langle (y, x) \in r^+ \rangle$ show $P y$
  proof cases
  case base
  show $P y$
  proof (rule induct-step)
    fix $y'$
    assume $(y', y) \in r^+$
    with $\langle (y, x) \in r \rangle$ show $P y'$
    by (rule hyp [of $y$ $y'$])
  qed
  next
  case step
  then obtain $x'$ where $(x', x) \in r$ and $(y, x') \in r^+$
  by simp
  then show $P y$ by (rule hyp [of $x'$ $y$])
  qed
  qed
  qed
  then show $\text{thesis unfolding } \text{wf-def}$ by blast
qed

lemmas wP-trancl = wf-trancl [to-pred]

lemma $\text{wf-converse-trancl}$: $\text{wf } (r^{-1}) \Rightarrow \text{wf } ((r^+)^{-1})$
apply (subst trancl-converse [symmetric])
apply (erule wf-trancl)
done

Well-foundedness of subsets

lemma $\text{wf-subset}$: $\text{wf } r \Rightarrow p \subseteq r \Rightarrow \text{wf } p$
by (simp add: wf-eq-minimal) fast

lemmas wfP-subset = wf-subset [to-pred]

Well-foundedness of the empty relation

lemma wf-empty [iff]: ![lemma](http://example.com)
by (simp add: wf-def)

lemma wfP-empty [iff]: ![lemma](http://example.com)
proof	hyave ![lemma](http://example.com)
then show ![lemma](http://example.com)
by (simp add: bot-fun-def)
qed

lemma wf-Int1: ![lemma](http://example.com)
by (erule wf-subset) (rule Int-lower1)

lemma wf-Int2: ![lemma](http://example.com)
by (erule wf-subset) (rule Int-lower2)

Exponentiation.

lemma wf-exp:
assumes ![lemma](http://example.com)
shows ![lemma](http://example.com)
proof (rule wfI-pf)
fix ![lemma](http://example.com)
assume ![lemma](http://example.com)
then have ![lemma](http://example.com)
by (induct n) force+
with ![lemma](http://example.com)
show ![lemma](http://example.com)
by (rule wfE-pf)
qed

Well-foundedness of insert.

lemma wf-insert [iff]: ![lemma](http://example.com)
proof assume ![lemma](http://example.com)
then show ![lemma](http://example.com)
by (blast elim: wf-trancl [THEN wf-irrefl]
intro: rtrancl-into-trancl1 wf-subset rtrancl-mono [THEN subsetD])

next
assume ![lemma](http://example.com)
then have ![lemma](http://example.com)
by (auto simp: wf-eq-minimal)
show ![lemma](http://example.com)
unfolding wf-eq-minimal
proof clarify
fix ![lemma](http://example.com)
assume ![lemma](http://example.com)
then obtain \( a \) where \( a \in Q \) and \( a : \bigwedge y . (y, a) \in r \rightarrow y \notin Q \)
using \( R \) by (auto simp: wf-eq-minimal)
show \( \exists z \in Q . \forall y'. (y', z) \in \text{insert} (y, x) r \rightarrow y' \notin Q \)
proof (cases \( a = x \))
  case True
  then obtain \( z \) where \( z \in Q \) and \( z , y \in r^* \)
  using \( R' \) of \( \{ z \in Q . (z,y) \in r^* \} \) by auto
  thus \( \text{thesis} \) using \( \text{inj} \) unfolding \( A \)-def
  by (rule bexI) fact
next
  case False
  then show \( \text{thesis} \) using \( a \) \( a \in Q \) by blast
qed
next
  case False
  with \( a \) \( a \in Q \) show \( \text{thesis} \)
  by blast
qed
qed

22.6.4 Well-foundedness of image

lemma \( \text{wf-map-prod-image-Dom-Ran} \):
  fixes \( r :: (\{a \times \{a\} \text{ set} \)
  and \( f :: \{a \Rightarrow \{a\} \text{ set} \)
  assumes \( \text{wf-r} \): \( \text{wf r} \)
  and \( \text{inj} \): \( \bigwedge a a'. a \in \text{Domain} r \Rightarrow a' \in \text{Range} r \Rightarrow f a = f a' \Rightarrow a = a' \)
  shows \( \text{wf} (\text{map-prod} \ f \ f ' \ r) \)
proof (unfold \( \text{wf-eq-minimal} \), clarify)
  fix \( B :: \{b \text{ set} \}
  and \( b :: \{b\} \)
  assume \( b \in B \)
  define \( A \) where \( A = \{a \} \cap \text{Domain} r \)
  show \( \exists z \in B . \forall y . (y, z) \in \text{map-prod} f f ' r \rightarrow y \notin B \)
proof (cases \( A = \{\} \))
  case False
  then obtain \( a0 \) where \( a0 \in A \) and \( \forall a . (a, a0) \in r \rightarrow a \notin A \)
  using \( \text{wfE-min}[OF \text{ wf-r}] \) by auto
  thus \( \text{thesis} \)
  using \( \text{inj} \) unfolding \( A \)-def
  by (intro bexI[of - \( f \ a 0\)]) auto
qed (use \( b \in B \) in \( \text{unfold} \ A \)-def, auto)
lemma wf-map-prod-image: \( \text{wf } r \implies \text{inj } f \implies \text{wf } (\text{map-prod } f \cdot f' \cdot r) \)
by (rule wf-map-prod-image-Dom-Ran) (auto dest: inj-onD)

lemma wfp-on-image: \( \text{wfp-on } (f \cdot A) R \iff \text{wfp-on } A (\lambda a. R (f a) (f b)) \)
proof (rule iffI)
assume hyp: \( \text{wfp-on } (f \cdot A) R \)
show \( \text{wfp-on } A (\lambda a. R (f a) (f b)) \)
  unfolding \text{wfp-on-iff-ex-minimal}
proof (intro allI impI)
  fix \( B \)
  assume \( B \subseteq A \) and \( B \neq \{\} \)
  hence \( f : B \subseteq f : A \) and \( f : B \neq \{\} \)
  unfolding \text{atomize-conj image-is-empty}
  using \text{image-uniq} by \text{iprover}
  hence \( \exists z \in f : B. \forall y. R y z \implies y \notin f : B \)
  using \text{hyp[unfolded } \text{wfp-on-iff-ex-minimal, rule-format]} by \text{iprover}
  then obtain \( f z \) where \( f z \in f : B \) and \( f z \cdot \text{max}: \forall y. R y f z \implies y \notin f : B \)
  obtain \( z \) where \( z \in B \) and \( f z = f z \)
  using \( \{fz \in f : B\} \) unfolding \text{image-iff} ..
  show \( \exists z \in B. \forall y. R (f y) (f z) \implies y \notin B \)
  proof (intro bexI allI impI)
    show \( z \in B \)
      using \( \{z \in B\} \).
    next
      fix \( y \) assume \( R (f y) (f z) \)
      hence \( f y \notin f : B \)
      using \( f z \cdot \text{max} \{fz = f z\} \) by \text{iprover}
      thus \( y \notin B \)
      by (rule \text{contrapos-nn}) (rule \text{imageI})
  qed
  qed
next
assume hyp: \( \text{wfp-on } A (\lambda a. R (f a) (f b)) \)
show \( \text{wfp-on } (f \cdot A) R \)
  unfolding \text{wfp-on-iff-ex-minimal}
proof (intro allI impI)
  fix \( fA \)
  assume \( fA \subseteq f : A \) and \( fA \neq \{\} \)
  then obtain \( A' \) where \( A' \subseteq A \) and \( A' \neq \{\} \) and \( fA = f : A' \)
  by (auto simp only: subset-image-iff)
  obtain \( z \) where \( z \in A' \) and \( z \cdot \text{max}: \forall y. R (f y) (f z) \implies y \notin A' \)
  using \text{hyp[unfolded } \text{wfp-on-iff-ex-minimal, rule-format, OF } \{A' \subseteq A\} \{A' \neq \{\}\}] by \text{blast}
show \( \exists z \in fA. \forall y. R y z \rightarrow y \notin fA \)

**proof** (intro bexI allI impI)

- show \( f z \in fA \)
  - unfolding \( fA = f' A' \)
  - using imageI[OF \( \langle z \in A' \rangle \) .

next

- show \( \forall y. R y (f z) \rightarrow y \notin fA \)
  - unfolding \( fA = f' A' \)
  - using z-max by auto

qed

qed

### 22.7 Well-Foundedness Results for Unions

**lemma** \( \text{wf-union-compatible} \):
- assumes \( \text{wf } R \text{ wf } S \)
- assumes \( R \circ S \subseteq R \)
- shows \( \text{wf } (R \cup S) \)

**proof** (rule wfl-min)

- fix \( x :: 'a \) and \( Q \)
- let \( ?Q' = \{ x \in Q. \forall y. (y, x) \in R \rightarrow y \notin Q \} \)
- assume \( x \in Q \)
- obtain \( a \) where \( a \in ?Q' \)
  - by (rule wflE-min [OF \( \text{wf } R \text{ x } x \in Q \)]) blast
- with \( \text{wf } S \) obtain \( z \) where \( z \in ?Q' \) and \( \text{zmin: } \forall y. (y, z) \in S \rightarrow y \notin ?Q' \)
  - by (erule wflE-min)
- have \( y \notin Q \) if \( (y, z) \in S \) for \( y \)

**proof**

- from that have \( y \notin ?Q' \) by (rule zmin)
- assume \( y \in Q \)
- with \( \langle y, y \rangle \notin ?Q' \) obtain \( w \) where \( (w, y) \in R \) and \( w \in Q \) by auto
- from \( \langle w, y \rangle \in R \) obtain \( (y, z) \in S \) and \( \text{have } (w, z) \in R \circ S \) by (rule relcompI)
- with \( \langle R \circ S \subseteq R \rangle \) have \( (w, z) \in R \).
- with \( (w, z) \in ?Q' \) have \( w \notin Q \) by blast
- with \( \langle w \in Q \rangle \) show \( \text{False by contradiction} \)

qed

with \( \langle z \in ?Q' \rangle \) show \( \exists z \in Q. \forall y. (y, z) \in R \cup S \rightarrow y \notin Q \) by blast

qed

Well-foundedness of indexed union with disjoint domains and ranges.

**lemma** \( \text{wf-UN} \):
- assumes \( r: \bigwedge i. \ i \in I \implies \text{wf } (r \ i) \)
  - and \( \text{disj: } \bigwedge i. j. \ [i \in I; j \in I; r i \neq r j] \implies \text{Domain } (r \ i) \cap \text{Range } (r \ j) = \{ \} \)
- shows \( \text{wf } (\bigcup i \in I. \ r \ i) \)

**proof** clarify

- fix \( A \) and \( a :: 'b \)
- assume \( a \in A \)
show \( \exists z \in A. \forall y. (y, z) \in \bigcup (r \cdot i) \rightarrow y \notin A \)

proof (cases \( \exists i \in I. \exists a \in A. \exists b \in A. (b, a) \in r \cdot i \))
  case True
  then obtain \( i b c \) where \( ibc: i \in I \) \( b \in A \) \( c \in A \)
  have ri: \( i \cdot Q. Q \neq \{\} \implies \exists z \in Q. \forall y. (y, z) \in r \cdot i \rightarrow y \notin Q \)
  using r [OF \( i \in I \)] unfolding wf-eq-minimal by auto
  show ?thesis using ri [of \( \{ a \cdot a \in A \land (\exists b \in A. (b, a) \in r \cdot i) \} \) ibc disj
  by blast
  qed

next
  case False
  with \( \{ a \in A \cdot \} \) show ?thesis
  by blast
  qed

lemma \( \text{wfP-SUP} \): \( \forall i. \text{wfP} (r \cdot i) \implies \forall i j. r \cdot i \neq r \cdot j \rightarrow \inf (\text{Domainp} (r \cdot i)) (\text{Rangep} (r \cdot j)) = \text{bot} \)
  \( \implies \)
  \( \text{wfP} (\bigcup (\text{range } r)) \)
  by (rule wf-UN[to-pred]) simp-all

lemma \( \text{wf-Union} \):
  assumes \( \forall r \in R. \text{wf } r \)
  and \( \forall r \in R. \forall s \in R. r \neq s \rightarrow \text{Domain } r \cap \text{Range } s = \{\} \)
  shows \( \text{wf } (\bigcup R) \)
  using \( \text{assms } \text{wf-UN[of } R \land i] \) by simp

Intuition: We find an \( R \cup S \)-min element of a nonempty subset \( A \) by case distinction.

1. There is a step \( a \rightarrow_{R} b \) with \( a, b \in A \). Pick an \( R \)-min element \( z \) of the (nonempty) set \( \{a \in A \mid \exists b \in A. a \rightarrow_{R} b\} \). By definition, there is \( z' \in A \) s.t. \( z \rightarrow_{R} z' \). Because \( z \) is \( R \)-min in the subset, \( z' \) must be \( R \)-min in \( A \). Because \( z' \) has an \( R \)-predecessor, it cannot have an \( S \)-successor and is thus \( S \)-min in \( A \) as well.

2. There is no such step. Pick an \( S \)-min element of \( A \). In this case it must be an \( R \)-min element of \( A \) as well.

lemma \( \text{wf-Un} \): \( \text{wf } r \implies \text{wf } s \implies \text{Domain } r \cap \text{Range } s = \{\} \implies \text{wf } (r \cup s) \)
  using \( \text{wf-union-compatible[of } s \cdot r] \)
  by (auto simp: Un-ac)

lemma \( \text{wf-union-merge} \): \( \text{wf } (R \cup S) = \text{wf } (R \circ R \cup S \circ R \cup S) \)
  (is \( \text{wf } ?A = \text{wf } ?B) \)
  proof
assume \( wf \ ?A \)
with \( wf\text{-}trancl \) have \( wf\text{-}T: \ wf \ (\ ?A^+ ) \).
moreover have \( ?B \subseteq ?A^+ \)
by \( \text{subst trancl-unfold, subst trancl-unfold} \) blast
ultimately show \( wf \ ?B \) by \( \text{(rule wf-subset)} \)
next
assume \( wf \ ?B \)
show \( wf \ ?A \)
proof \( \text{(rule wfI-min)} \)
fix \( Q :: \ 'a \ set \) and \( x \)
assume \( x \in Q \)
with \( \langle \text{wf } ?B, \ obtain \ z \ \rangle \ where \ z \in Q \ and \ \land y . \ (y, z) \in ?B \Longrightarrow y \notin Q \)
by \( \text{(erule wfE-min)} \)
then have \( 1: \land y . \ (y, z) \in R \circ R \Longrightarrow y \notin Q \)
and \( 2: \land y . \ (y, z) \in S \circ R \Longrightarrow y \notin Q \)
and \( 3: \land y . \ (y, z) \in S \Longrightarrow y \notin Q \)
by \( \text{auto} \)
show \( \exists z \in Q . \ \forall y . \ (y, z) \in ?A \Longrightarrow y \notin Q \)
proof \( \text{(cases } \forall y . \ (y, z) \in R \Longrightarrow y \notin Q \) \)
case \( True \)
with \( \langle z \in Q, \ 3 \ \rangle \) show \( \langle \text{thesis } \rangle \) by \( \text{blast} \)
next
case \( False \)
then obtain \( z' \) where \( z' \in Q \ (z', z) \in R \) by \( \text{blast} \)
have \( \forall y . \ (y, z') \in ?A \Longrightarrow y \notin Q \)
proof \( \text{(intro allI impI)} \)
fix \( y \) assume \( (y, z') \in ?A \)
then show \( y \notin Q \)
proof
assume \( (y, z') \in R \)
then have \( (y, z) \in R \circ R \) using \( \langle z', z \rangle \in R \) \).
with \( 1 \) show \( y \notin Q \) .
next
assume \( (y, z') \in S \)
then have \( (y, z) \in S \circ R \) using \( \langle z', z \rangle \in R \) \).
with \( 2 \) show \( y \notin Q \) .
qed
qed
with \( \langle z' \in Q, \ 3 \ \rangle \) show \( \langle \text{thesis } \rangle \) ..
qed
qed
qed

lemma \( \text{wf-comp-self: } wf R \iff wf (R \circ R) \) — special case
by \( \text{(rule wf-union-merge [where } S = \{} , \ simplified]} \) \)

22.8 Well-Foundedness of Composition
Bachmair and Dershowitz 1986, Lemma 2. [Provided by Tjark Weber]
lemma qc-wf-relto-iff:
assumes $R O S \subseteq (R \cup S)^* O R$ — R quasi-commutes over S
shows $wf (S^* O R O S^*) \iff wf R$
(is $wf ?S \iff -$)
proof
  show $wf R$ if $wf ?S$
    proof
      have $R \subseteq ?S$ by auto
      with $wf$-subset [of $?S$] that show $wf R$
        by auto
    qed
next
  show $wf ?S$ if $wf R$
    proof (rule wfI-pf)
      fix $A$
      assume $A \subseteq ?S$ " $A$
      let $?X = (R \cup S)^* O R$
      have $*: R O (R \cup S)^* \subseteq (R \cup S)^* O R$
        proof
          have $(x, z) \in (R \cup S)^* O R$ if $(y, z) \in (R \cup S)^* O R$ for $x y z$
            using that
          proof (induct $y z$)
            case rtrancl-refl
            then show $?case$ by auto
          next
            case (rtrancl-into-rtrancl $a b c$)
            then have $(x, c) \in ((R \cup S)^* O (R \cup S)^*) O R$
              using $assms$ by blast
            then show $?case$ by simp
          qed
          then show $?thesis$ by auto
        qed
    qed
  qed
next
  have $R O S^* \subseteq (R \cup S)^* O R$
    using rtrancl-Un-subset by blast
  then have $?S \subseteq (R \cup S)^* O (R \cup S)^* O R$
    by (simp add: relcomp-mono rtrancl-mono)
  also have \ldots $= (R \cup S)^* O R$
    by (simp add: O-assoc[symmetric])
  finally have $?S O (R \cup S)^* \subseteq (R \cup S)^* O R O (R \cup S)^*$
    by (simp add: O-assoc[symmetric] relcomp-mono)
  also have \ldots $\subseteq (R \cup S)^* O (R \cup S)^* O R$
    using * by (simp add: relcomp-mono)
  finally have $?S O (R \cup S)^* \subseteq (R \cup S)^* O R$
    by (simp add: O-assoc[symmetric])
  then have $(?S O (R \cup S)^*)^* A \subseteq ((R \cup S)^* O R)^* A$
    by (simp add: Image-mono)
  moreover have $?X \subseteq (?S O (R \cup S)^*)^* A$
    using $A$ by (auto simp: relcomp-Image)
  ultimately have $?X \subseteq R$ " $?X
by (auto simp: relcomp-Image)
then have \( ?X = \{ \} \)
  using \( \text{wf } R \) by (simp add: wfE-pf)
moreover have \( A \subseteq ?X \) by auto
ultimately show \( A = \{ \} \) by simp
qed

**corollary** \( \text{wf-relcomp-compatible} : \)
assumes \( \text{wf } R \) and \( R \circ S \subseteq S \circ R \)
sows \( \text{wf } (S \circ R) \)
proof –
  have \( R \circ S \subseteq (R \cup S)^* \circ R \)
    using assms by blast
then have \( \text{wf } (S^* \circ R \circ O S^*) \)
    by (simp add: assms qc-wf-relto-iff)
then show \( ?\text{thesis} \)
    by (rule Wellfounded.wf-subset) blast
qed

22.9 Acyclic relations

**lemma** \( \text{wf-acyclic} : \) \( \text{wf } r \implies \text{acyclic } r \)
by (simp add: acyclic-def) (blast elim: wf-trancl [THEN wf-irrefl])

**lemmas** \( \text{wfP-acyclicP } = \text{wf-acyclic } [\text{to-pred}] \)

22.9.1 Wellfoundedness of finite acyclic relations

**lemma** \( \text{finite-acyclic-wf} : \)
assumes \( \text{finite } r \) \( \text{acyclic } r \)
sows \( \text{wf } r \)
using assms
proof (induction \( r \) rule: finite-induct)
case (insert \( x \) \( r \) )
then show \( ?\text{case} \)
  by (cases \( x \)) simp
qed simp

**lemma** \( \text{finite-acyclic-wf-converse} : \) \( \text{finite } r \implies \text{acyclic } r \implies \text{wf } (r^{-1}) \)
apply (erule finite-converse [THEN iffD2, THEN finite-acyclic-wf])
apply (erule acyclic-converse [THEN iffD2])
done

Observe that the converse of an irreflexive, transitive, and finite relation is again well-founded. Thus, we may employ it for well-founded induction.

**lemma** \( \text{wf-converse} : \)
assumes \( \text{irrefl } r \) \( \text{and } \text{trans } r \) \( \text{and } \text{finite } r \)
sows \( \text{wf } (r^{-1}) \)
proof –
THEORY “Wellfounded”

have acyclic r
using (irrefl r) and (trans r)
by (simp add: irrefl-def acyclic-irrefl)
with (finite r) show ?thesis
by (rule finite-acyclic-wf-converse)
qed

lemma wf-iff-acyclic-if-finite: finite r \implies wf r = acyclic r
by (blast intro: finite-acyclic-wf wf-acyclic)

22.10 \textbf{n}at is well-founded

lemma less-nat-rel: (<) = (\lambda m. n = Suc m)^++
proof (rule ext, rule ext, rule iffI)
fix n m :: nat
show (\lambda m. n = Suc m)^++ m n if m < n
using that
proof (induct n)
case 0
then show ?case by auto
next
case (Suc n)
then show ?case
by (auto simp add: less-Suc-eq-le reflexive le-less intro: tranclp.trancl-into-trancl)
qed
show m < n if (\lambda m. n = Suc m)^++ m n
using that by (induct n) (simp-all add: less-Suc-eq-le reflexive le-less)
qed

definition pred-nat :: (nat \times nat) set
where pred-nat = \{(m, n). n = Suc m\}

definition less-than :: (nat \times nat) set
where less-than = pred-nat^*

lemma less-eq: (m, n) \in pred-nat^* \iff m < n
unfolding less-nat-rel pred-nat-def trancl-def by simp

lemma pred-nat-trancl-eq-le: (m, n) \in pred-nat^* \iff m \leq n
unfolding less-eq rtrancl-eq-or-trancl by auto

lemma wf-pred-nat: wf pred-nat
unfolding wf-def
proof clarify
fix P x
assume \forall x'. (\forall y. (y, x') \in pred-nat \implies P y) \implies P x'
then show P x
unfolding pred-nat-def by (induction x) blast+
qed
lemma wf-less-than [iff]: \( \text{wf} \) less-than
  by (simp add: less-than-def wf-pred-nat [THEN wf-trancl])

lemma trans-less-than [iff]: trans less-than
  by (simp add: less-than-def)

lemma less-than-iff [iff]: \((x, y) \in \text{less-than}\) = \((x < y)\)
  by (simp add: less-than-def less-eq)

lemma irrefl-less-than: irrefl less-than
  using irrefl-def by blast

lemma asym-less-than: asym less-than
  by (rule asymI simp)

lemma total-less-than: total less-than and total-on-less-than
  simp: total-on A less-than
  using total-on-def by force+

lemma wf-less: \(\text{wf} \) \(\{(x, y::\text{nat}). x < y\}\)
  by (rule Wellfounded.wellorder-class.wf)

## 22.11 Accessible Part

Inductive definition of the accessible part \(\text{acc} \ r\) of a relation; see also [6].

inductive-set acc :: ('a × 'a) set ⇒ 'a set for r :: ('a × 'a) set
  where accI: \((\forall y. (y, x) \in r \Rightarrow y \in \text{acc} \ r) \Rightarrow x \in \text{acc} \ r\)

abbreviation termip :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool
  where termip r ≡ accp \((r^{-1})\)

abbreviation termi :: ('a × 'a) set ⇒ 'a set
  where termi r ≡ acc \((r^{-1})\)

lemmas accpI = accp.accI

lemma accp-eq-acc [code]: accp \(r = (\lambda x. x \in \text{Wellfounded.acc} \{(x, y). r \ x y\})\)
  by (simp add: acc-def)

Induction rules

theorem accp-induct:
  assumes major: accp \(r a\)
  assumes hyp: \(\forall x. \text{accp} \ r x \Rightarrow \forall y. r \ y x \Rightarrow P \ y \Rightarrow P x\)
  shows \(P a\)
  apply (rule major [THEN accp.induct])
  apply (rule hyp)
  apply (rule accp.accI)
  apply auto
done

lemmas accp-induct-rule = accp-induct [rule-format, induct set: accp]

theorem accp-downward: accp r b → r a b → accp r a
  by (cases rule: accp.cases)

lemma not-accp-down:
  assumes na: ¬ accp R x
  obtains z where R z x and ¬ accp R z
proof −
  assume a: \( \forall z. R z x \implies \neg \text{accp} R z \implies \text{thesis} \)
  show thesis
    proof (cases \( \forall z . R z x \rightarrow \text{accp} R z \))
      case True
      then have \( \forall z . R z x \implies \text{accp} R z \) by auto
      then have accp R x by (rule accp.accI)
      with na show thesis ..
    next
    case False then obtain z where R z x and \( \neg \text{accp} R z \)
      by auto
      with a show thesis .
    qed
  qed

lemma accp-downwards-aux: \( r \ast\ast b a \rightarrow \text{accp} r a \rightarrow \text{accp} r b \)
  by (erule rtranclp-induct) (blast dest: accp-downward)+

theorem accp-downwards: accp r a \rightarrow r \ast\ast b a \rightarrow accp r b
  by (blast dest: accp-downwards-aux)

theorem accp wfPI: \( \forall x . \text{accp} r x \rightarrow \text{wfP} r \)
proof (rule wfPUNIVI)
  fix P x
  assume \( \forall x . \text{accp} r x \forall x . (\forall y . r y x \rightarrow P y) \rightarrow P x \)
  then show P x
    using accp-induct[where P = P] by blast
  qed

theorem accp wfPD: \( \text{wfP} r \rightarrow \text{accp} r x \)
  apply (erule wfP.induct-rule)
  apply (rule accp.accI)
  apply blast
  done

theorem wfP accp iff: \( \text{wfP} r = (\forall x . \text{accp} r x) \)
  by (blast intro: accp wfPI dest: accp wfPD)

Smaller relations have bigger accessible parts:
lemma accp-subset:
assumes $R_1 \leq R_2$
shows $accp R_2 \leq accp R_1$
proof (rule predicate1I)
  fix $x$
  assume $accp R_2 x$
  then show $accp R_1 x$
proof (induct $x$)
  fix $x$
  assume $\forall y. R_2 y x \implies accp R_1 y$
  with assms show $accp R_1 x$
  by (blast intro: accp.accI)
qed
qed

This is a generalized induction theorem that works on subsets of the accessible part.

lemma accp-subset-induct:
assumes subset: $D \leq accp R$
and del: $\forall x z. D x \Longrightarrow R z x \Longrightarrow D z$
and $D x$
and istep: $\forall x. D x \implies (\forall z. R z x \Longrightarrow P z) \Longrightarrow P x$
shows $P x$
proof
  from subset and $\langle D x \rangle$
  have $accp R x$ ..
  then show $P x$ using $\langle D x \rangle$
proof (induct $x$)
  fix $x$
  assume $D x$ and $\forall y. R y x \Longrightarrow D y \Longrightarrow P y$
  with del and istep show $P x$ by blast
qed
qed

Set versions of the above theorems

lemmas acc-induct = accp-induct [to-set]
lemmas acc-induct-rule = acc-induct [rule-format, induct set: acc]
lemmas acc-downward = accp-downward [to-set]
lemmas not-acc-down = not-accp-down [to-set]
lemmas acc-downwards-aux = accp-downwards-aux [to-set]
lemmas acc-downwards = accp-downwards [to-set]
lemmas acc-wfI = accp-wfPI [to-set]
lemmas acc-wfD = accp-wfPD [to-set]
lemmas wf-acc-iff = wfP-accp-iff [to-set]
lemmas acc-subset = accp-subset [to-set]
lemmas acc-subset-induct = accp-subset-induct [to-set]
22.12 Tools for building wellfounded relations

Inverse Image

**lemma** `wf-inv-image [simp,intro]`:
- *fixes* `f :: 'a ⇒ 'b`
- *assumes* `wf r`
- *shows* `wf (inv-image r f)`

**proof** –
- *have* \( \forall x. x \in P \implies \exists y. (f y, f z) \in r \rightarrow y \notin P \)

**lemma** `wfp-on-inv-imagep`:
- *assumes* `wf wfp-on (f ' A) R`
- *shows* `wfp-on A (inv-imagep R f)`

**proof** `intro allI impI`
- *fix* `B` *assume* `B ⊆ A` *and* `B ≠ {}`
- *hence* `∃ z ∈ f ' B. ∀ y. R y z → y /∈ f ' B`
- *using* `wf [unfolded wfp-on-inv-imagep, rule_format, of f ' B]` *by* `blast`
- *thus* `∃ z ∈ B. ∀ y. inv-imagep R f y z → y /∈ B`
- *unfolding* `inv-imagep-def`
- *by* `auto`

**lemma** `wfp-on-if-convertible-to-wfp-on`:
- *assumes* `wf wfp-on (f ' A) Q` *and* `convertible: (\( \forall x y. x \in A \implies y \in A \implies R x y \implies f x \ (f y) \))` *shows* `wfp-on A R`

**proof** `intro allI impI`
- *fix* `B` *assume* `B ⊆ A` *and* `B ≠ {}`
- *moreover from* `wf` *have* `wfp-on A (inv-imagep Q f)` *by* `rule wfp-on-inv-imagep`
- *ultimately obtain* `y` *where* `y ∈ B` *and* `\( \exists z. Q (f z) \ (f y) \implies z /∈ B \)`
unfolding \textit{wfp-on-if-ex-minimal} in \textit{inv-imagep} by blast
thus \( \exists z \in B. \forall y. R \, y \, z \longrightarrow y \notin B \)
using \( \langle B \subseteq A \rangle \) convertible by blast
\textit{qed}

\textbf{lemma} \textit{wfp-on-if-converible-to-wf}: \( \textit{wfp-on} (f \, \langle A \rangle ) \, Q \Longrightarrow (\forall x. x \in A \Longrightarrow y \in A \Longrightarrow (x, y) \in R \Longrightarrow (f \, x, f \, y) \in Q) \Longrightarrow \textit{wfp-on} A \, R \)
using \textit{wfp-on-if-converible-to-wfp-on[\textit{to-set}]}.

\textbf{lemma} \textit{wfp-if-converible-to-wf}:
\begin{itemize}
  \item \textit{fixes} \( r :: 'a \, \textit{rel} \) and \( s :: 'b \, \textit{rel} \) and \( f :: 'a \Rightarrow 'b \)
  \item \textit{assumes} \( \textit{wfs} \) and \textit{convertible}: \( \forall x, y. (x, y) \in r \Longrightarrow (f \, x, f \, y) \in s \)
  \item \textit{shows} \( \textit{wfr} \)
\end{itemize}
\textit{proof} \( \textit{(rule\ wfp-on-if-converible-to-wf-on)} \)
\begin{itemize}
  \item \textit{show} \( \textit{wfp-on} \, (\textit{range} \, f) \, s \)
    using \textit{wfp-on-subset[\textit{OF} \langle \textit{wfs} \rangle \, \textit{subset-UNIV}]}.
\end{itemize}
\textit{next}
\begin{itemize}
  \item \textit{show} \( \forall x, y. (x, y) \in r \Longrightarrow (f \, x, f \, y) \in s \)
    using \textit{convertible}.
\end{itemize}
\textit{qed}

\textbf{lemma} \textit{wfpP-if-converible-to-wfP}:
\begin{itemize}
  \item \textit{fixes} \( f :: 'a \Rightarrow \textit{nat} \)
  \item \textit{shows} \( \forall x, y. R \, x \, y \Longrightarrow f \, x < f \, y \Longrightarrow \textit{wfpP} \, R \)
\end{itemize}
using \textit{wfp-if-converible-to-wfP[\textit{of} \langle \textit{<} \rangle :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{bool}, \textit{simplified}]}.
\textit{Converting to \textit{nat} is a very common special case that might be found more easily by Sledgehammer.}

\textbf{lemma} \textit{wfpP-if-converible-to-nat}:
\begin{itemize}
  \item \textit{fixes} \( f :: 'a \Rightarrow \textit{nat} \)
  \item \textit{shows} \( \forall x, y. R \, x \, y \Longrightarrow f \, x < f \, y \Longrightarrow \textit{wfpP} \, R \)
\end{itemize}
by \( \textit{(rule\ wfpP-if-converible-to-wfpP[\textit{of} \langle \textit{<} \rangle :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{bool}, \textit{simplified}]}\)

22.12.2 Measure functions into \textit{nat}

\textbf{definition} \textit{measure} :: \( 'a \Rightarrow \textit{nat} \) \Rightarrow \( 'a \times 'a \) \textit{set}
\begin{itemize}
  \item where \textit{measure} = \textit{inv-image} \textit{less-than}
\end{itemize}
\textit{lemma} \textit{in-measure[\textit{simpl, code-unfold}]}: \( (x, y) \in \textit{measure} \, f \longleftrightarrow f \, x < f \, y \)
by \( \textit{(simp add:measure-def)} \)

\textbf{lemma} \textit{wf-measure [iff]}: \( \textit{wf} \, (\textit{measure} \, f) \)
\textit{unfolding} \textit{measure-def} by \( \textit{(rule\ wf-less-than[\textit{THEN} \textit{wf-inv-image}]}\)
\textbf{lemma} \textit{wf-if-measure}: \( \forall x. P \, x \Longrightarrow f(g \, x) < f \, y \Longrightarrow \textit{wf} \, \{ (y, x). P \, x \land y = g \, x \} \)
for \( f :: 'a \Rightarrow \textit{nat} \)
\textbf{using} \textit{wf-measure[\textit{of} \, f]} \textit{unfolding} \textit{measure-def} \textit{inv-image-def} \textit{less-than-def} \textit{less-eq}
by \( \textit{(rule\ wf-subset) auto} \)
22.12.3 Lexicographic combinations

**definition** lex-prod :: 
(′a × a) set ⇒ (′b × b) set ⇒ ((′a × ′b) × (′a × ′b)) set

where ra <∗ lex∗> rb = {((a, b), (a′, b′)). (a, a′) ∈ ra ∨ a = a′ ∧ (b, b′) ∈ rb}

**lemma** in-lex-prod**: ((a, b), (a′, b′)) ∈ r <∗ lex∗> s ←→ (a, a′) ∈ r ∨ a = a′ ∧ (b, b′) ∈ s

by (auto simp: lex-prod-def)

**lemma** wf-lex-prod**: assumes wf ra wf rb
shows wf (ra <∗ lex∗> rb)

proof (rule wfI)

fix z :: ′a × ′b and P

assume * [rule-format]: ∀ u. (∀ v. (v, u) ∈ ra <∗ lex∗> rb → P v) → P u

obtain x y where zeq: z = (x,y)

by fastforce

have P(x,y) using ⟨wf ra⟩

proof (induction x arbitrary: y rule: wf-induct-rule)

case (less x)

note lessx = less

show ?case using ⟨wf rb⟩ less

proof (induction y rule: wf-induct-rule)

case (less y)

show ?case

by (force intro: * less.IH lessx)

qed

qed

then show P z

by (simp add: zeq)

qed auto

**lemma** refl-lex-prod**: refl r B ⇒ refl (r A <∗ lex∗> r B)

by (auto intro!: reflI dest: refl-onD)

**lemma** irrefl-on-lex-prod**: irrefl-on A r A ⇒ irrefl-on B r B ⇒ irrefl-on (A × B) (r A <∗ lex∗> r B)

by (auto intro!: irrefl-onI dest: irrefl-onD)

**lemma** irrefl-lex-prod**: irrefl r A ⇒ irrefl r B ⇒ irrefl (r A <∗ lex∗> r B)

by (rule irrefl-on-lex-prod[of UNIV - UNIV, unfolded UNIV-Times-UNIV])

**lemma** sym-on-lex-prod**: sym-on A r A ⇒ sym-on B r B ⇒ sym-on (A × B) (r A <∗ lex∗> r B)

by (auto intro!: sym-onI dest: sym-onD)

**lemma** sym-lex-prod**: sym r A ⇒ sym r B ⇒ sym (r A <∗ lex∗> r B)

by (rule sym-on-lex-prod[of UNIV - UNIV, unfolded UNIV-Times-UNIV])
THEORY “Wellfounded” 594

lemma asym-on-lex-prod[simp]:
  asym-on A r_A \implies asym-on B r_B \implies asym-on (A \times B) (r_A \preclex r_B)
by (auto intro!: asym-onD dest: asym-onD)

lemma asym-lex-prod[simp]:
  asym r_A \implies asym r_B \implies asym (r_A \preclex r_B)
by (rule asym-lex-prod[of UNIV - UNIV, unfolded UNIV-Times-UNIV])

lemma trans-on-lex-prod[simp]:
  assumes trans-on A r_A and trans-on B r_B
  shows trans-on (A \times B) (r_A \preclex r_B)
proof (rule trans-onI)
  fix x y z
  show (x, z) \in r_A \preclex r_B \implies (y, z) \in r_A \preclex r_B \implies (x, z) \in r_A \preclex r_B
  using trans-onD[of trans-on A r_A, of fst x fst y fst z]
  using trans-onD[of trans-on B r_B, of snd x snd y snd z]
by auto
qed

lemma trans-lex-prod [simp,intro!]: trans r_A \implies trans r_B \implies trans (r_A \preclex r_B)
by (rule trans-on-lex-prod[of UNIV - UNIV, unfolded UNIV-Times-UNIV])

lemma total-on-lex-prod[simp]:
  total-on A r_A \implies total-on B r_B \implies total-on (A \times B) (r_A \preclex r_B)
by (auto simp: total-on-def)

lemma total-lex-prod[simp]: total r_A \implies total r_B \implies total (r_A \preclex r_B)
by (rule total-on-lex-prod[of UNIV - UNIV, unfolded UNIV-Times-UNIV])

lexicographic combinations with measure functions

definition mlex-prod :: ('a ⇒ nat) ⇒ ('a × 'a) set ⇒ ('a × 'a) set (infixr \preclex)
  where f \preclex R = inv-image (less-than \preclex R) (λx. (f x, x))

lemma
  wf-mlex: wf R \implies wf (f \preclex R)
and
  mlex-les: f x < f y \implies (x, y) \in f \preclex R
and
  mlex-leq: f x \leq f y \implies (x, y) \in R \implies (x, y) \in f \preclex R
and
  mlex-iff: (x, y) \in f \preclex R \iff f x < f y \lor f x = f y \land (x, y) \in R
by (auto simp: mlex-prod-def)

Proper subset relation on finite sets.

definition finite-psubset :: ('a set × 'a set) set
  where finite-psubset = {(A, B). A \subset B ∧ finite B}
lemma wf-finite-psubset[simp]: wf finite-psubset
  apply (unfold finite-psubset-def)
  apply (rule wf-measure [THEN wf-subset])
  apply (simp add: measure-def inv-image-def less-than-def less-eq)
  apply (fast elim!: psubset-card-mono)
  done

lemma trans-finite-psubset: trans finite-psubset
  by (auto simp: finite-psubset-def less-le trans-def)

lemma in-finite-psubset[simp]: (A, B) ∈ finite-psubset ←→ A ⊂ B ∧ finite B
  unfolding finite-psubset-def by auto

max- and min-extension of order to finite sets

inductive-set max-ext :: ('a × 'a) set ⇒ ('a × 'a) set
for R :: ('a × 'a) set
where
  max-extI[intro]: finite X =⇒ finite Y =⇒ Y ∉ {} =⇒ (∀x. x ∈ X =⇒ ∃y ∈ Y. (x, y) ∈ R) =⇒ (X, Y) ∈ max-ext R

lemma max-ext-wf:
  assumes wf: wf r
  shows wf (max-ext r)
proof (rule acc-wfI, intro allI)
  show M ∈ acc (max-ext r) (is ∈ ?W) for M
  proof (induct M rule: infinite-finite-induct)
    case empty
    show ?case by (rule accI) (auto elim!: max-ext.cases)
  next
case (insert a M)
from wf ‹M ∈ ?W› ‹finite M› show insert a M ∈ ?W
proof (induct arbitrary: M)
  fix M a
  assume M ∈ ?W
  assume [intro]: finite M
  assume hyp: ∀b M. (b, a) ∈ r =⇒ M ∈ ?W =⇒ finite M =⇒ insert b M ∈ ?W
  have add-less: M ∈ ?W =⇒ (∀y. y ∈ N =⇒ (y, a) ∈ r) =⇒ N ∪ M ∈ ?W
  if finite N finite M for N M :: 'a set
  using that by (induct N arbitrary: M) (auto simp: hyp)
  show insert a M ∈ ?W
  proof (rule accI)
    fix N
    assume Nless: (N, insert a M) ∈ max-ext r
    then have *: ∀x. x ∈ N =⇒ (x, a) ∈ r ∨ (∃y ∈ M. (x, y) ∈ r)
    by (auto elim!: max-ext.cases)
    let ?N1 = {n ∈ N. (n, a) ∈ r}
let \(?N2 = \{ n \in N. (n, a) \notin r \}\)

have \(?N1 \cup \?N2 = N\) by (rule set-eqI) auto

from Nless have finite N by (auto elim: max-ext.cases)

then have finites: finite \(?N1\) finite \(?N2\) by auto

have \(?N2 \in \?W\)
proof (cases \(M = \{\}\))
  case [simp]: True
    have \(Mw: \{\} \in \?W\) by (rule accI) (auto elim: max-ext.cases)
    from \(\ast\) have \(?N2 = \{\}\) by auto
    with \(Mw\) show \(?N2 \in \?W\) by (simp only:)
  next
    case False
    from \(\ast\) finites have \(?N1 \cup \?N2 \in \?W\)
      by (rule add-less simp)
    then show \(N \in \?W\) by (simp only: N)
  qed
  qed
next
  case infinite
  show \(?case\)
    by (rule accI) (auto elim: max-ext.cases simp: infinite)
  qed
  qed

lemma max-ext-additive: \((A, B) \in \text{max-ext } R \Rightarrow \(C, D) \in \text{max-ext } R \Rightarrow \(A \cup C, B \cup D) \in \text{max-ext } R\)
by (force elim!: max-ext.cases)

definition min-ext ::= \(\{a \times \{\}\\) set \(\Rightarrow \(\{\} \times a \text{ set}\) set\)
where min-ext r = \\{\{(X, Y) \mid X Y. X \neq \{\}\ \& \ (\forall y \in Y. (\exists x \in X. (x, y) \in r))\}\}\)

lemma min-ext-wf:
  assumes \(wfr\)
  shows \(\text{wf } min-ext\)
proof (rule wfl-min)
  show \(\exists m \in Q. (\forall n. (n, m) \in \text{min-ext } r \Rightarrow n \notin Q)\) if nonempty: \(x \in Q\)
    for \(Q ::= \{\}\) set and \(x\)
  proof (cases \(Q = \{\}\))
    case True
    then show \(?thesis\) by (simp add: min-ext-def)
  next
    case False
    with nonempty obtain \(x x\) where \(x \in Q e \in x\) by force
    then have \(e U. e \in \bigcup Q\) by auto
with \( \text{wf } r \)

obtain \( z \) where \( z \in \bigcup Q \land y. (y, z) \in r \Rightarrow y \notin \bigcup Q \)

by (erule wfE-min)

from \( z \) obtain \( m \) where \( m \in Q \) \( z \in m \) by auto

proof (intro rev-bexI allI impI)

fix \( n \)

assume smaller: \( (n, m) \in \text{min-ext } r \)

with \( \langle z \in m \rangle \) obtain \( y \) where \( y \in n \) \( (y, z) \in r \)

by (auto simp: min-ext-def)

with \( z \) (2) show \( n \notin Q \) by auto

qed

qed

22.12.4 Bounded increase must terminate

lemma \( \text{wf-bounded-measure} \):

fixes \( \text{ub} :: 'a \Rightarrow \text{nat} \)

and \( f :: 'a \Rightarrow \text{nat} \)

assumes \( \forall a b. (b, a) \in r \Rightarrow \text{ub } b \leq \text{ub } a \land \text{ub } a \geq f b \land f b > f a \)

shows \( \text{wf } r \)

by (rule wf-subset[OF \( \text{wf-measure[of } \lambda a. \text{ub } a - f a\] \))) (auto dest: assms)

lemma \( \text{wf-bounded-set} \):

fixes \( \text{ub} :: 'a \Rightarrow 'b \text{ set} \)

and \( f :: 'a \Rightarrow 'b \text{ set} \)

assumes \( \forall a b. (b, a) \in r \Rightarrow \text{finite } (\text{ub } a) \land \text{ub } b \subseteq \text{ub } a \land \text{ub } a \geq f b \land f b > f a \)

shows \( \text{wf } r \)

apply (rule wf-bounded-measure[of \( r \) \( \lambda a. \text{card } (\text{ub } a) \) \( \lambda a. \text{card } (f a)\] \))

apply (drule assms)

apply (blast intro: card-mono finite-subset psubset-card-mono dest: psubset-eq[THEN iffD2])

done

lemma \( \text{finite-subset-wf} \):

assumes \( \text{finite } A \)

shows \( \text{wf } \{ (X, Y). X \subseteq Y \land Y \subseteq A \} \)

by (rule wf-subset[OF \( \text{finite-psubset[unfolded finite-psubset-def]}\] \))

(auto intro: finite-subset[OF - assms])

hide-const (open) \( \text{acc accp} \)

22.13 Code Generation Setup

Code equations with \( \text{wf} \) or \( \text{wfp} \) on the left-hand side are not supported by the code generation module because of the \text{UNIV} hidden behind the abbreviations. To sidestep this problem, we provide the following wrapper
definitions and use code-abbrev to register the definitions with the pre- and
post-processors of the code generator.

definition wf-code :: (('a × 'a) set ⇒ bool) where
[code-abbrev]: wf-code r ←→ wf r

definition wfp-code :: ('a ⇒ 'a ⇒ bool) ⇒ bool where
[code-abbrev]: wfp-code R ←→ wfp R
end

23 Well-Founded Recursion Combinator

theory Wfrec
  imports Wellfounded
begin

inductive wfrec-rel :: ('a × 'a) set ⇒ ((('a ⇒ 'b) ⇒ ('a ⇒ 'b)) ⇒ 'a ⇒ 'b ⇒ bool)
  for R F
  where wfrecI: (∀ z. (z, x) ∈ R =⇒ wfrec-rel R F z (g z)) =⇒ wfrec-rel R F x (F g x)

definition cut :: ('a ⇒ 'b) ⇒ ('a × 'a) set ⇒ 'a ⇒ 'a ⇒ 'b
  where cut f R x = (λ y. if (y, x) ∈ R then f y else undefined)

definition adm-wf :: ('a × 'a) set ⇒ ((('a ⇒ 'b) ⇒ ('a ⇒ 'b)) ⇒ 'a ⇒ 'b ⇒ bool)
  where adm-wf R F ←→ (∀ f g x. (∀ z. (z, x) ∈ R =⇒ f z = g z) =⇒ F f x = F g x)

definition wfrec :: ('a × 'a) set ⇒ ((('a ⇒ 'b) ⇒ ('a ⇒ 'b)) ⇒ 'a ⇒ 'b)
  where wfrec R F = (λ x. THE y. wfrec-rel R (λ f x. F (cut f R x) x) x y)

lemma cuts-eq: (cut f R x = cut g R x) =⇒ (∀ y. (y, x) ∈ R =⇒ f y = g y)
  by (simp add: fun-eq-iff cut-def)

lemma cut-apply: (x, a) ∈ R =⇒ cut f R a x = f x
  by (simp add: cut-def)

  Inductive characterization of wfrec combinator; for details see: John Harrison,
  "Inductive definitions: automation and application".

lemma theI-unique: ∃! x. P x =⇒ P x =⇒ x = The P
  by (auto intro: the-equality[symmetric] theI)

lemma wfrec-unique:
  assumes adm-wf R F wf R
  shows ∃! y. wfrec-rel R F x y
  using (wf R)
  proof (induct)
    define f where f y = (THE z. wfrec-rel R F y z) for y
theory "Wfrec"

case (less x)
then have \( \forall y. (y, x) \in R \mapsto \text{wfrec-rel } R \ F \ y \ z \iff z = f \ y \)
  unfolding f-def by (rule theI-unique)
with \( \langle \text{adm-wf } R \ F \rangle \) show ?case
  by (subst wfrec-rel.simps) (auto simp: adm-wf-def)
qed

lemma adm-lemma: adm-wf R (\( \lambda \_ x. \_ \))
  by (auto simp: adm-wf-def intro: \( \text{arg-cong} \) \[ \langle \text{where } f = \lambda x. \_ \text{ for } y \rangle \) cuts-eq \[ \langle \text{THEN } \text{iffD2} \rangle \)

lemma wfrec: \( \text{wf } R \mapsto \text{wfrec } R \ F \ a = F \ (\text{cut } f \ R \ x) \ a \)
  apply (simp add: wfrec-def)
  apply (rule adm-lemma [THEN wfrec-unique, THEN the1-equality])
  apply assumption
  apply (rule wfrec-rel.wfrecI)
  done

This form avoids giant explosions in proofs. NOTE USE OF \( \equiv \).

lemma def-wfrec: \( f \equiv \text{wfrec } R \ F \mapsto \text{wf } R \mapsto f \ a = F \ (\text{cut } f \ R \ a) \ a \)
  by (auto intro: wfrec)

23.0.1 Well-founded recursion via genuine fixpoints

lemma wfrec-fixpoint:
  assumes \( \text{wf } R \)
  and adm: \( \text{adm-wf } R \ F \)
  shows \( \text{wfrec } R \ F = F \) (wfrec R F)
proof (rule ext)
  fix x
  have \( \text{wfrec } R \ F \ x = F \ (\text{cut } \text{wfrec } R \ F) \ R \ x \) \( x \)
    using \( \text{wfrec[of } R \ F) \) \( \text{by simp} \)
  also have \( \lambda y. (y, x) \in R \mapsto \text{cut } \text{wfrec } R \ F \) \( R \ x \ y = \text{wfrec } R \ F \ y \)
    by (auto simp add: cut-apply)
  then have \( F \ (\text{cut } \text{wfrec } R \ F) \ R \ x \) \( x = F \) (wfrec R F) \( x \)
    using \( \text{adm-adm-wf-def[of } R \ F) \) \( \text{by auto} \)
  finally show \( \text{wfrec } R \ F \ x = F \ ) (wfrec R F) \( x \).
qed

lemma wfrec-def-adm: \( f \equiv \text{wfrec } R \ F \mapsto \text{wf } R \mapsto \text{adm-wf } R \ F \mapsto f = F \)
  using \( \text{wfrec-fixpoint) \ by simp} \)

23.1 Wellfoundedness of same-fst

definition same-fst :: \( \langle 'a \Rightarrow \text{bool} \rangle \Rightarrow \langle 'a \Rightarrow ('b \times 'b) \text{ set} \rangle \Rightarrow (('a \times 'b) \times (\text{'a \times 'b})) \text{ set} \)
  where \( \text{same-fst } P \ R \cdot \langle \langle x', y' \rangle, (x, y) \rangle \cdot x' = x \wedge P \ x \wedge \langle y', y \rangle \in R \ x \rangle \)
— For \texttt{wfrecrec} declarations where the first \( n \) parameters stay unchanged in the recursive call.

\textbf{lemma} \texttt{same-fstI [intro!]}: \( P \ x \Rightarrow (y', y) \in R \ x \Rightarrow ((x, y'), (x, y)) \in \text{same-fst} \]
\( P \ R \)
by (simp add: same-fst-def)

\textbf{lemma} \texttt{wf-same-fst}:  
\begin{itemize}
\item \texttt{assumes} \( \forall x. P \ x \Rightarrow \text{wf} (R \ x) \)
\item \texttt{shows} \( \text{wf} (\text{same-fst} \ P \ R) \)
\end{itemize}
\textbf{proof} –  
\begin{itemize}
\item \texttt{have} \( \forall a \ b \ Q. \forall a \ b. (\forall x. P \ a \land (x, b) \in R \ a \Rightarrow Q (a, x)) \Rightarrow Q (a, b) \)
\item \texttt{proof} –  
\begin{itemize}
\item \texttt{fix} \( Q \ a \ b \)
\item \texttt{assume} \( \ast. \forall a \ b. (\forall x. P \ a \land (x,b) \in R \ a \Rightarrow Q (a,x)) \Rightarrow Q (a,b) \)
\item \texttt{show} \( Q(a,b) \)
\item \texttt{proof} (cases \( \text{wf} (R \ a) \))
\item \texttt{case} \( \text{True} \)
\item \texttt{then show} \( \ast \text{thesis} \)
\item \texttt{by} (\textit{induction} b \texttt{rule: wf-induct-rule} \ (use \( \ast \) in blast))
\item \texttt{qed} (use \( \ast \) \texttt{assms in blast})
\item \texttt{qed}
\end{itemize}
\item \texttt{then show} \( \ast \text{thesis} \)
\item \texttt{by} (\textit{clarsimp} simp add: \texttt{wf-def same-fst-def})
\item \texttt{qed}
\end{itemize}
end

24 Orders as Relations

\texttt{theory Order-Relation import Wfrec begin}

24.1 Orders on a set

\texttt{definition preorder-on} \( A \ r \equiv \text{refl-on} \ A \ r \land \text{trans} \ r \)

\texttt{definition partial-order-on} \( A \ r \equiv \text{preorder-on} \ A \ r \land \text{antisym} \ r \)

\texttt{definition linear-order-on} \( A \ r \equiv \text{partial-order-on} \ A \ r \land \text{total-on} \ A \ r \)

\texttt{definition strict-linear-order-on} \( A \ r \equiv \text{trans} \ r \land \text{irrefl} \ r \land \text{total-on} \ A \ r \)

\texttt{definition well-order-on} \( A \ r \equiv \text{linear-order-on} \ A \ r \land \text{wf}(r \land \text{Id}) \)

\texttt{lemmas order-on-defs =}
\begin{itemize}
\item \texttt{preorder-on-def partial-order-on-def linear-order-on-def}
\end{itemize}
lemma partial-order-on-def:
  assumes partial-order-on A r shows refl-on A r and trans r and antisym r
  using assms unfolding partial-order-on-def preorder-on-def by auto

lemma preorder-on-empty[simp]: preorder-on {} {} by (simp add: preorder-on-def trans-def)

lemma partial-order-on-empty[simp]: partial-order-on {} {} by (simp add: partial-order-on-def)

lemma linear-order-on-empty[simp]: linear-order-on {} {} by (simp add: linear-order-on-def)

lemma well-order-on-empty[simp]: well-order-on {} {} by (simp add: well-order-on-def)

lemma preorder-on-converse[simp]: preorder-on A (r\-1) = preorder-on A r by (simp add: preorder-on-def)

lemma partial-order-on-converse[simp]: partial-order-on A (r\-1) = partial-order-on A r by (simp add: partial-order-on-def)

lemma linear-order-on-converse[simp]: linear-order-on A (r\-1) = linear-order-on A r by (simp add: linear-order-on-def)

lemma partial-order-on-acyclic:
  partial-order-on A r \implies acyclic (r \- Id)
  by (simp add: acyclic-irrefl partial-order-on-def preorder-on-def trans-diff-Id)

lemma partial-order-on-well-order-on:
  finite r \implies partial-order-on A r \implies wf (r \- Id)
  by (simp add: finite-acyclic-wf partial-order-on-acyclic)

lemma strict-linear-order-on-diff-Id: linear-order-on A r \implies strict-linear-order-on A (r \- Id)
  by (simp add: order-on-defs trans-diff-Id)

lemma linear-order-on-singleton [simp]: linear-order-on \{x\} \{(x, x)\}
  by (simp add: order-on-defs)

lemma linear-order-on-acyclic:
  assumes linear-order-on A r
  shows acyclic (r \- Id)
using strict-linear-order-on-diff-Id[OF assms]
by (auto simp add: acyclic-irrefl strict-linear-order-on-def)

lemma linear-order-on-well-order-on:
  assumes finite r
  shows linear-order-on A r ←→ well-order-on A r
unfolding well-order-on-def
  using assms finite-acyclic-wf[OF linear-order-on-acyclic, OF r]
  by blast

24.2 Orders on the field

abbreviation Refl r ≡ refl-on (Field r) r
abbreviation Preorder r ≡ preorder-on (Field r) r
abbreviation Partial-order r ≡ partial-order-on (Field r) r
abbreviation Total r ≡ total-on (Field r) r
abbreviation Linear-order r ≡ linear-order-on (Field r) r
abbreviation Well-order r ≡ well-order-on (Field r) r

lemma subset-Image-Image-iff:
  Preorder r =⇒ A ⊆ Field r =⇒ B ⊆ Field r =⇒
  r " " A ⊆ r " " B =⇒ (∀ a ∈ A. ∃ b ∈ B. (b, a) ∈ r)
apply (simp add: preorder-on-def refl-on-def Image-def subset-eq)
apply (simp only: trans-def)
apply fast
done

lemma subset-Image1-Image1-iff:
  Preorder r =⇒ a ∈ Field r =⇒ b ∈ Field r =⇒ r " " {a} ⊆ r " " {b} =⇒ (b, a) ∈ r
  by (simp add: subset-Image-Image-iff)

lemma Refl-antisym-eq-Image1-Image1-iff:
  assumes Refl r
  and as: antisym r
  and abf: a ∈ Field r b ∈ Field r
  shows r " " {a} = r " " {b} =⇒ a = b
(is ?lhs =⇒ ?rhs)
proof
  assume ?lhs
then have *=: (∀ x. (a, x) ∈ r =⇒ (b, x) ∈ r
  by (simp add: set-eq-iff)
have (a, a) ∈ r (b, b) ∈ r using *Refl r abf by (simp-all add: refl-on-def)
then have (a, b) ∈ r (b, a) ∈ r using *[of a] *[of b] by simp-all
then show \(?rhs\)
  using \(\langle\text{antisym } r\rangle[\text{unfolded antisym-def}]\) by blast
next
  assume \(?rhs\)
  then show \(?lhs\) by fast
qed

lemma \textit{Partial-order-eq-Image1-Image1-iff}:
  \(\text{Partial-order } r \implies a \in \text{Field } r \implies b \in \text{Field } r \implies r \{a\} = r \{b\} \iff a = b\)
  by \(\langle\text{auto simp: order-on-defs Refl-antisym-eq-Image1-Image1-iff}\rangle\)

lemma \textit{Total-Id-Field}:
  assumes \(\text{Total } r\) and \(\not\text{Id}: \neg r \subseteq \text{Id}\)
  shows \(\text{Field } r = \text{Field } (r - \text{Id})\)
proof
  have \(\text{Field } r \subseteq \text{Field } (r - \text{Id})\)
proof (rule subsetI)
    fix \(a\) assume \(\ast\): \(a \in \text{Field } r\)
    from \(\not\text{Id}\) have \(r \neq \{\}\) by fast
    with \(\not\text{Id}\) obtain \(b\) and \(c\) where \(b \neq c \land (b, c) \in r\) by auto
    then have \(b \neq c \land \{b, c\} \subseteq \text{Field } r\) by \(\langle\text{auto simp: Field-def}\rangle\)
    with \(\ast\) obtain \(d\) where \(d \in \text{Field } r\) \(d \neq a\) by auto
    with \(\ast\) have \((a, d) \in r \lor (d, a) \in r\) by \(\langle\text{simp add: total-on-def}\rangle\)
    with \(\ast\) show \(a \in \text{Field } (r - \text{Id})\) unfolding Field-def by blast
qed
then show \(?\text{thesis}\)
  using \(\langle\text{mono-Field[of } r - \text{Id } r\rangle\) \(\text{Diff-subset[of } r \text{ Id}\rangle\) by auto
qed

24.3 Relations given by a predicate and the field

definition \textit{relation-of} :: \((\tau \Rightarrow \tau \Rightarrow \text{bool}) \Rightarrow \tau \text{ set} \Rightarrow (\tau \times \tau) \text{ set}\)
where \(\text{relation-of } P A \equiv \{ (a, b) \in A \times A. P a b \}\)

lemma \textit{Field-relation-of}:
  assumes \(\text{refl-on } A\) (\(\text{relation-of } P A\)) shows \(\text{Field } (\text{relation-of } P A) = A\)
using assms unfolding refl-on-def Field-def by auto

lemma \textit{partial-order-on-relation-off}:
  assumes \(\text{refl}: \forall a. a \in A \Rightarrow P a a\)
  and \(\text{trans}: \forall a b c. \{ a \in A; b \in A; c \in A \} \Rightarrow P a b \Rightarrow P b c \Rightarrow P a c\)
  and \(\text{antisym}: \forall a b. \{ a \in A; b \in A \} \Rightarrow P a b \Rightarrow P b a \Rightarrow a = b\)
shows \(\text{partial-order-on } A\) (\(\text{relation-of } P A\))
proof
  from \(\text{refl}\) have \(\text{refl-on } A\) (\(\text{relation-of } P A\))
  unfolding refl-on-def relation-of-def by auto
moreover have \(\text{trans } (\text{relation-of } P A)\) and \(\text{antisym } (\text{relation-of } P A)\)
unfolding relation-of-def
by (auto intro: transI dest: trans, auto intro: antisymI dest: antisym)
ultimately show thesis
unfolding partial-order-on-def preorder-on-def by simp
qed

lemma Partial-order-relation-ofI:
  assumes partial-order-on A (relation-of P A) shows Partial-order (relation-of P A)
  using Field-relation-of assms partial-order-on-def preorder-on-def by fastforce

24.4 Orders on a type
abbreviation strict-linear-order ≡ strict-linear-order-on UNIV
abbreviation linear-order ≡ linear-order-on UNIV
abbreviation well-order ≡ well-order-on UNIV

24.5 Order-like relations

In this subsection, we develop basic concepts and results pertaining to order-like relations, i.e., to reflexive and/or transitive and/or symmetric and/or total relations. We also further define upper and lower bounds operators.

24.5.1 Auxiliaries

lemma refl-on-domain: refl-on A r ⟹ (a, b) ∈ r ⟹ a ∈ A ∧ b ∈ A
  by (auto simp add: refl-on-def)

corollary well-order-on-domain: well-order-on A r ⟹ (a, b) ∈ r ⟹ a ∈ A ∧ b ∈ A
  by (auto simp add: refl-on-domain order-on-defs)

lemma well-order-on-Field: well-order-on A r ⟹ A = Field r
  by (auto simp add: refl-on-def Field-def order-on-defs)

lemma well-order-on-Well-order: well-order-on A r ⟹ A = Field r ∧ Well-order r
  using well-order-on-Field [of A] by auto

lemma Total-subset-Id:
  assumes Total r
  and r ⊆ Id
  shows r = {} ∨ (∃ a. r = {(a, a)})
proof
  have ∃ a. r = {(a, a)} if r ≠ {} 
    proof
      from that obtain a b where ab: (a, b) ∈ r by fast
with \( r \subseteq Id \) have \( a = b \) by blast
with \( ab \) have \( (a, a) \in r \) by simp
have \( a = c \land a = d \) if \( (c, d) \in r \) for \( c \) \( d \)
proof –
  from that have \( \{a, c, d\} \subseteq Field r \)
  using \( ab \) unfolding Field-def by blast
then have \( ((a, c) \in r \lor (c, a) \in r \lor a = c) \land ((a, d) \in r \lor (d, a) \in r \lor a = d) \)
  using \( \text{Total } r \) unfolding total-on-def by blast
then have \( ?thesis \) by blast
then have \( r \subseteq \{(a, a)\} \) by auto
then show \( ?thesis \) by blast
qed

then have \( r \subseteq \{(a, a)\} \) by auto
then show \( ?thesis \) by blast
qed

lemma Linear-order-in-diff-Id:
  assumes Linear-order \( r \)
  and \( a \in Field r \)
  and \( b \in Field r \)
  shows \( (a, b) \in r \iff (b, a) \notin r - Id \)
  using assms unfolding order-on-defs total-on-def antisym-def Id-def refl-on-def by force

24.5.2 The upper and lower bounds operators

Here we define upper ("above") and lower ("below") bounds operators. We think of \( r \) as a non-strict relation. The suffix \( S \) at the names of some operators indicates that the bounds are strict – e.g., \( \text{underS } a \) is the set of all strict lower bounds of \( a \) (w.r.t. \( r \)). Capitalization of the first letter in the name reminds that the operator acts on sets, rather than on individual elements.

definition under :: \( 'a \ rel \Rightarrow 'a \Rightarrow 'a \ set \)
  where \( \text{under } r \ a \equiv \{b. (b, a) \in r\} \)

definition underS :: \( 'a \ rel \Rightarrow 'a \Rightarrow 'a \ set \)
  where \( \text{underS } r \ a \equiv \{b. b \neq a \land (b, a) \in r\} \)

definition Under :: \( 'a \ rel \Rightarrow 'a \ set \Rightarrow 'a \ set \)
  where \( \text{Under } r \ A \equiv \{b \in Field r. \forall a \in A. (b, a) \in r\} \)

definition UnderS :: \( 'a \ rel \Rightarrow 'a \ set \Rightarrow 'a \ set \)
  where \( \text{UnderS } r \ A \equiv \{b \in Field r. \forall a \in A. b \neq a \land (b, a) \in r\} \)

definition above :: \( 'a \ rel \Rightarrow 'a \Rightarrow 'a \ set \)
  where \( \text{above } r \ a \equiv \{b. (a, b) \in r\} \)
THEORY "Order-Relation"

definition aboveS :: 'a rel ⇒ 'a ⇒ 'a set
  where aboveS r a ≡ \{ b. b ≠ a ∧ (a, b) ∈ r \}

definition Above :: 'a rel ⇒ 'a set ⇒ 'a set
  where Above r A ≡ \{ b ∈ Field r. ∀ a ∈ A. (a, b) ∈ r \}

definition AboveS :: 'a rel ⇒ 'a set ⇒ 'a set
  where AboveS r A ≡ \{ b ∈ Field r. ∀ a ∈ A. b ≠ a ∧ (a, b) ∈ r \}

definition ofilter :: 'a rel ⇒ 'a set ⇒ bool
  where ofilter r A ≡ A ⊆ Field r ∧ (∀ a ∈ A. under r a ⊆ A)

Note: In the definitions of Above[S] and Under[S], we bounded comprehension by Field r in order to properly cover the case of A being empty.

lemma underS-subset-under: underS r a ⊆ under r a
  by (auto simp add: underS-def under-def)

lemma underS-notIn: a /∈ underS r a
  by (simp add: underS-def)

lemma Refl-under-in: Refl r ⇒ a ∈ Field r ⇒ a ∈ under r a
  by (simp add: refl-on-def under-def)

lemma AboveS-disjoint: A ∩ (AboveS r A) = {}
  by (auto simp add: AboveS-def)

lemma in-AboveS-underS: a ∈ Field r ⇒ a ∈ AboveS r (underS r a)
  by (auto simp add: AboveS-def underS-def)

lemma Refl-under-underS: Refl r ⇒ a ∈ Field r ⇒ under r a = underS r a ∪ \{a\}
  unfolding under-def underS-def
  using refl-on-def[of - r] by fastforce

lemma underS-empty: a /∈ Field r ⇒ underS r a = {}
  by (auto simp: Field-def underS-def)

lemma under-Field: under r a ⊆ Field r
  by (auto simp: under-def Field-def)

lemma underS-Field: underS r a ⊆ Field r
  by (auto simp: underS-def Field-def)

lemma underS-Field2: a ∈ Field r ⇒ underS r a ⊆ Field r
  using underS-notIn underS-Field by fast

lemma underS-Field3: Field r ≠ \{\} ⇒ underS r a ⊆ Field r
  by (cases a ∈ Field r) (auto simp: underS-Field2 underS-empty)
lemma AboveS-Field: AboveS r A ⊆ Field r
  by (auto simp: AboveS-def Field-def)

lemma under-incr:
  assumes trans r
  and (a, b) ∈ r
  shows under r a ⊆ under r b
proof safe
  fix x assume (x, a) ∈ r
  with assms trans-def[of r] show (x, b) ∈ r by blast
qed

lemma underS-incr:
  assumes trans r
  and antisym r
  and ab: (a, b) ∈ r
  shows underS r a ⊆ underS r b
proof safe
  assume *: b ≠ a and **: (b, a) ∈ r
  with (antisym r) antisym-def[of r] ab show False
    by blast
next
  fix x assume x ≠ a (x, a) ∈ r
  with ab (trans r) trans-def[of r] show (x, b) ∈ r
    by blast
qed

lemma underS-incl-iff:
  assumes LO: Linear-order r
  and INA: a ∈ Field r
  and INb: b ∈ Field r
  shows underS r a ⊆ underS r b ←→ (a, b) ∈ r
(is ?lhs ←→ ?rhs)
proof
  assume ?rhs
  with (Linear-order r) show ?lhs
    by (simp add: order-on-defs underS-incr)
next
  assume *: ?lhs
  have (a, b) ∈ r if a = b
    using assms that by (simp add: order-on-defs refl-on-def)
  moreover have False if a ≠ b (b, a) ∈ r
  proof
    from that have b ∈ underS r a unfolding underS-def by blast
    with * have b ∈ underS r b by blast
    then show ?thesis by (simp add: underS-notIn)
  qed
ultimately show \((a, b) \in r\) using assms order-on-defs[of Field r r] total-on-def[of Field r r] by blast

qed

lemma finite-Partial-order-induct[consumes 3, case-names step]:
assumes Partial-order r
and \(x \in\) Field r
and finite r
and step: \(\forall x. x \in\) Field r \(\Rightarrow (\forall y. y \in\) aboveS \(r\) \(x \Rightarrow P y) \Rightarrow P x\)
shows \(P x\)
using assms(2)
proof (induct rule: wf-induct[of \(r^{-1} - Id\)])
case 1
from assms(1,3) show \(wf (r^{-1} - Id)\)
using partial-order-on-well-order-on partial-order-on-converse by blast
next
case prems: (2 \(x\))
show ?case
by (rule step) (use prems in \‹auto simp: aboveS-def intro: FieldI2›)
qed

lemma finite-Linear-order-induct[consumes 3, case-names step]:
assumes Linear-order r
and \(x \in\) Field r
and finite r
and step: \(\forall x. x \in\) Field r \(\Rightarrow (\forall y. y \in\) aboveS \(r\) \(x \Rightarrow P y) \Rightarrow P x\)
shows \(P x\)
using assms(2)
proof (induct rule: wf-induct[of \(r^{-1} - Id\)])
case 1
from assms(1,3) show \(wf (r^{-1} - Id)\)
using linear-order-on-well-order-on linear-order-on-converse
unfolding well-order-on-def by blast
next
case prems: (2 \(x\))
show ?case
by (rule step) (use prems in \‹auto simp: aboveS-def intro: FieldI2›)
qed

24.6 Variations on Well-Founded Relations

This subsection contains some variations of the results from HOL. Wellfounded:

- means for slightly more direct definitions by well-founded recursion;
- variations of well-founded induction;
- means for proving a linear order to be a well-order.
24.6.1 Characterizations of well-foundedness

A transitive relation is well-founded iff it is “locally” well-founded, i.e., iff its restriction to the lower bounds of of any element is well-founded.

**lemma** trans-wf-iff:

- **assumes** trans \( r \)
- **shows** \( \forall a. \ wf (r \cap (r^{-1}\cdot\{a\} \times r^{-1}\cdot\{a\}))) \)

**proof**
- **define** \( R \) where \( R a = r \cap (r^{-1}\cdot\{a\} \times r^{-1}\cdot\{a\}) \) \( \) for \( a \)
- **have** \( \forall a. \ wf (R a) \) if \( \forall a. \ wf (r) \) for \( a \)
- **unfolding** \( \) proud

**proof**
- **fix** \( \phi \) \( a \)
- **assume** \( **: \forall a. (\forall b. (b, a) \in r \rightarrow \phi b) \rightarrow \phi a \)
- **define** \( \chi \) where \( \chi b \leftarrow (b, a) \in r \rightarrow \phi b \) for \( b \)
- **with** \( ** \) have \( \forall (R a) \) by auto
- **then have** \( (\forall b. (\forall c. (c, b) \in R a \rightarrow \chi c) \rightarrow \chi b) \rightarrow (\forall b. \ chi b) \)
- **unfolding** \( \) proud by blast
- **also have** \( \forall b. (\forall c. (c, b) \in R a \rightarrow \chi c) \rightarrow \chi b \)
- **proof**
  - **fix** \( b \)
  - **assume** \( \forall c. (c, b) \in R a \rightarrow \chi c \)
  - **moreover** have \( (b, a) \in r \Rightarrow \forall c. (c, b) \in r \land (c, a) \in r \rightarrow \phi c \Rightarrow \phi \) \( b \)
  - **proof**
    - **assume** \( (b, a) \in r \) and \( \forall c. (c, b) \in r \land (c, a) \in r \rightarrow \phi c \)
    - **then have** \( \forall c. (c, b) \in r \rightarrow \phi c \)
      - **using** \( ** \) have \( \forall (of r R a) \) by blast
    - **with** \( ** \) show \( \phi b \) by blast
  - **qed**
  - ultimately show \( \chi b \)
    - **by** \( (auto simp add: chi-def R-def) \)
  - **qed**
  - finally have \( \forall b. \chi b \).
    - **with** \( ** \) chi-def show \( \phi a \) by blast
  - **qed**
  - ultimately show \( \)thesis unfolding \( R-def \) by blast
- ** qed**

A transitive relation is well-founded if all initial segments are finite.

**corollary** wf-finite-segments:

- **assumes** irrefl \( r \) and trans \( r \) and \( \forall x. \ finite \{y. (y, x) \in r\} \)
- **shows** \( \forall a. \ wf (r \cap \{x. (x, a) \in r\} \times \{x. (x, a) \in r\}) \)

**proof**
- **have** \( \forall a. \ acyclic (r \cap \{x. (x, a) \in r\} \times \{x. (x, a) \in r\}) \)
- **proof**
  -
THEORY “Order-Relation”

fix a
have trans (r ∩ (\{x. (x, a) ∈ r\} × \{x. (x, a) ∈ r\}))
  using assms unfolding trans-def Field-def by blast
then show acyclic (r ∩ \{x. (x, a) ∈ r\} × \{x. (x, a) ∈ r\})
  using assms acyclic-def assms irrefl-def by fastforce
qed
then show thesis
  by (clarsimp simp: trans-wf-iff wf-iff-acyclic-if-finite converse-def assms)
qed

The next lemma is a variation of wf-eq-minimal from Wellfounded, allowing
one to assume the set included in the field.

lemma wf-eq-minimal2: wf r ←→ (∀ A. A ⊆ Field r ∧ A ≠ {} → (∃ a ∈ A. ∀ a’ ∈ A. (a’, a) ∉ r))

proof
  let ?phi = λA. A ≠ {} → (∃ a ∈ A. ∀ a’ ∈ A. (a’, a) ∉ r)
  have wf r ←→ (∀ A. ?phi A)
    proof
      clarify
      fix A:: 'a set
      assume A ≠ {}
      then obtain x where x ∈ A
        by auto
      show (∃ a∈A. ∀ a'∈A. (a', a) ∉ r)
        apply (rule wfE-min[of r x A])
        apply fact+
        by blast
    qed

next
  assume *: ∀ A. ?phi A
  then show wf r
    apply (clarsimp simp: ex-in-conv [THEN sym])
    apply (rule wfI-min)
    by fast
  qed

also have (∀ A. ?phi A) ←→ (∀ B ⊆ Field r. ?phi B)
  proof
    assume ∀ A. ?phi A
    then show ∀ B ⊆ Field r. ?phi B by simp
  next
    assume *: ∀ B ⊆ Field r. ?phi B
    show ∀ A. ?phi A
      proof clarify
        fix A:: 'a set
        assume **: A ≠ {}
        define B where B = A ∩ Field r
        show (∃ a ∈ A. ∀ a’ ∈ A. (a’, a) ∉ r)
proof (cases $B = \{\}$)
  case True
  with ** obtain $a$ where $a: a \in A \land a \notin \text{Field } r$
  unfolding $B$-def by blast
  with $a$ have $\forall a' \in A. (a',a) \notin r$
  unfolding $\text{Field-def}$ by blast
  with $a$ show $?thesis$ by blast

next
  case False
  have $B \subseteq \text{Field } r$ unfolding $B$-def by blast
  with False * obtain $a$ where $a: a \in B \land \forall a' \in B. (a',a) \notin r$
  by blast
  have $(a',a) \notin r$ if $a' \in A$ for $a'$
proof
  assume $a'a: (a',a) \in r$
  with that have $a' \in B$ unfolding $B$-def $\text{Field-def}$ by blast
  with $a$ a'a show False by blast
qed
  with $a$ show $?thesis$ unfolding $B$-def by blast
qed

finally show $?thesis$ by blast
qed

24.6.2 Characterizations of well-foundedness

The next lemma and its corollary enable one to prove that a linear order is
a well-order in a way which is more standard than via well-foundedness of
the strict version of the relation.

lemma Linear-order-wf-diff-Id:
assumes Linear-order $r$
shows $\forall A \subseteq \text{Field } r. A \neq \{\} \rightarrow (\exists a \in A. \forall a' \in A. (a,a') \in r)$
proof (cases $r \subseteq \text{Id}$)
  case True
  then have $*: r - \text{Id} = \{\}$ by blast
  have $\text{wf} \ (r - \text{Id})$ by ($\text{simp add: } *$)
  moreover have $\exists a \in A. \forall a' \in A. (a,a') \in r$
  if $*: A \subseteq \text{Field } r$ and $**: A \neq \{\}$ for $A$
proof
  from (Linear-order $r$) True
  obtain $a$ where $a: r = \{\} \lor r = \{(a,a)\}$
  unfolding $\text{order-on-defs using } \text{Total-subset-Id}$ of $r$ by blast
  with $**$ have $A = \{a\} \land r = \{(a,a)\}$
  unfolding $\text{Field-def}$ by blast
  with $a$ show $?thesis$ by blast
qed
  ultimately show $?thesis$ by blast
qed
theorem "Hilbert-Choice"

next
case False
with \langle Linear-order r \rangle have Field: Field r = Field (r \setminus Id)
  unfolding order-on-defs using Total-Id-Field [of r] by blast
show \?thesis
proof
assume *: wf (r \setminus Id)
show \forall A \subseteq Field r. A \neq \{\} \implies (\exists a \in A. \forall a' \in A. (a, a') \in r)
proof clarify
fix A
assume **: A \subseteq Field r and ***: A \neq \{\}
then have \exists a \in A. \forall a' \in A. (a', a) \notin r \setminus Id
  using Field * unfolding wf-eq-minimal2 by simp
moreover have \forall a \in A. \forall a' \in A. (a, a') \in r \iff (a', a) \notin r \setminus Id
  using Linear-order-in-diff-Id [OF \langle Linear-order r \rangle] ** by blast
ultimately show \exists a \in A. \forall a' \in A. (a, a') \in r by blast
qed
next
assume *: \forall A \subseteq Field r. A \neq \{\} \implies (\exists a \in A. \forall a' \in A. (a, a') \in r)
show wf (r \setminus Id)
  unfolding wf-eq-minimal2
proof clarify
fix A
assume **: A \subseteq Field(r \setminus Id) and ***: A \neq \{\}
then have \exists a \in A. \forall a' \in A. (a, a') \in r
  using Field * by simp
moreover have \forall a \in A. \forall a' \in A. (a, a') \in r \iff (a', a) \notin r \setminus Id
  using Linear-order-in-diff-Id [OF \langle Linear-order r \rangle] ** mono-Field[of r \setminus Id r] by blast
ultimately show \exists a \in A. \forall a' \in A. (a', a) \notin r \setminus Id
  by blast
qed
qed

corollary Linear-order-Well-order-iff:
Linear-order r \implies
Well-order r \iff \langle \forall A \subseteq Field r. A \neq \{\} \implies (\exists a \in A. \forall a' \in A. (a, a') \in r)\rangle
unfolding well-order-on-def using Linear-order-wf-diff-Id[of r] by blast

end

25 Hilbert’s Epsilon-Operator and the Axiom of Choice

theory Hilbert-Choice
imports Wellfounded
keywords specification :: thy-goal-defn
25.1 Hilbert’s epsilon

axiomatization $Eps :: \{a \Rightarrow bool\} \Rightarrow \{a\}$

where someI: $P x \Longrightarrow P (Eps P)$

syntax (epsilon)

$-Eps :: \{ pttrn \Rightarrow bool \Rightarrow \{a\} ((3\epsilon -/-) [0, 10])$

syntax (input)

$-Eps :: \{ pttrn \Rightarrow bool \Rightarrow \{a\} ((3@ -/-) [0, 10])$

syntax

$-Eps :: \{ pttrn \Rightarrow bool \Rightarrow \{a\} ((3SOME -/-) [0, 10])$

translations

$SOME x. P \Rightarrow CONST Eps (\lambda x. P)$

print-translation

$\langle \langle const-synt\xrightarrow{\langle Eps, fn \Rightarrow fn Abs abs \Rightarrow} >
\rangle let val (x, t) = Syntax-Trans.atomic-abs-tr \langle abs
\rangle in Syntax.const syntax-const \langle Eps> \$ x \$ t end]$

definition inv-into :: \{a set \Rightarrow \{a \Rightarrow \{b\} \Rightarrow \{b \Rightarrow \{a\}\}\}$ where

$inv-into A f = (\lambda x. SOME y. y \in A \land f y = x)$

lemma inv-into-def2: $inv-into A f x = (SOME y. y \in A \land f y = x)$

by (simp add: inv-into-def)

abbreviation inv :: \{a \Rightarrow \{b\} \Rightarrow \{b \Rightarrow \{a\}\}\} where

$inv \equiv inv-into UNIV$

25.2 Hilbert’s Epsilon-operator

lemma $Eps-cong$:

assumes $\forall x. P x = Q x$

shows $Eps P = Eps Q$

using ext[of P Q, OF assms] by simp

Easier to use than someI if the witness comes from an existential formula.

lemma someI-ex [elim?] $\exists x. P x \Longrightarrow P (SOME x. P x)$

by (elim exE someI)

lemma some-eq-imp:

assumes $Eps P = a P b$ shows $P a$

using assms someI-ex by force

Easier to use than someI because the conclusion has only one occurrence of $P$.

lemma someI2: $P a \Longrightarrow (\forall x. P x \Longrightarrow Q x) \Longrightarrow Q (SOME x. P x)$
Easier to use than someI2 if the witness comes from an existential formula.

**lemma** someI2-ex: \( \exists a. P a \implies (\forall x. P x \implies Q x) \implies Q (\text{SOME} x. P x) \)
by (blast intro: someI)

**lemma** someI2-bex: \( \exists a \in A. P a \implies (\forall x. x \in A \land P x \implies Q x) \implies Q (\text{SOME} x. x \in A \land P x) \)
by (blast intro: someI)

**lemma** some-equality [intro]: \( P a \implies (\forall x. P x \implies x = a) \implies (\text{SOME} x. P x) = a \)
by (blast intro: someI)

**lemma** some1-equality: \( \exists! x. P x \implies P a \implies (\text{SOME} x. P x) = a \)
by blast

**lemma** some-eq-ex: \( P (\text{SOME} x. P x) \iff (\exists x. P x) \)
by (fast elim: someI)

**lemma** bchoice: \( \forall x \in S. \exists y. Q x y \implies \exists f. \forall x \in S. Q x (f x) \)
by (fast elim: someI)

**lemma** choice-iff: \( (\forall x y. Q x y) \iff (\exists f. \forall y. Q x (f y)) \)
by (fast elim: someI)

**lemma** choice-iff': \( (\forall x. P x \implies (\exists y. Q x y)) \iff (\exists f. \forall x. P x \implies Q x (f x)) \)
by (fast elim: someI)

**lemma** bchoice-iff: \( (\forall x \in S. \exists y. Q x y) \iff (\exists f. \forall x \in S. Q x (f x)) \)
by (fast elim: someI)

**lemma** bchoice-iff': \( (\forall x \in S. P x \implies (\exists y. Q x y)) \iff (\exists f. \forall x \in S. P x \implies Q x (f x)) \)
THEORY "Hilbert-Choice"

by (fast elim: someI)

lemma dependent-nat-choice:
  assumes 1: \( \exists x. P(0, x) \)
    and 2: \( \forall n. P(n, x) \implies \exists y. P(\text{Suc } n, y) \land Q(n, x, y) \)
  shows \( \exists f. \forall n. P(n, f(n)) \land Q(n, f(n)) \land (f(\text{Suc } n)) \)
proof (intro exI allI conjI)
  fix \( n \)
  define \( f \) where \( f = \text{rec-nat}(\text{SOME } x. P(0, x)) (\lambda n x. \text{SOME } y. P(\text{Suc } n, y) \land Q(n, x, y)) \)
  then have \( P(0, f(0)) \land P(n, f(n)) \implies P(\text{Suc } n, f(\text{Suc } n)) \land Q(n, f(n), f(\text{Suc } n)) \)
    using someI-ex[OF 1] someI-ex[OF 2] by simp-all
  then show \( P(n, f(n)) \land Q(n, f(n)) \land (f(\text{Suc } n)) \)
    by (induct \( n \)) auto
qed

lemma finite-subset-Union:
  assumes finite \( A, A \subseteq \bigcup B \)
  obtains \( F \) where finite \( F, F \subseteq B \land A \subseteq \bigcup F \)
proof
  have \( \forall x \in A. \exists B \in B. x \in B \)
    using assms by blast
  then obtain \( f \) where \( f = \bigwedge x. x \in A \implies f(x) \in B \land x \in f(x) \)
    by (auto simp add: bchoice-iff Bex-def)
  show thesis
  proof
    show finite \( f('A) \)
      using assms by auto
    qed (use \( f \) in auto)
  qed

25.4 Function Inverse

lemma inv-def: \( \text{inv } f = (\lambda y. \text{SOME } x. f(x) = y) \)
by (simp add: inv-into-def)

lemma inv-into-into: \( x \in f('A) \implies \text{inv-into } A f x \in A \)
by (simp add: inv-into-def) (fast intro: someI2)

lemma inv-identity [simp]: \( \text{inv } (\lambda a. a) = (\lambda a. a) \)
by (simp add: inv-def)

lemma inv-id [simp]: \( \text{inv } id = id \)
by (simp add: id-def)

lemma inv-into-f-f [simp]: \( \text{inj-on } A f \in A \implies \text{inv-into } A f (f x) = x \)
by (simp add: inv-into-def inj-on-def) (blast intro: someI2)
THEORY "Hilbert-Choice"

lemma \textit{inv-f-f}: \(\text{inj } f \implies \text{inv } f \ (f \ x) = x\)
by simp

lemma \textit{f-inv-into-f}: \(\forall y \in f\ A \implies f \ (\text{inv-into } A \ f \ y) = y\)
by (simp add: inv-into-def) (fast intro: someI2)

lemma \textit{inv-into-f-eq}: \(\text{inj-on } f \ A \implies \forall x \in A \implies f x = y \implies \text{inv-into } A \ f \ y = x\)
by (erule subst) (fast intro: inv-into-f-f)

lemma \textit{inv-f-eq}: \(\text{inj } f \implies \forall x \ y \in f \ y = x \implies \text{inv } f \ y = x\)
by simp add: inv-into-f-eq

lemma \textit{inj-imp-inv-eq}: \(\text{inj } f \implies \forall x \ s.t. f (g x) = x \implies \text{inv } f = g\)
by blast intro: inv-into-f-eq

But is it useful?

lemma \textit{inj-transfer}:
assumes \textit{inj}: \(\text{inj } f\)
and \textit{minor}: \(\forall y \in \text{range } f \implies \text{P } (\text{inv } f \ y)\)
shows \(\text{P } x\)
proof -
have \(f x \in \text{range } f\) by auto
then have \(\text{P } (\text{inv } f \ (f \ x))\) by (rule minor)
then show \(\text{P } x\) by (simp add: inv-into-f-f [OF inj])
qed

lemma \textit{inj-iff}: \(\text{inj } f \iff \text{inv } f \circ f = \text{id}\)
by (simp add: o-def fun-eq-iff) (blast intro: inj-on-inverseI inv-into-f-f)

lemma \textit{inv-o-cancel}[simp]: \(\text{inj } f \implies \text{inv } f \circ f = \text{id}\)
by (simp add: inj-iff)

lemma \textit{o-inv-o-cancel}[simp]: \(\text{inj } f \implies g \circ \text{inv } f \circ f = g\)
by (simp add: comp-assoc)

lemma \textit{inv-into-image-cancel}[simp]: \(\text{inj-on } f \ A \implies S \subseteq A \implies \text{inv-into } A \ f \ f^{-1} S = S\)
by (fastforce simp: image-def)

lemma \textit{inj-imp-surj-inv}: \(\text{inj } f \implies \text{surj } (\text{inv } f)\)
by (blast intro: surjI inv-into-f-f)

lemma \textit{surj-f-inv-f}: \(\text{surj } f \implies f \ (\text{inv } f \ y) = y\)
by (simp add: f.inv-into-f)

lemma \textit{bij-inv-eq-iff}: \(\text{bij } p \implies x = \text{inv } p \ y \iff p \ x = y\)
using surj-f-inv-f[of p] by (auto simp add: bij-def)

lemma \textit{inv-into-injective}:
assumes \( eq: \text{inv-into} A f x = \text{inv-into} A f y \)
and \( x: x \in f'A \)
and \( y: y \in f'A \)
says \( x = y \)
proof –
from \( eq \) have \( f (\text{inv-into} A f x) = f (\text{inv-into} A f y) \)
by simp
with \( x y \) show \( ?\text{thesis} \)
by (simp add: f-inv-into-f)
qed

**lemma** inj-on-inv-into: \( B \subseteq f'A \Rightarrow \text{inj-on} (\text{inv-into} A f) B \)
by (blast intro: inj-onI dest: inv-into-injective injD)

**lemma** bij-imp-bij-betw-inv: \( \text{bij} f \Rightarrow \text{bij-betw} (\text{inv} f) (f \ M) M \)
by (simp add: bij-betw-def image-subsetI inj-on-inv-into)

**lemma** bij-betw-inv-into: \( \text{bij-betw} f A B \Rightarrow \text{bij-betw} (\text{inv-into} A f) B A \)
by (auto simp add: bij-betw-def inj-on-inv-into)

**lemma** surj-imp-inv-eq:
assumes \( \text{surj} f \)
and \( gf: \forall x. g (f x) = x \)
says \( \text{inv} f = g \)
proof (rule ext)
fix \( x \)
have \( g (f (\text{inv} f x)) = \text{inv} f x \)
by (rule gf)
then show \( \text{inv} f x = g x \)
by (simp add: surj-f-inv-f surj-f)
qed

**lemma** bij-imp-bij-inv: \( \text{bij} f \Rightarrow \text{bij} (\text{inv} f) \)
by (simp add: bij-def inj-on-inv-surj-imp-surj-imp-inj-inv)

**lemma** inv-equality: \( (\forall x. g (f x) = x) \Rightarrow (\forall y. f (g y) = y) \Rightarrow \text{inv} f = g \)
by (rule ext) (auto simp add: inv-into-def)

**lemma** inv-inv-eq: \( \text{bij} f \Rightarrow \text{inv} (\text{inv} f) = f \)
by (rule inv-equality) (auto simp add: bij-def surj-f-inv-f)

\( \text{bij} (\text{inv} f) \) implies little about \( f \). Consider \( f :: \text{bool} \Rightarrow \text{bool} \) such that \( f \text{ True} \)
\[ f = \text{False} = \text{True}. \] Then it is consistent with axiom \textit{someI} that \( \text{inv} f \) could be any function at all, including the identity function. If \( \text{inv} f = \text{id} \) then \( \text{inv} f \) is a bijection, but \( \text{inv} f, \text{surj} f \) and \( \text{inv} (\text{inv} f) = f \) all fail.

**Lemma** \text{inv}-\text{into}-\text{comp}:
\[
\begin{align*}
\text{inj-on} f (g \cdot A) & \implies \text{inj-on} g A \implies x \in f \cdot g \cdot A \implies \\
\text{inv-into} A (f \circ g) x & = (\text{inv-into} A g \circ \text{inv-into} (g \cdot A) f) x \\
\text{by} & \ (\text{auto simp: f-inv-into-f inv-into-into intro: inv-into-f-eq comp-inj-on})
\end{align*}
\]

**Lemma** \text{o-inv-distrib}:
\[
\begin{align*}
\text{bij} f & \implies \text{bij} g \implies \text{inv} (f \circ g) = \text{inv} g \circ \text{inv} f \\
\text{by} & \ (\text{rule inv-equality}) \ (\text{auto simp add: bij-def surj-f-inv-f})
\end{align*}
\]

**Lemma** \text{image-f-inv-f}:
\[
\begin{align*}
\text{surj} f & \implies f \cdot (\text{inv} f \cdot A) = A \\
\text{by} & \ \text{simp}
\end{align*}
\]

**Lemma** \text{bij-image-Collect-eq}:
\[
\begin{align*}
\text{assumes} & \ \text{bij} f \\
\text{shows} & \ f \cdot \text{Collect} P = \{y. \ \text{P} (\text{inv} f y)\} \\
\text{proof} & \ (\text{show} \ f \cdot \text{Collect} P \subseteq \{y. \ \text{P} (\text{inv} f y)\}) \\
& \ \text{using} \ \text{assms by} \ (\text{force simp add: bij-is-inj}) \\
& \ (\text{show} \ \{y. \ \text{P} (\text{inv} f y)\} \subseteq f \cdot \text{Collect} P) \\
& \ \text{using} \ \text{assms by} \ (\text{blast intro: bij-is-surj \ [THEN surj-f-inv-f, symmetric]])}
\end{align*}
\]

**Lemma** \text{bij-vimage-eq-inv-image}:
\[
\begin{align*}
\text{assumes} & \ \text{bij} f \\
\text{shows} & \ f^{-} A = \text{inv} f \cdot A \\
\text{proof} & \ (\text{show} \ f^{-} A \subseteq \text{inv} f \cdot A) \\
& \ \text{using} \ \text{assms by} \ (\text{blast intro: bij-is-surj \ [THEN inv-into-f-f, symmetric]])} \\
& \ (\text{show} \ \text{inv} f \cdot A \subseteq f^{-} A) \\
& \ \text{using} \ \text{assms by} \ (\text{auto simp add: bij-is-surj \ [THEN surj-f-inv-f]])}
\end{align*}
\]

**Lemma** \text{ind-fn-o-fn-is-id}:
\[
\begin{align*}
\text{fixes} & \ f: \!'a \Rightarrow \!'a \\
\text{assumes} & \ \text{bij} f \\
\text{shows} & \ ((\text{inv} f) \cdot \!\!\!\!n) \circ (f \cdot \!\!\!\!n) = (\lambda x. \ x) \\
\text{proof} & \ (\text{case \ Suc n}) \\
& \ (\text{have} \ ((\text{inv} f) \cdot \!\!\!\!n)\cdot (f \cdot \!\!\!\!n) \ x = x \ \text{for} \ x \ n) \\
& \ \text{proof \ (induction \ n)} \\
& \ \text{case \ Suc \ n} \\
& \ \text{have} \ (\text{inv} f) \cdot \!\!\!\!n \ (f \cdot \!\!\!\!n) \ x = y \ \text{for} \ y \\
& \ \text{by} \ (\text{simpl \ assms: bij-is-inj}) \\
& \ \text{have} \ (\text{inv} f) \cdot \!\!\!\!n \ \text{Suc} \ n \ (f \cdot \!\!\!\!n) \ x = (\text{inv} f) \cdot \!\!\!\!n \ (\text{inv} f \cdot f ((f \cdot \!\!\!\!n) \ x)) \\
& \ \text{by} \ (\text{simpl \ add: funpow-swap1})
\end{align*}
\]
also have ... = (inv f ≺≺ n) ((f ≺≺ n) x)
  using * by auto
also have ... = x using Suc.IH by auto
finally show ?case by simp
qed (auto)
then show ?thesis unfolding o-def by blast
qed

lemma fn-o-inv-fn-is-id:
  fixes f::'a ⇒ 'a
  assumes bij f
  shows (f ≺≺ n) o ((inv f) ≺≺ n) = (λx. x)
proof –
  have (f ≺≺ n) (((inv f) ≺≺ n) x) = x for x n
  proof (induction n)
    case (Suc n)
    have ∗: f(inv f y) = y for y
      using bij-inv-eq-iff [OF assms] by auto
    have (f ≺ Suc n) (((inv f) ≺ Suc n) x) = (f ≺ n) (f (inv f ((inv f ≺ n) x)))
      by (simp add: funpow-swap1)
    also have ... = (f ≺ n) (inv f ≺ n) x
      using ∗ by auto
    also have ... = x using Suc.IH by auto
    finally show ?case by simp
  qed (auto)
  then show ?thesis unfolding o-def by blast
qed

lemma inv-fn:
  fixes f::'a ⇒ 'a
  assumes bij f
  shows inv (f ≺≺ n) = ((inv f) ≺≺ n)
proof –
  have inv (f ≺≺ n) x = ((inv f) ≺≺ n) x for x
  proof (rule inv-into-f-eq)
    show inj (f ≺ n)
      by (simp add: inj-fn[OF bij-is-inj[OF assms]])
    show (f ≺ n) ((inv f ≺ n) x) = x
      using fn-o-inv-fn-is-id[OF assms, THEN fun-cong] by force
  qed auto
  then show ?thesis by auto
qed

lemma funpow-inj-finite:
  assumes inj p, finite {y. ∃ n. y = (p ≺ n) x}
  obtains n where ∃ n > 0. ((p ≺ n) x) = x
proof –
  have {infinite (UNIV :: nat set)}
    by simp
moreover have \( \{ y. \exists n. y = (p \uparrow n) x \} = (\lambda n. (p \uparrow n) x) \cdot \text{UNIV} \)

by auto

with assms have \( \text{finite} \ldots \)

by simp

ultimately have \( \exists n \in \text{UNIV}. \neg \text{finite} \{ m \in \text{UNIV}. (p \uparrow m) x = (p \uparrow n) x \} \)

by \( \text{rule pigeonhole-infinite} \)

then obtain \( n \) where \( \text{infinite} \{ m \in \text{UNIV}. (p \uparrow m) x = (p \uparrow n) x \} \)

by \( \text{auto} \)

then have \( \text{infinite} \{ m \in \text{UNIV}. (p \uparrow m) x = (p \uparrow n) x \} \)

by \( \text{auto} \)

then have \( \{ m \in \text{UNIV}. (p \uparrow m) x = (p \uparrow n) x \} \neq \{ \}

by \( \text{auto simp add: subset-singleton-iff} \)

then obtain \( m \) where \( m \neq n \)

by \( \text{auto} \)

{ fix \( m n \) assume \( (p \uparrow n) x = (p \uparrow m) x < n \)

have \( (p \uparrow (n - m)) x = (p \uparrow m) ((p \uparrow m) ((p \uparrow (n - m)) x)) \)

using \( \text{inj p} \) by \( \text{simp add: inv-f-f} \)

also have \( ((p \uparrow m) ((p \uparrow (n - m)) x)) = (p \uparrow n) x \)

using \( \text{funpow-add[of m \cdot n - m \cdot p]} \) by simp

also have \( \text{inv (p \uparrow m)} \ldots = x \)

using \( \text{inj p} \) by \( \text{simp add: \cdot (p \uparrow n) x = \cdot} \)

finally have \( (p \uparrow (n - m)) x = x \)

by \( \text{auto} \)

note \( \text{general = this} \)

show \( \text{thesis} \)

proof (cases \( m n \) rule: \text{linorder-cases})

  case less

  then have \( \langle n - m > 0, (p \uparrow (n - m)) x = x \rangle \)

  using \( \text{general[of n m]} \) \( m \) by simp-all

  then show \( \text{thesis} \) by \text{blast intro: that}

next

  case equal

  then show \( \text{thesis} \) using \( m \) by simp

next

  case greater

  then have \( \langle m - n > 0, (p \uparrow (m - n)) x = x \rangle \)

  using \( \text{general[of m n]} \) \( m \) by simp-all

  then show \( \text{thesis} \) by \text{blast intro: that}

qed

qed

lemma \( \text{mono-inv} \):

fixes \( f::'a::\text{linorder} \Rightarrow 'b::\text{linorder} \)

assumes \( \text{mono f bij f} \)

shows \( \text{mono (inv f)} \)

proof

fix \( x y::'b \) assume \( x \leq y \)

from \( \text{bij f} \) obtain \( a b \) where \( x = f a \) and \( y = f b \) by \( \text{fastforce simp: bij-def surj-def} \)
show $\text{inv } f \ x \leq \text{inv } f \ y$
proof (rule le-cases)
  assume $a \leq b$
  thus $\text{thesis using } \langle \text{bij } f \rangle, \ x \ y \ \text{by (simp add: bij-def \text{-}inv-f-f)}$
next
  assume $b \leq a$
  hence $f b \leq f a$ by (rule monoD[OF \langle mono f \rangle])
  hence $y \leq x$ using $x \ y \ \text{by simp}$
  hence $x = y$ using $\langle x \leq y \rangle \ \text{by auto}$
  thus $\text{thesis by simp}$
qed
qed

lemma strict-mono-inv-on-range:
  fixes $f :: 'a::\text{linorder} \Rightarrow 'b::\text{order}$
  assumes $\text{strict-mono } f$
  shows $\text{strict-mono-on } (\text{range } f) \ (\text{inv } f)$
proof (clarsimp simp: strict-mono-on-def)
  fix $x \ y$
  assume $f \ x < f \ y$
  then show $\text{inv } f \ (f \ x) < \text{inv } f \ (f \ y)$
    using assms strict-mono-imp-inj-on strict-mono-less by fastforce
qed

lemma mono-bij-Inf:
  fixes $f :: 'a::\text{complete-linorder} \Rightarrow 'b::\text{complete-linorder}$
  assumes $\text{mono } f \ \text{bij } f$
  shows $f \ (\text{Inf } A) = \text{Inf } (f' A)$
proof
  have $\text{surj } f$ using $\langle \text{bij } f \rangle$ by (auto simp: bij-betw-def)
  have $* : (\text{inv } f) \ (\text{Inf } (f' A)) \leq \text{Inf } ((\text{inv } f) \ (f' A))$
    using mono-Inf[OF mono-inv[OF assms], of f' A] by simp
  have $\text{Inf } (f' A) \leq f \ (\text{Inf } ((\text{inv } f) \ (f' A)))$
    using monoD[OF \langle mono f \rangle $* $] by(simp add: surj-f-inv-f[OF \langle surj f \rangle])
  also have $\ldots = f (\text{Inf } A)$
    using assms by (simp add: bij-is-înj)
  finally show $\text{thesis using mono-\text{-}Inf}[\text{OF assms}(1), \ \text{of } A]$ by auto
qed

lemma finite-fun-UNIVD1:
  assumes $\text{fin : finite } (\text{UNIV :: } 'a \Rightarrow 'b \ \text{set})$
  and $\text{card : card } (\text{UNIV :: } 'b \ \text{set}) \neq \text{Suc 0}$
  shows $\text{finite } (\text{UNIV :: } 'a \ \text{set})$
proof
  let $?UNIV-b = \text{UNIV :: } 'b \ \text{set}$
  from $\text{fin}$ have $\text{finite } ?UNIV-b$
    by (rule finite-fun-UNIVD2)
  with $\text{card}$ have $\text{card } ?UNIV-b \geq \text{Suc } (\text{Suc 0})$
    by (cases card $?UNIV-b$) (auto simp: card-eq-0-iff)
then have \( \text{card } \text{UNIV-b} = \text{Suc } (\text{card } \text{UNIV-b} - \text{Suc } (\text{Suc } 0)) \)
  by simp
then obtain \( b1 b2 :: 'b \) where \( b1b2; b1 \neq b2 \)
  by (auto simp: card-Suc-eq)
from \( \text{fin} \) have \( \text{fin'}: \text{finite } (\text{range } (\lambda f :: 'a \Rightarrow 'b. \text{inv } f b1)) \)
  by (rule finite-imageI)
have \( \text{UNIV} = \text{range } (\lambda f :: 'a \Rightarrow 'b. \text{inv } f b1) \)
proof (rule UNIV-eq-I)
  fix \( x :: 'a \)
  from \( b1b2 \) have \( x = \text{inv } (\lambda y. \text{if } y = x \text{ then } b1 \text{ else } b2) b1 \)
    by (simp add: inv-into-def)
then show \( x \in \text{range } (\lambda f :: 'a \Rightarrow 'b. \text{inv } f b1) \)
  by blast
qed
with \( \text{fin'} \) show \( ?\text{thesis} \)
  by simp
qed

Every infinite set contains a countable subset. More precisely we show that
a set \( S \) is infinite if and only if there exists an injective function from the
naturals into \( S \).

The “only if” direction is harder because it requires the construction of a
sequence of pairwise different elements of an infinite set \( S \). The idea is to
construct a sequence of non-empty and infinite subsets of \( S \) obtained by
successively removing elements of \( S \).

lemma infinite-countable-subset:
assumes \( \text{inf} :: \neg \text{finite } S \)
shows \( \exists f :: \text{nat } \Rightarrow 'a. \text{inj } f \land \text{range } f \subseteq S \)
— Courtesy of Stephan Merz
proof
  define \( Sseq \) where \( Sseq = \text{rec-nat } S \ (\lambda n \ T. \ T - \{\text{SOME } e. e \in T\}) \)
  define \( \text{pick} \) where \( \text{pick } n = (\text{SOME } e. e \in Sseq n) \text{ for } n \)
  have \( *: Sseq n \subseteq S \land \text{finite } (Sseq n) \text{ for } n \)
    by (induct n) (auto simp: Sseq-def inf)
then have \( **: \forall n. \text{pick } n \in Sseq n \)
  unfolding pick-def by (subst (asm) finite.simps) (auto simp add: ex-in-conv
  intro: someI-ex)
with \( * \) have \( \text{range } \text{pick } \subseteq S \) by auto
moreover have \( \text{pick } n \neq \text{pick } (n + \text{Suc } m) \text{ for } m n \)
proof
  have \( \text{pick } n \notin Sseq (n + \text{Suc } m) \)
    by (induct m) (auto simp add: Sseq-def pick-def)
  with \( ** \) show \( ?\text{thesis} \) by auto
qed
then have \( \text{inj } \text{pick} \)
  by (intro linorder-injI) (auto simp add: less-iff-Suc-add)
ultimately show \( ?\text{thesis} \) by blast
qed
lemma infinite-iff-countable-subset: \(\neg \text{finite } S \iff (\exists f :: \text{nat} \Rightarrow 'a. \text{inj } f \land \text{range } f \subseteq S)\)
— Courtesy of Stephan Merz

using finite-imageD finite-subset infinite-UNIV-char-0 infinite-countable-subset
by auto

lemma image-inv-into-cancel:
assumes surj: \(f' A = A'\)
   and sub: \(B' \subseteq A'\)
shows \(f \langle (\text{inv-into } A f)' B' \rangle = B'\)
using assms
proof (auto simp: f-inv-into-f)
let \(?f' = \text{inv-into } A f\)
fix \(a'\)
assume \(*: a' \in B'\)
with sub have \(a' \in A'\) by auto
with surj have \(a' = f \langle ?f' a' \rangle\)
   by (auto simp: f-inv-into-f)
with \(*\) show \(a' \in f \langle ?f' \cdot B' \rangle\) by blast
qed

lemma inv-into-inv-into-eq:
assumes bij-betw f A A'
and a: \(a \in A\)
shows \(\text{inv-into } A' \langle \text{inv-into } A f \rangle a = f a\)
proof (auto simp: bij-betw-def)
let \(?f' = \text{inv-into } A f\)
let \(?f'' = \text{inv-into } A' ?f'\)
from assms have \(*: \text{bij-betw } ?f' A'\)
   by (auto simp: bij-betw-inv-into)
with a obtain \(a'\) where \(a': a' \in A' \iff a' = a\)
   unfolding bij-betw-def by force
with \(*\) have \(?f'' a = a'\)
   by (auto simp: f-inv-into-f bij-betw-def)
moreover from assms a' have \(f a = a'\)
   by (auto simp: bij-betw-def)
ultimately show \(?f'' a = f a\) by simp
qed

lemma inj-on-iff-surj:
assumes \(A \neq \{\}\)\)
shows \((\exists f. \text{inj-on } f A \land f \cdot A \subseteq A') \iff (\exists g. g \cdot A' = A)\)
proof safe
fix f
assume inj: \(\text{inj-on } f A\) and incl: \(f \cdot A \subseteq A'\)
let \(?phi = \lambda a'. a \in A \land f a = a'\)
let \(?csi = \lambda a. a \in A\)
let \(?g = \lambda a'. \text{if } a' \in f \cdot A \then \text{SOME } a. \text{?phi a'} \else \text{SOME } a. \text{?csi a}\)
have \( \exists g : A' = A \)

proof
  show \( \exists g : A' \subseteq A \)
  proof clarify
    fix \( a' \)
    assume \( \ast: a' \in A' \)
    show \( \exists g : a' \in A \)
    proof (cases \( a' \in f' A \))
      case True
      then obtain \( a \) where \( \phi a' a \) by blast
      then have \( \phi a' (\text{SOME } a. \phi a' a) \)
        using \( \text{someI[of } \phi a' a \] by blast \)
      with \( \text{True show } \phi \text{thesis by auto} \)
    next
      case False
      with \( \text{assms have } \psi (\text{SOME } a. \psi a) \)
        using \( \text{someI-ex[of } \psi \] by blast \)
      with \( \text{False show } \phi \text{thesis by auto} \)
  qed

next
  show \( A \subseteq \exists g : A' \)
  proof
    have \( \exists g : (f a) = a \land f a \in A' \) if \( a : a \in A \) for \( a \)
    proof
      let \( \phi = \lambda b'. \exists a. a' \in A' \land g b' = g a' \)
      from \( a' \) have \( \phi a' \) by auto
      then have \( \phi (\text{SOME } b'. \phi b) \)
        using \( \text{inj-on-def} \)
      with \( \text{incl show } \phi \text{thesis by auto} \)
      qed
    qed

then show \( \exists g : A' = A \) by blast

next
  fix \( g \)
  let \( f = \text{inv-into } A' g \)
  have \( \text{inj-on } f : (g' A') \)
    by (\( \text{auto simp: inj-on-inv-into} \))

moreover have \( \exists (g a') : A' \) if \( a' : a' \in A' \) for \( a' \)
  proof
    let \( \phi = \lambda a'. a' \in A' \land g b' = g a' \)
    from \( a' \) have \( \phi a' \) by auto
    then have \( \phi (\text{SOME } b'. \phi b) \)
using someI[of ?phi] by blast
then show ?thesis by (auto simp: inv-into-def)
qed
ultimately show \( \exists f. \text{inj-on } f \ (g \cdot A') \land f \cdot g \cdot A' \subseteq A' \)
  by auto
qed

lemma Ex-inj-on-UNION-Sigma:
\( \exists f. (\text{inj-on } f \ (\bigcup i \in I. \ A \ i) \land f \cdot (\bigcup i \in I. \ A \ i) \subseteq (\Sigma i : I. \ A \ i)) \)
proof
  let \( ?\phi = \lambda a \ i. \ i \in I \land a \in A \ i \)
  let \( ?\text{sm} = \lambda a. \ \text{SOME } i. \ ?\phi a \ i \)
  let \( ?f = \lambda a. \ (?\text{sm} a, a) \)
  have \( \text{inj-on } ?f \ (\bigcup i \in I. \ A \ i) \)
    by (auto simp: inj-on-def)
  moreover
  have \( ?\text{sm} a \in I \land a \in A \ (/?\text{sm} a) \) \text{ if } i \in I \text{ and } a \in A \ i \text{ for } i a
    using that someI[of \( ?\phi a \ i \)] by auto
  then have \( ?f \cdot (\bigcup i \in I. \ A \ i) \subseteq (\Sigma i : I. \ A \ i) \)
    by auto
  ultimately show \( \text{inj-on } ?f \ (\bigcup i \in I. \ A \ i) \land ?f \cdot (\bigcup i \in I. \ A \ i) \subseteq (\Sigma i : I. \ A \ i) \)
    by auto
qed

lemma inv-unique-comp:
assumes \( fg: f \circ g = \text{id} \)
  and \( gf: g \circ f = \text{id} \)
shows \( \text{inv } f = g \)
using \( fg \) \text{ inv-equality}[of \( g \ f \)] by (auto simp add: fun-eq_iff)

lemma subset-image-inj:
\( S \subseteq f \cdot T \longleftrightarrow (\exists U. \ U \subseteq T \land \text{inj-on } f \ U \land S = f \cdot U) \)
proof safe
  show \( \exists U \subseteq T. \text{inj-on } f \ U \land S = f \cdot U \)
    if \( S \subseteq f \cdot T \)
proof –
  from that [unfolded subset-image-iff subset-iff]
  obtain \( g \) \text{ where } \( g: \bigwedge x. \ x \in S \Rightarrow g \ x \in T \land x = f \ (g \ x) \)
    by (auto simp add: image-iff Bex-def choice iff)
  show \( \text{thesis} \)
proof (intro exI conjI)
  show \( ?S \subseteq T \)
    by (simp add: \( g \) image-subsetI)
  show \( \text{inj-on } f \ (g \cdot S) \)
    using \( g \) \text{ by (auto simp: inj-on-def)}
  show \( S = f \cdot (g \cdot S) \)
    using \( g \) \text{ image-subset-iff by auto}
qed
25.5 Other Consequences of Hilbert’s Epsilon

Hilbert’s Epsilon and the split Operator

Looping simp rule!

**Lemma** split-paired-Eps: $(\text{SOME } x. P x) = (\text{SOME } (a, b). P (a, b))$

*by simp*

**Lemma** Eps-case-prod: $Eps (\text{case-prod } P) = (\text{SOME } (xy). P (\text{fst } xy) (\text{snd } xy))$

*by (simp add: split-def)*

**Lemma** Eps-case-prod-eq [simp]: $(\text{SOME } (x', y'). x = x' \land y = y') = (x, y)$

*by blast*

A relation is wellfounded iff it has no infinite descending chain.

**Lemma** wf-iff-no-infinite-down-chain: $wf r \iff \neg \exists f. \forall i. (f (\text{Suc } i), f i) \in r$

*(is - \iff \neg ?ex)*

**Proof**
- **Assume** $wf r$
- **Show** $\neg ?ex$
  - **Proof**
    - **Assume** $?ex$
    - Then **obtain** $f$ where $f (\text{Suc } i), f i) \in i$ for $i$
      - **by blast**
    - From $wf r$ have $\text{minimal}$: $x \in Q \implies \exists y. (y, z) \in r \implies y \notin Q$ for $x$
      - **by (auto simp: wf-eq-minimal)**
    - Let $?Q = \{w. \exists i. w = f i\}$
    - Fix $n$
    - Have $f n \in ?Q$ by blast
      - From $\text{minimal}$ [OF this] obtain $j$ where $(y, f j) \in r$ \implies $y \notin ?Q$ for $y$ by blast
        - With this [OF $((f (\text{Suc } j), f j) \in r)$] have $f (\text{Suc } j) \notin ?Q$ by simp
        - Then **show** $False$ by blast
  - **Qed**

**Next**
- **Assume** $\neg ?ex$
- **Then show** $wf r$
  - **Proof** (rule contrapos-np)
    - **Assume** $\neg wf r$
    - Then **obtain** $Q$ where $x: x \in Q$ and $\text{rec}$: $z \in Q \implies \exists y. (y, z) \in r \land y \in Q$ for $z$
      - **by (auto simp add: wf-eq-minimal)**
    - Obtain $\text{descend} :: \text{nat} \Rightarrow 'a$
      - Where $\text{descend-0}$: $\text{descend } 0 = x$
and descend-Suc: descend (Suc n) = (SOME y. y ∈ Q ∧ (y, descend n) ∈ r) for n
by (rule that [of rec-nat x (λ- rec. (SOME y. y ∈ Q ∧ (y, rec) ∈ r))]) simp-all
have descend-Q: descend n ∈ Q for n
proof (induct n)
case 0
  with x show ?case by (simp only: descend-0)
next
case Suc
  then show ?case by (simp only: descend-Suc) (rule someI2-ex; use rec in blast)
qed

lemma wf-no-infinite-down-chainE:
assumes wf r
obtains k where (f (Suc k), f k) /∈ r
using assms wf-iff-no-infinite-down-chain[of r] by blast

A dynamically-scoped fact for TFL

lemma tfl-some: ∀ P x. P x → P (Eps P)
by (blast intro: someI)

25.6 An aside: bounded accessible part

Finite monotone eventually stable sequences

lemma finite-mono-remains-stable-implies-strict-prefix:
fixes f :: nat ⇒ 'a::order
assumes S: finite (range f) mono f
and eq: ∀ n. f n = f (Suc n) → f (Suc n) = f (Suc (Suc n))
shows ∃ N. (∀ n≤N. ∀ m≤N. m < n → f m < f n) ∧ (∀ n≥N. f N = f n)
using assms
proof –
  have ∃ n. f n = f (Suc n)
  proof (rule ccontr)
  assume ¬ ?thesis
  then have ∀ n. f n ≠ f (Suc n) by auto
  with (mono f) have ∀ n. f n < f (Suc n)
  by (auto simp: le-less mono-iff-le-Suc)
  with lift-Suc-mono-less-iff[of f] have *: ∀ n m. n < m → f n < f m
  by auto
  have inj f
  proof (intro injI)
  fix x y
  assume f x = f y
then show \( x = y \)
  by (cases \( x \) \( y \) rule: linorder-cases) (auto dest: *)
qed

with finite (range \( f \)) have finite (UNIV::nat set)
  by (rule finite-imageD)
then show False by simp

qed

then obtain \( n \) where \( n: f n = f (\text{Suc} \ n) \)
define \( N \) where \( N = (\text{LEAST} \ n. \ f n = f (\text{Suc} \ n)) \)
have \( f N = f (\text{Suc} \ N) \)
  unfolding \( N \)-def using \( n \) by (rule LeastI)
show \(?thesis\)
proof (intro exI [of - \( N \)])
  conjI allI impI
fix \( n \)
assume \( \text{N} \leq n \)
then have \( \forall \( m \). \text{N} \leq m \Longrightarrow m \leq n \Longrightarrow f m = f N \)
proof (induct rule: dec-induct)
  case base
  then show \(?case\) by simp
next
  case (step \( n \))
  then show \(?case\)
    using eq [rule-format, of \( n - 1 \)] \( N \)
    by (cases \( n \)) (auto simp add: le-Suc-eq)
qed
from this [of \( n \)] \(< \text{N} \leq n \)> show \( f \text{N} = f \text{n} \) by auto
next
fix \( m \) \( n \) :: nat
assume \( \text{m} < \text{n} \) \( \text{n} \leq \text{N} \)
then show \( f \text{m} < f \text{n} \)
proof (induct rule: less-Suc-induct)
  case (1 \( i \))
  then have \( i < \text{N} \) by simp
  then have \( f i \neq f (\text{Suc} \ i) \)
    unfolding \( \text{N} \)-def by (rule not-less-Least)
  with \( \langle \text{mono} \text{\text{f}} \rangle \) show \(?case\) by (simp add: mono-iff-le-Suc less-le)
next
  case 2
  then show \(?case\) by simp
qed

lemma finite-mono-strict-prefix-implies-finite-fixpoint:
fixes \( f :: \text{nat} \Rightarrow 'a \text{ set} \)
assumes \( S : \forall i. f i \subseteq S \ \text{finite} \ S \)
and \( \exists N. (\forall n \leq N. \forall m \leq N. m < n \Longrightarrow f m \subset f n) \ \land \ (\forall n \geq N. f N = f n) \)
shows \( f (\text{card} \ S) = (\bigcup n. f n) \)
proof –
from ex obtain \( N \) where \( \text{inj} \): \( \forall m. \ n \leq N \implies m \leq N \implies m < n \implies f m \subset f n \)
and \( \text{eq} : \forall n \geq N. \ f N = f n \)
by atomize auto
have \( i \leq N \implies i \leq \text{card} (f i) \) for \( i \)
proof (induct \( i \))
  case 0
  then show \( ?\text{case} \) by simp
next
  case (\text{Suc} \( i \))
  with \( \text{inj} \) [of \( \text{Suc} \ i \) \( i \)] have \( (f i) \subset (f (\text{Suc} \ i)) \) by auto
moreover have \( \text{finite} (f (\text{Suc} \ i)) \) using \( S \) by (rule finite-subset)
ultimately have \( \text{card} (f i) < \text{card} (f (\text{Suc} \ i)) \) by (intro psubset-card-mono)
with \( \text{Suc} \ \text{inj} \) show \( ?\text{case} \) by auto
qed
then have \( N \leq \text{card} (f N) \) by simp
also have \( \ldots \leq \text{card} S \) using \( S \) by (intro card-mono)
finally have \( \exists: f (\text{card} S) = f N \) using \( \text{eq} \) by auto
moreover have \( \bigcup (\text{range} f) \subseteq f N \)
proof clarify
  fix \( x \) \( n \)
  assume \( x \in f n \)
  with \( \text{eq inj} \) [of \( N \)] show \( x \in f N \)
  by (cases \( n < N \)) (auto simp: not-less)
qed
ultimately show \( ?\text{thesis} \)
  by auto
qed

25.7 More on injections, bijections, and inverses

locale bijection =
  fixes \( f :: 'a \Rightarrow 'a \)
  assumes bij: \( \text{bij} f \)
begin

lemma bij-inv: \( \text{bij} \) (\( \text{inv} f \))
  using bij by (rule bij-imp-bij-inv)

lemma surj \([\text{simp}]\): \( \text{surj} f \)
  using bij by (rule bij-is-surj)

lemma inj: \( \text{inj} f \)
  using bij by (rule bij-is-inj)

lemma surj-inv \([\text{simp}]\): \( \text{surj} \) (\( \text{inv} f \))
  using inj by (rule inj-imp-surj-inv)

lemma inj-inv: \( \text{inj} \) (\( \text{inv} f \))
using surj by (rule surj-imp-inj-inv)

lemma eqI: \( f \ a = f \ b \Rightarrow a = b \)
using inj by (rule injD)

lemma eq-iff [simp]: \( f \ a = f \ b \iff a = b \)
by (auto intro: eqI)

lemma eq-invI: \( \text{inv} f \ a = \text{inv} f \ b \Rightarrow a = b \)
using inj-inv by (rule injD)

lemma eq-inv-iff [simp]: \( \text{inv} f \ a = \text{inv} f \ b \iff a = b \)
by (auto intro: eq-invI)

lemma inv-left [simp]: \( \text{inv} f \ (f \ a) = a \)
using inj by (simp add: inv-f-eq)

lemma inv-comp-left [simp]: \( \text{inv} f \circ f = \text{id} \)
by (simp add: fun-eq-iff)

lemma inv-right [simp]: \( f \ (\text{inv} f \ a) = a \)
using surj by (simp add: surj-f-inv-f)

lemma inv-comp-right [simp]: \( f \circ \text{inv} f \ = \text{id} \)
by (simp add: fun-eq-iff)

lemma inv-comp-inv [simp]: \( \text{inv} f \ a = \text{inv} f \ b \iff a = b \)
by auto

lemma infinite-imp-bij-betw:
assumes infinite: \( \neg \text{finite} \ A \)
shows \( \exists h. \text{bij-betw} \ h \ A \ (A - \{a\}) \)
proof (cases \( a \in A \))
case False
then have \( A - \{a\} = A \) by blast
then show \( ?\text{thesis} \)
  using bij-betw-id[of A] by auto
next
case True
with infinite have \( \neg \text{finite} \ (A - \{a\}) \) by auto
with infinite-iff-countable-subset[of A - \{a\}] obtain \( f : \text{nat} \Rightarrow \{a\} \) where \( \text{inj} f \) and \( f \ : \text{UNIV} \subseteq A - \{a\} \) by blast
define \( g \) where \( g \ n = (\text{if} \ n = 0 \ \text{then} \ a \ \text{else} \ f \ (\text{Suc} \ n)) \) for \( n \)
define \( A' \) where \( A' = g \cdot \text{UNIV} \)
have \(*\) \(\forall y, f y \neq a\) using \(f\) by blast
have \(3\): \(\text{inj-on } g \ UNIV \land g \cdot \text{UNIV} \subseteq A \land a \in g \cdot \text{UNIV}\)
using \(\text{inj } f\) \(\ast\) unfolding \(\text{inj-on-def } g\)-def
by (auto simp add: True image-subset-iff)
then have \(4\): \(\text{bij-betw } g \ UNIV A' \land a \in A' \land A' \subseteq A\)
using \(\text{inj-on-imp-bij-betw[of } g\) by (auto simp: A'-def)
then have \(5\): \(\text{bij-betw } (\text{inv } g \cdot A') \ UNIV\)
by (auto simp add: bij-betw-inv-into)
from \(3\) obtain \(n\) where \(n\): \(g n = a\) by auto
have \(6\): \(\text{bij-betw } g \ (\text{UNIV} - \{n\}) (A' - \{a\})\)
by (rule bij-betw-subset) (use \(3\) \(4\) \(n\) in \(\text{auto simp: image-set-diff A'-def}\))
define \(v\) where \(v m = (\text{if } m < n \text{ then } m \text{ else } \text{Suc } m)\) for \(m\)
have \(m < n \lor m = n\) if \(\forall k. k < n \lor m \neq \text{Suc } k\) for \(m\)
using that \([\text{of } m-1]\) by auto
then have \(7\): \(\text{bij-betw } v \ UNIV (\text{UNIV} - \{n\})\)
unfolding \(\text{bij-betw-def } \text{inj-on-def } v\)-def by auto
define \(h'\) where \(h' = g \circ v \circ (\text{inv } g)\)
with \(5\) \(6\) \(7\) have \(8\): \(\text{bij-betw-def } \text{inj-on-def } v\)-def by auto
by (auto simp add: bij-betw-trans)
define \(b\) where \(h b = (\text{if } b \in A' \text{ then } h' b \text{ else } b)\) for \(b\)
with \(8\) have \(\text{bij-betw } h\): \(A' (A' - \{a\})\)
using \(\text{bij-betw-cong[of } A' h\) by auto
moreover
have \(\forall b \in A - A'. h b = b\) by (auto simp: h-def)
then have \(\text{bij-betw } h\): \(A - A' (A - A')\)
using \(\text{bij-betw-cong[of } A - A' h \text{ id}\) \(\text{bij-betw-id[of } A - A'\) by auto
moreover
from \(4\) have \(A' \cap (A - A') = \{\} \land A' \cup (A - A') = A\) \(\land\)
\(((A' - \{a\}) \cap (A - A') = \{\} \land (A' - \{a\}) \cup (A - A') = A - \{a\})\)
by blast
ultimately have \(\text{bij-betw } h\): \(A - \{a\}\)
using \(\text{bij-betw-combine[of } h A' A' - \{a\} A - A' A - A'\) by simp
then show \(\neg \text{thesis}\) by blast
qed

lemma \(\text{infinite-imp-bij-betw2}\):
assumes \(\neg \text{finite } A\)
shows \(\exists h. \text{bij-betw } h A (A \cup \{a\})\)
proof (cases \(a \in A\))
case \(\text{True}\)
then have \(A \cup \{a\} = A\) by blast
then show \(\neg \text{thesis}\) using \(\text{bij-betw-id[of } A\) by auto
next
case \(\text{False}\)
let \(\forall A' = A \cup \{a\}\)
from \(\text{False}\) have \(A = \exists A' - \{a\}\) by blast
moreover from \(\text{assms}\) have \(\neg \text{finite } A'\) by auto
ultimately obtain \(f\) where \(\text{bij-betw } f\) \(\forall A' A\)
using \(\text{infinite-imp-bij-betw[of } ?A' a\) by auto
then have bij-betw (inv-into ?A' f) A ?A' by (rule bij-betw-inv-into)
then show ?thesis by auto
qed

lemma bij-betw-inv-into-left: bij-betw f A A' =⇒ a ∈ A =⇒ inv-into A f (f a) = a
unfolding bij-betw-def by clarify (rule inv-into-f-f)

lemma bij-betw-inv-into-right: bij-betw f A A' =⇒ a' ∈ A' =⇒ f (inv-into A f a') = a'
unfolding bij-betw-def using f-inv-into-f by force

lemma bij-betw-inv-into-subset:
bij-betw f A A' =⇒ B ⊆ A =⇒ f ' B = B' =⇒ bij-betw (inv-into A f) B' B
by (auto simp: bij-betw-def intro: inj-on-inv-into)

25.8 Specification package – Hilbertized version

ML-file ⟨Tools/choice-specification.ML⟩

25.9 Complete Distributive Lattices – Properties depending on Hilbert Choice

context complete-distrib-lattice
begin

lemma Sup-Inf:⨆ (Inf ' A) = ⨂ (Sup ' {f ' A | f. ∀B∈A. f B ∈ B})
proof (rule order.antisym)
  show ⨆ (Inf ' A) ≤ ⨂ (Sup ' {f ' A | f. ∀B∈A. f B ∈ B})
    using Inf-lower2 Sup-upper
    by (fastforce simp add: intro: SUP-least INF-greatest)
next
  show ⨂ (Sup ' {f ' A | f. ∀B∈A. f B ∈ B}) ≤ ⨆ (Inf ' A)
proof (simp add: Inf-Sup, rule SUP-least, simp, safe)
  fix f
  assume ∀ Y. (∃ f. Y = f ' A ∧ (∀ Y∈A. f Y ∈ Y)) =⇒ f Y ∈ Y
  then have B: ⨂ F. (∀ Y∈A. F Y ∈ Y) =⇒ ∃ Z∈A. f (F ' A) = F Z
    by auto
  show ⨂ (f ' { f ' A | f. ∀ Y∈A. f Y ∈ Y}) ≤ ⨆ (Inf ' A)
proof (cases Z∈A. ⨂ (f ' { f ' A | f. ∀ Y∈A. f Y ∈ Y}) ≤ Inf Z)
  case True
    from this obtain Z where [simp]: Z ∈ A and A: ⨂ (f ' { f ' A | f. ∀ Y∈A. f Y ∈ Y}) ≤ Inf Z
    by blast
    have B: ... ≤ ⨆ (Inf ' A)
      by (simp add: SUP-upper)
    from A and B show ?thesis
THEORY "Hilbert-Choice"

by simp
next
case False
then have X: \( \bigwedge Z . Z \in A \implies \exists x . x \in Z \land \neg \bigcap (f : \{ f : A | f. \forall Y \in A. f Y \in Y \}) \leq x \)
  using Inf-greatest by blast
define F where F = (\( \lambda Z . \) SOME x . x \( \in \) Z \( \land \) \( \neg \) d (f (\{ f : A | f. \forall Y \in A. f Y \in Y \})) \leq x \))
have C: \( \bigwedge Y . Y \in A \implies F Y \in Y \)
  using X by (simp add: F-def, rule someI2-ex, auto)
have E: \( \bigwedge Y . Y \in A \implies \neg \bigcap (f : \{ f : A | f. \forall Y \in A. f Y \in Y \}) \leq F Y \)
  using X by (simp add: F-def, rule someI2-ex, auto)
from C and B obtain Z where D: Z \( \in \) A \and Y: f (\{ f : A \}) = F Z
  by blast
from E and D have W: \( \neg \bigcap (f : \{ f : A | f. \forall Y \in A. f Y \in Y \}) \leq F Z \)
  by simp
have \( \bigcap (f : \{ f : A | f. \forall Y \in A. f Y \in Y \}) \leq f (\{ f : A \}) \)
  using C by (blast intro: INF-greatest)
with W Y show \?thesis
  by simp
qed

lemma dual-complete-distrib-lattice:
  class.complete-distrib-lattice sup Inf Sup Inf sup (\( \geq \)) (\( \succ \)) inf \( \top \) \( \bot \)
  by (simp add: class.complete-distrib-lattice.intro [OF dual-complete-lattice]
  class.complete-distrib-lattice-axioms-def Sup-Inf)

lemma sup-Inf: a \( \cup \) \( \bigcap \) B = \( \bigcap \) (a \( \cup \) B)
proof (rule order.antisym)
  show a \( \cup \) \( \bigcap \) B \( \leq \) \( \bigcap \) (a \( \cup \) B)
    using Inf-lower sup mono by (fastforce intro: INF-greatest)
next
  have \( \bigcap \) (a \( \cup \) B) \( \leq \) \( \bigcap \) (Sup \( \{ \{ f \{ a \}, f B \} | f. f \{ a \} = a \land f B \in B \})
    by (rule INF-greatest, auto simp add: INF-lower)
  also have \( \bigcap \) (a \( \cup \) B) = \( \bigcap \) (\( \bigcup \) (a \( \cup \) B))
    by (unfold Sup-Inf, simp)
  finally show \( \bigcap \) (a \( \cup \) B) \( \leq \) a \( \cup \) \( \bigcap \) B
    by simp
qed

lemma inf-Sup: a \( \cap \) \( \bigcup \) B = \( \bigcup \) (a \( \cap \) B)
proof (rule complete-distrib-lattice)
  using dual-complete-distrib-lattice
by (rule complete-distrib-lattice.sup-Inf)

lemma INF-SUP: (\( \bigcap \) y. \( \bigcup \) x. P x y) = (\( \bigcup \) y. \( \bigcap \) x. P (f x) x)
proof (rule order.antisym)
  show (SUP x. INF y. P x y) \( \leq \) \( \bigcup \) y. \( \bigcap \) x. P x y
by (simp add: INF-lower sup mono)
by (rule SUP-least, rule INF-greatest, rule SUP-upper2, simp-all, rule INF-lower2, simp, blast)
next
have \( (\text{INF} \ y. \ \text{SUP} \ x. \ (\{P \ x \ y\})) \leq \text{Inf} \ (\text{Sup} \ \{\{P \ x \ y \mid x \ . \ True\} \mid y . \ True\}) \)
(is \( ?A \leq ?B \))
proof (rule INF-greatest, clarsimp)
  fix \( y \)
  have \( ?A \leq (\text{SUP} \ x. \ P \ x \ y) \)
     by (rule INF-lower, simp)
  also have \( \ldots \leq \text{Sup} \ \{\text{uu} . \ \exists x. \ uu = P \ x \ y\} \)
     by (simp add: full-SetCompr-eq)
finally show \( ?A \leq \text{Sup} \ \{\text{uu} . \ \exists x. \ uu = P \ x \ y\} \)
     by simp
qed
also have \( \ldots \leq (\text{SUP} \ x. \ \text{INF} \ y. \ P \ (x \ y) \ y) \)
proof (subst Inf-Sup, rule SUP-least, clarsimp)
  fix \( f \)
  assume \( A: \forall Y. \ (\exists y. \ Y = \{\text{uu} . \ \exists x. \ uu = P \ x \ y\}) \longrightarrow f \ Y \in Y \)

  have \( \prod(f \ \{\text{uu} . \ \exists y. \ uu = \{\text{uu} . \ \exists x. \ uu = P \ x \ y\}\}) \leq
       (\prod y. \ P \ (\text{SOME} \ x. \ f \ \{P \ x \ y \mid x . \ True\} = P \ x \ y) \ y) \)
proof (rule INF-greatest, clarsimp)
  fix \( y \)
  have \( \text{INF} \ x \in \{\text{uu} . \ \exists y. \ uu = \{\text{uu} . \ \exists x. \ uu = P \ x \ y\}\}. \ f \ x \) \leq \( f \ \{\text{uu} . \ \exists x. \ uu = P \ x \ y\} \)
    by (rule INF-lower, blast)
  also have \( \ldots \leq P \ (\text{SOME} \ x. \ f \ \{\text{uu} . \ \exists x. \ uu = P \ x \ y\} = P \ x \ y) \ y \)
    by (rule SOME2-ex) (use \( A \) in auto)
finally show \( \prod(f \ \{\text{uu} . \ \exists y. \ uu = \{\text{uu} . \ \exists x. \ uu = P \ x \ y\}\}) \leq
                  P \ (\text{SOME} \ x. \ f \ \{\text{uu} . \ \exists x. \ uu = P \ x \ y\} = P \ x \ y) \ y \)
    by simp
qed
also have \( \ldots \leq (\text{SUP} \ x. \ \text{INF} \ y. \ P \ (x \ y) \ y) \)
by (rule SUP-upper, simp)
finally show \( \prod(f \ \{\text{uu} . \ \exists y. \ uu = \{\text{uu} . \ \exists x. \ uu = P \ x \ y\}\}) \leq (\bigcup x. \ \prod y. \ P \ (x \ y) \ y) \)
    by simp
qed

lemma INF-SUP-set: \( (\prod B \in A. \ \bigcup(g \ f \ B)) = \bigcup B \in \{f \ A \mid f. \ \forall C \in A. \ f \ C \in C\}. \ \prod(g \ f \ B) \)
(is \( = (\bigcup B \in ?F. \ -) \))
proof (rule order.antisym)
  have \( \prod ((g \ f) \ A) \leq \bigcup (g \ f \ B) \) if \( \forall B. \ B \in A \Longrightarrow f \ B \in B \in A \) for \( f \ B \)
  using that by (auto intro: SUP-upper2 INF-lower2)
then show \( (\bigcup x \in ?F. \ \prod a \in x. \ g \ a) \leq (\prod x \in A. \ \bigcup a \in x. \ g \ a) \)
by (auto intro!: SUP-least INF-greatest simp add: image-comp)

next
show \((\bigsqcup x \in A. \bigcap a \in x. \, g \, a) \leq (\bigsqcup x \in ?F. \bigcap a \in x. \, g \, a)\)
proof (cases \(\{\} \in A\))
  case True
  then show \(?\)thesis
  by (rule INF-lower2 simp-all)
next
  case False
  \{fix \(x\)
  have \((\bigsqcup x \in A. \bigcup x \in x. \, g \, x) \leq (\bigsqcup ai \in A. \, g \, ai \leq _\bot \, else \top)\)
  proof (cases \(x \in A\))
  case True
  then show \(?\)thesis
  by (intro INF-lower2 SUP-least SUP-upper2 auto)
  qed auto

next
then have \((\bigsqcup Y \in A. \bigcap a \in Y. \, g \, a) \leq (\bigsqcup Y \in ?F. \bigcap a \in Y. \, g \, a)\)
by (rule INF-greatest)
also have \(\ldots \leq (\bigsqcup i \in A. \, g \, i)\)
proof (rule SUP-least)
  show \((\bigsqcup B. \, if B \in A \, then \, if x \in B \, then \, g \, (x \, B) \, else \bot \, e \bot \, else \top)\)
  \leq (\bigsqcup x \in ?F. \bigcap x \in x. \, g \, x) \, for \, x\)
  proof -
  define \(G\) where \(G \equiv \lambda Y. \, if \, x \, Y \in \, Y \, then \, x \, Y \, else \, (\text{SOME} \, x. \, x \in Y)\)
  have \(\forall Y \in A. \, G \, Y \in \, Y\)
  \(\text{using False some-in-eq G-def by auto}\)
  then have \(A: \, G \, ' A \in ?F\)
  by blast
  show \((\bigsqcup Y. \, if Y \in A \, then \, if x \, Y \in Y \, then \, g \, (x \, Y) \, else \bot \, e \bot \, else \top) \leq (\bigsqcup x \in ?F. \bigcap x \in x. \, g \, x)\)
  by (fastforce simp: G-def intro: SUP-upper2 [OF A] INF-greatest INF-lower2)
  qed
  \qed
  finally show \(?\)thesis by simp
  \qed
\qed

lemma SUP-INF: \((\bigcup i \in A. \, \bigcap x. \, P \, x \, y) = (\bigsqcup x. \, \bigcup y. \, P \, (x \, y) \, y)\)
\using dual-complete-distrib-lattice
by (rule complete-distrib-lattice.INF-SUP)

lemma SUP-INF-set: \((\bigsqcup x \in A. \, \bigcap g \, ' x) = (\bigcup x \in \{f \, ' A \, | \, f \, \forall Y \in A. \, f \, Y \in Y\}. \bigcup (g \, ' x)\))
THEORY “Hilbert-Choice”

using dual-complete-distrib-lattice
by (rule complete-distrib-lattice.INF-SUP-set)

end

context complete-distrib-lattice
begin

lemma sup-INF: a ⊔ (∏ b ∈ B. f b) = (∏ b ∈ B. a ⊔ f b)
by (simp add: sup-Inf image-comp)

lemma inf-SUP: a ∩ (∐ b ∈ B. f b) = (∐ b ∈ B. a ∩ f b)
by (simp add: inf-Sup image-comp)

lemma Inf-sup: (∏ B ⊔ a) = (∏ b ∈ B. b ⊔ a)
by (simp add: sup-Inf sup-commute)

lemma Sup-inf: (∐ B ⊓ a) = (∐ b ∈ B. b ⊓ a)
by (simp add: inf-Sup inf-commute)

lemma INF-sup: (∏ b ∈ B. f b) ⊔ a = (∏ b ∈ B. f b ⊔ a)
by (simp add: sup-INF sup-commute)

lemma SUP-inf: (∐ b ∈ B. f b) ⊓ a = (∐ b ∈ B. f b ⊓ a)
by (simp add: inf-SUP inf-commute)

lemma Inf-sup-eq-top-iff: (∏ B ⊔ a = ⊤) <-> (∀ b ∈ B. b ⊔ a = ⊤)
by (simp only: Inf-sup INF-top-conv)

lemma Sup-inf-eq-bot-iff: (∐ B ⊓ a = ⊥) <-> (∀ b ∈ B. b ⊓ a = ⊥)
by (simp only: Sup-inf SUP-bot-conv)

lemma INF-sup-distrib2: (∏ a ∈ A. f a) ⊔ (∏ b ∈ B. g b) = (∏ a ∈ A. ∏ b ∈ B. f a ⊔ g b)
by (subst INF-commute) (simp add: sup-INF INF-sup)

lemma SUP-inf-distrib2: (∐ a ∈ A. f a) ∩ (∐ b ∈ B. g b) = (∐ a ∈ A. ∐ b ∈ B. f a ∩ g b)
by (subst SUP-commute) (simp add: inf-SUP SUP-inf)

end

instantiation set :: (type) complete-distrib-lattice
begin

instance proof (standard, clarsimp)
fix A :: "('a set) set"
fix x::'a
assume A: ∀ S ∈ A. ∃ x ∈ S. x ∈ X
define $F$ where $F \equiv \lambda Y. \text{SOME} X. Y \in A \land X \in Y \land x \in X$

have $(\forall S \in F \cdot A. x \in S)$
    using $A$ unfolding $F$-def by (fastforce intro: someI2-ex)

moreover have $\forall Y \in A. F Y \in Y$
    using $A$ unfolding $F$-def by (fastforce intro: someI2-ex)
then have $\exists f. F \cdot A = f \cdot A \land (\forall Y \in A. f Y \in Y)$
    by blast

ultimately show $\exists X. (\exists f. X = f \cdot A \land (\forall Y \in A. f Y \in Y)) \land (\forall S \in X. x \in S)$
    by auto

qed

end

instance $set :: (\text{type})$ complete-boolean-algebra ..

instantiation $fun :: (\text{type}, \text{complete-distrib-lattice})$ complete-distrib-lattice
begin
instance by standard ($\text{simp add: le-fun-def INF-SUP-set image-comp}$)
end

instance $fun :: (\text{type}, \text{complete-boolean-algebra})$ complete-boolean-algebra ..

class complete-linorder
begin
subclass complete-distrib-lattice
proof (standard, rule ccontr)
    fix $A :: 'a \text{ set set}$
    let $?F = \{f \cdot A \mid f. \forall Y \in A, f Y \in Y\}$
    assume $? \neg \exists z \cdot (\exists f. f \cdot A > z \land z > (\exists f. f \cdot A))$
    then have $C: (\exists f. f \cdot A > z \land z > (\exists f. f \cdot A))$
        by blast
    then have $B: \forall Y. Y \in A \implies \exists k \in Y. z < k$
        using local.less-Sup-iff (force dest: less-INF-D)
    define $G$ where $G \equiv \lambda Y. \text{SOME} k . k \in Y \land z < k$
    have $E: \lambda Y. Y \in A \implies G Y \in Y$
        using $B$ unfolding $G$-def by (fastforce intro: someI2-ex)
    have $z \leq \text{Inf} (G \cdot A)$
        proof (rule INF-greatest)
            show $\forall Y. Y \in A \implies z \leq G Y$
                using $B$ unfolding $G$-def by (fastforce intro: someI2-ex)
            qed
        also have $\ldots \leq (\exists f. f \cdot A)$
            by (rule SUP-upper) (use $E$ in blast)
finally have $z \leq \bigsqcup (\text{Inf } \ ?F)$
  by simp

with $X$ show $\text{thesis}$
  using local.not-less by blast
next
  case False
  have $B: \bigwedge Y. \ Y \in A \Rightarrow \exists k \in Y . \ \bigsqcup (\text{Inf } \ ?F) < k$
    using $C$ local.less-Sup-iff by (force dest: less-INF-D)
  define $G$ where $G \equiv \lambda Y. \ \text{SOME } k \in Y \land \bigsqcup (\text{Inf } \ ?F) < k$
  have $E: \bigwedge Y. \ Y \in A \Rightarrow G \ Y \in Y$
    using $B$ unfolding $G$-def by (fastforce intro: someI2-ex)
  have $\bigwedge Y. \ Y \in A \Rightarrow \prod (\text{Sup } \ A) \leq G \ Y$
    using $B$ False local.leI unfolding $G$-def by (fastforce intro: someI2-ex)
  then have $\prod (\text{Sup } \ A) \leq \text{Inf } (G \ A)$
    by (simp add: local.INF-greatest)
  also have $\text{Inf } (G \ A) \leq \bigsqcup (\text{Inf } \ ?F)$
    by (rule SUP-upper) (use $E$ in blast)
  finally have $\prod (\text{Sup } \ A) \leq \bigsqcup (\text{Inf } \ ?F)$
    by simp
  with $C$ show $\text{thesis}$
    using not-less by blast
qed
qed
end

26  Zorn’s Lemma and the Well-ordering Theorem

theory Zorn
  imports Order-Relation Hilbert-Choice
begin

26.1  Zorn’s Lemma for the Subset Relation

26.1.1  Results that do not require an order

Let $P$ be a binary predicate on the set $A$.

locale pred-on =
  fixes $A :: \text{'}a$ set
  and $P :: \text{'}a \Rightarrow \text{'}a \Rightarrow \text{bool}$ (infix $\sqsubseteq$ 50)
begin

abbreviation $P_{eq} :: \text{'}a \Rightarrow \text{'}a \Rightarrow \text{bool}$ (infix $\subseteq$ 50)
  where $x \sqsubseteq y \equiv P_{eq} \ x \ y$

A chain is a totally ordered subset of $A$. 
THEORY "Zorn"

definition chain :: 'a set ⇒ bool
  where chain C ⇔ C ⊆ A ∧ (∀ x∈C. ∀ y∈C. x ⊆ y ∨ y ⊆ x)

We call a chain that is a proper superset of some set X, but not necessarily a chain itself, a superchain of X.

abbreviation superchain :: 'a set ⇒ 'a set ⇒ bool (infix <c)
  where X <c C ⇔ chain C ∧ X ⊂ C

A maximal chain is a chain that does not have a superchain.

definition maxchain :: 'a set ⇒ bool
  where maxchain C ⇔ chain C ∧ (∄ S. C <c S)

We define the successor of a set to be an arbitrary superchain, if such exists, or the set itself, otherwise.

definition suc :: 'a set ⇒ 'a set
  where suc C = (if ¬ chain C ∨ maxchain C then SOME D. C <c D)

lemma chainI [Pure.intro?): C ⊆ A ⇒ (∀ x y. x ∈ C ⇒ y ∈ C ⇒ x ⊆ y ∨ y ⊆ x) ⇒ chain C
  unfolding chain-def by blast

lemma chain-total [simp]: chain C ⇒ x ∈ C ⇒ y ∈ C ⇒ x ⊆ y ∨ y ⊆ x
  by (simp add: chain-def)

lemma not-chain-suc [simp]: ¬ chain X ⇒ suc X = X
  by (simp add: suc-def)

lemma maxchain-suc [simp]: maxchain X ⇒ suc X = X
  by (simp add: suc-def)

lemma suc-subset [simp]: X ⊆ suc X
  by (auto simp: suc-def maxchain-def intro: someI2)

lemma chain-empty [simp]: chain {}
  by (auto simp: chain-def)

lemma not-maxchain-Some: chain C ⇒ ¬ maxchain C ⇒ C <c (SOME D. C <c D)
  by (rule someI-ex) (auto simp: maxchain-def)

lemma suc-not-equals: chain C ⇒ ¬ maxchain C ⇒ suc C ≠ C
  using not-maxchain-Some by (auto simp: suc-def)

lemma subset-suc:
  assumes X ⊆ Y
  shows X ⊆ suc Y
  using assms by (rule subset-trans) (rule suc-subset)

We build a set C that is closed under applications of suc and contains the
union of all its subsets.

**inductive-set** *suc-Union-closed* \((C)\)

where

\[
suc \colon X \in C \implies suc X \in C
\]

\[
\mid \text{Union [unfolded Pow-if]: } X \in \text{Pow} C \implies \bigcup X \in C
\]

Since the empty set as well as the set itself is a subset of every set, \(C\) contains at least \(\{\} \in C\) and \(\bigcup C \in C\).

**lemma** *suc-Union-closed-empty*: \(\{} \in C\)

and *suc-Union-closed-Union*: \(\bigcup C \in C\)

**using** Union [of \(\{}\)] and Union [of \(C\)] by simp-all

Thus closure under *suc* will hit a maximal chain eventually, as is shown below.

**lemma** *suc-Union-closed-induct* [consumes 1, case-names suc Union, induct pred: *suc-Union-closed*]:

assumes \(X \in C\)

and \(\forall X. X \subseteq C \implies \forall x \in X. Q x \implies Q (\bigcup X)\)

shows \(Q X\)

using assms by induct blast+

**lemma** *suc-Union-closed-cases* [consumes 1, case-names suc Union, cases pred: *suc-Union-closed*]:

assumes \(X \in C\)

and \(\forall Y. X = suc Y \implies Y \in C \implies Q\)

and \(\forall Y. X = \bigcup Y \implies Y \subseteq C \implies Q\)

shows \(Q\)

using assms by cases simp-all

On chains, *suc* yields a chain.

**lemma** *chain-suc*:

assumes \(\text{chain} X\)

shows \(\text{chain} (suc X)\)

using assms

by (cases ~ chain X \lor maxchain X) (force simp: suc-def dest: not-maxchain-Some)+

**lemma** *chain-sucD*:

assumes \(\text{chain} X\)

shows \(suc X \subseteq A \land \text{chain} (suc X)\)

**proof** –

from ⟨chain X⟩ have *: chain (suc X)

by (rule chain-suc)

then have suc X \(\subseteq A\)

unfolding chain-def by blast

with * show ?thesis by blast

qed
lemma suc-Union-closed-total':
assumes $X \in \mathcal{C}$ and $Y \in \mathcal{C}$
and $\ast \forall Z. Z \in \mathcal{C} \implies Z \subseteq Y \implies Z = Y \lor \text{suc } Z \subseteq Y$
shows $X \subseteq Y \lor \text{suc } Y \subseteq X$
using $\langle X \in \mathcal{C} \rangle$
proof (induct)
case (suc $X$)
with $\ast$ show $\text{thesis}$ by (blast del: subsetI intro: subset-suc)
next
case Union
then show $\text{thesis}$ by blast
qed

lemma suc-Union-closed-subsetD:
assumes $Y \subseteq X$ and $X \in \mathcal{C}$ and $Y \in \mathcal{C}$
shows $X = Y \lor \text{suc } Y \subseteq X$
using assms(2,3,1)
proof (induct arbitrary: $Y$)
case (suc $X$)
note $\ast = \langle \forall Y. Y \in \mathcal{C} \implies Y \subseteq X \implies X = Y \lor \text{suc } Y \subseteq X \rangle$
with suc-Union-closed-total' $[OF \langle Y \in \mathcal{C} \rangle \langle X \in \mathcal{C} \rangle]$
have $Y \subseteq X \lor \text{suc } X \subseteq Y$ by blast
then show $\text{thesis}$
proof
assume $Y \subseteq X$
with $\ast$ and $\langle Y \in \mathcal{C} \rangle$ subset-suc show $\text{thesis}$
by fastforce
next
assume suc $X \subseteq Y$
with $\langle Y \subseteq \text{suc } X \rangle$ show $\text{thesis}$ by blast
qed
next
case (Union $X$)
show $\text{thesis}$
proof (rule ccontr)
assume $\neg \text{thesis}$
with $\langle Y \subseteq \bigcup X \rangle$ obtain $x y z$
where $\neg \text{suc } Y \subseteq \bigcup X$
and $x \in X$ and $y \in x$ and $y \notin Y$
and $z \in \text{suc } Y$ and $\forall x \in X. z \notin x$ by blast
with $\langle X \subseteq \mathcal{C} \rangle$ have $x \in \mathcal{C}$ by blast
from Union and $\langle x \in X \rangle$ have $\ast: \forall y. y \in \mathcal{C} \implies y \subseteq x \implies x = y \lor \text{suc } y \subseteq x$
by blast
with suc-Union-closed-total' $[OF \langle Y \in \mathcal{C} \rangle \langle x \in \mathcal{C} \rangle]$ have $Y \subseteq x \lor \text{suc } x \subseteq Y$
by blast
then show False
proof
assume $Y \subseteq x$
THEORY "Zorn"

with * [OF \( \cdot \in C \mapsto y \not\in Y \cdot x \in X \cdot \neg \text{suc } Y \subseteq \bigcup X \)] show False by blast

next
  assume suc x \subseteq Y
  with \( y \not\in Y \cdot \text{suc-subset } y \in x \) show False by blast
qed

The elements of \( C \) are totally ordered by the subset relation.

lemma suc-Union-closed-total:
  assumes \( X \in C \) and \( Y \in C \)
  shows \( X \subseteq Y \lor Y \subseteq X \)
proof (cases \( \forall Z \in C. \ Z \subseteq Y \longrightarrow Z = Y \lor \text{suc } Z \subseteq Y \))
  case True
  with suc-Union-closed-total' [OF assms]
  have \( X \subseteq Y \lor \text{suc } Y \subseteq X \) by blast
  with suc-subset [of Y] show ?thesis by blast
next
  case False
  then obtain \( Z \) where \( Z \in C \) and \( Z \subseteq Y \) and \( Z \not= Y \) and \( \neg \text{suc } Z \subseteq Y \)
  by blast
  with suc-Union-closed-subsetD and \( Y \in C \) show ?thesis by blast
qed

Once we hit a fixed point w.r.t. \( \text{suc} \), all other elements of \( C \) are subsets of this fixed point.

lemma suc-Union-closed-suc:
  assumes \( X \in C \) and \( Y \in C \) and \( \text{suc } Y = Y \)
  shows \( X \subseteq Y \)
  using \( \langle X \in C \rangle \)
proof induct
  case \( \text{suc } X \)
  with \( Y \in C \) and suc-Union-closed-subsetD have \( X = Y \lor \text{suc } X \subseteq Y \)
  by blast
  then show ?case
  by (auto simp: \( \text{suc } Y = Y \))
next
  case Union
  then show ?case by blast
qed

lemma eq-suc-Union:
  assumes \( X \in C \)
  shows \( \text{suc } X = X \leftrightarrow X = \bigcup C \)
  (is ?lhs \longleftrightarrow ?rhs)
proof
  assume ?lhs
then have $\bigcup C \subseteq X$
  by (rule suc-Union-closed-suc [OF suc-Union-closed-Union \(X \in C\)])
with \(X \in C\) show \(?rhs\)
  by blast
next
  from \(X \in C\) have \(\text{suc } X \in C\) by (rule suc)
  then have \(\text{suc } X \subseteq \bigcup C\)
  by blast
  moreover assume \(?rhs\)
  ultimately have \(\text{suc } X \subseteq X\) by simp
  moreover have \(X \subseteq \text{suc } X\)
  by (rule suc-subset)
  ultimately show \(?lhs\) ..
qed

lemma \textit{suc-in-carrier}:
  assumes \(X \subseteq A\)
  shows \(\text{suc } X \subseteq A\)
  using \(\text{assms}\)
  by (cases \(\neg \text{chain } X \lor \text{maxchain } X\))
  (auto dest: chain-sucD)

lemma \textit{suc-Union-closed-in-carrier}:
  assumes \(X \in C\)
  shows \(X \subseteq A\)
  using \(\text{assms}\)
  by induct (auto dest: suc-in-carrier)

All elements of \(C\) are chains.

lemma \textit{suc-Union-closed-chain}:
  assumes \(X \in C\)
  shows \(\text{chain } X\)
  using \(\text{assms}\)
proof induct
  case (suc \(X\))
  then show \(?case\)
    using not-maxchain-Some by (simp add: suc-def)
next
  case (\(\text{Union } X\))
  then have \(\bigcup X \subseteq A\)
    by (auto dest: suc-Union-closed-in-carrier)
  moreover have \(\forall x \in \bigcup X. \forall y \in \bigcup X. \ x \subseteq y \lor y \subseteq x\)
proof (intro ballI)
  fix \(x\) \(y\)
  assume \(x \in \bigcup X\) and \(y \in \bigcup X\)
  then obtain \(u\) \(v\) where \(x \in u\) and \(u \in X\) and \(y \in v\) and \(v \in X\)
  by blast
with \(\text{Union } X\)
  have \(u \in C\) and \(v \in C\) and \(\text{chain } u\) and \(\text{chain } v\)
  by blast+
with \(\text{suc-Union-closed-total}\)
  have \(u \subseteq v \lor v \subseteq u\)
  by blast
then show \(x \subseteq y \lor y \subseteq x\)
proof
assume \( u \subseteq v \)
from \( \{ \text{chain } v \} \) show \?thesis
proof (rule chain-total)
  show \( y \in v \) by fact
  show \( x \in v \) using \( \{ u \subseteq v \} \) and \( \{ x \in u \} \) by blast
qed
next
assume \( v \subseteq u \)
from \( \{ \text{chain } u \} \) show \?thesis
proof (rule chain-total)
  show \( x \in u \) by fact
  show \( y \in u \) using \( \{ v \subseteq u \} \) and \( \{ y \in v \} \) by blast
qed
qed
qed
ultimately show \?case unfolding chain-def ..
qed

\textbf{26.1.2 Hausdorff’s Maximum Principle}

There exists a maximal totally ordered subset of \( A \). (Note that we do not require \( A \) to be partially ordered.)

\textbf{theorem} Hausdorff: \( \exists C. \maxchain C \)

proof –
  let \(?M = \bigcup C\)
  have \maxchain ?M
  proof (rule ccontr)
    assume \( \neg \?thesis \)
    then have \( \text{suc } \neg ?M \neq ?M \)
      using suc-not-equals and suc-Union-closed-chain [OF suc-Union-closed-Union]
      by simp
    moreover have \( \text{suc } ?M = ?M \)
      using eq-suc-Union [OF suc-Union-closed-Union] by simp
    ultimately show \( \text{False by contradiction} \)
      by blast
    qed
  then show \( ?thesis \)
    by blast
  qed

Make notation \( C \) available again.

\textbf{no-notation} suc-Union-closed \( (C) \)

\textbf{lemma} chain-extend: \( \text{chain } C \implies \exists z \in A \implies \forall x \in C. \ x \subseteq z \implies \text{chain } (\{ z \} \cup C) \)

\textbf{unfolding} chain-def by blast

\textbf{lemma} maxchain-imp-chain: \( \maxchain C \implies \text{chain } C \)

by (simp add: maxchain-def)
end

Hide constant \textit{pred-on.suc-Union-closed}, which was just needed for the proof of Hausforff’s maximum principle.

\texttt{hide-const pred-on.suc-Union-closed}

\texttt{lemma chain-mono:}
\begin{itemize}
  \item \texttt{assumes} \( \forall x \ y. \ x \in A \implies y \in A \implies P \ x \ y \implies Q \ x \ y \)
  \item \texttt{and} \texttt{pred-on.chain} \( A \ P \ C \)
  \item \texttt{shows} \texttt{pred-on.chain} \( A \ Q \ C \)
  \item \texttt{using} \texttt{assms unfolding pred-on.chain-def by blast}
\end{itemize}

26.1.3 Results for the proper subset relation

\texttt{interpretation subset: pred-on} \( A (\subset) \) \texttt{for} \( A \).

\texttt{lemma subset-maxchain-max:}
\begin{itemize}
  \item \texttt{assumes} \texttt{subset.maxchain} \( A \ C \)
  \item \texttt{and} \( X \in A \)
  \item \texttt{and} \( \bigcup C \subseteq X \)
  \item \texttt{shows} \( \bigcup C = X \)
\end{itemize}

\texttt{proof (rule ccontr)}
\begin{itemize}
  \item \texttt{let} \( \forall C = \{X\} \cup C \)
  \item \texttt{from} \texttt{subset.maxchain} \( A \ C \) \texttt{have} \texttt{subset.chain} \( A \ C \)
  \item \texttt{and} \( \forall S \in C \texttt{. subset.chain} A S \implies \neg C \subseteq S \)
  \item \texttt{by (auto simp: subset.maxchain-def)}
  \item \texttt{moreover have} \( \forall x \in C \texttt{. x \subseteq X} \texttt{using} \bigcup C \subseteq X \) \texttt{by auto}
  \item \texttt{ultimately have} \texttt{subset.chain} \( A \ ?C \)
  \item \texttt{using} \texttt{subset.chain-extend [of} \( A \ C \ X \) \texttt{and} \( X \in A \) \texttt{by auto}
  \item \texttt{moreover assume} \( \forall C \in \bigcup C \neq X \)
  \item \texttt{moreover from} \( \forall C \in \bigcup C \neq X \) \texttt{using} \( \bigcup C \subseteq X \) \texttt{by auto}
  \item \texttt{ultimately show False using} \( \forall C \in \bigcup C \neq X \) \texttt{by blast}
\end{itemize}
\texttt{qed}

\texttt{lemma subset-chain-def: } \( \forall A \texttt{. subset.chain} A \ C = \{C \subseteq A \land (\forall X \in C \texttt{.} \forall Y \in C \texttt{.} X \subseteq Y \vee Y \subseteq X)\} \)
\begin{itemize}
  \item \texttt{by (auto simp: subset.chain-def)}
\end{itemize}

\texttt{lemma subset-chain-insert:}
\begin{itemize}
  \item \texttt{subset.chain} \( A \texttt{ (insert} B \texttt{ B)} \iff B \in A \land (\forall X \in B \texttt{.} X \subseteq B \lor B \subseteq X) \land \texttt{subset.chain} A B \)
  \item \texttt{by (fastforce simp add: subset.chain-def)}
\end{itemize}

26.1.4 Zorn’s lemma

If every chain has an upper bound, then there is a maximal set.

\texttt{theorem subset-Zorn:}
\begin{itemize}
  \item \texttt{assumes} \( \forall C \texttt{. subset.chain} A \ C \implies \exists U \in A \texttt{.} \forall X \in C \texttt{.} X \subseteq U \)
  \item \texttt{shows} \( \exists M \in A \texttt{.} \forall X \in A \texttt{.} M \subseteq X \implies X = M \)
\end{itemize}
THEORY “Zorn” 646

proof –
  from subset.Hausdorff [of A] obtain M where subset.maxchain A M ..
  then have subset.chain A M
    by (rule subset.maxchain-imp-chain)
  with assms obtain Y where Y ∈ A and ∀ X∈M. X ⊆ Y
    by blast
  moreover have ∀ X∈A. Y ⊆ X → Y = X
  proof (intro ballI impI)
    fix X
    assume X ∈ A and Y ⊆ X
    show Y = X
    proof (rule ccontr)
      assume ¬ ?thesis
      with ⟨Y ⊆ X⟩ have ¬ X ⊆ Y by blast
      from subset.chain-extend [OF ⟨subset.chain A M⟩ ⟨X ∈ A⟩] and ⟨∀ X∈M. X ⊆ Y⟩
      have subset.chain A (⟨X⟩ ∪ M)
      using ⟨Y ⊆ X⟩ by auto
      moreover have M ⊂ ⟨X⟩ ∪ M
        using ⟨∀ X∈M. X ⊆ Y⟩ and ⟨¬ X ⊆ Y⟩ by auto
      ultimately show False
        using ⟨subset.maxchain A M⟩ by (auto simp: subset.maxchain-def)
    qed
    qed
  ultimately show ?thesis by blast
  qed

Alternative version of Zorn’s lemma for the subset relation.

lemma subset-Zorn':
  assumes ⟨C. subset.chain A C ⇒ ∪ C ∈ A⟩
  shows ∃ M∈A. ∀ X∈A. M ⊆ X → X = M
proof –
  from subset.Hausdorff [of A] obtain M where subset.maxchain A M ..
  then have subset.chain A M
    by (rule subset.maxchain-imp-chain)
  with assms have ∪ M ∈ A .
  moreover have ∀ Z∈A. ∪ M ⊆ Z → ∪ M = Z
  proof (intro ballI impI)
    fix Z
    assume Z ∈ A and ∪ M ⊆ Z
      with subset-maxchain-max [OF ⟨subset.maxchain A M⟩]
        show ∪ M = Z .
    qed
  ultimately show ?thesis by blast
  qed

26.2 Zorn’s Lemma for Partial Orders

Relate old to new definitions.
THEORY "Zorn"

definition chain-subset :: 'a set set ⇒ bool (chain≦)
  where chain≦ C ⇔ (∀ A∈C. ∀ B∈C. A ⊆ B ∨ B ⊆ A)

definition chains :: 'a set set ⇒ 'a set set
  where chains A = { C. C ⊆ A ∧ chain≦ C }

definition Chains :: (′a × ′a) set ⇒ ′a set set
  where Chains r = { C. (∀ a∈C. ∀ b∈C. (a, b) ∈ r ∨ (b, a) ∈ r) }

lemma chains-extend: c ∈ chains S =⇒ z ∈ S =⇒ (∀ x∈c. x ⊆ z =⇒ {z} ∪ c ∈ chains S )
  for z :: 'a set
  unfolding chains-def chain-subset-def by blast

lemma mono-Chains: r ⊆ s =⇒ Chains r ⊆ Chains s
  unfolding Chains-def by blast

lemma chain-subset-alt-def: chain≦ C = subset.chain UNIV C
  unfolding chain-subset-def subset.chain-def by fast

lemma chains-alt-def: chains A = { C. subset.chain A C }
  by (simp add: chains-def chain-subset-alt-def subset.chain-def)

lemma Chains-subset: Chains r ⊆ { C. pred-on.chain UNIV (λx y. (x, y) ∈ r) C }
  by (force simp add: Chains-def pred-on.chain-def)

lemma Chains-subset':
  assumes refl r
  shows { C. pred-on.chain UNIV (λx y. (x, y) ∈ r) C } ⊆ Chains r
  using assms
  by (auto simp add: Chains-def pred-on.chain-def refl-on-def)

lemma Chains-alt-def:
  assumes refl r
  shows Chains r = { C. pred-on.chain UNIV (λx y. (x, y) ∈ r) C }
  using assms Chains-subset Chains-subset' by blast

lemma Chains-relation-of:
  assumes C ∈ Chains (relation-of P A) shows C ⊆ A
  using assms unfolding Chains-def relation-of-def by auto

lemma pairwise-chain-Union:
  assumes P: ∃ S. S ∈ C =⇒ pairwise R S and chain≦ C
  shows pairwise R (∪ C)
  using chain≦ C unfolding pairwise-def chain-subset-def
  by (blast intro: P [unfolded pairwise-def, rule-format])

lemma Zorn-Lemma: ∀ C∈chains A. ∪ C ∈ A =⇒ ∃ M∈A. ∀ X∈A. M ⊆ X =⇒ X = M
THEORY "Zorn"

using subset-Zorn' [of A] by (force simp: chains-alt-def)

lemma Zorn-Lemma2: \( \forall C \in \text{chains } A \exists U \in A. \forall X \in C. X \subseteq U \implies \exists M \in A. \forall X \in A. M \subseteq X \)
using subset-Zorn [of A] by (auto simp: chains-alt-def)

26.3 Other variants of Zorn’s Lemma

lemma chainsD: \( c \in \text{chains } S \implies x \in c \implies y \in c \implies x \leq y \vee y \leq x \)
unfolding chains-def chain-subset-def by blast

lemma chainsD2: \( c \in \text{chains } S \implies c \subseteq S \)
unfolding chains-def by blast

lemma Zorns-po-lemma:
  assumes po: Partial-order r
  and u: \( \forall C \in \text{Chains } r \implies \exists u \in \text{Field } r. \forall a \in C. (m, a) \in r \implies a = m \)
proof
  have Preorder r using po by (simp add: partial-order-on-def)
  Mirror r in the set of subsets below (wrt r) elements of A.
  let ?B = \( \lambda x. r^{-1} \{ x \} \)
  let ?S = ?B ' Field r
  have \( \exists u \in \text{Field } r. \forall A \in C. A \subseteq r^{-1} \{ u \} \) (is \( \exists u \in \text{Field } r. \exists P u \))
  if 1: \( C \subseteq ?S \) and 2: \( \forall A \in C. \forall B \in C. A \subseteq B \vee B \subseteq A \) for \( C \)
proof
  let \( ?A = \{ x \in \text{Field } r. \exists M \in C. M = ?B x \} \)
  from 1 have \( C = ?B ' ?A \) by (auto simp: image-def)
  have \( ?A \in \text{Chains } r \)
proof (simp add: Chains-def, intro allI impI, elim conjE)
    fix a b
    assume a in Field r and ?B a \in C and b in Field r and ?B b \in C
    with 2 have ?B a \subseteq ?B b \vee ?B b \subseteq ?B a by auto
    then show \( (a, b) \in r \vee (b, a) \in r \)
      using \( \langle a \in \text{Field } r \rangle \) and \( \langle b \in \text{Field } r \rangle \)
      by (simp add:subset-Image1-Image1-iff)
  qed
  then obtain u where aA: \( u \in \text{Field } r \forall a \in ?A. (a, u) \in r \)
    by (auto simp: dest: u)
  have ?P u
proof auto
  fix a B assume aB: \( B \in C \ a \in B \)
  with 1 obtain x where x in Field r and B = r^{-1} \{ x \} by auto
  then show \( (a, u) \in r \)
    using aA and aB and \( \langle \text{Preorder } r \rangle \)
    unfolding preorder-on-on refl-on-on by simp (fast dest: transD)
  qed
  then show ?thesis
THEORY "Zorn"

using (u ∈ Field r) by blast

qed

then have ∀ C ∈ chains ?S. ∃ U ∈ ?S. ∀ A ∈ C. A ⊆ U
  by (auto simp: chains-def chain-subset-def)

from Zorn-Lemma2 [OF this] obtain m B
  where m ∈ Field r and B = r⁻¹ "{m}"
  and ∀ x ∈ Field r. B ⊆ r⁻¹ "{x}" "{x}" = B
  by auto

then have ∀ a ∈ Field r. (m, a) ∈ r → a = m
  by (auto simp: subset-Image1-Image1-iff Partial-order-eq-Image1-Image1-iff)

then show ?thesis
  using (m ∈ Field r) by blast
qed

lemma predicate-Zorn:
  assumes po: partial-order-on A (relation-of P A)
  and ch: ⋀ C ∈ Chains (relation-of P A) =⇒ ∃ u ∈ A. ∀ a ∈ C. P a u
  shows ∃ m ∈ A. ∀ a ∈ A. (m, a) ∈ relation-of P A
  proof
    have a ∈ A if C ∈ Chains (relation-of P A) and a ∈ C for C a
      using that unfolding Chains-def relation-of-def by auto
    moreover have (a, u) ∈ relation-of P A if a ∈ A and u ∈ A and P a u for a u
      unfolding relation-of-def using that by auto
    ultimately have ∃ m ∈ A. ∀ a ∈ A. (m, a) ∈ relation-of P A =⇒ a = m
      using Zorns-po-lemma[OF Partial-order-relation-ofD[OF po, rule-format] ch]
      unfolding Field-relation-of[OF partial-order-onD(1)[OF po]] by blast
    then show ?thesis
      by (auto simp: relation-of-def)
  qed

lemma Union-in-chain: [finite B; B ≠ {}]: subset.chain A B] ⇒ ∪ B ∈ B
  proof (induction B rule: finite-induct)
    case (insert B B)
    show ?case
    proof (cases B = {})
      case False
      then show ?thesis
        using insert sup.absorb2 by (auto simp: subset-chain-insert dest!: bspec [where x=∪B])
    qed auto
  qed simp

lemma Inter-in-chain: [finite B; B ≠ {}]: subset.chain A B] ⇒ ∩ B ∈ B
  proof (induction B rule: finite-induct)
    case (insert B B)
    show ?case
    proof (cases B = {})
      qed auto
    qed simp
  qed

lemma Inter-in-chain: [finite B; B ≠ {}]: subset.chain A B] ⇒ ∩ B ∈ B
  proof (induction B rule: finite-induct)
    case (insert B B)
    show ?case
    proof (cases B = {})
      qed auto
    qed simp
  qed

THEORY “Zorn”

---

case False 
then show ‚thesis 
  using insert inf.absorb2 by (auto simp: subset-chain-insert dest!: bspec [where
x=|B|]) 
qed auto
qed simp

lemma finite-subset-Union-chain: 
  assumes finite A A ⊆ |B| B ≠ |} and sub: subset.chain A B
  obtains B where B ∈ B A ⊆ B
proof –
  obtain F where F: finite F F ⊆ |B| A ⊆ |∪|F
  using assms by (auto intro: finite-subset-Union)
  show ‚thesis 
    proof (cases F = |})
      case True 
      then show ‚thesis 
        using ‹A ⊆ |∪|F› ‹B ≠ |}› that by fastforce
    next 
    case False 
    show ‚thesis 
      proof
        show |∪|F ∈ B 
          using sub ‹F ⊆ |B› ‹finite F›, 
          by (simp add: Union-in-chain False subset.chain-def subset-iff)
        show A ⊆ |∪|F 
          using ‹A ⊆ |∪|F› by blast
        qed
      qed
    qed

lemma subset-Zorn-nonempty: 
  assumes A ≠ |} and ch: ∃C. [C≠|}; subset.chain A C] ⇒ |∪|C ∈ A
  shows ∃M∈A. ∀X∈A. M ⊆ X → X = M
proof (rule subset-Zorn)
  show ∃U∈A. ∀X∈C. X ⊆ U if subset.chain A C for C
  proof (cases C = |})
    case True 
    then show ‚thesis 
      using ‹A ≠ |}› by blast
  next 
  case False 
  show ‚thesis 
    by (blast intro!: ch False that Union-upper)
qed
qed
26.4 The Well Ordering Theorem

definition init-seg-of :: (('a × 'a) set × ('a × 'a) set) set
  where init-seg-of = {(r, s). r ⊆ s ∧ (∀ a b c. (a, b) ∈ s ∧ (b, c) ∈ r → (a, b) ∈ r)}

abbreviation initial-segment-of-syntax :: ('a × 'a) set ⇒ ('a × 'a) set ⇒ bool
  (infix initial_segment_of Syntax 55)
  where r initial-segment-of s ≡ (r, s) ∈ init-seg-of

lemma refl-on-init-seg-of [simp]: r initial-segment-of r
  by (simp add: init-seg-of-def)

lemma trans-init-seg-of:
  r initial-segment-of s =⇒ s initial-segment-of t =⇒ r initial-segment-of t
  by (simp (no_asm-use) add: init-seg-of-def) blast

lemma antisym-init-seg-of:
  r initial-segment-of s =⇒ s initial-segment-of r =⇒ r = s
  unfolding init-seg-of-def by safe

lemma Chains-init-seg-of-Union:
  R ∈ Chains init-seg-of =⇒ r ∈ R =⇒ r initial-segment-of ∪ R
  by (auto simp: init-seg-of-def Ball_def Chains_def) blast

lemma chain-subset-trans-Union:
  assumes chain ⊆ R ∀ r ∈ R. trans r
  shows trans (∪ R)

proof (intro transI, elim UnionE)
  fix S1 S2 :: 'a rel and x y z :: 'a
  assume S1 ∈ R S2 ∈ R
  with assms(1) have S1 ⊆ S2 ∨ S2 ⊆ S1
    unfolding chain-subset-def by blast
  moreover assume (x, y) ∈ S1 (y, z) ∈ S2
  ultimately have ((x, y) ∈ S1 ∧ (y, z) ∈ S1) ∨ ((x, y) ∈ S2 ∧ (y, z) ∈ S2)
    by blast
  with ⟨S1 ∈ R, S2 ∈ R⟩ assms(2) show (x, z) ∈ ∪ R
    by (auto elim: transE)
qed

lemma chain-subset-antisym-Union:
  assumes chain ⊆ R ∀ r ∈ R. antisym r
  shows antisym (∪ R)

proof (intro antisymI, elim UnionE)
  fix S1 S2 :: 'a rel and x y :: 'a
  assume S1 ∈ R S2 ∈ R
  with assms(1) have S1 ⊆ S2 ∨ S2 ⊆ S1
    unfolding chain-subset-def by blast
  moreover assume (x, y) ∈ S1 (y, x) ∈ S2
  ultimately have ((x, y) ∈ S1 ∧ (y, x) ∈ S1) ∨ ((x, y) ∈ S2 ∧ (y, x) ∈ S2)
by blast
with \langle S1 \in R \rangle \langle S2 \in R \rangle \text{assms}(2) \text{ show } x = y
unfolding antisym-def by auto
qed

lemma chain-subset-Total-Union:
assumes chain\subseteq R and \forall r \in R. \text{Total } r
shows \text{Total } (\bigcup R)
proof (simp add: total-on-def Ball-def, auto del: disjCI)
fix r s a b
assume A: r \in R s \in R a \in \text{Field } b \in \text{Field } s a \neq b
from \langle \text{chain} \subseteq R \rangle \langle r \in R \rangle \text{ and } \langle s \in R \rangle \text{ have } r \subseteq s \vee s \subseteq r
by (auto simp add: chain-subset-def)
then show (\exists r \in R. (a, b) \in r) \vee (\exists r \in R. (b, a) \in r)
proof
assume r \subseteq s
then have (a, b) \in s \vee (b, a) \in s
using assms(2) \text{ A mono-Field[of } s r]\text{]}
by (auto simp add: total-on-def)
then show ?thesis
using \text{$s \in R$} by blast
next
assume s \subseteq r
then have (a, b) \in r \vee (b, a) \in r
using assms(2) \text{ A mono-Field[of } s r]\text{]}
by (fastforce simp add: total-on-def)
then show ?thesis
using \langle r \in R \rangle by blast
qed

lemma wf-Union-wf-init-segs:
assumes R \in \text{Chains init-seg-of}
and \forall r \in R. \text{wf } r
shows \text{wf } (\bigcup R)
proof (simp add: wf_iff_no_infinite_down_chain, rule ccontr, auto)
fix f
assume 1: \forall i. \exists r \in R. (f (\text{Suc } i), f i) \in r
then obtain r where r \in R \text{ and } (f (\text{Suc } 0), f 0) \in r \text{ by auto}
have (f (\text{Suc } i), f i) \in r \text{ for } i
proof (induct i)
case 0
show ?case by fact
next
case (Suc i)
then obtain s where s \in R (f (\text{Suc } (\text{Suc } i)), f (\text{Suc } i)) \in s
using f by auto
then have s initial-segment-of r \vee r initial-segment-of s
using assms(1) \langle r \in R \rangle by (simp add: Chains-def)
with Suc s show ?case by (simp add: init-seg-of-def) blast
qed

then show False
  using assms(2) and \( r \in R \)
  by (simp add: wf_iff_no_infinite_down_chain) blast
qed

lemma initial-segment-of-Diff: \( p \) initial-segment-of \( q \) \implies \( p - s \) initial-segment-of \( q - s \)
  unfolding init-seg-of_def by blast

lemma Chains-init-seg-I: \( R \in \text{Chains} \) init-seg-of \implies \( \{ r - s \mid r, r \in R \} \in \text{Chains} \) init-seg-of
  unfolding Chains_def by (blast intro: initial-segment-of-Diff)

theorem well-ordering: \( \exists r::'a \text{ rel}. \text{Well-order} \ r \land \text{Field} \ r = \text{UNIV} \)
proof
  — The initial segment relation on well-orders:
  let \( \{ r::'a \text{ rel}. \text{Well-order} \ r \} \)
  define I where I = init-seg-of \cap \( \{ r - s \mid r, r \in R \} \)
  unfolding chain_subset_def Chains_def by blast
  have I-init: \( I \subseteq \text{init-seg-of} \) by simp
  then have subch: \( \forall R. \ R \in \text{Chains} \ I \implies \text{chain} \subseteq R \)
    unfolding init-seg-of_def chain_subset_def Chains_def by blast
  have Chains-wo: \( \forall R. \ R \in \text{Chains} \ I \implies r \in R \implies \text{Well-order} \ r \)
    by (simp add: Chains_def I_def Field_def)
  have FI: Field I = \( \{ r - s \mid r, r \in R \} \)
    by (auto simp add: I-def init-seg-of_def Field_def)
  then have 0: Partial-order I
    by (auto simp: partial_order_on_def preorder_on_def antisym_def antisym_init-seg-of refl_on_def
      trans_def I_def elim!: trans-init-seg-of)
  — \( I \)-chains have upper bounds in \( \{ r - s \mid r, r \in R \} \) wrt \( I \):
  their Union
  have \( \bigcup R \in \{ r - s \mid r, r \in R \} \) if \( R \in \text{Chains} \ I \) for \( R \)
  proof
    — from that have Rix: \( R \in \text{Chains} \) init-seg-of
    using mono-Chains [OF I-init] by blast
    have subch: \( \text{chain} \subseteq R \)
      using \( \forall R. \ R \in \text{Chains} \ I \) I-init by (auto simp: init-seg-of_def chain_subset_def Chains_def)
  have \( \forall r \in R. \ \text{Refl} \ r \) and \( \forall r \in R. \ \text{trans} \ r \) and \( \forall r \in R. \ \text{antisym} \ r \)
    and \( \forall r \in R. \ \text{Total} \ r \) and \( \forall r \in R. \ \text{wf} \ (r - \text{Id}) \)
    using Chains-wo [OF \( \forall R. \ R \in \text{Chains} \ I \)] by (simp-all add: order_on_defs)
  have Refl (\( \bigcup R \))
    using \( \forall r \in R. \ \text{Refl} \ r \) unfolding refl_on_def by fastforce
  moreover have trans (\( \bigcup R \))
    by (rule chain_subset_transUnion [OF subch \( \forall r \in R. \ \text{trans} \ r \)])
  moreover have antisym (\( \bigcup R \))
    by (rule chain_subset_antisymUnion [OF subch \( \forall r \in R. \ \text{antisym} \ r \)])
  moreover have Total (\( \bigcup R \))
by (rule chain-subset-Total-Union [OF subch \( \forall r \in R. \ Total \ r \)])
moreover have \( wf \ (\bigcup R) \sim Id \)
proof 
  have \( (\bigcup R) \sim Id = \bigcup \{ r \sim Id \mid r \in R \} \) by blast
with \( \bigvee r \in R. \ wf \ (r \sim Id) \) and \( wf-Union-wf-init-segs \ [OF \ Chains-init-segs] \)
[of Ris]
  show \(?thesis\) by fastforce
qed
ultimately have Well-order \( (\bigcup R) \)
by (simp add: order-on-defs refl-on-def trans-def antisym-def total-on-def Field-def)
— Zorn’s Lemma yields a maximal well-order \( m \):
then obtain \( m :: 'a \ rel \)
where Well-order \( m \)
  and \( max: \forall r. \ Well-order \ r \land (m, r) \in I \implies r = m \)
using Zorns-po-lemma[OF 0 1] unfolding FI by fastforce
— Now show by contradiction that \( m \) covers the whole type:
have False if \( x \notin Field m \) for \( x :: 'a \)
proof 
— Assuming that \( x \) is not covered and extend \( m \) at the top with \( x \)
  have \( m \neq \{ \} \)
proof 
  assume \( m = \{ \} \)
  moreover have Well-order \{ (x, x) \}
    by (simp add: order-on-defs refl-on-def trans-def antisym-def total-on-def Field-def)
  ultimately show False using max
    by (auto simp: I-def init-seg-of-def simp del: Field-insert)
qed
then have Field \( m \neq \{ \} \) by (auto simp: Field-def)
moreover have \( wf \ (m \sim Id) \)
using Well-order \( m \) by (simp add: well-order-on-def)
— The extension of \( m \) by \( x \):
  let \( ?s = \{ (a, x) \mid a, a \in Field m \} \)
  let \( ?m = \text{insert} \ (x, x) \ m \cup ?s \)
  have \( Fm: Field ?m = \text{insert} \ x \ (Field m) \)
    by (auto simp: Field-def)
  have Refl \( m \) and \( \text{trans} \ m \) and \( \text{antisym} \ m \) and \( \text{Total} \ m \) and \( \text{wf} \ (m \sim Id) \)
    using Well-order \( m \) by (simp-all add: order-on-defs)
— We show that the extension is a well-order
  have Refl \( ?m \)
    using (Refl \( m \) \ Fm unfolding refl-on-def by blast
moreover have \( \text{trans } ?m \) using \( \langle \text{trans } m \rangle \) and \( \langle x \notin \text{Field } m \rangle \) unfolding \( \text{trans-def Field-def} \) by blast

moreover have \( \text{antisym } ?m \) using \( \langle \text{antisym } m \rangle \) and \( \langle x \notin \text{Field } m \rangle \) unfolding \( \text{antisym-def Field-def} \) by blast

moreover have \( \text{Total } ?m \) using \( \langle \text{Total } m \rangle \) and \( \text{Fm} \) by (auto simp: total-on-def)

moreover have \( \text{wf } (\ ?m - \text{Id}) \)

proof –

have \( \text{wf } ?s \) using \( \langle x \notin \text{Field } m \rangle \) by (auto simp: wf-eq-minimal Field-def Bex-def)

then show \( \text{thesis} \) using \( \langle \text{wf } (\ ?m - \text{Id}) \rangle \) and \( \langle x \notin \text{Field } m \rangle \) wf-subset [OF \( \langle \text{wf } ?s \rangle \) Diff-subset]

by (auto simp: Un-Diff Field-def intro: wf-Un)

qed

ultimately have \( \text{Well-order } ?m \)

by (simp add: order-on-defs)

— We show that the extension is above \( m \)

moreover have \( (m, ?m) \in I \)

using \( \langle \text{Well-order } ?m \rangle \) and \( \langle \text{Well-order } m \rangle \) and \( \langle x \notin \text{Field } m \rangle \)

by (fastforce simp: I-def init-seg-of-def Field-def)

ultimately

— This contradicts maximality of \( m \):

show False

using max and \( \langle x \notin \text{Field } m \rangle \) unfolding Field-def by blast

qed

then have \( \text{Field } m = \text{UNIV} \) by auto

with \( \langle \text{Well-order } m \rangle \) show \( \text{thesis} \) by blast

qed

corollary well-order-on: \( \exists r :: \forall a. \text{well-order-on } A r \)

proof –

obtain \( r :: \forall a. \text{well-ordering \ [where } 'a = 'a] \) by blast

let \( ?r = \{ (x, y). x \in A \land y \in A \land (x, y) \in r \} \)

have 1: Field ?r = A

using wo univ by (fastforce simp: Field-def order-on-defs refl-on-def)

from \( \langle \text{Well-order } r \rangle \) have Refl r trans r antisym r Total r wf (\( r - \text{Id} \))

by (simp-all add: order-on-defs)

from \( \langle \text{Refl } r \rangle \) have Refl ?r

by (auto simp: refl-on-def 1 univ)

moreover from \( \langle \text{trans } r \rangle \) have trans ?r

unfolding trans-def by blast

moreover from \( \langle \text{antisym } r \rangle \) have antisym ?r

unfolding antisym-def by blast

moreover from \( \langle \text{Total } r \rangle \) have Total ?r

by (simp add: total-on-def 1 univ)

moreover have \( \text{wf } (\ ?r - \text{Id}) \)

by (rule wf-subset [OF \( \langle \text{wf } (\ ?r - \text{Id}) \rangle \)) blast


ultimately have Well-order ?r by (simp add: order-on-defs)
with I show ?thesis by auto
qed

lemma dependent-wf-choice:
fixes P :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ bool
assumes wf R
and adm: (∃ x. (x ∈ R) =⇒ P x)
shows (∃ f. ∀ x. P f x)
proof (intro exI allI)
fix x
define f where f ≡ wfrec R (λf x. SOME r. P f x)
from wf R show P f x
proof (induct x)
case (less x)
show P f x
proof (auto simp: adm-wf-def intro: arg-cong where f=λ x. SOME r. P f x)
 using wf by (rule someI-ex) fact
qed
qed

lemma (in wellorder) dependent-wellorder-choice:
assumes (∀ x. (∀ y. y < x =⇒ f y = g y) =⇒ P f x = P g x)
and P: (∀ x. (∀ y. y < x =⇒ P f y (f y)) =⇒ ∃ r. P f x)
shows (∃ f. ∀ x. P f x)
using R by (rule dependent-wf-choice) (auto intro: assms)

end

27 Well-Order Relations as Needed by Bounded Natural Functors

theory BNF-Wellorder-Relation
imports Order-Relation
begin

In this section, we develop basic concepts and results pertaining to well-order relations. Note that we consider well-order relations as non-strict relations, i.e., as containing the diagonals of their fields.

locale wo-rel =
fixes R :: 'a rel
assumes WELL: Well-order R
begin

The following context encompasses all this section. In other words, for the whole section, we consider a fixed well-order relation $r$.

abbreviation under where under $\equiv$ Order-Relation.under $r$
abbreviation underS where underS $\equiv$ Order-Relation.underS $r$
abbreviation Under where Under $\equiv$ Order-Relation.Under $r$
abbreviation above where above $\equiv$ Order-Relation.above $r$
abbreviation aboveS where aboveS $\equiv$ Order-Relation.aboveS $r$
abbreviation Above where Above $\equiv$ Order-Relation.Above $r$
abbreviation ofilter where ofilter $\equiv$ Order-Relation.ofilter $r$

lemmas ofilter-def = Order-Relation.ofilter-def[of $r$]

\[ \text{27.1 Auxiliaries} \]

lemma REFL: Refl $r$
using WELL order-on-defs[of - $r$] by auto

lemma TRANS: trans $r$
using WELL order-on-defs[of - $r$] by auto

lemma ANTISYM: antisym $r$
using WELL order-on-defs[of - $r$] by auto

lemma TOTAL: Total $r$
using WELL order-on-defs[of - $r$] by auto

lemma TOTALS: $\forall a \in \text{Field } r. \forall b \in \text{Field } r. (a,b) \in r \lor (b,a) \in r$
using REFL TOTAL refl-on-def[of - $r$] total-on-def[of - $r$] by force

lemma LIN: Linear-order $r$
using WELL well-order-on-def[of - $r$] by auto

lemma WF: wf ($r - \text{Id}$)
using WELL well-order-on-def[of - $r$] by auto

lemma cases-Total:
$\bigwedge \phi i a b. \{\{a,b\} \leq \text{Field } r; ((a,b) \in r \implies \phi i a b); ((b,a) \in r \implies \phi i a b)\}$
$\implies \phi i a b$
using TOTALS by auto

lemma cases-Total3:
$\bigwedge \phi i a b. \{\{a,b\} \leq \text{Field } r; ((a,b) \in r - \text{Id} \lor (b,a) \in r - \text{Id} \implies \phi i a b); (a = b \implies \phi i a b)\}$
$\implies \phi i a b$
using TOTALS by auto
27.2 Well-founded induction and recursion adapted to non-strict well-order relations

Here we provide induction and recursion principles specific to non-strict well-order relations. Although minor variations of those for well-founded relations, they will be useful for doing away with the tediousness of having to take out the diagonal each time in order to switch to a well-founded relation.

**Lemma** well-order-induct:
- **Assumes**: \( \forall x. \forall y. y \neq x \land (y, x) \in r \implies P y \implies P x \)
- **Shows**: \( P a \)

**Proof** 
- Have: \( \forall x. \forall y. (y, x) \in r - Id \implies P y \implies P x \)
- Using IND by blast
- Thus \( P a \) using WF wf-induct[of \( r - Id \) \( P a \)] by blast

**Qed**

**Definition** worec :: \( (\forall a \Rightarrow b) \Rightarrow a \Rightarrow b \)
- Where
  \( \text{worec } F \equiv \text{wfrec } (r - Id) F \)

**Definition** adm-wo :: \( (\forall a \Rightarrow b) \Rightarrow a \Rightarrow b \Rightarrow \text{bool} \)
- Where
  \( \text{adm-wo } H \equiv \forall f \ g \ x. (\forall y \in \text{underS } x. f y = g y) \implies H f x = H g x \)

**Lemma** worec-fixpoint:
- **Assumes**: ADM: adm-wo \( H \)
- **Shows**: \( \text{worec } H = H (\text{worec } H) \)

**Proof**
- Let \( r' = r - Id \)
- Have: adm-wo \( r - Id \) \( H \)
- Unfolding adm-wo-def
  - Using ADM adm-wo-def[of \( H \)] underS-def[of \( r \)] by auto
- Hence \( \text{wfrec } r' S H = H (\text{wfrec } r' S H) \)
  - Using WF wfrec-fixpoint[of \( r' S H \)] by simp
- Thus ?thesis unfolding worec-def.

**Qed**

27.3 The notions of maximum, minimum, supremum, successor and order filter

We define the successor of a set, and not of an element (the latter is of course a particular case). Also, we define the maximum of two elements, \( \text{max2} \), and the minimum of a set, \( \text{minim} \) – we chose these variants since we consider them the most useful for well-orders. The minimum is defined in terms of the auxiliary relational operator \( \text{isMinim} \). Then, supremum and
successor are defined in terms of minimum as expected. The minimum is only meaningful for non-empty sets, and the successor is only meaningful for sets for which strict upper bounds exist. Order filters for well-orders are also known as “initial segments”.

**definition** \( \text{max2} :: \ 'a \Rightarrow \ 'a \rightarrow \ 'a \) where \( \text{max2} a b \equiv \begin{cases} b & \text{if } (a, b) \in r \text{ then } b \text{ else } a \end{cases} \)

**definition** \( \text{isMinim} :: \ 'a \text{ set } \Rightarrow \ 'a \Rightarrow \ ) where \( \text{isMinim} A b \equiv b \in A \land (\forall a \in A. (b, a) \in r) \)

**definition** \( \text{minim} :: \ 'a \text{ set } \Rightarrow \ 'a \) where \( \text{minim} A \equiv \text{THE } \text{b. isMinim } A \text{ b} \)

**definition** \( \text{supr} :: \ 'a \text{ set } \Rightarrow \ 'a \) where \( \text{supr} A \equiv \text{minim } \text{(Above } A) \)

**definition** \( \text{suc} :: \ 'a \text{ set } \Rightarrow \ 'a \) where \( \text{suc} A \equiv \text{minim } \text{(AboveS } A) \)

### 27.3.1 Properties of \( \text{max2} \)

**lemma** \( \text{max2-greater-among}: \) 
\( \begin{align*} &\text{assumes } a \in \text{Field } r \text{ and } b \in \text{Field } r \\
&\text{shows } (a, \text{max2 } a b) \in r \land (b, \text{max2 } a b) \in r \land \text{max2 } a b \in \{a, b\} \\
&\text{proof} - \ \\
&\{\text{assume } (a, b) \in r \\
&\quad \text{hence } ?\text{thesis using } \text{max2-def } \text{assms } \text{REFL } \text{refl-on-def} \\
&\quad \text{by } (\text{auto simp add: refl-on-def}) \\
&\} \\
&\text{moreover} \\
&\{\text{assume } a = b \\
&\quad \text{hence } (a, b) \in r \text{ using } \text{REFL } \text{assms} \\
&\quad \text{by } (\text{auto simp add: refl-on-def}) \\
&\} \\
&\text{moreover} \\
&\{\text{assume } *: a \neq b \land (b, a) \in r \\
&\quad \text{hence } (a, b) \notin r \text{ using } \text{ANTISYM} \\
&\quad \text{by } (\text{auto simp add: antisym-def}) \\
&\quad \text{hence } ?\text{thesis using } * \text{max2-def } \text{assms } \text{REFL } \text{refl-on-def} \\
&\quad \text{by } (\text{auto simp add: refl-on-def}) \\
&\} \\
&\text{ultimately show } ?\text{thesis using } \text{assms } \text{TOTAL} \\
&\quad \text{total-on-def[of Field } r r] \text{ by blast} \\
&\text{qed} \\
\)

**lemma** \( \text{max2-greater}: \) 
\( \begin{align*} &\text{assumes } a \in \text{Field } r \text{ and } b \in \text{Field } r \\
&\text{shows } (a, \text{max2 } a b) \in r \land (b, \text{max2 } a b) \in r \\
&\text{using } \text{assms } \text{by } (\text{auto simp add: max2-greater-among}) \\
\)
lemma max2-among:
    assumes a ∈ Field r and b ∈ Field r
    shows max2 a b ∈ {a, b}
    using assms max2-greater-among[of a b] by simp

lemma max2-equals1:
    assumes a ∈ Field r and b ∈ Field r
    shows (max2 a b = a) = ((a, b) ∈ r)
    using assms ANTISYM unfolding antisym-def using TOTALS
    by(auto simp add: max2-def max2-among)

lemma max2-equals2:
    assumes a ∈ Field r and b ∈ Field r
    shows (max2 a b = b) = ((a, b) ∈ r)
    using assms ANTISYM unfolding antisym-def using TOTALS
    unfolding max2-def
    by auto

lemma in-notinI:
    assumes (j, i) /∈ r ∨ j = i and i ∈ Field r and j ∈ Field r
    shows (i, j) ∈ r using assms max2-def max2-greater-among
    by fastforce

27.3.2 Existence and uniqueness for isMinim and well-definedness of minim

lemma isMinim-unique:
    assumes isMinim B a isMinim B a'
    shows a = a'
    using assms ANTISYM antisym-def[of r] by (auto simp: isMinim-def)

lemma Well-order-isMinim-exists:
    assumes SUB: B ≤ Field r and NE: B ≠ {} 
    shows ∃b. isMinim B b
    proof
    from spec[OF WF[unfolded wf-eq-minimal[of r – Id]], of B] NE obtain b where
    *: b ∈ B ∧ (∀b'. b' ≠ b ∧ (b', b) ∈ r → b' /∈ B) by auto
    have ∀b'. b' ∈ B → (b, b') ∈ r
    proof
      fix b'
      show b' ∈ B → (b, b') ∈ r
      proof
        assume As: b' ∈ B
        hence **: b ∈ Field r ∧ b' ∈ Field r using As SUB * by auto
        from As * have b' = b ∨ (b', b) /∈ r by auto
        moreover have b' = b =⇒ (b, b') ∈ r
        using ** refl by (auto simp add: refl-on-def)
        moreover have b' ≠ b ∧ (b', b) /∈ r =⇒ (b, b') ∈ r
        using ** total by (auto simp add: total-on-def)
        ultimately show (b, b') ∈ r by blast
    qed
qed

then show ?thesis
  unfolding isMinim-def using * by auto
qed

lemma minim-isMinim:
  assumes SUB: B ≤ Field r and NE: B ≠ {} 
  shows isMinim B (minim B)
proof –
  let ?phi = (λ b. isMinim B b) 
  from assms Well-order-isMinim-exists
  obtain b where *: ?phi b by blast 
  moreover
  have ⋀ b′. ?phi b′ =⇒ b′ = b 
  using isMinim-unique * by auto 
  ultimately show ?thesis
  unfolding minim-def using theI[of ?phi b] by blast
qed

27.3.3 Properties of minim

lemma minim-in:
  assumes B ≤ Field r and B ≠ {} 
  shows minim B ∈ B
  using assms minim-isMinim[of B] by (auto simp: isMinim-def)

lemma minim-inField:
  assumes B ≤ Field r and B ≠ {} 
  shows minim B ∈ Field r
proof –
  have minim B ∈ B using assms by (simp add: minim-in) 
  thus ?thesis using assms by blast
qed

lemma minim-least:
  assumes SUB: B ≤ Field r and IN: b ∈ B 
  shows (minim B, b) ∈ r 
proof –
  from minim-isMinim[of B] assms 
  have isMinim B (minim B) by auto 
  thus ?thesis by (auto simp add: isMinim-def IN)
qed

lemma equals-minim:
  assumes SUB: B ≤ Field r and IN: a ∈ B and 
  LEAST: ⋀ b. b ∈ B =⇒ (a,b) ∈ r 
  shows a = minim B 
proof –
from minim-isMinim[of B] assms
have isMinim B (minim B) by auto
moreover have isMinim B a using IN LEAST isMinim-def by auto
ultimately show ?thesis using isMinim-unique by auto
qed

27.3.4 Properties of successor

lemma suc-AboveS:
  assumes SUB: B ≤ Field r and ABOVES: AboveS B ≠ {}
  shows suc B ∈ AboveS B
proof (unfold suc-def)
  have AboveS B ≤ Field r
    using AboveS-Field[of r] by auto
  thus minim (AboveS B) ∈ AboveS B
    using assms by (simp add: minim-in)
qed

lemma suc-greater:
  assumes SUB: B ≤ Field r and ABOVES: AboveS B ≠ {} and IN: b ∈ B
  shows suc B ≠ b ∧ (b, suc B) ∈ r
  using IN AboveS-def[of r] assms suc-AboveS by auto

lemma suc-least-AboveS:
  assumes ABOVES: a ∈ AboveS B
  shows (suc B, a) ∈ r
  using assms minim-least AboveS-Field[of r] by (auto simp: suc-def)

lemma suc-inField:
  assumes B ≤ Field r and AboveS B ≠ {}
  shows suc B ∈ Field r
  using suc-AboveS assms AboveS-Field[of r] by auto

lemma equals-suc-AboveS:
  assumes B ≤ Field r and a ∈ AboveS B and \( a', a' \in AboveS B \implies (a,a') \in r \)
  shows a = suc B
  using assms equals-minim AboveS-Field[of r B] by (auto simp: suc-def)

lemma suc-underS:
  assumes IN: a ∈ Field r
  shows a = suc (underS a)
proof –
  have underS a ≤ Field r
    using underS-Field[of r] by auto
  moreover
  have a ∈ AboveS (underS a)
    using in-AboveS-underS IN by fast
moreover
have ∀ a′ ∈ AboveS (underS a). (a,a′) ∈ r
proof(clarify)
  fix a'
  assume *: a′ ∈ AboveS (underS a)
  hence **: a′ ∈ Field r
      using AboveS-Field by fast
{assume (a,a') ∉ r
  hence a′ = a ∨ (a',a) ∈ r
      using TOTAL IN ** by (auto simp add: total-on-def)
moreover
{assume a′ = a
  hence (a,a') ∈ r
      using REFL IN ** by (auto simp add: refl-on-def)
}
moreover
{assume a′ ≠ a ∧ (a',a) ∈ r
  hence a′ ∈ underS a
      unfolding underS-def by simp
  hence a' ∉ AboveS (underS a)
      using AboveS-disjoint by fast
  with * have False by simp
}
ultimately have (a,a') ∈ r by blast
}
thus (a, a') ∈ r by blast
qed
ultimately show ?thesis
  using equals-suc-AboveS by auto
qed

27.3.5 Properties of order filters

lemma under-ofilter: ofilter (under a)
  using TRANS by (auto simp: ofilter-def under-def Field-iff trans-def)

lemma underS-ofilter: ofilter (underS a)
  unfolding ofilter-def underS-def under-def
proof safe
  fix b assume (a, b) ∈ r (b, a) ∈ r and DIFF: b ≠ a
  thus False
      using ANTSYM antisym-def[of r] by blast
next
  fix b x
  assume (b,a) ∈ r b ≠ a (x,b) ∈ r
  thus (x,a) ∈ r
      using TRANS trans-def[of r] by blast
next
  fix x
assume \( x \neq a \) and \((x, a) \in r\)
then show \( x \in \text{Field } r \)
unfolding Field-def
by auto
qed

lemma Field-ofilter:
ofilter (Field r)
by (unfold ofilter-def under-def, auto simp add: Field-def)

lemma ofilter-underS-Field:
ofilter A = ((\exists a \in \text{Field } r. \ A = \text{underS } a) \lor (A = \text{Field } r))
proof
assume \((\exists a \in \text{Field } r. \ A = \text{underS } a) \lor A = \text{Field } r\)
thus ofilter A
by (auto simp: underS-ofilter Field-ofilter)
next
assume \(*\): ofilter A
let \(?One = (\exists a \in \text{Field } r. \ A = \text{underS } a)\)
let \(?Two = (A = \text{Field } r)\)
show \(?One \lor ?Two\)
proof(cases ?Two)
let \(?B = (\text{Field } r) - A\)
let \(?a = \text{minim } ?B\)
assume \(A \neq \text{Field } r\)
moreover have \(A \leq \text{Field } r\) using * ofilter-def by simp
ultimately have 1: \(?B \neq \{\}\) by blast
hence 2: \(?a \in \text{Field } r\) using minim-inField[of \(?B\)] by blast
have 3: \(?a \in ?B\) using minim-in[of \(?B\)] 1 by blast
hence 4: \(?a \notin A\) by blast
have 5: \(A \leq \text{Field } r\) using * ofilter-def by auto
moreover
have \(A = \text{underS } ?a\)
proof
show \(A \leq \text{underS } ?a\)
proof
fix \(x\) assume **: \(x \in A\)
hence 11: \(x \in \text{Field } r\) using 5 by auto
have 12: \(x \neq ?a\) using 4 ** by auto
have 13: \(\text{under } x \leq A\) using * ofilter-def ** by auto
{assume \((x, ?a) \notin r\)
hence \(?a, x\) \in r
using TOTAL total-on-def[of Field r r]
2 4 11 12 by auto
hence \(?a \in \text{under } x\) using under-def[of r] by auto
hence \(?a \in A\) using ** 13 by blast
with 4 have False by simp
}
then have \((x, ?a) \in r\) by blast
thus \(x \in \text{underS } ?a\)
  unfolding \text{underS-def} by (auto simp add: 12)
qed

next
show \(\text{underS } ?a \leq A\)
proof
fix \(x\)
assume **: \(x \in \text{underS } ?a\)
  hence 11: \(x \in \text{Field } r\)
  using Field-def unfolding \text{underS-def} by fastforce
{assume \(x \notin A\)
  hence \(x \in \text{?B}\) using 11 by auto
  hence \((?a, x) \in r\) using 3 minim-least[of \(?B\) \(x\)] by blast
  hence False
    using ANTISYM antisym-def[of \(r\)] ** unfolding \text{underS-def} by auto
}
  thus \(x \in A\) by blast
qed qed

ultimately have \(?One\) using 2 by blast
  thus \(?thesis\) by simp

next
  assume \(A = \text{Field } r\)
  then show \(?thesis\)
    by simp
qed qed

\textbf{lemma ofilter-UNION: }
\((\land i. \ i \in I \Rightarrow \text{ofilter}(A \ i)) \Rightarrow \text{ofilter} (\bigcup i \in I. \ A \ i)\)
  unfolding \text{ofilter-def} by blast

\textbf{lemma ofilter-under-UNION: }
  \textbf{assumes} ofilter \(A\)
  \textbf{shows} \(A = (\bigcup a \in A. \ \text{under } a)\)
proof
  have \(\forall a \in A. \ \text{under } a \leq A\)
    using \text{assms ofilter-def} by auto
  thus \((\bigcup a \in A. \ \text{under } a) \leq A\) by blast
next
  have \(\forall a \in A. \ a \in \text{under } a\)
    using REFL Refl-under-in[of \(r\)] \text{assms ofilter-def[of } A\) by blast
  thus \(A \leq (\bigcup a \in A. \ \text{under } a)\) by blast
qed

\textbf{27.3.6 Other properties}

\textbf{lemma ofilter-linord: }
assumes $OF1$: ofilter $A$ and $OF2$: ofilter $B$
shows $A \leq B \lor B \leq A$

proof (cases $A = Field\ r$)
  assume Case1: $A = Field\ r$
  hence $B \leq A$ using $OF2$ ofilter-def by auto
  thus ?thesis by simp
next
  assume Case2: $A \neq Field\ r$
  with ofilter-underS-Field $OF1$ obtain a where
    $1$: $a \in Field\ r \land A = underS\ a$ by auto
  show ?thesis
  proof (cases $B = Field\ r$)
    assume Case21: $B = Field\ r$
    hence $A \leq B$ using $OF1$ ofilter-def by auto
    thus ?thesis by simp
  next
    assume Case22: $B \neq Field\ r$
    with ofilter-underS-Field $OF2$ obtain b where
      $2$: $b \in Field\ r \land B = underS\ b$ by auto
    have $a = b \lor (a, b) \in r \lor (b, a) \in r$
      using $1\ 2$ TOTAL total-on-def[of - r] by auto
    moreover
      {assume $a = b$ with $1\ 2$ have ?thesis by auto}
    } moreover
      {assume $(a, b) \in r$
        with underS-incr[of r] TRANS ANTISYM $1\ 2$
        have $A \leq B$ by auto
        hence ?thesis by auto
      }
    } moreover
      {assume $(b, a) \in r$
        with underS-incr[of r] TRANS ANTISYM $1\ 2$
        have $B \leq A$ by auto
        hence ?thesis by auto
      }
    }
  ultimately show ?thesis by blast
qed

lemma ofilter-AboveS-Field:
assumes ofilter $A$
shows $A \cup (AboveS\ A) = Field\ r$

proof
  show $A \cup (AboveS\ A) \leq Field\ r$
    using assms ofilter-def AboveS-Field[of r] by auto
next
  {fix $x$ assume *: $x \in Field\ r$ and **: $x \notin A$
    {fix $y$ assume ***: $y \in A$
  qed

qed
with ** have 1: y ≠ x by auto
{assume (y,x) ∉ r
moreover
have y ∈ Field r using assms ofilter-def *** by auto
ultimately have (x,y) ∈ r
using 1 * TOTAL total-on-def[of - r] by auto
with *** assms ofilter-def under-def[of r] have x ∈ A by auto
with ** have False by contradiction
}
hence (y,x) ∈ r by blast
with 1 have y ≠ x ∧ (y,x) ∈ r by auto
}
with * have x ∈ AboveS A unfolding AboveS-def by auto
}
thus Field r ⊆ A ∪ (AboveS A) by blast
qed

lemma suc-ofilter-in:
assumes OF: ofilter A and ABOVE-NE: AboveS A ≠ {} and
REL: (b,suc A) ∈ r and DIFF: b ≠ suc A
shows b ∈ A
proof –
have *: suc A ∈ Field r ∧ b ∈ Field r
using WELL REL well-order-on-domain[of Field r] by auto
{assume **: b ∉ A
hence b ∈ AboveS A
using OF * ofilter-AboveS-Field by auto
hence (suc A, b) ∈ r
using suc-least-AboveS by auto
hence False using REL DIFF ANTISYM *
by (auto simp add: antisym-def)
}
thus ?thesis by blast
qed

end

end

28 Well-Order Embeddings as Needed by Bounded Natural Functors

theory BNF-Wellorder-Embedding
imports Hilbert-Choice BNF-Wellorder-Relation
begin
In this section, we introduce well-order embeddings and isomorphisms and prove their basic properties. The notion of embedding is considered from the
point of view of the theory of ordinals, and therefore requires the source to be injected as an initial segment (i.e., order filter) of the target. A main result of this section is the existence of embeddings (in one direction or another) between any two well-orders, having as a consequence the fact that, given any two sets on any two types, one is smaller than (i.e., can be injected into) the other.

### 28.1 Auxiliaries

**Lemma** *UNION-inj-on-ofilter*: 
- **Assumes**: WELL: Well-order \( r \) and 
  \( OF: \bigwedge i. i \in I \Rightarrow wo\cdotrel\cdotfilter r (A i) \) and 
  INJ: \( \bigwedge i. i \in I \Rightarrow inj\cdoton f (A i) \) 
- **Shows**: \( inj\cdoton f (\bigcup i \in I. A i) \)

**Proof** –
- have \( wo\cdotrel r \) using WELL by (simp add: wo\cdotrel\cdotdef)
  hence \( \bigwedge i. j. [i \in I; j \in I] \Rightarrow A i \leq A j \vee A j \leq A i \)
  using wo\cdotrel\cdotfilter\cdotlinord[of r] OF by blast
- with WELL INJ show \(?thesis\)
  by (auto simp add: inj\cdoton\cdotUNION\cdotchain)

qed

**Lemma** *under\cdotunderS\cdotbij\cdotbetw*:
- **Assumes**: WELL: Well-order \( r \) and WELL’: Well-order \( r’ \) and 
  IN: \( a \in \text{Field } r \) and IN’: \( f a \in \text{Field } r’ \) and 
  BIJ: bij\cdotbetw f (underS r a) (underS r’ (f a))
- **Shows**: bij\cdotbetw f (under r a) (under r’ (f a))

**Proof** –
- have \( a \notin underS r a \land f a \notin underS r’ (f a) \)
  unfolding underS\cdotdef by auto
- moreover
  \{ have \( \text{Refl } r \land \text{Refl } r’ \) using WELL WELL’
    by (auto simp add: order\cdoton\cdotdefs)
    hence \( \text{under } r a = \text{underS } r a \cup \{ a \} \land \text{under } r’ (f a) = \text{underS } r’ (f a) \cup \{ f a \} \)
    using IN IN’ by(auto simp add: Refl\cdotunder\cdotunderS) \}
- ultimately show \(?thesis\)
  using BIJ notIn\cdotUn\cdotbij\cdotbetw[of a underS r a f underS r’ (f a)] by auto

qed

### 28.2 (Well-order) embeddings, strict embeddings, isomorphisms and order-compatible functions

Standardly, a function is an embedding of a well-order in another if it injectively and order-compatibly maps the former into an order filter of the latter. Here we opt for a more succinct definition (operator *embed*), asking that, for any element in the source, the function should be a bijection
between the set of strict lower bounds of that element and the set of strict lower bounds of its image. (Later we prove equivalence with the standard definition – lemma embed-iff-compat-inj-ofilter.) A strict embedding (operator embedS) is a non-bijective embedding and an isomorphism (operator iso) is a bijective embedding.

**definition** embed :: 'a rel ⇒ 'a' rel ⇒ ('a ⇒ 'a') ⇒ bool
  where
  embed r r' f ≡ ∀ a ∈ Field r. bij-betw f (under r a) (under r' (f a))

**lemmas** embed-defs = embed-def embed-def[abs-def]

Strict embeddings:

**definition** embedS :: 'a rel ⇒ 'a' rel ⇒ ('a ⇒ 'a') ⇒ bool
  where
  embedS r r' f ≡ embed r r' f ∧ ¬ bij-betw f (Field r) (Field r')

**lemmas** embedS-defs = embedS-def embedS-def[abs-def]

**definition** iso :: 'a rel ⇒ 'a' rel ⇒ ('a ⇒ 'a') ⇒ bool
  where
  iso r r' f ≡ embed r r' f ∧ bij-betw f (Field r) (Field r')

**lemmas** iso-defs = iso-def iso-def[abs-def]

**definition** compat :: 'a rel ⇒ 'a' rel ⇒ ('a ⇒ 'a') ⇒ bool
  where
  compat r r' f ≡ ∀ a b. (a, b) ∈ r −→ (f a, f b) ∈ r'

**lemma** compat-wf:
  assumes CMP: compat r r' f and WF: wf r'
  shows wf r
  proof -
  have r ≤ inv-image r' f
    unfolding inv-image-def using CMP
    by (auto simp add: compat-def)
  with WF show ?thesis
    using wf-inv-image[of r' f] wf-subset[of inv-image r' f] by auto
  qed

**lemma** id-embed: embed r r id
  by(auto simp add: id-def embed-def bij-betw-def)

**lemma** id-iso: iso r r id
  by(auto simp add: id-def embed-def iso-def bij-betw-def)

**lemma** embed-compat:
  assumes EMB: embed r r' f
  shows compat r r' f
unfolding compat-def
proof clarify
  fix a b
  assume *: (a,b) ∈ r
  hence 1: b ∈ Field r using Field-def[of r] by blast
  have a ∈ under r b
    using * under-def[of r] by simp
  hence f a ∈ under r’ (f b)
    using EMB embed-def[of r r’ f]
    bij-betw-def[of f under r b under r’ (f b)]
    image-def[of f under r b] 1 by auto
  thus (f a, f b) ∈ r’
    by (auto simp add: under-def)
qed

lemma embed-in-Field:
  assumes EMB: embed r r’ f and IN: a ∈ Field r
  shows f a ∈ Field r’
proof –
  have a ∈ Domain r ∨ a ∈ Range r
    using IN unfolding Field-def by blast
  then show ?thesis
    using embed-compat [OF EMB]
    unfolding Field-def compat-def by force
qed

lemma comp-embed:
  assumes EMB: embed r r’ f and EMB’: embed r’ r’’ f’
  shows embed r r’’ (f’ o f)
proof(unfold embed-def, auto)
  fix a assume *: a ∈ Field r
  hence bij-betw f (under r a) (under r’ (f a))
    using embed-def[of r] EMB by auto
  moreover
  {have f a ∈ Field r’
    using EMB * by (auto simp add: embed-in-Field)
    hence bij-betw f’ (under r’ (f a)) (under r’’ (f’ (f a)))
      using embed-def[of r’] EMB’ by auto
  }
  ultimately
  show bij-betw (f’ o f) (under r a) (under r’’ (f’(f a)))
    by(auto simp add: bij-betw-trans)
qed

lemma comp-iso:
  assumes EMB: iso r r’ f and EMB’: iso r’ r’’ f’
  shows iso r r’’ (f’ o f)
using assms unfolding iso-def
by (auto simp add: comp-embed bij-betw-trans)
That embedS is also preserved by function composition shall be proved only later.

**lemma** embed-Field: embed r r' f \implies f'(Field r) \leq Field r'  
by (auto simp add: embed-in-Field)

**lemma** embed-preserves-ofilter:  
assumes WELL: Well-order r and WELL': Well-order r' and  
EMB: embed r r' f and OF: wo-rel.ofilter r A  
shows wo-rel.ofilter r' (f' A)

**proof** –

from WELL have Well: wo-rel r unfolding wo-rel-def .  
from WELL' have Well': wo-rel r' unfolding wo-rel-def .  
from OF have 0: A \leq Field r by(auto simp add: Well wo-rel.ofilter-def)

show ?thesis using Well' WELL EMB 0 embed-Field[of r r' f]
proof(unfold wo-rel.ofilter-def, auto simp add: image-def)
  fix a b'  
  assume *: a \in A and **: b' \in under r' (f a)  
  hence a \in Field r using 0 by auto  
  hence bij-betw f (under r a) (under r' (f a))  
    using * EMB by (auto simp add: embed-def)  
  hence f(under r a) = under r' (f a)  
    by (simp add: bij-betw-def)  
  with ** image-def[of f under r a] obtain b where  
    1: b \in under r a \land b' = f b by blast  
  hence b \in A using Well * OF  
    by (auto simp add: wo-rel.ofilter-def)  
  with 1 show \exists b \in A. b' = f b by blast
qed

**lemma** embed-Field-ofilter:  
assumes WELL: Well-order r and WELL': Well-order r' and  
EMB: embed r r' f  
shows wo-rel.ofilter r' (f'(Field r))

**proof** –

have wo-rel.ofilter r (Field r)  
  using WELL by (auto simp add: wo-rel-def wo-rel.Field-ofilter)  
with WELL WELL' EMB  
show ?thesis by (auto simp add: embed-preserves-ofilter)
qed

**lemma** embed-inj-on:  
assumes WELL: Well-order r and EMB: embed r r' f  
shows inj-on f (Field r)

**proof**(unfold inj-on-def, clarify)

  from WELL have Well: wo-rel r unfolding wo-rel-def .
with wo-rel.TOTAL[of r]
have Total: Total r by simp
from Well wo-rel.REFL[of r]
have Refl: Refl r by simp

fix a b
assume #: a ∈ Field r and #: b ∈ Field r and
###: f a = f b
hence I: a ∈ Field r ∧ b ∈ Field r

unfolding Field-def by auto
{assume (a,b) ∈ r
hence a ∈ under r b ∧ b ∈ under r b
using Refl by(auto simp add: under-def refl-on-def)
hence a = b
using EMB 1 ###
by (auto simp add: embed-def bij-betw-def inj-on-def)
}
moreover
{assume (b,a) ∈ r
hence a ∈ under r a ∧ b ∈ under r a
using Refl by(auto simp add: under-def refl-on-def)
hence a = b
using EMB 1 ###
by (auto simp add: embed-def bij-betw-def inj-on-def)
}
ultimately
show a = b using Total 1
by (auto simp add: total-on-def)
qed

lemma embed-underS:
assumes WELL: Well-order r and
EMB: embed r r' f and IN: a ∈ Field r
shows bij-betw f (underS r a) (underS r' (f a))
proof−
have f a ∈ Field r' using assms embed-Field[of r r' f] by auto
then have 0: under r a = underS r a ∪ {a}
by (simp add: IN Refl-under-underS WELL wo-rel.REFL wo-rel.intro)
moreover have 1: bij-betw f (under r a) (under r' (f a))
using assms by (auto simp add: embed-def)
moreover have under r' (f a) = underS r' (f a) ∪ {f a}
proof
show under r' (f a) ⊆ underS r' (f a) ∪ {f a}
using underS-def under-def by fastforce
show underS r' (f a) ∪ {f a} ⊆ under r' (f a)
using bij-betwE 0 1 underS-subset-under by fastforce
qed
moreover have a /∈ underS r a ∧ f a /∈ underS r' (f a)
unfolding underS-def by blast
ultimately show thesis
    by (auto simp add: notIn-Un-bij-betw3)
qed

lemma embed-iff-compat-inj-on-ofilter:
    assumes WELL: Well-order r and WELL': Well-order r'
    shows embed r r' \ f = (compat r r' f \ inj-on f (Field r) \ wo-rel.ofilter r' (f(Field r)))
    using assms
proof (auto simp add: embed-compat embed-inj-on embed-Field-ofilter, unfold embed-def, auto)
fix a
assume *: inj-on f (Field r) and
**: compat r r' f and
***: wo-rel.ofilter r' (f(Field r)) and
****: a \ Field r
have Well: wo-rel r
    using WELL wo-rel-def[of r] by simp
hence Refl: Refl r
    using wo-rel.REFL[of r] by simp
have Total: Total r
    using Well wo-rel.TOTAL[of r] by simp
have WELL': wo-rel r'
    using WELL' wo-rel-def[of r'] by simp
hence Antisym': antisym r'
    using wo-rel.ANTISYM[of r'] by simp
have (a,a) \ r
    using **** Well wo-rel.REFL[of r]
    refl-on-def[of - r] by auto
hence (f a, f a) \ r'
    using ** by (auto simp add: compat-def)
hence 0: f a \ Field r'
    unfolding Field-def by auto
have f a \ f(Field r)
    using **** by auto
hence 2: under r' (f a) \ f(Field r)
    using WELL' *** wo-rel.ofilter-def[of r' f(Field r)] by fastforce

show bij-betw f (under r a) (under r' (f a))
proof (unfold bij-betw-def, auto)
    show inj-on f (under r a) by (rule subset-inj-on[OF * under-Field])
next
fix b assume b \ under r a
thus f b \ under r' (f a)
    unfolding under-def using **
    by (auto simp add: compat-def)
next
fix b' assume *****: b' \ under r' (f a)
hence \( b' \in f'(\text{Field } r) \)

using 2 by auto

with \text{Field-def[of } r\text{]} obtain \( b \) where

3: \( b \in \text{Field } r \) and 4: \( b' = f \ b \) by auto

have \((b, a) \in r\)

proof−
  \{assume \((a, b) \in r\)
   with ** 4 have \((f \ a, b') \in r'\)
     by (auto simp add: compat-def)
   with ***** \text{Antisym'} have \( f \ a = b'\)
     by(auto simp add: under-def antisym-def)
   with 3 ***** 4 \* have \( a = b\)
     by(auto simp add: inj-on-def)
  }

moreover
  \{assume \( a = b\)
   hence \((b, a) \in r\) using \text{Refl} ***** 3
     by (auto simp add: refl-on-def)
  }

ultimately

show \?thesis using \text{Total} ***** 3 by (fastforce simp add: total-on-def)

qed

with \( f \) show \( b' \in f'(\text{under } r \ a)\)

unfolding \text{under-def} by auto

qed

qed

lemma \text{inv-into-ofilter-embed}:
assumes \( \text{WELL}: \text{Well-order } r \) and \( \text{OF}: \text{wo-rel.ofilter } r \ A \) and
  \( \text{BIJ}: \forall \ b \in A. \ \text{bij-betw } f \ (\text{under } r \ b) \ (\text{under } r' \ (f \ b)) \) and
  \( \text{IMAGE}: f : A = \text{Field } r'\)
shows \( \text{embed } r' \ r \ (\text{inv-into } A \ f)\)

proof−

have \( \text{Well}: \text{wo-rel } r\)
  using \( \text{WELL} \ \text{wo-rel-def[of } r\text{]} \) by simp

have \( \text{Refl}: \text{Refl } r\)
  using \( \text{Well} \ \text{wo-rel.REFL[of } r\text{]} \) by simp

have \( \text{Total}: \text{Total } r\)
  using \( \text{Well} \ \text{wo-rel.TOTAL[of } r\text{]} \) by simp

have 1: \text{bij-betw} \( f \ A \ (\text{Field } r')\)

proof(unfold \text{bij-betw-def inj-on-def}, auto simp add: \text{IMAGE})
  fix \( b1 \ b2\)
  assume *: \( b1 \in A \) and **: \( b2 \in A \) and
    ***: \( f \ b1 = f \ b2\)
  have 11: \( b1 \in \text{Field } r \ \& \ b2 \in \text{Field } r\)
    using ** \( \text{Well } \text{OF} \) by (auto simp add: \text{wo-rel.ofilter-def})

moreover
\{\text{assume } (b_1,b_2) \in r \\
\text{hence } b_1 \in \text{under } r \ b_2 \land b_2 \in \text{under } r \ b_2 \\
\text{unfolding } \text{under-def using } 11 \text{ Refl} \\
\text{by } (\text{auto simp add: refl-on-def}) \\
\text{hence } b_1 = b_2 \text{ using BIJ} \ \ast \ \ast \ \ast \ \ast \\
\text{by } (\text{simp add: bij-betw-def inj-on-def}) \}\}

\text{moreover}
\{\text{assume } (b_2,b_1) \in r \\
\text{hence } b_1 \in \text{under } r \ b_1 \land b_2 \in \text{under } r \ b_1 \\
\text{unfolding } \text{under-def using } 11 \text{ Refl} \\
\text{by } (\text{auto simp add: refl-on-def}) \\
\text{hence } b_1 = b_2 \text{ using BIJ} \ \ast \ \ast \ \ast \ \ast \\
\text{by } (\text{simp add: bij-betw-def inj-on-def}) \}\}

\text{ultimately}
\text{show } b_1 = b_2 \\
\text{using Total by } (\text{auto simp add: total-on-def})
\text{qed}

\text{let } f' = (\text{inv-into } A f) \\
\text{have } 2: \forall b \in A. \text{bij-betw } f' (\text{under } r' (f b)) (\text{under } r b) \\
\text{proof(clarify)} \\
\text{fix } b \text{ assume } \ast: b \in A \\
\text{hence } \text{under } r \ b \leq A \\
\text{using \text{Well OF} by}(\text{auto simp add: wo-rel.ofilter-def}) \\
\text{moreover}
\text{have } f' (\text{under } r b) = \text{under } r' (f b) \\
\text{using } \ast \text{ BIJ by } (\text{auto simp add: bij-betw-def}) \\
\text{ultimately}
\text{show } \text{bij-betw } f' (\text{under } r' (f b)) (\text{under } r b) \\
\text{using } 1 \text{ by } (\text{auto simp add: bij-betw-inv-into-subset})
\text{qed}

\text{have } 3: \forall b' \in \text{Field } r'. \text{bij-betw } f' (\text{under } r' b') (\text{under } r (f b')) \\
\text{proof(clarify)} \\
\text{fix } b' \text{ assume } \ast: b' \in \text{Field } r' \\
\text{have } b' = f (f' b') \text{ using } 1 \\
\text{by } (\text{auto simp add: bij-betw-inv-into-right}) \\
\text{moreover}
\{\text{obtain } b \text{ where } 31: b \in A \text{ and } f b = b' \text{ using } \text{IMAGE} \ \ast \ \text{ by force} \\
\text{hence } f' b' = b \text{ using } 1 \text{ by } (\text{auto simp add: bij-betw-inv-into-left}) \\
\text{with } 31 \text{ have } f' b' \in A \text{ by } \text{auto} \} \\
\text{ultimately}
\text{show } \text{bij-betw } f' (\text{under } r' b') (\text{under } r (f b')) \\
\text{using } 2 \text{ by } \text{auto}
\text{qed}
thus \( ?\text{thesis unfolding embed-def} \).

\textbf{qed}

\textbf{lemma} \textit{inv-into-underS-embed}:
\textit{assumes WELL: Well-order \( r \) and BIJ: \( \forall b \in \text{underS} \ r \ a. \ bij-betw \ f (\text{under} \ r \ b) (\text{under} \ r' \ (f \ b)) \) and IN: \( a \in \text{Field} \ r \) and IMAGE: \( f' (\text{underS} \ r \ a) = \text{Field} \ r' \) shows embed \( r' \ r \ (\text{inv-into} (\text{underS} \ r \ a) \ f) \) using assms by(auto simp add: wo-rel-def wo-rel.underS-ofilter inv-into-ofilter-embed)

\textbf{lemma} \textit{inv-into-Field-embed}:
\textit{assumes WELL: Well-order \( r \) and EMB: embed \( r \ r' \ f \) and IMAGE: \( \text{Field} \ r' \leq f' (\text{Field} \ r) \) shows embed \( r' \ r \ (\text{inv-into} (\text{Field} \ r) \ f) \) proof − have \( (\forall b \in \text{Field} \ r. \ bij-betw \ f (\text{under} \ r \ b) (\text{under} \ r' \ (f \ b))) \) using EMB by(auto simp add: embed-def)
moreover have \( f' (\text{Field} \ r) \leq \text{Field} \ r' \) using EMB WELL by(auto simp add: embed-Field)
ultimately show \( ?\text{thesis using assms} \) by(auto simp add: wo-rel-def wo-rel.Field-ofilter inv-into-ofilter-embed)
\textbf{qed}

\textbf{lemma} \textit{inv-into-Field-embed-bij-betw}:
\textit{assumes EMB: embed \( r \ r' \ f \) and BIJ: bij-betw \( f (\text{Field} \ r) (\text{Field} r') \) shows embed \( r' \ r \ (\text{inv-into} (\text{Field} \ r) \ f) \) proof − have \( \text{Field} \ r' \leq f' (\text{Field} \ r) \) using BIJ by(auto simp add: bij-betw-def)
then have iso: \( iso \ r \ r' \ f \) by(simp add: BIJ EMB iso-def)
have \( *: \forall a. \ a \in \text{Field} \ r \rightarrow \text{bij-betw} \ f (\text{under} \ r \ a) (\text{under} \ r' \ (f \ a)) \) using EMB embed-def by fastforce
show \( ?\text{thesis} \)
\textbf{proof (clarsimp simp add: embed-def)}
fix \( a \)
assume \( a: \ a \in \text{Field} \ r' \)
then have \( ar: \ a \in f' (\text{Field} \ r) \)
using BIJ bij-betw-imp-surj-on by blast
have [simp]: \( f (\text{inv-into} (\text{Field} \ r) \ f \ a) = a \) by(simp add: ar f-inv-into-f)
show bij-betw (inv-into (Field r) f) (under r' a) (under r (inv-into (Field r) f a))
proof (rule bij-betw-inv-into-subset [OF BIJ])
28.3 Given any two well-orders, one can be embedded in the other

Here is an overview of the proof of of this fact, stated in theorem wellorders-totally-ordered:

Fix the well-orders \( r :: 'a \ rel \) and \( r' :: 'a' \ rel \). Attempt to define an embedding \( f :: 'a \Rightarrow 'a' \) from \( r \) to \( r' \) in the natural way by well-order recursion ("hoping" that \( \text{Field } r \) turns out to be smaller than \( \text{Field } r' \)), but also record, at the recursive step, in a function \( g :: 'a \Rightarrow \text{bool} \), the extra information of whether \( \text{Field } r' \) gets exhausted or not.

If \( \text{Field } r' \) does not get exhausted, then \( \text{Field } r \) is indeed smaller and \( f \) is the desired embedding from \( r \) to \( r' \) (lemma wellorders-totally-ordered-aux).

Otherwise, it means that \( \text{Field } r' \) is the smaller one, and the inverse of (the 'good' segment of) \( f \) is the desired embedding from \( r' \) to \( r \) (lemma wellorders-totally-ordered-aux2).

lemma wellorders-totally-ordered-aux:
fixes \( r :: 'a \ rel \) and \( r' :: 'a' \ rel \) and
\( f :: 'a \Rightarrow 'a' \) and \( a :: 'a' \)
assumes WELL: Well-order \( r \) and \( r' \) and\( IN: a \in \text{Field } r \)
and
IH: \( \forall b \in \text{underS } r \ a. \ bij-betw f (\text{under } r \ b) \ (\text{under } r' \ (f b)) \) and
NOT: \( f ' (\text{underS } r \ a) \neq \text{Field } r' \) and\( SUC: f a = \text{wo-rel.suc } r' \ (f(\text{underS } r \ a)) \)
shows \( \text{bij-betw } f (\text{under } r \ a) \ (\text{under } r' \ (f a)) \)
proof−

have Well: wo-rel \( r \) using WELL unfolding wo-rel-def .
hence Refl: Refl \( r \) using wo-rel.REFL[of \( r \)] by auto
have Trans: trans \( r \) using Well wo-rel.TRANS[of \( r \)] by auto
have Well': wo-rel \( r' \) using WELL' unfolding wo-rel-def .
have OF: wo-rel.ofilter \( r \) (underS \( r \ a) \)
by (auto simp add: Well wo-rel.underS-ofilter)
hence \( UN: \text{underS } r \ a = (\bigsqcup b \in \text{underS } r \ a. \ \text{under } r \ b) \)
using Well wo-rel.ofilter-under-UNION[of \( r \) underS \( r \ a)] \ by blast

\{fix \( b \) assume \( *: b \in \text{underS } r \ a \)
hence \( t0: (b,a) \in r \land b \neq a \) unfolding underS-def \ by auto
have \( t1: b \in \text{Field } r \)
THEORY "BNF-Wellorder-Embedding"

using * underS-Field[of r a] by auto
have t2: \( f'(\under r b) = \under r' (f b) \)
using IH * by (auto simp add: bij-betw-def)

hence t3: wo-rel.ofilter r' (f'(\under r b))
using Well' by (auto simp add: wo-rel.under-ofilter)
have f'(\under r b) \leq Field r'
using t2 by (auto simp add: under-Field)

moreover
have b \in \under r b
using t1 by (auto simp add: Refl Refl-under-in)
ultimately
have t4: \( f b \in Field r' \)
using t2 t3 t4 by auto

hence bFact:
\[ \forall b \in \underS r a. f'(\under r b) = under r' (f b) \land \]
wo-rel.ofilter r' (f'(\under r b)) \land
\( f b \in Field r' \)
by blast

have subField: f'(\underS r a) \leq Field r'
using bFact by blast

have OF': wo-rel.ofilter r' (f'(\underS r a))
proof
have f'(\underS r a) = f'(\bigcup b \in \underS r a. \under r b)
using UN by auto
also have \ldots = (\bigcup b \in \underS r a. f'(\under r b)) by blast
also have \ldots = (\bigcup b \in \underS r a. (under r' (f b)))
using bFact by auto
finally
have f'(\underS r a) = (\bigcup b \in \underS r a. (under r' (f b))) .
thus ?thesis
using Well' bFact
wo-rel.ofilter-UNION[of r' underS r a \lambda b. under r' (f b)] by fastforce
qed

have f'(\underS r a) \cup AboveS r' (f'(\underS r a)) = Field r'
using Well' OF' by (auto simp add: wo-rel.ofilter-AboveS-Field)

hence NE: AboveS r' (f'(\underS r a)) \neq \{
using subField NOT by blast

have INCL1: f'(\underS r a) \leq \underS r' (f a)
proof(auto)
fix b assume *: b \in \underS r a
have f b \neq f a \land \{f b, f a\} \in r'
using subField Well' SUC NE *
wo-rel.suc-greater[of r' f(underS r a) f b] by force
thus f b ∈ underS r' (f a)
unfolding underS-def by simp
qed

have INCL2: underS r' (f a) ≤ f'(underS r a)
proof
  fix b' assume b' ∈ underS r' (f a)
  hence b' ≠ f a ∧ (b', f a) ∈ r'
    unfolding underS-def by simp
  thus b' ∈ f'(underS r a)
    using Well' SUC NE OF'
    wo-rel.suc-ofilter-in[of r' f' underS r a b' by auto
qed

have INJ: inj-on f (underS r a)
proof
  have ∀ b ∈ underS r a. inj-on f (under r b)
    using IH by (auto simp add: bij-beta-def)
  moreover
  have ∀ b. wo-rel.ofilter r (under r b)
    using Well by (auto simp add: wo-rel.ofilter)
  ultimately show ?thesis
    using WELL bFact UN
    UNION-inj-on-ofilter[of r underS r a λb. under r b f]
    by auto
qed

have BIJ: bij-betw f (underS r a) (underS r' (f a))
    unfolding bij-beta-def
    using INJ INCL1 INCL2 by auto
have f a ∈ Field r'
  using Well' subField NE SUC
  by (auto simp add: wo-rel.suc-inField)
thus ?thesis
  using WELL WELL' IN BIJ under-underS-bij-betw[of r r' a f] by auto
qed

lemma wellorders-totally-ordered-aux2:
fixes r :: 'a rel and r' :: 'a' rel and
  f :: 'a ⇒ 'a' and g :: 'a ⇒ bool and a :: 'a
assumes WELL: Well-order r and WELL': Well-order r' and
MAIN1:
  ∀ a. (False ̸∈ g' (underS r a) ∧ f' (underS r a) ̸= Field r')
    → f a = wo-rel.suc r' (f' (underS r a)) ∧ g a = True
  ∧
  (¬(False ̸∈ (g' (underS r a)) ∧ f' (underS r a) ̸= Field r'))
    → g a = False) and
\[ \bigwedge a, a \in \text{Field } r \land \text{False } \notin g'(\text{under } r \ a) \rightarrow \]

bij-betw \( f (\text{under } r \ a) \) (\( \text{under } r' (f a) \)) \text{ and}

Case: \( a \in \text{Field } r \land \text{False } \notin g'(\text{under } r \ a) \)

shows \( \exists f'. \text{ embed } r' r f' \)

proof

have Well: wo-rel \( r \) using WELL unfolding wo-rel-def .

hence Refl: \( r \) using wo-rel.REFL[of \( r \)] by auto

have Trans: \( r \) using Well wo-rel.TRANS[of \( r \)] by auto

have Antisym: \( r \) using Well wo-rel.ANTISYM[of \( r \)] by auto

have Well': wo-rel \( r' \) using WELL' unfolding wo-rel-def .

have 0: \( \text{under } r \ a = \text{underS } r \ a \cup \{a\} \)

using Refl Case by(auto simp add: Refl-under-underS)

have 1: \( g \ a = \text{False} \)

proof

\{assume \( g \ a \neq \text{False} \)

with 0 Case have \( \text{False } \in g'(\text{underS } r \ a) \) by blast

with MAIN1 have \( g \ a = \text{False} \) by blast\}

thus \( ?\text{thesis} \) by blast

qed

let \( ?A = \{a \in \text{Field } r. \ g \ a = \text{False}\} \)

let \( ?a = (\text{wo-rel.minim } r \ ?A) \)

have 2: \( ?A \neq \{\} \land ?A \leq \text{Field } r \) using Case 1 by blast

have 3: \( \text{False } \notin g'(\text{underS } r \ ?a) \)

proof

assume \( \text{False } \in g'(\text{underS } r \ ?a) \)

then obtain \( b \) where \( b \in \text{underS } r \ ?a \) and 31: \( g \ b = \text{False} \) by auto

hence 32: \( (b, ?a) \in r \land b \neq ?a \)

by (auto simp add: underS-def)

hence \( b \in \text{Field } r \) unfolding Field-def by auto

with 31 have \( b \in ?A \) by auto

hence \( (?a, b) \in r \) using wo-rel.minim-least 2 Well by fastforce

with 32 Antisym show \( \text{False} \)

by (auto simp add: antisym-def)

qed

have temp: \( ?a \in ?A \)

using Well 2 wo-rel.minim-in[of \( r \ ?A \)] by auto

hence 4: \( ?a \in \text{Field } r \) by auto

have 5: \( g \ ?a = \text{False} \) using temp by blast

have 6: \( f'(\text{underS } r \ ?a) = \text{Field } r' \)

using MAIN1[of \( ?a \)] 3 5 by blast

have 7: \( \forall b \in \text{underS } r \ ?a. \text{ bij-betw } f (\text{under } r \ b) (\text{under } r' (f b)) \)
THEORY “BNF-Wellorder-Embedding” 681

proof
  fix b assume as: b ∈ underS r ?a
moreover
  have wo-rel.ofilter r (underS r ?a)
    using Well by (auto simp add: wo-rel.underS-ofilter)
ultimately
  have False ⋄ g′(under r b) using 3 Well by (subst (asm) wo-rel.ofilter-def)

fast+
  moreover have b ∈ Field r
    unfolding Field-def using as by (auto simp add: underS-def)
ultimately
  show bij-betw f (under r b) (under r′ (f b))
    using MAIN2 by auto
qed

have embed r′ r (inv-into (underS r ?a) f)
  using WELL WELL′ 7 4 6 inv-into-underS-embed[of r ?a f r′] by auto
thus ?thesis
  unfolding embed-def by blast
qed

theorem wellorders-totally-ordered:
  fixes r ‚: a rel and r′ ‚: † a rel
  assumes WELL: Well-order r and WELL′: Well-order r′
  shows (∃f. embed r r′ f) ∨ (∃f′. embed r′ r f′)
proof –
  have Well: wo-rel r using WELL unfolding wo-rel-def .
  hence Refl: Refl r using wo-rel.REFL[of r] by auto
  have Trans: trans r using Well wo-rel.TRANS[of r] by auto
  have Well′: wo-rel r′ using WELL′ unfolding wo-rel-def .

obtain H where H-def: H =
  (λh a. if False ⋄ (snd ⋄ h)(underS r a) ∧ (fst ⋄ h)(underS r a) ≠ Field r′
    then (wo-rel.suc r′ ((fst ⋄ h)(underS r a)), True)
    else (undefined, False)) by blast
  have Adm: wo-rel.adm-wo r H
    using Well
  proof(unfold wo-rel.adm-wo-def, clarify)
    fix h1::a ⇒ ′a ′* bool and h2::′a ′ ⇒ ′a ′* bool and x
    assume ∀ y ∈ underS r x. h1 y = h2 y
    hence ∀ y ∈ underS r x. (fst ∘ h1) y = (fst ∘ h2) y ∧
      (snd ∘ h1) y = (snd ∘ h2) y by auto
    hence (fst ∘ h1)(underS r x) = (fst ∘ h2)(underS r x) ∧
      (snd ∘ h1)(underS r x) = (snd ∘ h2)(underS r x)
      by (auto simp add: image-def)
    thus H h1 x = H h2 x by (simp add: H-def del: not-False-in-image-Ball)
  qed
obtain \( h \colon \alpha \rightarrow \alpha \rightarrow \text{bool} \) and \( f \colon \alpha \rightarrow \alpha \rightarrow \text{bool} \)
where \( h \text{-def} : h = \text{wo-rel\_worerec} r H \) and
\( f \text{-def} : f = \text{fst} \circ h \) and \( g \text{-def} : g = \text{snd} \circ h \) by blast
obtain \( \text{test} \) where \( \text{test} \text{-def} : \text{test} = (\lambda a. \text{False} \notin (g'(\text{underS} r a)) \land f'(\text{underS} r a) \neq \text{Field} r') \) by blast

have \( \star \colon \bigwedge a. \text{test} a = \text{True} \)

have \( \star \star \colon \alpha \in \text{Field} r \)

have \( \star \star \star \colon \text{test} a \rightarrow g a = \text{False} \)

have \( \text{Main1} : \bigwedge a. (\text{test} a 

have \( \text{Main2} : \bigwedge a. \text{?phi} a \)

have \( \text{21} : \text{False} \notin g'(\text{underS} r a) \)

have \( \text{22} : g'(\text{underS} r a) \leq \{ \text{True} \} \) using \( \star \star \star \)

have \( \text{23} : a \in \text{underS} r a \)

ultimately have \( \text{24} : g a = \text{True} \)

have \( \text{24} : g a = \text{True} \) by blast

proof

assume \( f'(\text{underS} r a) \neq \text{Field} r' \)

proof

assume \( f'(\text{underS} r a) = \text{Field} r' \)
hence $g \ a = \text{False}$ using Main1 test-def by blast
with 24 show False using ** by blast
qed

have $3: f \ a = \text{wo-rel.suc } r' (f'(\text{underS } r \ a))$
using 21 2 Main1 test-def by blast

show bij-betw $f (\text{under } r \ a) (\text{under } r' (f \ a))$
using WELL WELL' 1 2 3 *
wellorders-totally-ordered-aux[of $r \ r' \ a \ f$] by auto
qed

let $?chi = (\lambda a. \ a \in \text{Field } r \land \text{False } \in g(\text{under } r \ a))$
show ?thesis
proof (cases $\exists a. ?chi \ a$
  assume $\neg (\exists a. ?chi \ a)$
  hence $\forall a \in \text{Field } r. \ \text{bij-betw } f (\text{under } r \ a) (\text{under } r' (f \ a))$
  using Main2 by blast
  thus ?thesis unfolding embed-def by blast
next
  assume $\exists a. ?chi \ a$
  then obtain $a$ where $?chi \ a$ by blast
  hence $\exists f'. \ \text{embed } r' r \ f'$
  using wellorders-totally-ordered-aux[of $r \ r' \ g \ f \ a$
  WELL WELL' Main1 Main2 test-def by fast
  thus ?thesis by blast
qed

28.4 Uniqueness of embeddings

Here we show a fact complementary to the one from the previous subsection – namely, that between any two well-orders there is at most one embedding, and is the one definable by the expected well-order recursive equation. As a consequence, any two embeddings of opposite directions are mutually inverse.

lemma embed-determined:
assumes WELL: Well-order $r$ and WELL': Well-order $r'$ and
  EMB: embed $r \ r' \ f$ and IN: $a \in \text{Field } r$
shows $f \ a = \text{wo-rel.suc } r' (f'(\text{underS } r \ a))$
proof –
  have bij-betw $f (\text{underS } r \ a) (\text{underS } r' (f \ a))$
  using assms by (auto simp add: embed-underS)
  hence $f(\text{underS } r \ a) = \text{underS } r' (f \ a)$
    by (auto simp add: bij-betw-def)
  moreover
  have $f \ a \in \text{Field } r'$ using IN
using EMB WELL embed-Field[of r r’ f] by auto
hence f a = wo-rel.suc r’ (underS r’ (f a))
using WELL’ by (auto simp add: wo-rel-def wo-rel-suc-underS)
}
ultimately show thesis by simp
qed

lemma embed-unique:
assumes WELL: Well-order r and WELL’: Well-order r’ and
EMBf: embed r r’ f and EMBg: embed r r’ g
shows a ∈ Field r → f a = g a
proof(rule wo-rel.well-order-induct[of r], auto simp add: WELL wo-rel-def)
fix a
assume IH: ∀ b. b ≠ a ∧ (b,a) ∈ r → b ∈ Field r → f b = g b and
*: a ∈ Field r
hence ∀ b ∈ underS r a. f b = g b
unfolding underS-def by (auto simp add: Field-def)
hence f(underS r a) = g(underS r a) by force
thus f a = g a
using assms * embed-determined[of r r’ f a] embed-determined[of r r’ g a] by auto
qed

lemma embed-bothWays-inverse:
assumes WELL: Well-order r and WELL’: Well-order r’ and
EMB: embed r r’ f and EMB’: embed r’ r f’
shows (∀ a ∈ Field r. f'(f a) = a) ∧ (∀ a’ ∈ Field r’. f(f’ a’) = a’)
proof
have embed r r (f’ o f) using assms
  by(auto simp add: comp-embed)
moreover have embed r r id using assms
  by (auto simp add: id-embed)
ultimately have ∀ a ∈ Field r. f'(f a) = a
  using assms embed-unique[of r r f’ o f id] id-def by auto
moreover
{have embed r’ r’ (f o f’) using assms
  by(auto simp add: comp-embed)
  moreover have embed r’ r’ id using assms
    by (auto simp add: id-embed)
  ultimately have ∀ a’ ∈ Field r’. f(f’ a’) = a’
    using assms embed-unique[of r’ r’ f o f’ id] id-def by auto
}
ultimately show thesis by blast
qed

lemma embed-bothWays-bij-betw:
assumes WELL: Well-order r and WELL’: Well-order r’ and
EMB: embed r r’ f and EMB’: embed r’ r g
shows bij-betw f (Field r) (Field r’)

proof
  let ?A = Field r  let ?A' = Field r'
  have embed r r (g o f) ∧ embed r' r' (f o g)
    using assms by (auto simp add: comp-embed)
  hence 1: (∀a ∈ ?A. g(f a) = a) ∧ (∀a' ∈ ?A'. f(g a') = a')
    using WELL id-embed[of r] embed-unique[of r r g o f id]
    id-def by auto
  have 2: (∀a ∈ ?A. f a ∈ ?A') ∧ (∀a' ∈ ?A'. g a' ∈ ?A)
    using assms embed-Field[of r r'] embed-Field[of r' r g]
    by blast
  show ?thesis
proof (unfold bij-betw-def inj-on-def, auto simp add: 2)
  fix a b assume *: a ∈ ?A b ∈ ?A and **: f a = f b
  have a = g(f a) ∧ b = g(f b) using * 1 by auto
  with ** show a = b by auto
next
  fix a' assume *: a' ∈ ?A'
  hence g a' ∈ ?A ∧ f(g a') = a' using 1 2 by auto
  thus a' ∈ f' ?A by force
qed

lemma embed-bothWays-iso:
  assumes WELL: Well-order r and WELL': Well-order r'
  and EMB: embed r r' f and EMB': embed r' r g
  shows iso r r' f
unfolding iso-def using assms by (auto simp add: embed-bothWays-bij-betw)

28.5 More properties of embeddings, strict embeddings and isomorphisms

lemma embed-bothWays-Field-bij-betw:
  assumes WELL: Well-order r and WELL': Well-order r'
  and EMB: embed r r' f and EMB': embed r' r f'
  shows bij-betw f (Field r) (Field r')
proof
  have (∀a ∈ Field r. f'(f a) = a) ∧ (∀a' ∈ Field r'. f(f' a') = a')
    using assms by (auto simp add: embed-bothWays-inverse)
  moreover
  have f'(Field r) ⊆ Field r' ∧ f (Field r') ⊆ Field r
    using assms by (auto simp add: embed-Field)
  ultimately
  show ?thesis using bij-betw-byWitness[of Field r f' f Field r'] by auto
qed

lemma embedS-comp-embed:
  assumes WELL: Well-order r and WELL': Well-order r'
  and EMB: embedS r r' f and EMB': embed r' r'' f'

shows \( \text{embed} S \ r r'' (f' \circ f) \)

**proof**

- let \(?g = (f' \circ f)\)  let \(?h = \text{inv-into} (\text{Field} r) ?g\)
- have 1: \(\text{embed} r r' f \land \lnot (\text{bij-betw} f (\text{Field} r) (\text{Field} r'))\)
  - using \(\text{EMB}\) by (auto simp add: embedS-def)
- hence 2: \(\text{embed} r r'' ?g\)
  - using \(\text{EMB}' \text{comp-embed}[of r r' f r'' f']\) by auto

**moreover**

{assume \(\text{bij-betw} ?g (\text{Field} r) (\text{Field} r'')\)
  - hence \(\text{embed} r'' r ?h\) using 2
    - by (auto simp add: inv-into-Field-embed-bij-betw)
  - hence \(\text{embed} r' r (?h \circ f')\) using \(\text{EMB}'\)
    - by (auto simp add: comp-embed)
  - hence \(\text{bij-betw} f (\text{Field} r) (\text{Field} r')\) using \(\text{WELL WELL}' 1\)
    - by (auto simp add: embed-bothWays-Field-bij-betw)
  - with 1 have False by blast}

ultimately show \(?\text{thesis}\) unfolding embedS-def by auto

qed

**lemma** \(\text{embed-comp-embedS}\):

**assumes** \(\text{WELL}: \text{Well-order} r \text{ and WELL}': \text{Well-order} r'\)
  - and \(\text{EMB}: \text{embed} r r' f \text{ and EMB'}: \text{embed} S r' r'' f'\)

**shows** \(\text{embed} S r r'' (f' \circ f)\)

**proof**

- let \(?g = (f' \circ f)\)  let \(?h = \text{inv-into} (\text{Field} r) ?g\)
- have 1: \(\text{embed} r' r'' f' \land \lnot (\text{bij-betw} f' (\text{Field} r') (\text{Field} r''))\)
  - using \(\text{EMB}'\) by (auto simp add: embedS-def)
- hence 2: \(\text{embed} r r'' ?g\)
  - using \(\text{WELL EMB} \text{comp-embed}[of r r' f r'' f']\) by auto

**moreover** have §: \(f' \cdot \text{Field} r' \subseteq \text{Field} r''\)
  - by (simp add: 1 embed-Field)
{assume §: \(\text{bij-betw} ?g (\text{Field} r) (\text{Field} r'')\)
  - hence \(\text{embed} r'' r ?h\) using 2 WELL
    - by (auto simp add: inv-into-Field-embed-bij-betw)
  - hence \(\text{embed} r' r (\text{inv-into} (\text{Field} r) ?g \circ f')\)
    - using 1 BNF-Wellorder-Embedding.comp-embed WELL' by blast
  - then have \(\text{bij-betw} f' (\text{Field} r') (\text{Field} r'')\)
    - using \(\text{EMB} \text{WELL WELL}' § \text{bij-betw-comp-iff}\) by (blast intro: embed-bothWays-Field-bij-betw)
  - with 1 have False by blast}

ultimately show \(?\text{thesis}\) unfolding embedS-def by auto

qed

**lemma** \(\text{embed-comp-iso}\):

**assumes** \(\text{EMB}: \text{embed} r r' f \text{ and EMB'}: \text{iso} r r'' f'\)

**shows** \(\text{embed} r r'' (f' \circ f)\)
  - using assms unfolding iso-def
  - by (auto simp add: comp-embed)
lemma iso-comp-embed:
assumes EMB: iso \( r \rightarrow f \) and EMB': iso \( r' \rightarrow f' \)
shows embed \( r'' \)(f' \circ f)
using assms unfolding iso-def by (auto simp add: comp-embed)

lemma embedS-comp-iso:
assumes EMB: embedS \( r \rightarrow f \) and EMB': iso \( r' \rightarrow f' \)
shows embedS \( r'' \)(f' \circ f)
proof |
  have §: embed \( r \rightarrow f \) \& \& \& \neg bij-betw f (Field r) (Field r')
    using EMB embedS-def by blast
  then have embed \( r'' \)(f' \circ f)
    using embed-comp-iso EMB' by blast
  then have f' Field r \subseteq Field r'
    by (simp add: embed-Field §)
  then have \neg bij-betw (f' \circ f) (Field r) (Field r'')
    using § EMB' by (auto simp: bij-betw-comp-iff2 iso-def)
  then show thesis
    by (simp add: embed-Field f' Field r \& \& \& \neg bij-betw (f' \circ f) (Field r) (Field r''))
qed

lemma iso-comp-embedS:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \)
  and EMB: iso \( r \rightarrow f \) and EMB': embedS \( r' \rightarrow f' \)
shows embedS \( r'' \)(f' \circ f)
using assms unfolding iso-def by (auto simp add: embed-comp-embedS)

lemma embedS-Field:
assumes WELL: Well-order \( r \) and EMB: embedS \( r \rightarrow f \)
shows f ' (Field r) \lt Field r'
proof |
  have f' (Field r) \leq Field r' using assms
    by (auto simp add: embed-Field embedS-def)
  moreover
  {have inj-on f (Field r) using assms
    by (auto simp add: embedS-def embed-inj-on)
    hence f' (Field r) \neq Field r' using EMB
      by (auto simp add: embedS-def bij-betw-def)
  }
  ultimately show thesis by blast
qed

lemma embedS-iff:
assumes WELL: Well-order \( r \) and ISO: embed \( r \rightarrow f \)
shows embedS \( r \rightarrow f \) = (f' (Field r) \lt Field r')
proof
assume embedS \( r \rightarrow f \)
thus f' Field r \subseteq Field r'
  using WELL by (auto simp add: embedS-Field)
next
  assume \( f : \text{Field } r \subseteq \text{Field } r' \)
  hence \( \sim \text{bij-betw } f (\text{Field } r) (\text{Field } r') \)
  unfolding \text{bij-betw-def} by blast
thus embedS \( r \to r' \) unfolding embedS-def
  using ISO by auto
qed

lemma iso-Field: \( \text{iso } r \to r' \implies f \cdot (\text{Field } r) = \text{Field } r' \)
  by (auto simp add: iso-def bij-betw-def)

lemma iso-iff:
  assumes \( \text{Well-order } r \)
  shows \( \text{iso } r \to r' \) \( \equiv \) \( (\text{embed } r \to r' \ f \wedge f \cdot (\text{Field } r) = \text{Field } r') \)
proof
  assume \( \text{iso } r \to r' \ f \)
  thus \( \text{embed } r \to r' \ f \wedge f \cdot (\text{Field } r) = \text{Field } r' \)
      by (auto simp add: iso-Field iso-def)
next
  assume \( \ast \): \( \text{embed } r \to r' \ f \wedge f \cdot (\text{Field } r) = \text{Field } r' \)
  hence inj-on \( f \) (\text{Field } r) using assms by (auto simp add: embed-inj-on)
  with \( \ast \) have \( \text{bij-betw } f \) (\text{Field } r) (\text{Field } r')
      unfolding \text{bij-betw-def} by simp
  with \( \ast \) show \( \text{iso } r \to r' \) unfolding iso-def by auto
qed

lemma iso-iff2: \( \text{iso } r \to r' \ f \iff \) \( \text{bij-betw } f \) (\text{Field } r) (\text{Field } r') \wedge
  \( \forall a \in \text{Field } r. \forall b \in \text{Field } r. (a, b) \in r \iff (f a, f b) \in r' \)
(is ?lhs = ?rhs)
proof
assumption L: ?lhs
then have \( \text{bij-betw } f \) (\text{Field } r) (\text{Field } r') \text{ and } \text{emb: embed } r \to r' \ f
  by (auto simp: bij-betw-def iso-def)
then obtain \( g \) where \( g \cdot \) (\text{Field } r) = \( g (f x) = x \)
  by (auto simp: bij-betw-iff-bijections)
moreover
have \( (a, b) \in r \) if \( a \in \text{Field } r \) \( b \in \text{Field } r \) \( (f a, f b) \in r' \) for \( a \) \( b \)
  using that \( \text{emb } g \) \( [\text{OF FieldI1]} \) — yes it's weird
  by (force simp add: embed-def under-def bij-betw-iff-bijections)
ultimately show ?rhs
using L by (auto simp: compat-def iso-def dest: embed-compat)
next
assume R: ?rhs
then show ?rhs
  apply (clarsimp simp add: iso-def embed-def under-def bij-betw-iff-bijections)
  apply (rule-tac \( x=g \) in exI)
  apply (fastforce simp add: intro: FieldII)+
done
lemma iso-iff3:
  assumes WELL: Well-order r and WELL': Well-order r'
  shows iso r r' f = (bij-betw f (Field r) (Field r') ∧ compat r r' f)
proof
  assume iso r r' f
  thus bij-betw f (Field r) (Field r') ∧ compat r r' f
    unfolding compat-def using WELL by (auto simp add: iso-iff2 Field-def)
next
  have Well: wo-rel r ∧ wo-rel r' using WELL WELL'
    by (auto simp add: wo-rel-def)
  assume ∗: bij-betw f (Field r) (Field r') ∧ compat r r' f
  thus iso r r' f
    unfolding compat-def using assms
    proof (auto simp add: iso-iff2)
      fix a b assume ∗∗: a ∈ Field r b ∈ Field r and
      ***: (f a, f b) ∈ r'
      {assume (b,a) ∈ r ∨ b = a
        hence (b,a) ∈ using Well ** wo-rel.REFL[of r] refl-on-def[of - r] by blast
        hence (f b, f a) ∈ r' using * unfolding compat-def by auto
        hence f a = f b
          using Well *** wo-rel.ANTISYM[of r'] antisym-def[of r'] by blast
        hence a = b using * ** unfolding bij-betw-def inj-on-def by auto
        hence (a,b) ∈ r using Well ** wo-rel.REFL[of r] refl-on-def[of - r] by blast
      }
      thus (a,b) ∈ r
        using Well ** wo-rel.TOTAL[of r] total-on-def[of - r] by blast
    qed
  qed

lemma iso-imp-inj-on:
  assumes iso r r' f shows inj-on f (Field r)
  using assms unfolding iso-iff2 bij-betw-def by blast

lemma iso-backward-Field:
  assumes x ∈ Field r' iso r r' f
  shows inv-into (Field r) f x ∈ Field r
  using assms iso-Field by (blast intro: inv-into-into)

lemma iso-backward:
  assumes (x,y) ∈ r' and iso: iso r r' f
  shows (inv-into (Field r) f x, inv-into (Field r) f y) ∈ r
proof
  have §: \x. (∃xa ∈ Field r. x = f xa) = (x ∈ Field r')
    using assms iso-Field [OF iso] by (force simp add: )
  have x ∈ Field r' y ∈ Field r'
    using assms by (auto simp: Field-def)
  with § [of x] § [of y] assms show ?thesis
by (auto simp add: iso-iff2 bij-betw-inv-into-left)

qed

lemma iso-forward:
  assumes \((x, y) \in r\) iso \(r \leftrightarrow f\)
  shows \((f x, f y) \in r'\)
  using assms by (auto simp: Field-def iso-iff2)

lemma iso-trans:
  assumes \(\text{trans } r\) and \(\text{iso } r \leftrightarrow f\)
  shows \(\text{trans } r'\)
  unfolding trans-def
  proof clarify
  fix \(x\) \(y\) \(z\)
  assume \(xyz\): \((x, y) \in r'\) \((y, z) \in r'\)
  then have \(\ast\): \((\text{inv-into } (\text{Field } r) f x, \text{inv-into } (\text{Field } r) f y) \in r\)
    using iso-backward \([OF - iso]\) by blast+
  then have \((\text{inv-into } (\text{Field } r) f x) \in \text{Field } r\) \((\text{inv-into } (\text{Field } r) f y) \in \text{Field } r\)
    by (auto simp: Field-def)
  moreover have \((\text{inv-into } (\text{Field } r) f x, \text{inv-into } (\text{Field } r) f z) \in r'\)
    using assms \(1\) by (blast intro: transD)
  ultimately have \(False\)
    if \((\text{inv-into } (\text{Field } r) f x, \text{inv-into } (\text{Field } r) f y) \in r\)
    using assms \(\ast\) by simp
  then have \((\text{inv-into } (\text{Field } r) f x, \text{inv-into } (\text{Field } r) f z) \in r\)
    using assms \(\ast\) by (auto simp: Field-def)
  moreove have \((x, f) \in f' \text{Field } r\) \((z, f) \in f' \text{Field } r\)
    using \(xyz\) iso \(\text{Field } r\) \(\text{Field } r\)
    by (blast intro: FieldI1 FieldI2)+
  ultimately show \((x, z) \in r'\)
    by (simp add: f-inv-into-f)
  qed

lemma iso-Total:
  assumes \(\text{Total } r\) and \(\text{iso } r \leftrightarrow f\)
  shows \(\text{Total } r'\)
  unfolding total-on-def
  proof clarify
  fix \(x\) \(y\)
  assume \(xy\): \((x, y) \in r'\) \((y, x) \notin r'\)
  then have \(\$: \text{inv-into } (\text{Field } r) f x \in \text{Field } r\) \((\text{inv-into } (\text{Field } r) f y) \in \text{Field } r\)
    using iso-backward-Field \([OF - iso]\) by auto
  moreover have \((x, f') \in f' \text{Field } r\) \((y, f') \in f' \text{Field } r\)
    using \(xy\) iso \(\text{Field } r\) \(\text{Field } r\)
    by (blast intro: FieldI1 FieldI2)+
  ultimately have \(False\) if \((\text{inv-into } (\text{Field } r) f x) = \text{inv-into } (\text{Field } r) f y\)
    using inv-into-injective \([OF that]\) \(x \neq y\) by simp
  then have \((\text{inv-into } (\text{Field } r) f x, \text{inv-into } (\text{Field } r) f y) \in r \lor \text{inv-into } (\text{Field } r) f x \in r\)
    using assms \(\$: \text{total-on-def}\)
  then show \((x, y) \in r'\)
    using assms \(\$: \text{auto simp: iso-Field f-inv-into-f dest!: iso-forward [OF - iso]}\)
  qed

lemma iso-wf:
assumes $\text{wf } r$ and $\text{iso } r \rightarrow r'$ shows $\text{wf } r'$

proof –

have bij-betw $f$ (Field $r$) (Field $r'$)
  and iff: $(\forall a \in \text{Field } r. \forall b \in \text{Field } r. (a, b) \in r \leftrightarrow (f a, f b) \in r')$
  using assms by (auto simp: iso-iff2)
show ?thesis
proof (rule wfI-min)
  fix $x$ and $Q$
  assume $x \in Q$
  let $?g = \text{inv-into } (\text{Field } r) f$
  obtain $z0$ where $z0 \in ?g' Q$
    using $x \in Q$ by blast
  then obtain $z$ where $z : z \in ?g' Q$ and $\forall x y. [(y, z) \in r; x \in Q] \Longrightarrow y \neq ?g x$
    by (rule wfE-min [OF $\text{wf } r$]) auto
  then have $\forall y. (y, \text{inv-into } Q ?g z) \in r' \Longrightarrow y \notin Q$
    by (clarsimp simp: f-inv-into-f [OF $z$ dest: iso-backward [OF - iso]] blast)
moreover have $\text{inv-into } Q ?g z \in Q$
  by (simp add: inv-into-into $z$
ultimately show $\exists z \in Q. \forall y. (y, z) \in r' \Longrightarrow y \notin Q$ ..
qed
qed

29 Constructions on Wellorders as Needed by Bounded Natural Functors

theory BNF-Wellorder-Constructions
  imports BNF-Wellorder-Embedding
begin

In this section, we study basic constructions on well-orders, such as restriction to a set/order filter, copy via direct images, ordinal-like sum of disjoint well-orders, and bounded square. We also define between well-orders the relations $\text{ordLeq}$, of being embedded (abbreviated $\leq o$), $\text{ordLess}$, of being strictly embedded (abbreviated $< o$), and $\text{ordIso}$, of being isomorphic (abbreviated $= o$). We study the connections between these relations, order filters, and the aforementioned constructions. A main result of this section is that $< o$ is well-founded.

29.1 Restriction to a set

abbreviation Restr :: 'a rel $\Rightarrow$ 'a set $\Rightarrow$ 'a rel
  where Restr $r A \equiv r \text{ Int } (A \times A)$

lemma Restr-subset:
\( A \leq B \implies \text{Restr} (\text{Restr} r B) A = \text{Restr} r A \)

by blast

lemma Restr-Field: \( \text{Restr} r (\text{Field} r) = r \)
unfolding Field-def by auto

lemma Refl-Restr: \( \text{Refl} r \implies \text{Refl} (\text{Restr} r A) \)
unfolding refl-on-def Field-def by auto

lemma linear-order-on-Restr:
linear-order-on A r \implies linear-order-on \( A \cap \text{above} r x \) (Restr r (above r x))
by(simp add: order-on-defs refl-on-def trans-def antisym-def total-on-def)(safe; blast)

lemma antisym-Restr:
antisym r \implies antisym (Restr r A)
unfolding antisym-def Field-def by auto

lemma Total-Restr:
Total r \implies Total (Restr r A)
unfolding total-on-def Field-def by auto

lemma total-on-imp-Total-Restr: total-on A r \implies Total (Restr r A)
by (auto simp: Field-def total-on-def)

lemma trans-Restr:
trans r \implies trans (Restr r A)
unfolding trans-def Field-def by blast

lemma Preorder-Restr:
Preorder r \implies Preorder (Restr r A)
unfolding preorder-on-def by (simp add: Refl-Restr trans-Restr)

lemma Partial-order-Restr:
Partial-order r \implies Partial-order (Restr r A)
unfolding partial-order-on-def by (simp add: Preorder-Restr antisym-Restr)

lemma Linear-order-Restr:
Linear-order r \implies Linear-order (Restr r A)
unfolding linear-order-on-def by (simp add: Partial-order-Restr Total-Restr)

lemma Well-order-Restr:
assumes Well-order r
shows Well-order (Restr r A)
using assms
by (auto simp: well-order-on-def Linear-order-Restr elim: wf-subset)

lemma Field-Restr-subset: Field (Restr r A) \leq A
by (auto simp add: Field-def)
lemma Refl-Field-Restr:
Refl r ⇒ Field(Restr r A) = (Field r) Int A
unfolding refl-on-def Field-def by blast

lemma Refl-Field-Restr2:
[Refl r; A ≤ Field r] ⇒ Field(Restr r A) = A
by (auto simp add: Refl-Field-Restr)

lemma well-order-on-Restr:
assumes WELL: Well-order r and SUB: A ≤ Field r
shows well-order-on A (Restr r A)
using assms
using Well-order-Restr[of r A] Refl-Field-Restr2[of r A]
order-on-defs[of Field r r]
by auto

29.2 Order filters versus restrictions and embeddings

lemma Field-Restr-ofilter:
[Well-order r; wo-rel.ofilter r A] ⇒ Field(Restr r A) = A
by (auto simp add: wo-rel-def wo-rel.ofilter-def wo-rel.REFL Refl-Field-Restr2)

lemma ofilter-Restr-under:
assumes WELL: Well-order r and OF: wo-rel.ofilter r A and IN: a ∈ A
shows under (Restr r A) a = under r a
unfolding wo-rel.ofilter-def under-def
proof
show {b. (b, a) ∈ Restr r A} ⊆ {b. (b, a) ∈ r}
  by auto
next
have under r a ⊆ A
proof
  fix x
  assume #: x ∈ under r a
  then have a ∈ Field r
  unfolding under-def using Field-def by fastforce
  then show x ∈ A using IN assms #
    by (auto simp add: wo-rel-def wo-rel.ofilter-def)
qed
then show {b. (b, a) ∈ r} ⊆ {b. (b, a) ∈ Restr r A}
  unfolding under-def using assms by auto
qed

lemma ofilter-embed:
assumes Well-order r
shows wo-rel.ofilter r A = (A ≤ Field r ∧ embed (Restr r A) r id)
proof
assume #: wo-rel.ofilter r A
show A ≤ Field r ∧ embed (Restr r A) r id
unfolding embed-def
proof safe
  fix a assume a ∈ A thus a ∈ Field r using assms *
    by (auto simp add: wo-rel-def wo-rel_ofilter-def)
next
  fix a assume a ∈ Field (Restr r A)
  thus bij-betw id (under (Restr r A) a) (under r (id a)) using assms *
    by (simp add: ofilter-Restr-under Field-Restr-ofilter)
qed
next
assume *: A ≤ Field r ∧ embed (Restr r A) r id
hence Field(Restr r A) ≤ Field r
  using assms embed-Field[of Restr r A r id] id-def
  Well-order-Restr[of r] by auto
{ fix a assume a ∈ A
  hence a ∈ Field(Restr r A) using * assms
    by (simp add: order-on-defs Refl-Field-Restr2)
hence bij-betw id (under (Restr r A) a) (under r a)
  using * unfolding embed-def by auto
hence under r a ≤ under (Restr r A) a
  unfolding bij-betw-def by auto
also have ... ≤ Field(Restr r A) by (simp add: under-Field)
also have ... ≤ A by (simp add: Field-Restr-subset)
finally have under r a ≤ A .
}
thus wo-rel_ofilter r A using assms * by (simp add: wo-rel-def wo-rel_ofilter-def)
qed

lemma ofilter-Restr-Int:
  assumes WELL: Well-order r and OFA: wo-rel_ofilter r A
  shows wo-rel_ofilter (Restr r B) (A Int B)
proof –
  let ?rB = Restr r B
  have Well: wo-rel r unfolding wo-rel-def using WELL .
hence Refl: Refl r by (simp add: wo-rel_REFL)
hence Field: Field ?rB = Field r Int B
  using Refl-Field-Restr by blast
  have WellB: wo-rel ?rB ∧ Well-order ?rB using WELL
    by (simp add: Well-order-Restr wo-rel-def)
show *thesis using WellB assms
unfolding wo-rel_ofilter-def under-def ofilter-def
proof safe
  fix a assume a ∈ A and *: a ∈ B
  hence a ∈ Field r using OFA Well by (auto simp add: wo-rel_ofilter-def)
  with * show a ∈ Field ?rB using Field by auto
next
  fix a b assume a ∈ A and (b,a) ∈ r
  thus b ∈ A using Well OFA by (auto simp add: wo-rel_ofilter-def under-def)
lemma `ofilter-Restr-subset`:
  assumes WELL: `Well-order r` and OFA: `wo-rel.ofilter r A` and SUB: `A ≤ B`
  shows `wo-rel.ofilter (Restr r B) A`
proof –
  have `A ∩ B = A` using `SUB` by blast
  thus `?thesis` using `assms ofilter-Restr-Int[of r A B]` by auto
qed

lemma `ofilter-subset-embed`:
  assumes WELL: `Well-order r` and OFA: `wo-rel.ofilter r A` and OFB: `wo-rel.ofilter r B`
  shows `(A ≤ B) = (embed (Restr r A) (Restr r B) id)`
proof –
  let `?rA = Restr r A` let `?rB = Restr r B`
  have `Well: wo-rel r unfolding wo-rel-def using WELL` .
  hence `Refl: Refl r by (simp add: wo-rel.REFL)`
  hence `FieldA: Field `?rA = Field r ∩ A`
    using `Refl-Field-Restr` by blast
  have `FieldB: Field `?rB = Field r ∩ B`
    using `Refl Refl-Field-Restr` by blast
  have `WellA: wo-rel ?rA ∧ Well-order ?rA using WELL`
    by (simp add: `Well-order-Restr wo-rel-def`)
  have `WellB: wo-rel ?rB ∧ Well-order ?rB using WELL`
    by (simp add: `Well-order-Restr wo-rel-def`)
show `?thesis`
proof
  assume `∗: A ≤ B`
  hence `wo-rel.ofilter (Restr r B) A` using `assms` by (simp add: `ofilter-Restr-subset`)
  hence `embed (Restr ?rB A) (Restr r B) id`
    using `WellB ofilter-embed[of ?rB A]` by auto
  thus `embed (Restr r A) (Restr r B) id`
    using `∗` by (simp add: `Restr-subset`)
next
  assume `∗: embed (Restr r A) (Restr r B) id`
  {fix `a` assume `∗∗: a ∈ A`
    hence `a ∈ Field r using Well OFA by (auto simp add: wo-rel.ofilter-def)`
      with `∗∗ FieldA have `a ∈ Field ?rA` by auto
    hence `a ∈ Field ?rB using `WellA embed-Field[of ?rA ?rB id] by auto
      hence `a ∈ B using FieldB by auto`}
  thus `A ≤ B` by blast
qed
lemma ofilter-subset-embedS-iso:
  assumes WELL: Well-order r and
          OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
  shows \((A < B) = (\text{embedS} (\text{Restr} r A) (\text{Restr} r B) \text{id})) \land
          ((A = B) = (\text{iso} (\text{Restr} r A) (\text{Restr} r B) \text{id}))
proof
  let \(?rA = \text{Restr} r A\) let \(?rB = \text{Restr} r B\)
  have Well: wo-rel r unfolding wo-rel-def using WELL.
  hence Field \(?rA = \text{Field} r \text{Int} A\)
    using Refl-Field-Restr by blast
  hence FieldA: Field \(?rA = A\) using OFA Well
    by (auto simp add: wo-rel.ofilter-def)
  have Field \(?rB = \text{Field} r \text{Int} B\)
    using Refl-Field-Restr by blast
  hence FieldB: Field \(?rB = B\) using OFB Well
    by (auto simp add: wo-rel.ofilter-def)
  show \(?thesis unfolding embedS-def iso-def
    using assms ofilter-subset-embedS-iso\)
  qed

lemma ofilter-subset-embedS:
  assumes WELL: Well-order r and
          OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
  shows \((A < B) = \text{embedS} (\text{Restr} r A) (\text{Restr} r B) \text{id})\)
  using assms by (simp add: ofilter-subset-embedS-iso)

lemma embed-implies-iso-Restr:
  assumes WELL: Well-order r and WELL': Well-order r' and
          EMB: \(\text{embed} r' r f\)
  shows iso r' (\(\text{Restr} r (f' (\text{Field} r'))\)) f
proof
  let \(?A' = \text{Field} r'\)
  let \(?r'' = \text{Restr} r (f' ?A')\)
  have 0: Well-order \(?r''\) using WELL Well-order-Restr by blast
  have 1: wo-rel.ofilter r (f' ?A') using assms embed-Field-ofilter by blast
  hence Field \(?r'' = f' (\text{Field} r')\) using WELL Field-Restr-ofilter by blast
  hence bij-betw f ?A' (Field \(?r''\))
    using EMB embed-inj-on WELL' unfolding bij-betw-def by blast
moreover
  have \(\forall a. b. (a,b) \in r' \rightarrow a \in \text{Field} r' \land b \in \text{Field} r'\)
    unfolding Field-def by auto
  hence compat r' ?r'' f
    using assms embed-iff-compat-inj-on-ofilter
    unfolding compat-def by blast
  }
ultimately show \(?thesis using WELL' 0 iso-iff3 by blast\)
29.3 The strict inclusion on proper ofilters is well-founded

**definition** ofilterIncl : 'a rel ⇒ 'a set rel

where

ofilterIncl r ≡ 
{ (A,B). wo-rel.ofilter r A ∧ A ≠ Field r ∧ wo-rel.ofilter r B ∧ B ≠ Field r ∧ A < B }

**lemma** wf-ofilterIncl:

assumes WELL: Well-order r

shows wf(ofilterIncl r)

**proof**

have Well: wo-rel r using WELL by (simp add: wo-rel-def)

hence Lo: Linear-order r by (simp add: wo-rel.LIN)

let ?h = (λ A. wo-rel.suc r A)

let ?rS = r − Id

have wf ?rS using WELL by (simp add: order-on-defs)

moreover

have compat (ofilterIncl r) ?rS ?h

**proof**(unfold compat-def ofilterIncl-def,
intro allI impI, simp, elim conjE)

fix A B

assume *: wo-rel.ofilter r A A ≠ Field r and

**: wo-rel.ofilter r B B ≠ Field r and ***: A < B

then obtain a and b where 0: a ∈ Field r ∧ b ∈ Field r and

1: A = underS r a ∧ B = underS r b

using Well by (auto simp add: wo-rel.ofilter-underS-Field)

hence a ≠ b using *** by auto

moreover

have (a,b) ∈ r using 0 1 Lo ***

by (auto simp add: underS-incl-iff)

moreover

have a = wo-rel.suc r A ∧ b = wo-rel.suc r B

using Well 0 1 by (simp add: wo-rel.suc-underS)

ultimately

show (wo-rel.suc r A, wo-rel.suc r B) ∈ r ∧ wo-rel.suc r A ≠ wo-rel.suc r B

by simp

qed

ultimately show wf (ofilterIncl r) by (simp add: compat-wf)

qed

29.4 Ordering the well-orders by existence of embeddings

We define three relations between well-orders:

- **ordLeq**, of being embedded (abbreviated ≤ o);
- **ordLess**, of being strictly embedded (abbreviated < o);
• ordIso, of being isomorphic (abbreviated =o).

The prefix "ord" and the index "o" in these names stand for "ordinal-like". These relations shall be proved to be inter-connected in a similar fashion as the trio \(\leq, <, =\) associated to a total order on a set.

**definition ordLeq :: \(\langle \mathcal{L} \rangle \rel \times \mathcal{L} \rel \) set**
where
\[
\text{ordLeq} = \{ (r, r') : \text{Well-order } r \land \text{Well-order } r' \land (\exists f. \text{embed } r, r' f) \}
\]

**abbreviation ordLeq2 :: \(\langle \mathcal{L} \rangle \rel \Rightarrow \mathcal{L} \rel \Rightarrow \) bool** (infix \(\leq_o\ 50\))
where \(r \leq_o r' \equiv (r, r') \in \text{ordLeq}\)

**abbreviation ordLeq3 :: \(\langle \mathcal{L} \rangle \rel \Rightarrow \mathcal{L} \rel \Rightarrow \) bool** (infix \(<_o\ 50\))
where \(r <_o r' \equiv (r, r') \in \text{ordLeq}\)

**definition ordLess :: \(\langle \mathcal{L} \rangle \rel \times \mathcal{L} \rel \) set**
where
\[
\text{ordLess} = \{ (r, r') : \text{Well-order } r \land \text{Well-order } r' \land (\exists f. \text{embedS } r, r' f) \}
\]

**abbreviation ordLess2 :: \(\langle \mathcal{L} \rangle \rel \Rightarrow \mathcal{L} \rel \Rightarrow \) bool** (infix \(<_o\ 50\))
where \(r <_o r' \equiv (r, r') \in \text{ordLess}\)

**definition ordIso :: \(\langle \mathcal{L} \rangle \rel \times \mathcal{L} \rel \) set**
where
\[
\text{ordIso} = \{ (r, r') : \text{Well-order } r \land \text{Well-order } r' \land (\exists f. \text{iso } r, r' f) \}
\]

**abbreviation ordIso2 :: \(\langle \mathcal{L} \rangle \rel \Rightarrow \mathcal{L} \rel \Rightarrow \) bool** (infix \(=_o\ 50\))
where \(r =_o r' \equiv (r, r') \in \text{ordIso}\)

**lemmas ordRels-def = ordLeq-def ordLess-def ordIso-def**

**lemma ordLeq-Well-order-simp:**
assumes \(r \leq_o r'\)
shows \(\text{Well-order } r \land \text{Well-order } r'\)
using assms unfolding ordLeq-def by simp

Notice that the relations \(\leq_o, <_o, =_o\) connect well-orders on potentially distinct types. However, some of the lemmas below, including the next one, restrict implicitly the type of these relations to \(\langle \mathcal{L} \rangle \rel \times \mathcal{L} \rel \) set, i.e., to \(\mathcal{L} \rel \).

**lemma ordLeq-reflexive:**
Well-order \(r \Rightarrow r \leq_o r\)
unfolding ordLeq-def using id-embed[of r] by blast

**lemma ordLeq-transitive[trans]:**
assumes \(r \leq_o r' \land r' \leq_o r''\)
shows \(r \leq_o r''\)
using assms by (auto simp: ordLeq-def intro: comp-embed)
lemma ordLeq-total:
[ Well-order r; Well-order r' ] \implies r \leq o r' \lor r' \leq o r
unfolding ordLeq-def using wellorders-totally-ordered by blast

lemma ordIso-reflexive:
Well-order r \implies r = o r
unfolding ordIso-def using id-iso[of r] by blast

lemma ordIso-transitive[trans]:
assumes *: r = o r' and **: r' = o r''
shows r = o r''
using assms by (auto simp: ordIso-def intro: comp-iso)

lemma ordIso-symmetric:
assumes *: r = o r'
shows r' = o r
proof
obtain f where 1: Well-order r \land Well-order r' and
  2: embed r' f \land bij-betw f (Field r) (Field r')
    using * by (auto simp add: ordIso-def iso-def)
let f' = inv-into (Field r) f
have embed r' r f' \land bij-betw f' (Field r') (Field r)
  using 1 2 by (simp add: bij-betw-inv-into inv-into-Field-embed-bij-betw)
thus r' = o r unfolding ordIso-def using 1 by (auto simp add: iso-def)
qed

lemma ordLeq-ordLess-trans[trans]:
assumes r \leq o r' and r' < o r''
shows r < o r''
proof
have Well-order r \land Well-order r''
  using assms unfolding ordLeq-def ordLess-def by auto
thus ?thesis using assms unfolding ordLeq-def ordLess-def
  using embed-comp-embedS by blast
qed

lemma ordLess-ordLeq-trans[trans]:
assumes r < o r' and r' \leq o r''
shows r \leq o r''
using embedS-comp-embed assms by (force simp: ordLeq-def ordLess-def)

lemma ordLeq-ordIso-trans[trans]:
assumes r \leq o r' and r' = o r''
shows r \leq o r''
using embed-comp-iso assms by (force simp: ordLeq-def ordIso-def)

lemma ordIso-ordLeq-trans[trans]:
assumes r = o r' and r' \leq o r''
shows $r \leq o r''$
using iso-comp-embed assms by (force simp: ordLeq-def ordIso-def)

lemma ordLess-ordIso-trans[trans]:
assumes $r < o r' \quad \text{and} \quad r' = o r''$
shows $r < o r''$
using embedS-comp-iso assms by (force simp: ordLess-def ordIso-def)

lemma ordIso-ordLess-trans[trans]:
assumes $r = o r' \quad \text{and} \quad r' < o r''$
shows $r < o r''$
using iso-comp-embedS assms by (force simp: ordLess-def ordIso-def)

lemma ordLess-not-embed:
assumes $r < o r'$
shows $\neg \exists f'. \text{embed } r \quad \text{r } f'$
proof
obtain f where 1: Well-order r \quad \text{and} \quad embed r \quad r f
3: \neg bij-betw f (Field r) (Field r')
using assms unfolding ordLess-def by (auto simp add: embedS-def)
{fix f' assume \*: embed r' \quad r \quad f'
  hence bij-betw f (Field r) (Field r') using 1 2 by (simp add: embed-bothWays-Field-bij-betw)
  with 3 have False by contradiction
} thus ?thesis by blast
qed

lemma ordLess-Field:
assumes OL: $r1 < o r2$ and EMB: embed r1 r2 f
shows $\neg (f'(\text{Field } r1) = \text{Field } r2)$
proof
let ?A1 = Field r1 let ?A2 = Field r2
obtain g where
0: Well-order r1 \quad \text{and} \quad embed r1 r2 g \quad \neg(bij-betw g ?A1 ?A2)
using OL unfolding ordLess-def by (auto simp add: embedS-def)
hence $\forall a \in ?A1. f a = g a$
using 0 EMB embed-unique[of r1] by auto
hence $\neg(bij-betw f ?A1 ?A2)$
using 1 bij-betw-cong[of ?A1] by blast
moreover
have inj-on f ?A1 using EMB 0 by (simp add: embed-inj-on)
ultimately show ?thesis by (simp add: bij-betw-def)
qed

lemma ordLess-iff:
$r < o r' = (\text{Well-order } r \quad \text{and} \quad \text{Well-order } r' \quad \neg(\exists f'. \text{embed } r' \quad r \quad f'))$
proof
assume *: \( r < o r' \)

hence \( \neg\exists f'. \text{embed} r' r f' \) using \texttt{ordLess-not-embed[of r r']} by simp

with * show \( \text{Well-order} r \land \text{Well-order} r' \land \neg \exists f'. \text{embed} r' r f' \)

unfolding \texttt{ordLess-def} by auto

next

assume *: \( \text{Well-order} r \land \text{Well-order} r' \land \neg \exists f'. \text{embed} r' r f' \)

then obtain \( f \) where \( 1: \text{embed} r r' f \)

using \texttt{wellorders-totally-ordered[of r r']} by blast

moreover

\{ assume \texttt{bij-betw f (Field r) (Field r')}

with * \texttt{have embed} r' r \texttt{(inv-into (Field r) f)}

using \texttt{inv-into-Field-embed-bij-betw[of r r' f]} by auto

with * \texttt{have False by blast}

\}

ultimately show \( (r, r') \in \text{ordLess} \)

unfolding \texttt{ordLess-def} using * by \texttt{(fastforce simp add: embedS-def)}

qed

lemma \texttt{ordLess-irreflexive}: \( \neg r < o r \)

using \texttt{id-embed[of r]} by \texttt{(auto simp: ordLess-iff)}

lemma \texttt{ordLeq-iff-ordLess-or-ordIso}: \( r \leq o r' = (r < o r' \lor r = o r') \)

unfolding \texttt{ordRels-def embedS-defs iso-defs} by blast

lemma \texttt{ordIso-iff-ordLeq}: \( (r = o r') = (r \leq o r' \land r' \leq o r) \)

proof

assume \( r = o r' \)

then obtain \( f \) where \( 1: \text{Well-order} r \land \text{Well-order} r' \land \text{embed} r r' f \land \texttt{bij-betw f (Field r) (Field r')} \)

unfolding \texttt{ordIso-def iso-defs} by auto

hence \( \text{embed} r r' f \land \text{embed} r' r \texttt{(inv-into (Field r) f)} \)

by \texttt{(simp add: inv-into-Field-embed-bij-betw)}

thus \( r \leq o r' \land r' \leq o r \)

unfolding \texttt{ordLeq-def} using \( 1 \) by auto

next

assume \( r \leq o r' \land r' \leq o r \)

then obtain \( f \) and \( g \) where \( 1: \text{Well-order} r \land \text{Well-order} r' \land \text{embed} r r' f \land \text{embed} r' r g \)

unfolding \texttt{ordLeq-def} by auto

hence \( \text{iso} r r' f \) by \texttt{(auto simp add: embed-bothWays-iso)}

thus \( r = o r' \) unfolding \texttt{ordIso-def} using \( 1 \) by auto

qed

lemma \texttt{not-ordLess-ordLeq}: \( r < o r' \Rightarrow \neg r' \leq o r \)

using \texttt{ordLess-ordLeq-trans ordLess-irreflexive} by blast
lemma not-ordLeq-ordLess:
  \( r \leq o r' \implies \neg r' < o r \)
  using not-ordLess-ordLeq by blast

lemma ordLess-or-ordLeq:
  assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \)
  shows \( r < o r' \lor r' \leq o r \)
proof
  have \( r \leq o r' \lor r' \leq o r \)
    using assms by (simp add: ordLeq-total)
  moreover
  { assume \( \neg r < o r' \land r \leq o r' \)
    hence \( r = o r' \) using ordLeq-iffLess-or-ordIso by blast
    hence \( r' \leq o r \) using ordIso-symmetric ordIso-iff-ordLeq by blast
  }
  ultimately show \(?thesis\) by blast
qed

lemma not-ordLess-ordIso:
  \( r < o r' \implies \neg r = o r' \)
  using ordLess-ordIso-trans ordIso-symmetric ordLess-irreflexive by blast

lemma not-ordLess-ordLeq:
  assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \)
  shows \( \neg r' \leq o r \) = (\( r < o r' \))
  using assms not-ordLess-ordLeq ordLess-or-ordLeq by blast

lemma ordLess-transitive[trans]:
  \[ [r < o r'; r' < o r''] \implies r < o r'' \]
  using ordLess-ordLeq-trans ordLeq-iffLess-or-ordIso by blast

corollary ordLess-trans: trans ordLess
  unfolding trans-def using ordLess-transitive by blast

lemmas ordIso-equivalence = ordIso-transitive ordIso-reflexive ordIso-symmetric

lemma ordIso-imp-ordLeq:
  \( r = o r' \implies r \leq o r' \)
  using ordIso-iff-ordLeq by blast

lemma ordLess-imp-ordLeq:
  \( r < o r' \implies r \leq o r' \)
  using ordLeq-iff-ordLess-or-ordIso by blast
lemma ofilter-subset-ordLeq:
  assumes WELL: Well-order r and
    OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
  shows \((A \leq B) = (\operatorname{Restr} r A \leq_o \operatorname{Restr} r B)\)
proof
  assume \(A \leq B\)
  thus \(\operatorname{Restr} r A \leq_o \operatorname{Restr} r B\)
  unfolding ordLeq-def using assms
  Well-order-Restr Well-order-Restr ofilter-subset-embed by blast
next
  assume \(*\): \(\operatorname{Restr} r A \leq_o \operatorname{Restr} r B\)
  then obtain \(f\) where embed \((\operatorname{Restr} r A) (\operatorname{Restr} r B)\) \(f\)
  unfolding ordLeq-def by blast
  {assume \(B < A\)
   hence \(\operatorname{Restr} r B <_o \operatorname{Restr} r A\)
   unfolding ordLess-def using assms
   Well-order-Restr Well-order-Restr ofilter-subset-embedS by blast
   hence False using \(*\) not-ordLess-ordLeq by blast
  }
  thus \(A \leq B\) using OFA OFB WELL wo-rel-def[of r] wo-rel.ofilter-linord[of r A B] by blast
qed

lemma ofilter-subset-ordLess:
  assumes WELL: Well-order r and
    OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
  shows \((A < B) = (\operatorname{Restr} r A <_o \operatorname{Restr} r B)\)
proof
  let \(?rA = \operatorname{Restr} r A\) let \(?rB = \operatorname{Restr} r B\)
  have 1: Well-order \(?rA\) \& Well-order \(?rB\)
    using WELL Well-order-Restr by blast
  have \((A < B) = (\neg B \leq A)\) using assms
    wo-rel-def wo-rel.ofilter-linord[of r A B] by blast
  also have \ldots = (\neg \operatorname{Restr} r B \leq_o \operatorname{Restr} r A)\)
    using assms ofilter-subset-ordLeq by blast
  also have \ldots = (\operatorname{Restr} r A <_o \operatorname{Restr} r B)\)
    using 1 not-ordLeq-iff-ordLess by blast
  finally show \(?\thesis\).
qed

lemma ofilter-ordLess:
\[\text{Well-order } r; \text{ wo-rel.ofilter } r A\] \implies \((A < \text{ Field } r) = (\operatorname{Restr} r A <_o r)\)
by (simp add: ofilter-subset-ordLess wo-rel.Field-ofilter
  wo-rel-def Restr-Field)

corollary underS-Restr-ordLess:
  assumes Well-order r and Field r \(\neq \{\}\)
  shows \(\operatorname{Restr} r (\text{underS } r a) <_o r\)
proof
have \textit{underS} \( r \ a < \text{Field} \ r \) using \textit{assms}
  by (simp add: \textit{underS-Field3})
thus \( \text{?thesis} \) using \textit{assms}
  by (simp add: \textit{ofilter-ordLess wo-rel.underS-ofilter wo-rel-def})
qed

\textbf{lemma} \textit{embed-ordLess-ofilterIncl}:
  \begin{itemize}
  \item assumes \( \text{OL12}: r1 < o r2 \) \textbf{and} \( \text{OL23}: r2 < o r3 \) \textbf{and}
  \item \( \text{EMB13}: \text{embed} \ r1 \ r3 \ f13 \) \textbf{and} \( \text{EMB23}: \text{embed} \ r2 \ r3 \ f23 \)
  \item shows \( (f13 \ (< \text{Field} \ r1), f23 \ (< \text{Field} \ r2)) \in (\text{ofilterIncl} \ r3) \)
  \end{itemize}
\textbf{proof}–
have \( \text{OL13}: r1 < o r3 \)
  using \( \text{OL12} \ \text{OL23} \) using \textit{ordLess-transitive} by auto
let \( \text{?A1} = \text{Field} \ r1 \) let \( \text{?A2} = \text{Field} \ r2 \) let \( \text{?A3} = \text{Field} \ r3 \)
obtain \( f12 \ g23 \) where
  \begin{itemize}
  \item \( \text{0}: \text{Well-order} \ r1 \ \wedge \ \text{Well-order} \ r2 \ \wedge \ \text{Well-order} \ r3 \) \textbf{and}
  \item \( \text{1}: \text{embed} \ r1 \ r2 \ f12 \ \land \lnot (\text{bij-betw} \ f12 \ ?A1 \ ?A2) \) \textbf{and}
  \item \( \text{2}: \text{embed} \ r2 \ r3 \ g23 \ \land \lnot (\text{bij-betw} \ g23 \ ?A2 \ ?A3) \)
  \end{itemize}
  using \( \text{OL12} \ \text{OL23} \) by (auto simp add: \textit{ordLess-def embedS-def})
  hence \( \forall a \in ?A2. \ f23 \ a = g23 \ a \)
  using \( \text{EMB23} \ \text{embed-unique}[o f r2 r3] \) by blast
  hence \( 3: \lnot (\text{bij-betw} \ f23 \ ?A2 \ ?A3) \)
  using \( 2 \ \text{bij-betw-cong}[o f ?A2 f23 g23] \) by blast

have \( 4: \text{wo-rel.ofilter} \ r2 \ (f12 \ ?A1) \land f12 \ ?A1 \neq \ ?A2 \)
  using \( 0 \ \text{OL12} \) by (simp add: \textit{embed-Field-ofilter ordLess-Field})
  have \( 5: \text{wo-rel.ofilter} \ r3 \ (f23 \ ?A2) \land f23 \ ?A2 \neq \ ?A3 \)
  using \( 0 \ \text{EMB23} \ \text{OL23} \) by (simp add: \textit{embed-Field-ofilter ordLess-Field})
  have \( 6: \text{wo-rel.ofilter} \ r3 \ (f13 \ ?A1) \land f13 \ ?A1 \neq \ ?A3 \)
  using \( 0 \ \text{EMB13} \ \text{OL13} \) by (simp add: \textit{embed-Field-ofilter ordLess-Field})
  have \( f12 \ ?A1 < ?A2 \)
  using \( 0 \ \text{OL12} \) by (auto simp add: \textit{wo-rel-def wo-rel.ofilter-def})
  moreover have \( \text{inj-on} \ f23 \ ?A2 \)
  using \( \text{EMB23} \ \text{OL23} \) by (simp add: \textit{wo-rel-def embed-inj-on})
ultimately
have \( f23 \ (f12 \ ?A1) < f23 \ ?A2 \) by (simp add: \textit{image-strict-mono})
  moreover
  \begin{itemize}
  \item have \( \text{embed} \ r1 \ r3 \ (f23 \circ f12) \)
    using \( 1 \ \text{EMB23} \ \text{OL23} \) by (auto simp add: \textit{comp-embed})
    hence \( \forall a \in ?A1. \ f23(f12 \ a) = f13 \ a \)
    using \( \text{EMB13} \ \text{EMB23} \ \text{embed-unique}[o f r1 r3 f23 \circ f12 f13] \) by auto
    hence \( f23 (f12 \ ?A1) = f13 \ ?A1 \) by force
  \end{itemize}
ultimately
have \( f13 \ ?A1 < f23 \ ?A2 \) by simp

with \( 5 \ \text{6} \) show \( \text{?thesis} \)
unfolding ofilterIncl-def by auto

qed

lemma ordLess-iff-ordIso-Restr:
  assumes WELL: Well-order r and WELL': Well-order r'
  shows \((r' < o r) = (\exists a \in \text{Field } r. r' = o \text{ Restr } r \text{ (underS } r \text{ a)})\)
  proof safe
    fix a assume *: a \in Field r and **: r' = o \text{ Restr } r \text{ (underS } r \text{ a)}
    hence \text{Restr } r \text{ (underS } r \text{ a) < o r using WELL underS-Restr-ordLess[of } r\) by blast
    thus \(r' < o r\) using ** ordIso-ordLess-trans by blast
  next
    assume r' < o r
    then obtain f where 1: Well-order r \land Well-order r' and
      2: embed r' r / f \land f' (Field r') \neq Field r
      unfolding ordLess-def embedS-def[abs-def] bij-betw-def using embed-inj-on by blast
    hence wo-rel. ofilter r (f' (Field r')) using embed-Field-ofilter by blast
    then obtain a where 3: a \in Field r and 4: underS r a = f' (Field r')
      using 1 2 by (auto simp add: wo-rel.ofilter-underS-Field wo-rel-def)
    have iso r' (Restr r (f' (Field r'))) f
      using embed-implies-iso-Restr 2 assms by blast
    let ?p = Restr r (underS r a)
    have wo-rel.ofilter r (underS r a) using 0 by (simp add: wo-rel-def wo-rel.underS-ofilter)
    hence Field ?p = underS r a using 0 Field-Restr-ofilter by blast
    hence Field ?p < Field r using underS-Field2 1 by fast
    moreover have ?p < o r using underS-Restr-ordLess[of r a] 0 1 by blast
    ultimately show \(\exists a \in \text{Field } r. r' = o \text{ Restr } r \text{ (underS } r \text{ a)}\ using 3 by auto
    qed

lemma internalize-ordLess:
  \((r' < o r) = (\exists p. \text{Field } p < \text{Field } r \land r' = o p \land p < o r)\)
  proof
    assume *: r' < o r
    hence 0: Well-order r \land Well-order r' unfolding ordLess-def by auto
    with * obtain a where 1: a \in Field r and 2: r' = o \text{ Restr } r \text{ (underS } r \text{ a)}
      using ordLess-iff-ordIso-Restr by blast
    let ?p = Restr r (underS r a)
    have wo-rel.ofilter r (underS r a) using 0 by (simp add: wo-rel-def wo-rel.underS-ofilter)
    hence Field ?p = underS r a using 0 Field-Restr-ofilter by blast
    hence Field ?p < Field r using underS-Field2 1 by fast
    moreover have ?p < o r using underS-Restr-ordLess[of r a] 0 1 by blast
    ultimately show \(\exists p. \text{Field } p < \text{Field } r \land r' = o p \land p < o r\) using 2 by blast
  next
    assume \(\exists p. \text{Field } p < \text{Field } r \land r' = o p \land p < o r\)
    thus \(r' < o r\) using ordIso-ordLess-trans by blast
  qed
lemma internalize-ordLeq:
(r' ≤ o r) = (∃ p. Field p ≤ Field r ∧ r' = o p ∧ p ≤ o r)
proof
assume *: r' ≤ o r
moreover
have r' < o r ⟹ ∃ p. Field p ≤ Field r ∧ r' = o p ∧ p ≤ o r
  using ordLeq-iff-ordLess-or-ordIso internalize-ordLess[of r' r] by blast
moreover
have r ≤ o r using * ordLeq-def ordLeq-reflexive by blast
ultimately show ∃ p. Field p ≤ Field r ∧ r' = o p ∧ p ≤ o r
  using ordLeq-iff-ordLess-or-ordIso by blast
next
assume ∃ p. Field p ≤ Field r ∧ r' = o p ∧ p ≤ o r
thus r' ≤ o r using ordIso-ordLeq-trans by blast
qed

lemma ordLeq-iff-ordLess-Restr:
assumes WELL: Well-order r and WELL': Well-order r'
shows (r ≤ o r') = (∀ a ∈ Field r. Restr r (underS r a) < o r')
proof safe
assume *: r ≤ o r'
fix a assume a ∈ Field r
hence Restr r (underS r a) < o r
  using WELL underS-Restr-ordLess[of r] by blast
thus Restr r (underS r a) < o r'
  using * ordLess-ordLeq-trans by blast
next
assume *: ∀ a ∈ Field r. Restr r (underS r a) < o r'
{assume r' < o r
  then obtain a where a ∈ Field r ∧ r' = o Restr r (underS r a)
    using assms ordLess-iff-ordIso-Restr by blast
  hence False using * not-ordLess-ordIso ordIso-symmetric by blast }
thus r ≤ o r' using ordLess-or-ordLeq assms by blast
qed

lemma finite-ordLess-infinite:
assumes WELL: Well-order r and WELL': Well-order r' and
FIN: finite(Field r) and INF: ¬ finite(Field r')
shows r < o r'
proof –
{assume r' ≤ o r
  then obtain h where inj-on h (Field r') ∧ h⁻¹(Field r') ≤ Field r
    unfolding ordLeq-def using assms embed-inj-on embed-Field by blast
  hence False using finite-imageD finite-subset FIN INF by blast }
thus ?thesis using WELL WELL' ordLess-or-ordLeq by blast
qed
lemma finite-well-order-on-ordIso:
assumes FIN: finite A and
WELL: well-order-on A r and WELL': well-order-on A r'
shows r = o r'
proof
have 0: Well-order r ∧ Well-order r' ∧ Field r = A ∧ Field r' = A
using assms well-order-on-Well-order by blast
moreover
have ∀ r r'. well-order-on A r ∧ well-order-on A r' ∧ r ≤ o r' → r = o r'
proof
fix r r' assume *: well-order-on A r and **: well-order-on A r'
have 2: Well-order r ∧ Well-order r' ∧ Field r = A ∧ Field r' = A
using * ** well-order-on-Well-order by blast
assume r ≤ o r' then obtain f where 1: embed r r' f
inj-on f A ∧ f ' A ≤ A
unfolding ordLeq-def using 2 embed-inj-on embed-Field by blast
hence bij-betw f A A unfolding bij-betw-def using FIN endo-inj-surj by blast
thus r = o r' unfolding ordIso-def iso-def[abs-def] using 1 2 by auto
qed
ultimately show thesis using assms ordLeq-total ordIso-symmetric by blast
qed

29.5 <o is well-founded

Of course, it only makes sense to state that the <o is well-founded on the restricted type 'a rel rel. We prove this by first showing that, for any set of well-orders all embedded in a fixed well-order, the function mapping each well-order in the set to an order filter of the fixed well-order is compatible w.r.t. to <o versus strict inclusion; and we already know that strict inclusion of order filters is well-founded.

definition ord-to-filter :: 'a rel rel ⇒ 'a set
where ord-to-filter r0 r ≡ (SOME f. embed r r0 f) ' (Field r)

lemma ord-to-filter-compat:
compat (ordLess Int (ordLess⁻¹·{r0} × ordLess⁻¹·{r0}))
(ofilterIncl r0)
(ord-to-filter r0)
proof(unfold compat-def ord-to-filter-def, clarify)
fix r1:: 'a rel and r2:: 'a rel
let ?A1 = Field r1 let ?A2 = Field r2 let ?A0 = Field r0
let ?phi10 = λ f10. embed r1 r0 f10 let ?f10 = SOME f. ?phi10 f
let ?phi20 = λ f20. embed r2 r0 f20 let ?f20 = SOME f. ?phi20 f
assume *: r1 <o r0 r2 <o r0 and **: r1 <o r2
hence (∃ f. ?phi10 f) ∧ (∃ f. ?phi20 f)
  by (auto simp add: ordLess-def embedS-def)
hence ?phi10 ?f10 ∧ ?phi20 ?f20 by (auto simp add: someEx)
thus \((\forall f1. \forall A1. \forall f2. \forall A2) \in \text{ofilterIncl} r0\)
using \(*\*\) by (simp add: embed-ordLess-ofilterIncl)
qed

theorem \(wf-ordLess\): \(wf\ ordLess\)
proof\-
\{fix \(r0:: ('a \times 'a)\) set\n
let \(?ordLess = ordLess::('d rel * 'd rel) set\nlet \(?R = ?ordLess Int (?ordLess^{-1}''\{r0\} \times ?ordLess^{-1}''\{r0\})\n{assume \(Case1\): Well-order \(r0\)
  hence \(wf ?R\)
  using \(wf-ofilterIncl[of r0]\)
  \(compat-wf[of \ ?R ofilterIncl r0 ord-to-filter r0]\)
  \(ord-to-filter-compat[of r0]\) by auto\}
moreover
{assume \(Case2\): \(\neg\) Well-order \(r0\)
  hence \(?R = \{\}\) unfolding \(ordLess-def\) by auto
  hence \(wf ?R\) using \(wf-empty\) by simp\}
ultimately have \(wf ?R\) by blast\}
thus \(?thesis\) by (simp add: trans-wf-iff ordLess-trans)
qed

corollary \(exists-minim-Well-order\):
  assumes \(NE: R \neq \{\}\) and \(WELL: \forall r \in R.\ Well-order r\)
  shows \(\exists r \in R.\ \forall r' \in R.\ r \leq o r'\)
proof\-
obtain \(r\ where r \in R \land (\forall r' \in R.\ \neg r' < o r)\)
  using \(NE \ spec[of \ spec[of subst[of OF wf-eq-minimal, of %x. x, OF wf-ordLess]], of - R]\)
  \(equals0I[of R]\) by blast
with \(not-ordLeq-iff-ordLess\ WELL show \(?thesis\) by blast\)
qed

29.6 Copy via direct images

The direct image operator is the dual of the inverse image operator \(\text{inv-image}\)
from \(\text{Relation.thy}\). It is useful for transporting a well-order between different
types.

definition \(dir-image:: 'a rel \Rightarrow ('a \Rightarrow 'a') \Rightarrow 'a' rel\)
where
\(dir-image r f = \{(f a, f b)| a b. (a,b) \in r\}\)

lemma \(dir-image-Field:\)
\(Field(dir-image r f) = f^{-1}(Field r)\)
unfolding \(dir-image-def Field-def Range-def Domain-def\) by fast
lemma dir-image-minus-Id:
\[ \text{inj-on } f (\text{Field } r) \implies (\text{dir-image } r f) - \text{Id} = \text{dir-image } (r - \text{Id}) f \]
unfolding inj-on-def Field-def dir-image-def by auto

lemma Refl-dir-image:
assumes Refl r
shows Refl(\text{dir-image } r f)
proof -
{fix \(a', b'\)
 assume \((a',b')\in \text{dir-image } r f\)
 then obtain \(a b\) where 1: \(a' = f a \land b' = f b \land (a,b) \in r\)
 unfolding dir-image-def by blast
 hence \(a \in \text{Field } r \land b \in \text{Field } r\) using Field-def by fastforce
 hence \((a,a)\in r \land (b,b) \in r\) using assms by (simp add: refl-on-def)
 with \(f\) have \((a',a')\in \text{dir-image } r f \land (b',b') \in \text{dir-image } r f\)
 unfolding dir-image-def by auto
}
thus ?thesis
by(unfold refl-on-def Field-def Domain-def Range-def, auto)
qed

lemma trans-dir-image:
assumes TRANS: \(\text{trans } r\) and INJ: \(\text{inj-on } f (\text{Field } r)\)
shows \(\text{trans } (\text{dir-image } r f)\)
unfolding trans-def
proof safe
fix \(a' b' c'\)
 assume \((a',b')\in \text{dir-image } r f \land (b',c') \in \text{dir-image } r f\)
 then obtain \(a b1 b2 c\) where 1: \(a' = f a \land b' = f b1 \land b' = f b2 \land c' = f c\)
 and
2: \((a,b1)\in r \land (b2,c) \in r\)
 unfolding dir-image-def by blast
 hence \(b1 \in \text{Field } r \land b2 \in \text{Field } r\)
 unfolding Field-def by auto
 hence \(b1 = b2\) using 1 INJ unfolding inj-on-def by auto
 hence \((a,c)\in r\) using 2 TRANS unfolding trans-def by blast
 thus \((a',c')\in \text{dir-image } r f\)
 unfolding dir-image-def using 1 by auto
qed

lemma Preorder-dir-image:
\([\text{Preorder } r ; \text{inj-on } f (\text{Field } r)] \implies \text{Preorder } (\text{dir-image } r f)\)
by (simp add: preorder-on-def Refl-dir-image trans-dir-image)

lemma antisym-dir-image:
assumes AN: \(\text{antisym } r\) and INJ: \(\text{inj-on } f (\text{Field } r)\)
shows \(\text{antisym } (\text{dir-image } r f)\)
unfolding antisym-def
proof safe
  fix $a^' b$
  assume $(a^', b^') \in \text{dir-image } r f \land (b', a') \in \text{dir-image } r f$
  then obtain $a1 b1 a2 b2$ where $1: a' = f a1 \land a'^' = f a2 \land b'^' = f b1 \land b' = f b2$
  and
  $2: (a1, b1) \in r \land (b2, a2) \in r$ and
  $3: \{a1, a2, b1, b2\} \leq \text{Field } r$
  unfolding dir-image-def Field-def by blast
  hence $a1 = a2 \land b1 = b2$ using INJ unfolding inj-on-def by auto
  hence $a1 = b2$ using $2\ AN$ unfolding antisym-def by auto
  thus $a' = b'$ using $1$ by auto
  qed

lemma Partial-order-dir-image:
  $[\text{Partial-order } r; \text{inj-on } f (\text{Field } r)] \implies \text{Partial-order } (\text{dir-image } r f)$
  by (simp add: partial-order-on-def Preorder-dir-image antisym-dir-image)

lemma Total-dir-image:
  assumes TOT: $\text{Total } r$ and INJ: $\text{inj-on } f (\text{Field } r)$
  shows $\text{Total}(\text{dir-image } r f)$
proof(unfold total-on-def, intro ballI impI)
  fix $a^' b$
  assume $a'^' \in \text{Field } (\text{dir-image } r f) \land b'^' \in \text{Field } (\text{dir-image } r f)$
  then obtain $a$ and $b$ where $1: a \in \text{Field } r \land b \in \text{Field } r \land f a = a'^' \land f b = b'^'$
  unfolding dir-image-Field[of $r f$] by blast
  moreover assume $a'^' \neq b'^'$
  ultimately have $a \neq b$ using INJ unfolding inj-on-def by auto
  hence $(a, b) \in r \lor (b, a) \in r$ using $1\ TOT$ unfolding total-on-def by auto
  thus $(a'^', b'^') \in \text{dir-image } r f \lor (b'^', a'^') \in \text{dir-image } r f$
  using $1$ unfolding dir-image-def by auto
  qed

lemma Linear-order-dir-image:
  $[\text{Linear-order } r; \text{inj-on } f (\text{Field } r)] \implies \text{Linear-order } (\text{dir-image } r f)$
  by (simp add: linear-order-on-def Partial-order-dir-image Total-dir-image)

lemma uf-dir-image:
  assumes WF: $\text{wf } r$ and INJ: $\text{inj-on } f (\text{Field } r)$
  shows $\text{wf}(\text{dir-image } r f)$
proof(unfold wf-eq-minimal2, intro allI impI, elim conjE)
  fix $A':='b$ set
  assume SUB: $A' \leq \text{Field}(\text{dir-image } r f)$ and NE: $A' \neq {}$
  obtain $A$ where A-def: $A = \{a \in \text{Field } r. f a \in A'\}$ by blast
  have $A \neq {}$ and $A \leq \text{Field } r$ using A-def SUB NE by (auto simp: dir-image-Field)
  then obtain $a$ where $1: a \in A \land (\forall b \in A. (b, a) \notin r)$
  unfolding spec[OF WF[unfolded uf-eq-minimal2], of $A$] by blast
  have $\forall b'^' \in A'. (b'^', a) \notin \text{dir-image } r f$
  proof(clarify)
  fix $b'$ assume $*: b' \in A'$ and $**: (b', f a) \in \text{dir-image } r f$
obtain $b_1 \ a_1$ where $2: b' = f b_1 \wedge f a = f a_1$ and
$3: (b_1,a_1) \in r \land \{a_1,b_1\} \subseteq \text{Field } r$
using ** unfolding \texttt{dir-image-def Field-def} by blast

hence $a = a_1$ using $1$ \texttt{A-def INJ unfolding inj-on-def} by auto
hence $b_1 \in A \land (b_1,a) \in r$ using $2$ $3$ \texttt{A-def} * by auto

with $1$ show False by auto

qed

thus $\exists a' \in A'. \forall b \in A'. (b', a') \notin \text{dir-image } r \ f$
using $\texttt{A-def} \ 1 \texttt{by blast}$

qed

\textbf{lemma} \textit{Well-order-dir-image:}

\quad \[[\text{Well-order } r; \ \text{inj-on } f (\text{Field } r)] \implies \text{Well-order } (\text{dir-image } r \ f)\]
\quad \text{unfolding \texttt{well-order-on-def}}
\quad \text{using \texttt{Linear-order-dir-image[of r f] \ \texttt{wf-dir-image[of r - Id f]}}}
\quad \texttt{dir-image-minus-Id[of f r]}
\quad \texttt{subset-inj-on[of f Field r Field(r - Id)]}
\quad \texttt{mono-Field[of r - Id r]} \texttt{by auto}

\textbf{lemma} \textit{dir-image-bij-betw:}

\quad \[[\text{inj-on } f (\text{Field } r)] \implies \text{bij-betw } f (\text{Field } r) (\text{Field } (\text{dir-image } r \ f))\]
\quad \text{unfolding \texttt{bij-betw-def} by (simp add: \texttt{dir-image-Field order-on-defs})}

\textbf{lemma} \textit{dir-image-compat:}

\quad \text{compat } r (\text{dir-image } r \ f) \ f
\quad \text{unfolding \texttt{compat-def dir-image-def} by auto}

\textbf{lemma} \textit{dir-image-iso:}

\quad \[[\text{Well-order } r; \ \text{inj-on } f (\text{Field } r)] \implies \text{iso } r (\text{dir-image } r \ f) \ f\]
\quad \text{using \texttt{iso-iff3 dir-image-compat dir-image-bij-betw Well-order-dir-image by blast}}

\textbf{lemma} \textit{dir-image-ordIso:}

\quad \[[\text{Well-order } r; \ \text{inj-on } f (\text{Field } r)] \implies r =o \text{dir-image } r \ f\]
\quad \text{unfolding \texttt{ordIso-def} using \texttt{dir-image-iso Well-order-dir-image} by blast}

\textbf{lemma} \textit{Well-order-iso-copy:}

\quad assumes WELL: \texttt{well-order-on } A r \ \texttt{and BIJ: bij-betw } f \ A \ A'
\quad shows $\exists r'. \texttt{well-order-on } A' \ r' \wedge r =o r'$

\textbf{proof} –

\quad let $?r' = \text{dir-image } r \ f$
\quad have $1: A = \text{Field } r \land \text{Well-order } r$
\quad \quad using WELL \texttt{well-order-on-Well-order} by blast
\quad hence $2: \text{iso } r \ ?r' \ f$
\quad \quad using \texttt{dir-image-iso using BIJ unfolding bij-betw-def} by auto
\quad hence $f'(\text{Field } r) = \text{Field } ?r' \texttt{ using } 1 \texttt{iso-if[of r ?r'] by blast}$
\quad hence $\text{Field } ?r' = A'$
\quad \quad using $1$ BIJ unfolding bij-betw-def by auto
\quad moreover have $\text{Well-order } ?r'$
\quad \quad using $1$ \texttt{Well-order-dir-image BIJ unfolding bij-betw-def} by blast
ultimately show \( \text{thesis unfolding ordIso-def using 1 2 by blast} \)
proof
qed

29.7 Bounded square

This construction essentially defines, for an order relation \( r \), a lexicographic order \( \text{bsqr} \, r \) on \((\text{Field} \, r) \times (\text{Field} \, r)\), applying the following criteria (in this order):

- compare the maximums;
- compare the first components;
- compare the second components.

The only application of this construction that we are aware of is at proving that the square of an infinite set has the same cardinal as that set. The essential property required there (and which is ensured by this construction) is that any proper order filter of the product order is included in a rectangle, i.e., in a product of proper filters on the original relation (assumed to be a well-order).

definition \( \text{bsqr} :: \{a \ rel \} = \{a * a \rel\}
where
\begin{align*}
\text{bsqr} & = \{(a1,a2),(b1,b2)\} \leq \text{Field} \, r \\
(a1 = b1 \land a2 = b2) & \lor \\
(\text{wo-rel.max2} \, r \, a1 \, a2, \text{wo-rel.max2} \, r \, b1 \, b2) & \in r - \text{Id} \lor \\
\text{wo-rel.max2} \, r \, a1 \, a2 = \text{wo-rel.max2} \, r \, b1 \, b2 \land (a1, b1) & \in r - \text{Id} \lor \\
\text{wo-rel.max2} \, r \, a1 \, a2 = \text{wo-rel.max2} \, r \, b1 \, b2 \land a1 = b1 \land (a2, b2) & \in r
\end{align*}

lemma Field-bsqr:
Field \((\text{bsqr} \, r)\) = Field \(r \times Field \, r\)
proof
show Field \((\text{bsqr} \, r)\) \(\leq Field \, r \times Field \, r\)
proof-
\{fix \ a1 \, a2 assume \((a1,a2)\) \in Field \((\text{bsqr} \, r)\)
moreover
have \(\land b1 b2. (a1,a2),(b1,b2)) \in \text{bsqr} \lor ((b1,b2),(a1,a2)) \in \text{bsqr} \implies a1 \in \text{Field} \, r \land a2 \in \text{Field} \, r\) unfolding \text{bsqr-def by auto}
ultimately have \(a1 \in \text{Field} \, r \land a2 \in \text{Field} \, r\) unfolding Field-def by auto
\}
thus \(\text{thesis unfolding Field-def by force}\)
qed
next
show Field \(r \times Field \, r\) \(\leq Field \, (\text{bsqr} \, r)\)
proof safe
THEORY “BNF-Wellorder-Constructions”

fix \( a_1 \) \( a_2 \) assume \( a_1 \in \text{Field } r \) and \( a_2 \in \text{Field } r \)
hence \( ((a_1,a_2),(a_1,a_2)) \in \text{bsqr } r \) unfolding bsqr-def by blast
thus \( (a_1,a_2) \in \text{Field } (\text{bsqr } r) \) unfolding Field-def by auto
qed

lemma bsqr-Refl: \( \text{Refl}(\text{bsqr } r) \)
by (unfold refl-on-def Field-bsqr, auto simp add: bsqr-def)

lemma bsqr-Trans:
  assumes Well-order \( r \)
  shows trans \((\text{bsqr } r)\)
unfolding trans-def proof
  safe
  have Well: \( \text{wo-rel } r \) using assms wo-rel-def by auto
  hence Trans: \( \text{trans } r \) using wo-rel.ANTISYM Well by auto
  have Anti: \( \text{antisym } r \) using wo-rel.ANTISYM Well by auto
  hence TransS: \((r - \text{Id})\) using Trans by (simp add: trans-diff-Id)
fix \( a_1 a_2 b_1 b_2 c_1 c_2 \)
assume *: \( ((a_1,a_2),(b_1,b_2)) \in \text{bsqr } r \) and **: \( ((b_1,b_2),(c_1,c_2)) \in \text{bsqr } r \)
hence 0: \( \{a_1,a_2,b_1,b_2,c_1,c_2\} \leq \text{Field } r \) unfolding bsqr-def by auto
have 1: \( a_1 = b_1 \wedge a_2 = b_2 \lor (\text{wo-rel.max2 } r \ a_1 \ a_2, \text{wo-rel.max2 } r \ b_1 \ b_2) \in r - \text{Id} \)
  using * unfolding bsqr-def by auto
have 2: \( b_1 = c_1 \wedge b_2 = c_2 \lor (\text{wo-rel.max2 } r \ b_1 \ b_2, \text{wo-rel.max2 } r \ c_1 \ c_2) \in r - \text{Id} \)
  using ** unfolding bsqr-def by auto
show \( ((a_1,a_2),(c_1,c_2)) \in \text{bsqr } r \)
proof
  {assume Case1: \( a_1 = b_1 \wedge a_2 = b_2 \)
  hence \( \text{?thesis using } ** \) by simp}
moreover
  {assume Case2: \( (\text{wo-rel.max2 } r \ a_1 \ a_2, \text{wo-rel.max2 } r \ b_1 \ b_2) \in r - \text{Id} \)
  {assume Case21: \( b_1 = c_1 \wedge b_2 = c_2 \)
  hence \( \text{?thesis using } * \) by simp}
}
moreover
  {assume Case22: \( (\text{wo-rel.max2 } r \ b_1 \ b_2, \text{wo-rel.max2 } r \ c_1 \ c_2) \in r - \text{Id} \)
  hence \( (\text{wo-rel.max2 } r \ a_1 \ a_2, \text{wo-rel.max2 } r \ c_1 \ c_2) \in r - \text{Id} \)
  using Case2 TransS trans-def[of r - Id] by blast}
hence \( \text{thesis using 0 unfolding bsqr-def by auto} \) }

moreover
{ assume Case23-4: wo-rel.max2 \( r \) \( b1 \) \( b2 \) = wo-rel.max2 \( r \) \( c1 \) \( c2 \)
  hence \( \text{thesis using Case2 0 unfolding bsqr-def by auto} \) }

ultimately have \( \text{thesis using 0 2 by auto} \) }

moreover
{ assume Case3: wo-rel.max2 \( r \) \( a1 \) \( a2 \) = wo-rel.max2 \( r \) \( b1 \) \( b2 \) \& \( (a1,b1) \in r - Id \)
  { assume Case31: \( b1 = c1 \land b2 = c2 \)
    hence \( \text{thesis using * by simp} \) }
  }

moreover
{ assume Case32: (wo-rel.max2 \( r \) \( b1 \) \( b2 \), wo-rel.max2 \( r \) \( c1 \) \( c2 \)) \in r - Id
  hence \( \text{thesis using Case3 0 unfolding bsqr-def by auto} \) }

moreover
{ assume Case33: wo-rel.max2 \( r \) \( b1 \) \( b2 \) = wo-rel.max2 \( r \) \( c1 \) \( c2 \) \& \( (b1,c1) \in r - Id \)
  hence \( (a1,c1) \in r - Id \)
    using Case3 TransS trans-def[of r - Id] by blast
    hence \( \text{thesis using Case3 Case33 0 unfolding bsqr-def by auto} \) }

moreover
{ assume Case4: wo-rel.max2 \( r \) \( a1 \) \( a2 \) = wo-rel.max2 \( r \) \( b1 \) \( b2 \) \& \( a1 = b1 \land (a2,b2) \in r - Id \)
  { assume Case41: \( b1 = c1 \land b2 = c2 \)
    hence \( \text{thesis using * by simp} \) }
  }

moreover
{ assume Case42: (wo-rel.max2 \( r \) \( b1 \) \( b2 \), wo-rel.max2 \( r \) \( c1 \) \( c2 \)) \in r - Id
  hence \( \text{thesis using Case4 0 unfolding bsqr-def by force} \) }

moreover
{ assume Case43: wo-rel.max2 \( r \) \( b1 \) \( b2 \) = wo-rel.max2 \( r \) \( c1 \) \( c2 \) \& \( (b1,c1) \in r - Id \)
  hence \( \text{thesis using Case4 0 unfolding bsqr-def by auto} \) }

moreover
{ assume Case44: wo-rel.max2 \( r \) \( b1 \) \( b2 \) = wo-rel.max2 \( r \) \( c1 \) \( c2 \) \& \( b1 = c1 \land (b2,c2) \in r - Id \)
hence \((a_2, c_2) \in r - \text{Id}\) using Case4 TransS trans-def[of \(r - \text{Id}\)] by blast
hence \(?\text{thesis}\) using Case4 Case44 0 unfolding bsqr-def by auto
}
ultimately have \(?\text{thesis}\) using 0 2 by auto
ultimately show \(?\text{thesis}\) using 0 1 by auto
qed
qed

lemma bsqr-antisym:
  assumes Well-order \(r\)
  shows antisym \((\text{bsqr } r)\)
proof(unfold antisym-def, clarify)

  have Well: wo-rel \(r\) using assms wo-rel-def by auto
  hence Trans: trans \(r\) using wo-rel.TRAN by auto
  have Anti: antisym \(r\) using wo-rel.antisym Well by auto
  hence TransS: trans\((r - \text{Id})\) using Trans by (simp add: trans-diff-Id)
    using Anti trans-def[of \(r - \text{Id}\)] antisym-def[of \(r - \text{Id}\)] by blast

  fix \(a_1 a_2 b_1 b_2\)
  assume \(*\): \(((a_1,a_2),(b_1,b_2)) \in \text{bsqr } r\) and \(\ast\ast\): \(((b_1,b_2),(a_1,a_2)) \in \text{bsqr } r\)

  hence \(\{a_1,a_2,b_1,b_2\} \leq \text{Field } r\) unfolding bsqr-def by auto
  moreover
  have \(a_1 = b_1 \land a_2 = b_2 \lor (\text{wo-rel.max2 } r \ a_1 \ a_2, \text{wo-rel.max2 } r \ b_1 \ b_2) \in r - \text{Id}\)
    using * unfolding bsqr-def by auto
  moreover
  have \(b_1 = a_1 \land b_2 = a_2 \lor (\text{wo-rel.max2 } r \ b_1 \ b_2, \text{wo-rel.max2 } r \ a_1 \ a_2) \in r - \text{Id}\)
    using ** unfolding bsqr-def by auto

ultimately show \(a_1 = b_1 \land a_2 = b_2\) using IrrS by auto
qed

lemma bsqr-Total:
  assumes Well-order \(r\)
  shows Total(\(\text{bsqr } r\))
proof–

  have Well: wo-rel \(r\) using assms wo-rel-def by auto
\textbf{THEORY “BNF-Wellorder-Constructions”}

hence Total: \( \forall a \in \text{Field } r. \forall b \in \text{Field } r. \ (a,b) \in r \lor (b,a) \in r \)
using wo-rel.TOTALS by auto

\{ fix \ a1 a2 b1 b2 assume \([(a1,a2),(b1,b2)] \leq \text{Field}(bsqr r)
    hence 0: a1 \in \text{Field } r \land a2 \in \text{Field } r \land b1 \in \text{Field } r \land b2 \in \text{Field } r
    using Field-bsqr by blast
    have \([(a1,a2),(b1,b2)] = (b1,b2) \lor [(a1,a2),(b1,b2)] \in bsqr r \lor [(b1,b2),(a1,a2)] \in bsqr r
    proof(rule wo-rel.cases-Total[of r a1 a2], clarsimp simp add: Well, simp add: 0)
        assume Case1: \((a1,a2) \in r
        hence 1: wo-rel.max2 r a1 a2 = a2
        using Well 0 by (simp add: wo-rel.max2-equals2)
        show ?thesis
        proof(rule wo-rel.cases-Total[of r b1 b2], clarsimp simp add: Well, simp add: 0)
            assume Case11: \((b1,b2) \in r
            hence 2: wo-rel.max2 r b1 b2 = b2
            using Well 0 by (simp add: wo-rel.max2-equals2)
            show ?thesis
            proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp add: 0)
                assume Case111: \((a2,b2) \in r \land \text{Id \lor (b2,a2) \in r \land \text{Id}
                thus ?thesis using 0 1 2 unfolding bsqr-def by auto
            next
                assume Case112: \(a2 = b2
                show ?thesis
                proof(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp add: 0)
                    assume Case1121: \((a1,b1) \in r \land \text{Id \lor (b1,a1) \in r \land \text{Id}
                    thus ?thesis using 0 1 2 Case112 unfolding bsqr-def by auto
                next
                    assume Case1122: \(a1 = b1
                    thus ?thesis using Case112 by auto
                qed
            qed
        next
            assume Case12: \((b2,b1) \in r
        hence 3: wo-rel.max2 r b1 b2 = b1 using Well 0 by (simp add: wo-rel.max2-equals1)
        show ?thesis
        proof(rule wo-rel.cases-Total3[of r a2 b1], clarsimp simp add: Well, simp add: 0)
            assume Case121: \((a2,b1) \in r \land \text{Id \lor (b1,a2) \in r \land \text{Id}
            thus ?thesis using 0 1 3 unfolding bsqr-def by auto
        next
            assume Case122: \(a2 = b1
            show ?thesis
            proof(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp add: 0)
    qed

"
add: 0)
assume Case1221: (a1,b1) ∈ r − Id ∨ (b1,a1) ∈ r − Id
thus ?thesis using 0 1 3 Case122 unfolding bsqr-def by auto
next
assume Case1222: a1 = b1
show ?thesis
proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp
add: 0)
assume Case1221: (a2,b2) ∈ r − Id ∨ (b2,a2) ∈ r − Id
thus ?thesis using 0 1 3 Case122 Case1222 unfolding bsqr-def by auto
next
assume Case1222: a2 = b2
thus ?thesis using Case122 Case1222 by auto

next
assume Case2: (a2,a1) ∈ r
hence 1: wo-rel.max2 r a1 a2 = a1 using Well 0 by (simp add: wo-rel.max2-equals1)
show ?thesis
proof(rule wo-rel.cases-Total[of r b1 b2], clarsimp simp add: Well, simp add: 0)
assume Case21: (b1,b2) ∈ r
hence 2: wo-rel.max2 r b1 b2 = b2 using Well 0 by (simp add: wo-rel.max2-equals2)
show ?thesis
proof(rule wo-rel.cases-Total3[of r a1 b2], clarsimp simp add: Well, simp add: 0)
assume Case211: (a1,b2) ∈ r − Id ∨ (b2,a1) ∈ r − Id
thus ?thesis using 0 1 2 unfolding bsqr-def by auto
next
assume Case212: a1 = b2
show ?thesis
proof(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp add: 0)
assume Case2121: (a1,b1) ∈ r − Id ∨ (b1,a1) ∈ r − Id
thus ?thesis using 0 1 2 Case212 unfolding bsqr-def by auto
next
assume Case2122: a1 = b1
show ?thesis
proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp add: 0)
assume Case2122: (a2,b2) ∈ r − Id ∨ (b2,a2) ∈ r − Id
thus ?thesis using 0 1 2 Case212 Case2122 unfolding bsqr-def by auto
next
assume Case2122: a2 = b2
thus ?thesis using Case2122 Case212 by auto
qed
qed
qed
next
assume Case22: \((b2,b1) \in r\)
    hence 3: wo-rel.max2 \(r b1 b2 = b1\) using Well 0 by (simp add: wo-rel.max2-equals1)
show \(?thesis\)
proof (rule wo-rel.cases-Total3[of \(r a1 b1\)], clarsimp simp add: Well, simp add: 0)
    assume Case221: \((a1,b1) \in r - Id \lor (b1,a1) \in r - Id\)
    thus \(?thesis using 0 1 3 unfolding bsqr-def by auto\)
next
assume Case222: \(a1 = b1\)
show \(?thesis\)
proof (rule wo-rel.cases-Total3[of \(r a2 b2\)], clarsimp simp add: Well, simp add: 0)
    assume Case2221: \((a2,b2) \in r - Id \lor (b2,a2) \in r - Id\)
    thus \(?thesis using 0 1 3 Case222 unfolding bsqr-def by auto\)
next
assume Case2222: \(a2 = b2\)
thus \(?thesis using Case222 by auto\)
qed
qed
qed
qed

Thus \(?thesis unfolding total-on-def by fast\)
qed

lemma bsqr-Linear-order:
assumes Well-order \(r\)
shows Linear-order(bsqr \(r\))
ounfolding order-on-defs
using assms bsqr-Refl bsqr-Trans bsqr-antisym bsqr-Total by blast

lemma bsqr-Well-order:
assumes Well-order \(r\)
shows Well-order(bsqr \(r\))
using assms
proof (simp add: bsqr-Linear-order Linear-order-Well-order-iff, intro allI impI)
  have 0: \(\forall A \leq \text{Field } r. A \neq \{\} \rightarrow (\exists a \in A. \forall a' \in A. (a,a') \in r)\)
    using assms well-order-on-def Linear-order-Well-order-iff by blast
  fix \(D\) assume \(*\): \(D \leq \text{Field } (\text{bsqr } r)\) and \(**\): \(D \neq \{\}\)
  hence I: \(D \leq \text{Field } r \times \text{Field } r\) unfolding Field-bsqr by simp
obtain \(M\) where \(M\)-def: \(M = \{\text{wo-rel.max2 } r a1 a2| a1 a2. (a1,a2) \in D\}\) by blast
  have \(M \neq \{\}\) using I \(M\)-def \(**\) by auto
moreover
have \( M \leq \text{Field } r \) unfolding \( M\)-def
  using 1 assms wo-rel-def[of \( r \)] wo-rel.max2-among[of \( r \)] by fastforce
ultimately obtain \( m \) where \( m\)-min: \( m \in M \land (\forall a \in M. (m,a) \in r) \)
  using 0 by blast
obtain \( A1 \) where \( A1\)-def: \( A1 = \{a1. \exists a2. (a1,a2) \in D \land wo-rel.max2 r a1 a2 = m\} \)
  by blast
have \( A1 \leq \text{Field } r \) unfolding \( A1\)-def using 1 by auto
moreover have \( A1 \neq \{\} \) unfolding \( A1\)-def using \( m\)-min unfolding \( M\)-def by blast
ultimately obtain \( a1 \) where \( a1\)-min: \( a1 \in A1 \land (\forall a \in A1. (a1,a) \in r) \)
  using 0 by blast
obtain \( A2 \) where \( A2\)-def: \( A2 = \{a2. (a1,a2) \in D \land wo-rel.max2 r a1 a2 = m\} \)
  by blast
have \( A2 \leq \text{Field } r \) unfolding \( A2\)-def using 1 by auto
moreover have \( A2 \neq \{\} \) unfolding \( A2\)-def
  using \( m\)-min \( a1\)-min unfolding \( A1\)-def \( M\)-def by blast
ultimately obtain \( a2 \) where \( a2\)-min: \( a2 \in A2 \land (\forall a \in A2. (a2,a) \in r) \)
  using 0 by blast
have 2: wo-rel.max2 r a1 a2 = m
  using \( a1\)-min \( a2\)-min unfolding \( A1\)-def \( A2\)-def by auto
have 3: \((a1,a2) \in D\) using \( a2\)-min unfolding \( A2\)-def by auto
moreover
\{fix \( b1 \) \( b2 \) assume \( ***: (b1,b2) \in D\)
  hence 4: \( \{a1,a2,b1,b2\} \leq \text{Field } r \) using 1 3 by blast
  have 5: \((wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) \in r \)
    using \( *** \) \( a1\)-min \( a2\)-min \( m\)-min unfolding \( A1\)-def \( A2\)-def \( M\)-def by auto
  have \(((a1,a2),(b1,b2)) \in \text{bsqr } r\)
  proof(cases wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2)
    assume Case1: wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2
    thus \( ?\)thesis unfolding bsqr-def using 4 5 by auto
next
  assume Case2: wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2
  hence \( b1 \in A1 \) unfolding \( A1\)-def using 2 *** by auto
  hence 6: \((a1,b1) \in r\) using \( a1\)-min by auto
  show \( ?\)thesis
  proof(cases \( a1 = b1 \))
    assume Case21: \( a1 \neq b1 \)
    thus \( ?\)thesis unfolding bsqr-def using 4 Case2 6 by auto
next
  assume Case22: \( a1 = b1 \)
  hence \( b2 \in A2 \) unfolding \( A2\)-def using 2 *** Case2 by auto
  hence 7: \((a2,b2) \in r\) using \( a2\)-min by auto
  thus \( ?\)thesis unfolding bsqr-def using 4 7 Case2 Case22 by auto
qed
ultimately show \( \exists d \in D. \forall d' \in D. (d,d') \in \text{bsqr} \ r \) by \texttt{fastforce}

lemma \texttt{bsqr-max2}:

assumes \( \text{WELL: Well-order } r \) and \( \text{LEQ: } ((a1,a2),(b1,b2)) \in \text{bsqr} \ r \)

shows \( (\text{wo-rel.max2} \ r \ a1 \ a2, \text{wo-rel.max2} \ r \ b1 \ b2) \in r \)

proof –

have \( \{(a1,a2),(b1,b2)\} \leq \text{Field(bsqr} \ r) \)

using \( \text{LEQ unfolding } \text{Field-def by auto} \)

hence \( \{a1,a2,b1,b2\} \leq \text{Field } \text{r unfolding } \text{Field-bsqr by auto} \)

hence \( \{\text{wo-rel.max2} \ r \ a1 \ a2, \text{wo-rel.max2} \ r \ b1 \ b2\} \leq \text{Field } \text{r} \)

using \( \text{WELL wo-rel-def[of } r \text{ } \text{wo-rel.max2-among[of } r \text{] by fastforce} \)

moreover have \( (\text{wo-rel.max2} \ r \ a1 \ a2, \text{wo-rel.max2} \ r \ b1 \ b2) \in r \lor \text{wo-rel.max2} \ r \ a1 \ a2 = \text{wo-rel.max2} \ r \ b1 \ b2 \)

using \( \text{LEQ unfolding } \text{bsqr-def by auto} \)

ultimately show \( \text{?thesis using WELL unfolding } \text{order-on-defs refl-on-def by auto} \)

qed

lemma \texttt{bsqr-ofilter}:

assumes \( \text{WELL: Well-order } r \) and \( \text{OF: } \text{wo-rel.ofilter (bsqr } r \) \( D \) and \( \text{SUB: } D < \text{Field } r \times \text{Field } r \) and \( \text{NE: } \neg (\exists a. \text{Field } r = \text{under } r \ a) \)

shows \( \exists A. \text{wo-rel.ofilter } r \ A \land A < \text{Field } r \land D \leq A \times A \)

proof –

let \( ?r' = \text{bsqr } r \)

have \( \text{Well: } \text{wo-rel } r \text{ using WELL wo-rel-def by blast} \)

hence \( \text{Trans: trans } r \text{ using } \text{wo-rel.TRANS by blast} \)

have \( \text{Well': Well-order } ?r' \land \text{wo-rel } ?r' \)

using \( \text{WELL bsqr-Well-order wo-rel-def by blast} \)

have \( D < \text{Field } ?r' \text{ unfolding } \text{Field-bsqr using SUB}. \)

with \( \text{OF obtain } a1 \text{ and } a2 \) where

\( (a1,a2) \in \text{Field } ?r' \text{ and } 1: D = \text{under } S ?r' (a1,a2) \)

using \( \text{Well' wo-rel.ofilter-underS } \text{Field[of } ?r' \text{ } D \text{ by auto} \)

hence \( 2: \{a1,a2\} \leq \text{Field } \text{r unfolding } \text{Field-bsqr by auto} \)

let \( ?m = \text{wo-rel.max2 } r \ a1 \ a2 \)

have \( D \leq (\text{under } r \ ?m) \times (\text{under } r \ ?m) \)

proof(unfold 1)

\{ fix b1 b2 \}

let \( ?n = \text{wo-rel.max2 } r \ b1 \ b2 \)

assume \( (b1,b2) \in \text{under } S ?r' (a1,a2) \)

hence \( 3: ((b1,b2),(a1,a2)) \in ?r' \)

unfolding \( \text{underS-def by blast} \)

hence \( (?n,?m) \in r \text{ using WELL by (simp add: bsqr-max2)} \)

moreover


{have \((b_1, b_2) \in \text{Field} \vdash r'\) using 3 unfolding \(\text{Field-def}\) by auto

hence \(\{b_1, b_2\} \leq \text{Field} r\) unfolding \(\text{Field-bsqr}\) by auto

hence \((b_1, ?n) \in r \land (b_2, ?m) \in r\)

using \(\text{Well} by (simp add: wo-rel\cdot max2\cdot greater)\)

}

ultimately have \((b_1, ?m) \in r \land (b_2, ?m) \in r\)

using \(\text{Trans trans-def[of} r\) by blast

hence \((b_1, b_2) \in (\text{under} r \vdash ?m) \times (\text{under} r \vdash ?m)\)

unfolding \(\text{under-def}\) by simp

thus \(\text{underS} ?r' (a_1, a_2) \leq (\text{under} r \vdash ?m) \times (\text{under} r \vdash ?m)\) by auto

qed

moreover have \(\text{wo-rel.ofilter} r (\text{under} r \vdash ?m)\)

using \(\text{Well by (simp add: wo-rel.under.ofilter)}\)

moreover have \(\text{under} r \vdash ?m < \text{Field} r\)

using \(\text{NE under-Field[of} r \vdash ?m]\) by blast

ultimately show \(\?thesis\) by blast

qed


definition \(\text{Func where}\)

\(\text{Func} A B = \{f . (\forall a \in A. f a \in B) \land (\forall a. a \notin A \rightarrow f a = \text{undefined})\}\)

lemma \(\text{Func-empty};\)

\(\text{Func} \{\} B = \{\lambda x. \text{undefined}\}\)

unfolding \(\text{Func-def}\) by auto

lemma \(\text{Func-elim};\)

assumes \(g \in \text{Func} A B\) and \(a \in A\)

shows \(\exists b. b \in B \land g a = b\)

using \(\text{assms unfolding Func-def by (cases g a = undefined) auto}\)


definition \(\text{curr where}\)

\(\text{curr} A f \equiv \lambda a. \text{if } a \in A \text{ then } \lambda b. f (a, b) \text{ else undefined}\)


definition \(\text{curr-in};\)

assumes \(f : f \in \text{Func} (A \times B) C\)

shows \(\text{curr} A f \in \text{Func} A (\text{Func} B C)\)

using \(\text{assms unfolding curr-def Func-def by auto}\)

lemma \(\text{curr-inj};\)

assumes \(f_1 \in \text{Func} (A \times B) C\) and \(f_2 \in \text{Func} (A \times B) C\)

shows \(\text{curr} A f_1 = \text{curr} A f_2 \iff f_1 = f_2\)

proof safe

assume \(c: \text{curr} A f_1 = \text{curr} A f_2\)

show \(f_1 = f_2\)

proof (rule ext, clarify)

fix \(a\ b\) show \(f_1 (a, b) = f_2 (a, b)\)

proof (cases \((a, b) \in A \times B)\)

case False

thus \(\?thesis\) using \(\text{assms unfolding Func-def by auto}\)
THEORY “BNF-Wellorder-Constructions” 722

next
case True hence a ∈ A and b ∈ B by auto
thus ?thesis
using c unfolding curr-def fun-eq-iff by (elim allE[of - a]) simp
qed
qed
qed

lemma curr-surj:
assumes g ∈ Func A (Func B C)
shows ∃ f ∈ Func (A × B) C. curr A f = g
proof
let ?f = λ ab. if fst ab ∈ A ∧ snd ab ∈ B then g (fst ab) (snd ab) else undefined
show curr A ?f = g
proof (rule ext)
fix a show curr A ?f a = g a
proof (cases a ∈ A)
case False
hence g a = undefined using assms unfolding Func-def by auto
thus ?thesis unfolding curr-def using False by simp
next
case True
obtain g1 where g1 ∈ Func B C and g a = g1 using assms using Func-elim[OF assms True]
thus ?thesis using True unfolding Func-def curr-def by auto
qed
qed
show ?f ∈ Func (A × B) C using assms unfolding Func-def mem-Collect-eq by auto
qed

lemma bij-betw-curr:
bij-betw (curr A) (Func (A × B) C) (Func A (Func B C))
unfolding bij-betw-def inj-on-def
using curr-surj curr-in curr-inj by blast

definition Func-map where
Func-map B2 f1 f2 g b2 ≡ if b2 ∈ B2 then f1 (g (f2 b2)) else undefined

lemma Func-map:
assumes g: g ∈ Func A2 A1 and f1: f1 : A1 ⊆ B1 and f2: f2 : B2 ⊆ A2
shows Func-map B2 f1 f2 g ∈ Func B2 B1
using assms unfolding Func-def Func-map-def mem-Collect-eq by auto

lemma Func-non-emp:
assumes B ≠ {}
shows Func A B ≠ {}
proof –
obtain b where b: b ∈ B using assms by auto
hence \((\lambda a. \text{if } a \in A \text{ then } b \text{ else undefined}) \in \text{Func } A \ B\) unfolding \text{Func-def} by auto

thus \(?thesis\) by blast

qed

lemma \text{Func-is-emp}:
\text{Func } A \ B = \{\} \iff A \neq \{\} \land B = \{\} (\text{is } ?L \iff ?R)

proof
assume \(?L\)
then show \(?R\)
  using \text{Func-empty} \text{Func-non-emp}[\text{of } B \ A]
  by blast

next
assume \(?R\)
then show \(?L\)
  using \text{Func-empty} \text{Func-non-emp}[\text{of } B \ A]
  by (auto simp: \text{Func-def})

qed

lemma \text{Func-map-surj}:
assumes \(B1: f1 \ A1 = B1\) and \(A2: \text{inj-on } f2 B2 f2 ' B2 \subseteq A2\)
and \(B2A2: B2 = \{\} \implies A2 = \{\}\)
shows \(\text{Func } B2 B1 = \text{Func-map } B2 f1 f2 ' \text{Func } A2 A1\)

proof (cases \(B2 = \{\}\))
  case True
  thus \(?thesis\) using \(B2A2\) by (auto simp: \text{Func-empty} \text{Func-map-def})

next
  case False
  note \(B2\)
  show \(?thesis\)
    proof safe
    fix \(h\)
    assume \(h\)
    define \(j1\) where \(j1 = \text{inv-into } A1 f1\)
    have \(\forall a2 \in f2 ' B2. \exists b2. b2 \in B2 \land f2 b2 = a2\) by blast
    then obtain \(k\) where \(k: \forall a2 \in f2 ' B2. k a2 \in B2 \land f2 (k a2) = a2\)
      by atomize-clim \(\text{rule bchoice}\)
    \{fix \(b2\) assume \(b2\): \(b2 \in B2\)
    hence \(f2 (k (f2 b2)) = f2 b2\) using \(k A2(2)\) by auto
    moreover have \(k (f2 b2) \in B2\) using \(b2 A2(2)\) \(k\) by auto
    ultimately have \(k (f2 b2) = b2\) using \(b2 A2(1)\) unfolding \text{inj-on-def} by blast\}
    note \(kk\) = \this
    obtain \(b22\) where \(b22: b22 \in B2\) using \(B2\) by auto
    define \(j2\) where \(\text{abs-def}\): \(j2 a2 = (a2 \in f2 ' B2 \text{ then } k a2 \text{ else } b22)\) for \(a2\)
    have \(j2A2: j2 ' A2 \subseteq B2\) unfolding \text{j2-def} using \(b22\) by auto
    have \(j2: \forall b2. b2 \in B2 \implies j2 (f2 b2) = b2\)
      using \(kk\) unfolding \text{j2-def} by auto
    define \(g\) where \(g = \text{Func-map } A2 j1 j2 h\)
    have \(\text{Func-map } B2 f1 f2 g = h\)
    proof \(\text{rule ext}\)
      fix \(b2\)
      show \(\text{Func-map } B2 f1 f2 g b2 = h b2\)
      proof (cases \(b2 \in B2\))
THEORY “BNF-Cardinal-Order-Relation”

case True
show ?thesis
proof (cases h b2 = undefined)
case True
  hence b1: h b2 ∈ f1 ' A1 using h (b2 ∈ B2) unfolding B1 Func-def by auto
show ?thesis using A2 f-inv-into-f[OF b1]
  unfolding True g-def Func-map-def j1-def j2[OF (b2 ∈ B2)] by auto
qed(insert A2 True j2[OF True] h B1, unfold j1-def g-def Func-def Func-map-def,
  auto intro: f-inv-into-f)
  auto intro: f-inv-into-f)
       unfolding Func-map-def[abs-def] by auto
  ultimately show h ∈ Func A2 A1 unfolding g-def apply(rule Func-map[OF h])
    using j2A2 B1 A2 unfolding j1-def by (fast intro: inv-into-into)+
    ultimately show h ∈ Func-map B2 f1 f2 ' Func A2 A1
    unfolding Func-map-def[abs-def] by auto
  qed(insert h, unfold Func-def Func-map-def, auto)
  qed

end

30  Cardinal-Order Relations as Needed by Bounded Natural Functors

theory BNF-Cardinal-Order-Relation
  imports Zorn BNF-Wellorder-Constructions
begin

In this section, we define cardinal-order relations to be minim well-orders on
their field. Then we define the cardinal of a set to be some cardinal-order
relation on that set, which will be unique up to order isomorphism. Then
we study the connection between cardinals and:

• standard set-theoretic constructions: products, sums, unions, lists, pow-
ersets, set-of finite sets operator;

• finiteness and infiniteness (in particular, with the numeric cardinal
  operator for finite sets, card, from the theory Finite-Sets.thy).

On the way, we define the canonical ω cardinal and finite cardinals. We
also define (again, up to order isomorphism) the successor of a cardinal, and
show that any cardinal admits a successor.

Main results of this section are the existence of cardinal relations and the
facts that, in the presence of infiniteness, most of the standard set-theoretic
constructions (except for the powerset) do not increase cardinality. In particular, e.g., the set of words/lists over any infinite set has the same cardinality (hence, is in bijection) with that set.

30.1 Cardinal orders

A cardinal order in our setting shall be a well-order \(\text{minim}\) w.r.t. the order-embedding relation, \(\leq_o\) (which is the same as being \(\text{minimal}\) w.r.t. the strict order-embedding relation, \(<_o\) ), among all the well-orders on its field.

**definition** \(\text{card-order-on}::'a\ set\Rightarrow'a\ rel\Rightarrow\text{bool}\)

\(\text{card-order-on } A\ r\equiv\text{well-order-on } A\ r\land(\forall r'.\text{well-order-on } A\ r'\rightarrow r\leq_o r')\)

**abbreviation** \(\text{Card-order } r\equiv\text{card-order-on } (\text{Field } r)\ r\)

**abbreviation** \(\text{card-order } r\equiv\text{card-order-on } \text{UNIV } r\)

**lemma** \(\text{card-order-on-well-order-on}:\)

assumes \(\text{card-order-on } A\ r\)

shows \(\text{well-order-on } A\ r\)

using assms unfolding card-order-on-def by simp

**lemma** \(\text{card-order-on-Card-order}:\)

\(\text{card-order-on } A\ r\Rightarrow A=\text{Field } r\land\text{Card-order } r\)

unfolding card-order-on-def using well-order-on-Field by blast

The existence of a cardinal relation on any given set (which will mean that any set has a cardinal) follows from two facts:

- Zermelo’s theorem (proved in Zorn.thy as theorem well-order-on), which states that on any given set there exists a well-order;

- The well-founded-ness of \(<_o\), ensuring that then there exists a minimal such well-order, i.e., a cardinal order.

**theorem** \(\text{card-order-on}::\exists r.\text{card-order-on } A\ r\)

**proof** –

define \(R\ where\ R\equiv\{r.\text{well-order-on } A\ r\}\)

have \(R\neq\\{\}\\land(\forall r\in R.\text{Well-order } r)\)

using well-order-on[of A] R-def well-order-on-Well-order by blast

with exists-minim-Well-order[of R] show ?thesis

by (auto simp: R-def card-order-on-def)

qed

**lemma** \(\text{card-order-on-ordIso}:\)

assumes \(\text{CO:card-order-on } A\ r\ \text{and } \text{CO':card-order-on } A\ r'\)

shows \(r=o\ r'\)

using assms unfolding card-order-on-def

using ordIso-iff-ordLeq by blast
lemma Card-order-ordIso:
  assumes CO: Card-order r and ISO: r' = o r
  shows Card-order r'
  using ISO unfolding ordIso-def
proof (unfold card-order-on-def, auto)
  fix p' assume well-order-on (Field r') p'
  hence 0: Well-order p' ∧ Field p' = Field r'
    using well-order-on-Well-order by blast
  obtain f where 1: iso r' r f and 2: Well-order r ∧ Well-order r'
    using ISO unfolding ordIso-def by auto
  hence 3: inj-on f (Field r') ∧ f ' (Field r') = Field r
    by (auto simp add: iso-iff embed-inj-on)
  let ?p = dir-image p' f
  have 4: p' = o ?p ∧ Well-order ?p
    using 0 2 3 by (auto simp add: dir-image-ordIso Well-order-dir-image)
  moreover have Field ?p = Field r
    using 0 3 by (auto simp add: dir-image-Field)
  ultimately have well-order-on (Field r) ?p by auto
  hence r ≤ o ?p using CO unfolding card-order-on-def by auto
  thus r' ≤ o p'
    using ISO 4 ordLeq-ordIso-trans ordIso-ordLeq-trans ordIso-symmetric by blast
qed

lemma Card-order-ordIso2:
  assumes CO: Card-order r and ISO: r' = o r'
  shows Card-order r'
  using assms Card-order-ordIso ordIso-symmetric by blast

30.2 Cardinal of a set

We define the cardinal of set to be some cardinal order on that set. We shall prove that this notion is unique up to order isomorphism, meaning that order isomorphism shall be the true identity of cardinals.
definition card-of :: 'a set ⇒ 'a rel (||-||)
  where card-of A = (SOME r. card-order-on A r)

lemma card-of-card-order-on: card-order-on A |A|
  unfolding card-of-def by (auto simp add: card-order-on someI-ex)

lemma card-of-well-order-on: well-order-on A |A|
  using card-of-card-order-on card-order-on-def by blast

lemma Field-card-of: Field |A| = A
  using card-of-card-order-on[of A] unfolding card-order-on-def
  using well-order-on-Field by blast

lemma card-of-Card-order: Card-order |A|
  by (simp only: card-of-card-order-on Field-card-of)
corollary ordIso-card-of-imp-Card-order:
  \( r = \circ |A| \implies \text{Card-order } r \)
  using card-of-Card-order Card-order-ordIso by blast

lemma card-of-Well-order: Well-order \(|A|\)
  using card-of-Card-order unfolding card-order-on-def by auto

lemma card-of-refl: \(|A| = \circ |A|\)
  using card-of-Well-order unfolding card-order-on-def by auto

lemma card-of-least: well-order-on \(A\) \(r\) \(\implies\) \(|A| \leq \circ r\)
  using card-of-card-order-on unfolding card-order-on-def by blast

lemma card-of-ordIso:
  \((\exists f. \text{bij-betw } f A B) = (|A| = \circ |B|)\)
  proof(auto)
  fix \(f\) assume \(\ast\): \(\text{bij-betw } f A B\)
  then obtain \(r\) where \(\text{well-order-on } B\) \(\land\) \(|A| = \circ r\)
    using Well-order-iso-copy card-of-well-order-on by blast
  hence \(|B| \leq \circ |A|\) using card-of-least
    ordLeq-ordIso-trans ordIso-symmetric by blast

moreover
  \{let \(?g = \text{inv-into } A f\)
  have \(\text{bij-betw } ?g B A\) using \(\ast\) \(\text{bij-betw-inv-into}\) by blast
  then obtain \(r\) where \(\text{well-order-on } A\) \(\land\) \(|B| = \circ r\)
    using Well-order-iso-copy card-of-well-order-on by blast
  hence \(|A| \leq \circ |B|\)
    using card-of-least ordLeq-ordIso-trans ordIso-symmetric by blast
  \}

ultimately show \(|A| = \circ |B|\) using ordIso-iff-ordLeq by blast
next
  assume \(|A| = \circ |B|\)
  then obtain \(f\) where \(\text{iso } (|A|) (|B|) f\)
    unfolding ordIso-def by auto
  hence \(\text{bij-betw } f A B\) unfolding iso-def Field-card-of by simp
  thus \(\exists f. \text{bij-betw } f A B\) by auto
qed

lemma card-of-ordLeq:
  \((\exists f. \text{inj-on } f A \land f \cdot A \leq B) = (|A| \leq \circ |B|)\)
  proof(auto)
  fix \(f\) assume \(\ast\): \(\text{inj-on } f A\) and \(\ast\ast\): \(f \cdot A \leq B\)
    \{assume \(|B| < \circ |A|\)
    hence \(|B| \leq \circ |A|\) using ordLeq-iff-ordLess-or-ordIso by blast
    then obtain \(g\) where \(\text{embed } (|B|) (|A|) g\)
      unfolding ordLeq-def by auto
    hence \(1: \text{inj-on } g B \land g \cdot B \leq A\) using embed-inj-on[of \(|B|\) \(|A|\) \(g\)]
      card-of-Well-order[of \(|B|\) Field-card-of[of \(|B|\) Field-card-of[of \(|A|\)]

THEORY “BNF-Cardinal-Order-Relation”

embed-Field [of |B| |A| g] by auto
obtain h where bij-betw h A B
  using ** 1 Schroeder-Bernstein[of f] by fastforce
  hence |A| ≤o |B| using card-of-ordIso ordIso-iff-ordLeq by auto
}
thus |A| ≤o |B| using ordLess-or-ordLeq by auto
by (auto simp: card-of-Well-order)
next
assume *: |A| ≤o |B|
obtain f where embed |A| |B| f
  using * unfolding ordLeq-def by auto
hence inj-on f A ∧ f ' A ≤ B
  using embed-inj-on[of |A| |B|] card-of-Well-order embed-Field[of |A| |B|]
  by (auto simp: Field-card-of)
thus ∃f. inj-on f A ∧ f ' A ≤ B by auto
qed

lemma card-of-ordLeq2:
A ≠ {} ⇒ (∃g. g ' B = A) = ( |A| ≤o |B| )
using card-of-ordLeq[of A B] inj-on-iff-surj[of A B] by auto

lemma card-of-ordLess:
(¬ (∃f. inj-on f A ∧ f ' A ≤ B)) = ( |B| <o |A| )
proof —
  have (¬ (∃f. inj-on f A ∧ f ' A ≤ B)) = (¬ |A| ≤o |B| )
    using card-of-ordLeq by blast
  also have ... = ( |B| <o |A| )
    using not-ordLeq-iff-ordLess by (auto intro: card-of-Well-order)
finally show ?thesis .
qed

lemma card-of-ordLess2:
B ≠ {} ⇒ (¬ (∃f. f ' A = B)) = ( |A| <o |B| )
using card-of-ordLess[of B A] inj-on-iff-surj[of B A] by auto

lemma card-of-ordIsoI:
  assumes bij-betw f A B
  shows |A| =o |B|
  using assms unfolding card-of-ordIso[symmetric] by auto

lemma card-of-ordLeqI:
  assumes inj-on f A and A. a ∈ A ⇒ f a ∈ B
  shows |A| ≤o |B|
  using assms unfolding card-of-ordLeq[symmetric] by auto

lemma card-of-unique:
card-order-on A r ≡ r =o |A|
by (simp only: card-order-on-ordIso card-of-card-order-on)
lemma card-of-mono1:
\[ A \leq B \implies |A| \leq_o |B| \]
using inj-on-id[of A] card-of-ordLeq[of A B] by fastforce

lemma card-of-mono2:
assumes \( r \leq_o r' \)
shows \( |\text{Field } r| \leq_o |\text{Field } r'| \)
proof -
obtain f where
  \( 1: \text{well-order-on } (\text{Field } r) \land \text{well-order-on } (\text{Field } r') \land \text{embed } r \\ r' f \)
using assms unfolding ordLeq-def
by (auto simp add: well-order-on-Well-order)
  hence inj-on f (\text{Field } r) \land f' (\text{Field } r) \leq \text{Field } r'
by (auto simp add: embed-inj-on embed-Field)
  thus |\text{Field } r| \leq_o |\text{Field } r'| using card-of-ordLeq by blast
qed

lemma card-of-cong: \( r =_o r' \implies |\text{Field } r| =_o |\text{Field } r'| \)
by (simp add: ordIso-iff-ordLeq card-of-mono2)

lemma card-of-Field-ordIso:
assumes Card-order r
shows |\text{Field } r| =_o r
proof -
  have \( \text{card-order-on } (\text{Field } r) \land \text{card-order-on } (\text{Field } r) \land \text{card-order-on } (\text{Field } r) \land \text{card-order-on } (\text{Field } r) \)
      using assms card-order-on-Card-order by blast
moreover have \( |\text{Field } r| =_o r \)
using card-of-card-order-on by blast
ultimately show ?thesis using card-order-on-ordIso by blast
qed

lemma Card-order-iff-ordIso-card-of:
\( \text{Card-order } r = (r =_o |\text{Field } r|) \)
using ordIso-card-of-imp-Card-order card-of-Field-ordIso ordIso-symmetric by blast

lemma Card-order-iff-ordLeq-card-of:
\( \text{Card-order } r = (r \leq_o |\text{Field } r|) \)
proof -
  have Card-order r = \( (r =_o |\text{Field } r|) \)
    unfolding Card-order-iff-ordIso-card-of by simp
  also have \( \ldots = (r \leq_o |\text{Field } r| \land |\text{Field } r| \leq_o r) \)
    unfolding ordIso-iff-ordLeq by simp
  also have \( \ldots = (r \leq_o |\text{Field } r|) \)
    using card-of-least
    by (auto simp: card-of-least ordLeq-Well-order-simp)
finally show ?thesis .
qed
lemma Card-order-iff-Restr-underS:
assumes Well-order r
shows Card-order r = (∀ a ∈ Field r. Restr r (underS r a) <o |Field r| )
using assms ordLeq-iff-ordLess-Restr card-of-Well-order
unfolding Card-order-iff-ordLeq-card-of by blast

lemma card-of-underS:
assumes r: Card-order r and a: a ∈ Field r
shows |underS r a| <o r
proof –
let ?A = underS r a let ?r' = Restr r ?A
have 1: Well-order r
using r unfolding card-order-on-def by simp
have Well-order ?r' using 1 Well-order-Restr by auto
with card-of-card-order-on have |Field ?r'| ≤o ?r'
  unfolding card-order-on-def by auto
moreover have Field ?r' = ?A
  using 1 wo-rel.underS-ofilter Field-Restr-ofilter
  unfolding wo-rel-def by fastforce
ultimately have |?A| ≤o ?r' by simp
also have ?r' <o |Field r| using 1 a r Card-order-iff-Restr-underS by blast
also have |Field r| =o r
  using r ordIso-symmetric unfolding Card-order-iff-ordIso-card-of by auto
finally show ?thesis.
qed

lemma ordLess-Field:
assumes r <o r'
shows |Field r| <o r'
proof –
have well-order-on (Field r) r using assms unfolding ordLess-def
  by (auto simp add: well-order-on-Well-order)
hence |Field r| ≤o r using card-of-least by blast
thus ?thesis using assms ordLeq-ordLess-trans by blast
qed

lemma internalize-card-of-ordLeq:
( |A| ≤o r ) = (∃ B ≤ Field r. |A| =o |B| ∧ |B| ≤o r )
proof
assume |A| ≤o r
then obtain p where 1: Field p ≤ Field r ∧ |A| =o p ∧ p ≤o r
  using internalize-ordLeq[of |A| r] by blast
hence Card-order p using card-of-Card-order Card-order-ordIso2 by blast
hence |Field p| =o p using card-of-Field-ordIso by blast
hence |A| =o |Field p| ∧ |Field p| ≤o r
  using 1 ordIso-equivalence ordIso-ordLeq-trans by blast
thus ∃ B ≤ Field r. |A| =o |B| ∧ |B| ≤o r using 1 by blast
next
THEORY “BNF-Cardinal-Order-Relation”

assume $\exists B \leq \text{Field } r. \ |A| = o \ |B| \land |B| \leq o \ r$

thus $|A| \leq o \ r$ using ordIso-ordLeq-trans by blast

qed

lemma internalize-card-of-ordLeq2:

$(|A| \leq o \ |C| ) = (\exists B \leq C. |A| = o \ |B| \land |B| \leq o \ |C| )$


30.3 Cardinals versus set operations on arbitrary sets

Here we embark in a long journey of simple results showing that the standard set-theoretic operations are well-behaved w.r.t. the notion of cardinal – essentially, this means that they preserve the “cardinal identity” $= o$ and are monotonic w.r.t. $\leq o$.

lemma card-of-empty: $|\{\}| \leq o \ |A|$

using card-of-ordLeq inj-on-id by blast

lemma card-of-empty1:

assumes Well-order $r \lor$ Card-order $r$

shows $|\{\}| \leq o \ r$

proof

have Well-order $r$ using assms unfolding card-order-on-def by auto

hence $|\text{Field } r| \leq o \ r$

using assms card-of-least by blast

moreover have $|\{\}| \leq o \ |\text{Field } r|$ by (simp add: card-of-empty)

ultimately show ?thesis using ordLeq-transitive by blast

qed

corollary Card-order-empty:

Card-order $r \implies |\{\}| \leq o \ r$ by (simp add: card-of-empty1)

lemma card-of-empty2:

assumes $|A| = o \ |\{\}|$

shows $A = \{\}$

using assms card-of-ordIso[of $A$] bij-betw-empty2 by blast

lemma card-of-empty3:

assumes $|A| \leq o \ |\{\}|$

shows $A = \{\}$

using assms

by (simp add: ordIso-iff-ordLeq card-of-empty1 card-of-empty2 ordLeq-Well-order-simp)

lemma card-of-empty-ordIso:

$|\{\}|::\text{a set} = o \ |\{\}|::\text{b set}$

using card-of-ordIso unfolding bij-betw-def inj-on-def by blast

lemma card-of-image:
|f · A| ≤ o |A|

proof (cases A = {}) 
  case False 
  hence f · A ≠ {} by auto 
  thus ?thesis using card-of-ordLeq2[of f · A] by auto 
qed (simp add: card-of-empty)

lemma surj-imp-ordLeq:
  assumes B ⊆ f · A 
  shows |B| ≤ o |A| 
proof − 
  have |B| ≤ o |f · A| using assms card-of-mono1 by auto 
  thus ?thesis using card-of-image ordLeq-transitive by blast 
qed

lemma card-of-singl-ordLeq:
  assumes A ≠ {} 
  shows |{b}| ≤ o |A| 
proof − 
  obtain a where *: a ∈ A using assms by auto 
  let ?h = λ b': b', if b' = b then a else undefined 
  have inj-on ?h {b} ∧ ?h ' {b} ≤ A 
    using * unfolding inj-on-def by auto 
  thus ?thesis unfolding card-of-ordLeq[symmetric] by (intro exI) 
qed

corollary Card-order-singl-ordLeq:
  [Card-order r; Field r ≠ {}] ⇒ |{b}| ≤ o r 
using card-of-singl-ordLeq[of Field r b] 
  card-of-Field-ordIso[of r] ordLeq-ordIso-trans by blast

lemma card-of-Pow: |A| < o |Pow A| 
using card-of-ordLess2[of Pow A A] Cantors-theorem[of A] 
  Pow-not-empty[of A] by auto 

corollary Card-order-Pow:
  Card-order r ⇒ r < o |Pow(Field r)| 
using card-of-Pow-card-of-Field-ordIso ordIso-ordLess-trans ordIso-symmetric by blast

lemma card-of-Plus1: |A| ≤ o |A <+> B| and card-of-Plus2: |B| ≤ o |A <+> B| 
using card-of-ordLeq by force+

corollary Card-order-Plus1:
  Card-order r ⇒ r ≤ o |(Field r) <+> B| 
using card-of-Plus1 card-of-Field-ordIso ordIso-ordLeq-trans ordIso-symmetric by blast
corollary  Card-order-Plus2:
  Card-order r \Rightarrow r \leq o |A <+> (Field r)|
using  card-of-Plus2 card-of-Field-ordIso ordIso-ordLeq-trans ordIso-symmetric by
blast

lemma  card-of-Plus-empty1: |A| = o |A <+> {}|
proof –
  have bij-betw Inl A (A <+> {}) unfolding bij-betw-def inj-on-def by auto
  thus ?thesis using card-of-ordIso by auto
qed

lemma  card-of-Plus-empty2: |A| = o |{} <+> A|
proof –
  have bij-betw Inr A ({} <+> A) unfolding bij-betw-def inj-on-def by auto
  thus ?thesis using card-of-ordIso by auto
qed

lemma  card-of-Plus-commute: |A <+> B| = o |B <+> A|
proof –
  let \( ?f = \lambda c. \text{case } c \text{ of } \text{Inl } a \Rightarrow \text{Inr } a | \text{Inr } b \Rightarrow \text{Inl } b \) 
  have bij-betw \( ?f (A <+> B) (B <+> A) \)
    unfolding bij-betw-def inj-on-def by force
  thus ?thesis using card-of-ordIso by blast
qed

lemma  card-of-Plus-assoc:
  fixes A :: 'a set and B :: 'b set and C :: 'c set
  shows |(A <+> B) <+> C| = o |A <+> B <+> C|
proof –
  define f :: ('a + 'b) + 'c \Rightarrow 'a + 'b + 'c
    where [abs-def]: f k =
      (case k of
        Inl ab \Rightarrow
        (case ab of
          Inl a \Rightarrow Inl a
          | Inr b \Rightarrow Inr (Inl b))
          | Inr c \Rightarrow Inr (Inr c))
    for k
  have A <+> B <+> C \subseteq f \cdot ((A <+> B) <+> C)
proof
  fix x assume x: x \in A <+> B <+> C
  show x \in f \cdot ((A <+> B) <+> C)
  proof (cases x)
    case (Inl a)
    hence a \in A x = f (Inl (Inl a))
      using x unfolding f-def by auto
    thus ?thesis by auto
  next
    case (Inr bc) with x show ?thesis
by (cases bc) (force simp: f-def)+
qed

hence bij-betw f ((A <+> B) <+> C) (A <+> B <+> C)
  unfolding bij-betw-def inj-on-def f-def by fastforce
thus ?thesis using card-of-ordIso by blast
qed

lemma card-of-Plus-mono1:
  assumes |A| ≤o |B|
  shows |A <+> C| ≤o |B <+> C|
proof (unfolding bij-betw-def inj-on-def f-def by fastforce)
  have inj-on g (A <+> C) ∧ g ' (A <+> C) ≤ (B <+> C)
    using f unfolding inj-on-def g-def by force
thus ?thesis using card-of-ordLeq by blast
qed

corollary ordLeq-Plus-mono1:
  assumes r ≤o r'
  shows |(Field r) <+> C| ≤o |(Field r') <+> C|
  using card-of-mono2 card-of-Plus-mono1 by blast

lemma card-of-Plus-mono2:
  assumes |A| ≤o |B|
  shows |C <+> A| ≤o |C <+> B|
  using card-of-Plus-mono1[of A B] card-of-Plus-mono2 by blast

corollary ordLeq-Plus-mono2:
  assumes r ≤o r'
  shows |A <+> (Field r)| ≤o |A <+> (Field r')|
  using card-of-mono2 card-of-Plus-mono2 by blast

lemma card-of-Plus-mono:
  assumes |A| ≤o |B| and |C| ≤o |D|
  shows |A <+> C| ≤o |B <+> D|

corollary ordLeq-Plus-mono:
  assumes r ≤o r' and p ≤o p'
  shows |(Field r) <+> (Field p)| ≤o |(Field r') <+> (Field p')|
  using card-of-mono2[of r r'] card-of-Plus-mono2[of p p'] by blast

lemma card-of-Plus-cong1:
assumes \(|A| = o \|B\|
shows \(|A <+> C| = o \|B <+> C|
using assms by (simp add: ordIso-iff-ordLeq card-of-Plus-mono1)

corollary ordIso-Plus-cong1:
assumes \(r = o \ r'\)
shows \(||(\text{Field } r) <+> C|| = o ||(\text{Field } r') <+> C||\)
using assms card-of-cong card-of-Plus-cong1 by blast

lemma card-of-Plus-cong2:
assumes \(|A| = o \|B\|\) and \(|C| = o \|D|\)
shows \(|A <+> C| = o \|B <+> D|\)
using assms by (simp add: ordIso-iff-ordLeq card-of-Plus-mono2)

corollary ordIso-Plus-cong2:
assumes \(r = o \ r'\) and \(p = o \ p'\)
shows \(||(\text{Field } r) <+> (\text{Field } p)|| = o ||(\text{Field } r') <+> (\text{Field } p')||\)
using assms card-of-cong[of \(r \ r'\)] card-of-cong[of \(p \ p'\)] card-of-Plus-cong by blast

lemma card-of-Un-Plus-ordLeq:
\(|A \cup B| \leq o \|A <+> B|\)
proof -
let \(\lambda c. \text{ if } c \in A \text{ then Inl } c \text{ else Inr } c\)
have inj-on \(?f. \text{ if } (A \cup B) \text{ and } \(?f' . (A \cup B) \leq o \|A <+> B|\)
  unfolding inj-on-def by auto
thus \(?thesis \text{ using card-of-ordLeq by blast}\)
qed

lemma card-of-Times1:
assumes \(A \not= \{\}\)
shows \(|B| \leq o \|B \times A|\)
proof (cases \(B = \{\}\))
case False
  have \(\text{fst } (B \times A) = B\) using assms by auto
thus \(?thesis \text{ using inj-on-iff-surj[of } B \times A\]
  card-of-ordLeq False by blast\)
qed (simp add: card-of-empty)

lemma card-of-Times-commute: \(|A \times B| = o \|B \times A|\)
proof –
  have bij-betw (λ(a,b), (b,a)) (A × B) (B × A)
  unfolding bij-betw-def inj-on-def by auto
thus ?thesis using card-of-ordIso by blast
qed

lemma card-of-Times2:
assumes A ≠ {} shows |B| ≤ o |A × B|
  ordLeq-ordIso-trans by blast

corollary Card-order-Times1:
[Card-order r; B ≠ {}] ⟹ r ≤ o |(Field r) × B|
using card-of-Times1[of B] card-of-Field-ordIso
  ordIso-ordLeq-trans ordIso-symmetric by blast

corollary Card-order-Times2:
[Card-order r; A ≠ {}] ⟹ r ≤ o |A × (Field r)|
using card-of-Times2[of A] card-of-Field-ordIso
  ordIso-ordLeq-trans ordIso-symmetric by blast

lemma card-of-Times3: |A| ≤ o |A × A|
using card-of-Times1[of A]
by(cases A = {}, simp add: card-of-empty)

lemma card-of-Plus-Times-bool: |A <+> A| = o |A × (UNIV::bool set)|
proof –
  let ?f = λc::'a + 'a. case c of Inl a ⇒ (a,True)
   Inr a ⇒ (a, False)
  have bij-betw ?f (A <+> A) (A × (UNIV::bool set))
  proof –
    have ∀c1 c2. ?f c1 = ?f c2 ⟹ c1 = c2
      by (force split: sum.split-asm)
    moreover
    have ∀c. c ∈ A <+> A ⟹ ?f c ∈ A × (UNIV::bool set)
      by (force split: sum.split-asm)
    moreover
    {fix a bl assume (a,bl) ∈ A × (UNIV::bool set)
      hence (a,bl) ∈ ?f ' (A <+> A)
        by (cases bl) (force split: sum.split-asm)+
    }
    ultimately show ?thesis unfolding bij-betw-def inj-on-def by auto
qeda!

lemma card-of-Times-mono1:
assumes |A| ≤ o |B|
shows |A × C| ≤ o |B × C|
proof

obtain \( f \) where \( f : \text{inj-on } f \ A \land f ' \ A \leq B \)
using assms card-of-ordLeq[of \( A \)] by fastforce

define \( g \) where \( g \equiv (\lambda (a,c::'c). (f a,c)) \)

have \( \text{inj-on } g \ (A \times C) \land g ' \ (A \times C) \leq (B \times C) \)
using \( f \) unfolding inj-on-def using \( g \)-def by auto

thus \( \text{thesis} \) using card-of-ordLeq by blast

qed

corollary ordLeq-Times-mono1:
assumes \( r \leq o \ r' \)
shows \( |(\text{Field } r) \times C| \leq o |(\text{Field } r') \times C| \)
using assms card-of-mono2 card-of-Times-mono1 by blast

lemma card-of-Times-mono2:
assumes \( |A| \leq o \ |B| \)
shows \( |C \times A| \leq o \ |C \times B| \)
using assms card-of-Times-mono1[of \( A \ B \ C \)]
by (blast intro: card-of-Times-commute ordIso-ordLeq-trans ordLeq-ordIso-trans)

corollary ordLeq-Times-mono2:
assumes \( r \leq o \ r' \)
shows \( |A \times (\text{Field } r)| \leq o \ |A \times (\text{Field } r')| \)
using assms card-of-mono2 card-of-Times-mono2 by blast

lemma card-of-Sigma-mono1:
assumes \( \forall i \in I. \ |A i| \leq o \ |B i| \)
shows \( |\Sigma i : I. A i| \leq o |\Sigma i : I. B i| \)
proof

have \( \forall i. i \in I \longrightarrow (\exists f. \text{inj-on } f (A i) \land f ' (A i) \leq B i) \)
using assms by (auto simp add: card-of-ordLeq)
with \( \text{choice} \)\( \lambda i f. \ i \in I \longrightarrow \text{inj-on } f (A i) \land f ' (A i) \leq B i \)
obtain \( F \) where \( F : \forall i \in I. \ \text{inj-on } (F i) (A i) \land (F i) ' (A i) \leq B i \)
by atomize-elim (auto intro: choice)

define \( g \) where \( g \equiv (\lambda (i,a::'b). (i,F i a)) \)

have \( \text{inj-on } g \ (\text{Sigma } I \ A) \land g ' \ (\text{Sigma } I \ A) \leq (\text{Sigma } I \ B) \)
using \( F \) unfolding inj-on-def using \( g \)-def by force

thus \( \text{thesis} \) using card-of-ordLeq by blast

qed

lemma card-of-UNION-Sigma:
\( |\bigcup i \in I. A i| \leq o |\Sigma i : I. A i| \)
using Ex-inj-on-UNION-Sigma[of \( A I \)] card-of-ordLeq by blast

lemma card-of-book:
assumes \( a1 \neq a2 \)
shows \( |\text{UNIV::bool set}| = o |\{a1,a2}\| \)
proof

let \( \text{if } = \lambda bl. \text{ if } bl \text{ then } a1 \text{ else } a2 \)
have bij-betw ?f UNIV \{a1,a2\}
proof -
  have \(bl1 \leq bl2\). ?f bl1 = ?f bl2 \implies bl1 = bl2
    using assms by (force split: if-split-asm)
moreover
  have \(bl\). ?f bl \in \{a1,a2\}
    using assms by (force split: if-split-asm)
ultimately show ?thesis unfolding bij-betw-def inj-on-def by force
qed
thus ?thesis using card-of-ordIso by blast
qed

lemma card-of-Plus-Times-aux:
  assumes A2: \(a1 \neq a2 \land \{a1,a2\} \leq A\) and
  LEQ: \(|A| \leq o \|B\|
shows \(|A <+> B| \leq o \|A \times B\|
proof -
  have 1: \(|UNIV::bool set| \leq o \|A|\)
    using A2 card-of-mono1[of \{a1,a2\}] card-of-bool[of a1 a2]
    by (blast intro: ordIso-ordLeq-trans)
  have \(|A <+> B| \leq o \|B <+> B|\)
    using LEQ card-of-Plus-mono1 by blast
moreover have \(|B <+> B| = o \|B \times (UNIV::bool set)|\)
    using card-of-Plus-Times-bool by blast
moreover have \(|B \times (UNIV::bool set)| \leq o \|B \times A|\)
    using 1 by (simp add: card-of-Times-mono2)
moreover have \(|B \times A| = o \|A \times B|\)
    using card-of-Times-commute by blast
ultimately show \(|A <+> B| \leq o \|A \times B|\)
    by (blast intro: ordLeq-transitive dest: ordLeq-ordIso-trans)
qed

lemma card-of-Plus-Times:
  assumes A2: \(a1 \neq a2 \land \{a1,a2\} \leq A\) and
  B2: \(b1 \neq b2 \land \{b1,b2\} \leq B\)
shows \(|A <+> B| \leq o \|A \times B|\)
proof -
  \{assume \(|A| \leq o \|B|\)
    hence ?thesis using assms by (auto simp add: card-of-Plus-Times-aux) \}
  moreover
  \{assume \(|B| \leq o \|A|\)
    hence \(|B <+> A| \leq o \|B \times A|\)
      using assms by (auto simp add: card-of-Plus-Times-aux)
    hence ?thesis
      using card-of-Plus-commute card-of-Times-commute
      ordIso-ordLeq-trans ordLeq-ordIso-trans by blast \}
ultimately show ?thesis
ordLeq-total[of |A|] by blast

qed

lemma card-of-Times-Plus-distrib:
|A × (B <+> C)| =o |A × B <+> A × C| (is |?RHS| =o |?LHS|)
proof
  let ?f = λ(a, bc). case bc of Inl b ⇒ Inl (a, b) | Inr c ⇒ Inr (a, c)
  have bij-betw ?f ?RHS ?LHS unfolding bij-betw-def inj-on-def by force
  thus ?thesis using card-of-ordIso by blast
qed

lemma card-of-ordLeq-finite:
  assumes |A| ≤o |B| and finite B
  shows finite A
  using assms unfolding ordLeq-def
  using embed-inj-on[of |A| |B|] embed-Field[of |A| |B|]

lemma card-of-ordLeq-infinite:
  assumes |A| ≤o |B| and ¬ finite A
  shows ¬ finite B
  using assms card-of-ordLeq-finite by auto

lemma card-of-ordIso-finite:
  assumes |A| =o |B|
  shows finite A = finite B
  using assms unfolding ordIso-def iso-def[abs-def]
  by (auto simp: bij-betw-finite Field-card-of)

lemma card-of-ordIso-finite-Field:
  assumes Card-order r and r =o |A|
  shows finite(Field r) = finite A
  using assms card-of-Field-ordIso card-of-ordIso-finite ordIso-equivalence by blast

30.4 Cardinals versus set operations involving infinite sets

Here we show that, for infinite sets, most set-theoretic constructions do not
increase the cardinality. The cornerstone for this is theorem Card-order-Times-same-infinite,
which states that self-product does not increase cardinality – the proof of
this fact adapts a standard set-theoretic argument, as presented, e.g., in the
proof of theorem 1.5.11 at page 47 in [4]. Then everything else follows fairly
easily.

lemma infinite-iff-card-of-nat:
  ¬ finite A ⇔ ( |UNIV::nat set| ≤o |A| )
  unfolding infinite-iff-countable-subset card-of-ordLeq ..

The next two results correspond to the ZF fact that all infinite cardinals are
limit ordinals:
idea Card-order-infinite-not-under:
assumes CARD: Card-order r and INF: ¬finite (Field r)
sows ¬ (∃a. Field r = under r a)
proof
have 0: Well-order r ∧ wo-rel r ∧ Refl r
  using CARD unfolding wo-rel-def card-order-on-def order-on-defs by auto
fix a assume *: Field r = under r a
show False
proof (cases a ∈ Field r)
  assume Case1: a /∈ Field r
  hence under r a = {} unfolding Field-def under-def by auto
  thus False using INF * by auto
next
  let ?r' = Restr r (underS r a)
  assume Case2: a ∈ Field r
  hence 1: under r a = underS r a ∪ {a} ∧ a /∈ underS r a
    using 0 Refl-under-underS[of r a] underS-notIn[of a r] by blast
  have 2: wo-rel.ofilter r (underS r a) ∧ underS r a < Field r
    using 0 wo-rel.underS-ofilter * 1 Case2 by fast
  hence ?r' < o r using 0 using ofilter-ordLess by blast
  moreover
  have Field ?r' = underS r a ∧ Well-order ?r'
    using 2 0 Field-Restr-ofilter[of r] Well-order-Restr[of r] by blast
  ultimately have |underS r a| < o r using ordLess-Field[of ?r'] by auto
  moreover have |under r a| = o r using * CARD card-of-Field-ordIso[of r] by auto
  ultimately have |underS r a| < o |under r a|
    using ordIso-symmetric ordLess-ordIso-trans by blast
  moreover
    { have ∃f. bij-betw f (under r a) (underS r a)
      using infinite-imp-bij-betw[of Field r a] INF * 1 by auto
      hence |under r a| = o |underS r a| using card-of-ordIso by blast
    }
  ultimately show False using not-ordLess-ordIso ordIso-symmetric by blast
qed
qed

idea infinite-Card-order-limit:
assumes r: Card-order r and ¬finite (Field r)
and a: a ∈ Field r
shows ∃b ∈ Field r. a ≠ b ∧ (a, b) ∈ r
proof
  have Field r ≠ under r a
    using assms Card-order-infinite-not-under by blast
  moreover have under r a ≤ Field r
    using under-Field .
  ultimately obtain b where b: b ∈ Field r ∧ ¬ (b, a) ∈ r
    unfolding under-def by blast
  moreover have ba: b ≠ a
using \( b \) unfolding card-order-on-def well-order-on-def
linear-order-on-def partial-order-on-def preorder-on-def refl-on-def by auto
ultimately have \((a,b) \in r\)
using \( a \) unfolding card-order-on-def well-order-on-def linear-order-on-def
total-on-def by blast
thus \( \exists \)thesis using \( b \) ba by auto
qed

theorem Card-order-Times-same-infinite:
assumes \( CO: \) Card-order \( r \) and \( INF: \neg \)finite\((Field r)\)
shows \(|Field r \times Field r| \leq o r\)
proof
  define \( \phi \) where
  \( \phi \equiv \lambda r::'a rel.\) Card-order \( r \) \( \land \neg \)finite\((Field r)\) \( \land \neg |Field r \times Field r| \leq o r\)
  have temp1: \( \forall r.\) phi \( r \) \( \longrightarrow \) Well-order \( r \)
    unfolding phi-def card-order-on-def by auto
  have Ft: \( \neg (\exists r.\) phi \( r)\)
    proof
      assume \( \exists r.\) phi \( r\)
      hence \( \{r.\) phi \( r\} \neq \{\} \land \{r.\) phi \( r\} \leq \{r.\) Well-order \( r\}\)
      using temp1 by auto
    then obtain \( r \) where 1: phi \( r \) and 2: \( \forall r',\) phi \( r ' \) \( \longrightarrow \) r \( \leq o r' \) and
      3: Card-order \( r \) \( \land \) Well-order \( r \)
    using exists-minim-Well-order[of \{r. phi r\}] temp1 phi-def by blast
    let \( \mathcal{A} = Field r \) let \( \mathcal{A}' = bsqr r \)
    have 4: Well-order \( \mathcal{A}' \) \( \land Field \mathcal{A}' = \mathcal{A} \times \mathcal{A} \land |\mathcal{A}| = o r\)
      using 3 bsqr-Well-order Field-bsqr card-of-Field-ordIso by blast
    have 5: Card-order \(|\mathcal{A} \times \mathcal{A}| \land Well-order \(|\mathcal{A} \times \mathcal{A}|\)
      using card-of-Card-order card-of-Well-order by blast
    have r < o \(|\mathcal{A} \times \mathcal{A}|\)
      using 1 3 5 ordLess-or-ordLeq unfolding phi-def by blast
    moreover have \(|\mathcal{A} \times \mathcal{A}| \leq o \mathcal{A}'\)
      using card-of-least[of \|\mathcal{A} \times \mathcal{A}|] 4 by auto
    ultimately have r < o \mathcal{A}' using ordLess-ordLeq-trans by auto
    then obtain \( f \) where 6: embed \( r \) \( \mathcal{A}' \) \( f \) and 7: \( \neg bij-betw f \) \( \mathcal{A} \) \(|\mathcal{A} \times \mathcal{A}|\)
      unfolding ordLess-def embedS-def[abs-def]
    by (auto simp add: Field-bsqr)
    let \( B = f ' \mathcal{A}\)
    have \|\mathcal{A}| = o \|\mathcal{B}|\)
      using 3 6 embed-ij-on inj-on-imp-bij-betw card-of-ordIso by blast
    hence 8: \( r = o \) \|\mathcal{B}|\) using 4 ordIso-transitive ordIso-symmetric by blast
    have wo-rel.ofilter \( \mathcal{A}' \) \( \mathcal{B}\)
      using 6 embed-Field-ofilter 3 4 by blast
    hence wo-rel.ofilter \( \mathcal{A}' \) \( \mathcal{B} \land \mathcal{B} \neq \mathcal{A} \land \mathcal{A} \land \mathcal{B} \neq Field \) \( \mathcal{A}'\)
      using 7 unfolding bij-betw-def using 6 3 embed-ij-on 4 by auto
    hence temp2: wo-rel.ofilter \( \mathcal{A}' \) \( \mathcal{B} \land \mathcal{B} < \mathcal{A} \times \mathcal{A}\)
      using 4 wo-rel-def[of \( \mathcal{A}'\) wo-rel.ofilter-def[of \( \mathcal{A}'\) \( \mathcal{B}\)] by blast
have \( \neg (\exists a. \text{Field } r = \text{under } r a) \)
using 1 unfolding phi-def using Card-order-infinite-not-under[of r] by auto
then obtain \( A1 \) where temp3: wo-rel.ofilter \( r \ A1 \land A1 < \ ?A \) and 9: \( \ ?B \leq A1 \times A1 \)
using temp2 3 bsqr-ofilter[of \( r \ ?B \)] by blast
hence \( |\ ?B | \leq o | A1 \times A1 | \) using card-of-mono1 by blast
let \( ?r1 = \operatorname{Restr} r A1 \)
have \( ?r1 < o r \) using temp3 ofilter-ordLess 3 by blast
moreover
{have well-order-on \( A1 \) ?r1 using 3 temp3 well-order-on-Restr by blast
hence \( |A1| \leq o \ ?r1 \) using 3 Well-order-Restr card-of-least by blast }
ultimately have 11: \( |A1| < o r \) using ordLeg-ordLess-trans by blast

have \( \neg \text{finite } (\text{Field } r) \) using 1 unfolding phi-def by simp
hence \( \neg \text{finite } ?B \) using 8 3 card-of-ordIso-finite-Field[of \( r \ ?B \)] by blast
hence \( \neg \text{finite } A1 \) using 9 finite-cartesian-product finite-subset by blast
moreover have temp4: \( \text{Field } |A1| = A1 \land \text{Well-order } |A1| \land \text{Card-order } |A1| \)
by (simp add: Field-card-of)
moreover have \( \neg r \leq o | A1 | \)
using temp4 11 3 using not-ordLeq-iff-ordLess by blast
ultimately have \( \neg \text{finite } (\text{Field } |A1| \land \text{Card-order } |A1| \land \neg r \leq o | A1 | \)
by (simp add: card-of-card-order-on)
hence \( |A1 \times A1 | \leq o |A1| \) using temp4 by auto
hence \( r \leq o |A1| \) using 10 ordLeq-transitive by blast
thus False using 11 not-ordLess-ordLeq by auto
qed
thus \(?\thesis\) using assms unfolding phi-def by blast
qed

corollary card-of-Times-same-infinite:
assumes \( \neg \text{finite } A \)
shows \( |A \times A| = o |A| \)
proof –
let \( ?r = |A| \)
have Field \( ?r = A \land \text{Card-order } ?r \)
using Field-card-of card-of-Card-order[of \( A \)] by fastforce
hence \( |A \times A| \leq o |A| \)
using Card-order-Times-same-infinite[of \( ?r \)] assms by auto
thus \(?\thesis\) using card-of-Times3 ordIso-iff-ordLeq by blast
qed

lemma card-of-Times-infinite:
assumes INF: \( \neg \text{finite } A \) and NE: \( B \neq \{\} \) and LEQ: \( |B| \leq o |A| \)
shows \( |A \times B| = o |A| \land |B \times A| = o |A| \)
proof
  have \(|A| \leq |A \times B| \land |A| \leq |B \times A|\)
    using assms by (simp add: card-of-Times1 card-of-Times2)
  moreover
    \{have \(|A \times B| \leq |A \times A| \land |B \times A| \leq |A \times A|\)
      using LEQ card-of-Times-mono1 card-of-Times-mono2 by blast
      moreover have \(|A \times A| = |A|\) using INF card-of-Times-same-infinite by blast
    ultimately have \(|A \times B| \leq |A| \land |B \times A| \leq |A|\)
      using ordLeq-ordIso-trans[of \(|A \times B|\)] ordLeq-ordIso-trans[of \(|B \times A|\)] by auto
  \}
  ultimately show ?thesis by (simp add: ordIso-iff-ordLeq)
qed

corollary Card-order-Times-infinite:
  assumes INF: \(\neg finite (Field r)\) and CARD: Card-order r and
  NE: Field p \(\neq\) \{\} and LEQ: \(p \leq o\) r
  shows \(|\{Field r\} \times (Field p)| = o \land \| (Field p) \times (Field r) | = o\) r
proof
  have \(|Field r \times Field p| = o \| Field r\) \land \| Field p \times Field r | = o \| Field r|\)
    using assms by (simp add: card-of-Times-infinite card-of-mono2)
  thus ?thesis
    using assms card-of-Field-ordIso by (blast intro: ordIso-transitive)
qed

lemma card-of-Sigma-ordLeq-infinite:
  assumes INF: \(\neg finite B\) and
  LEQ-I: \(|I| \leq o \| B\) and LEQ: \(\forall i \in I. \| A \ i| \leq o \| B|\)
  shows \(|SIGMA i : I. \ A \ i| \leq o \| B|\)
proof(cases \(I = \{\}\))
  case False
  have \(|SIGMA i : I. \ A \ i| \leq o \| I \times B|\)
    using card-of-Sigma-mono1[of LEQ] by blast
  moreover have \(|I \times B| = o \| B|\)
    using INF False LEQ-I by (auto simp add: card-of-Times-infinite)
  ultimately show ?thesis using ordLeq-ordIso-trans by blast
qed (simp add: card-of-empty)

lemma card-of-Sigma-ordLeq-infinite-Field:
  assumes INF: \(\neg finite (Field r)\) and r: Card-order r and
  LEQ-I: \(|I| \leq o \ r\) and LEQ: \(\forall i \in I. \| A \ i| \leq o \ r|\)
  shows \(|SIGMA i : I. \ A \ i| \leq o \ r|\)
proof
  let \(?B = Field r\)
  have \(1: r = o \| ?B\) \land \(?B| = o\) r
    using r card-of-Field-ordIso ordIso-symmetric by blast
  hence \(|I| \leq o \| ?B|\) \land \(\forall i \in I. \| A \ i| \leq o \| ?B|\)
    using LEQ-I LEQ ordLeq-ordIso-trans by blast
hence $|\Sigma I : A | \leq o |?B|$ using INF LEQ

card-of-Sigma-ordLeq-infinite by blast

thus $?thesis$ using 1 ordLeq-ordIso-trans by blast

qed

lemma card-of-Times-ordLeq-infinite-Field:

$\neg finite (Field r) ; |A| \leq o r ; |B| \leq o r ; Card-order r \implies |A \times B| \leq o r$

by (simp add: card-of-Sigma-ordLeq-infinite-Field)

lemma card-of-Times-infinite-simps:

$\neg finite A ; B \neq {} ; |B| \leq o |A| \implies |A \times B| = o |A|$

$\neg finite A ; B \neq {} ; |B| \leq o |A| \implies |B \times A| = o |A|$

$\neg finite A ; B \neq {} ; |B| \leq o |A| \implies |A| = o |B \times A|$

by (auto simp add: card-of-Times-infinite ordIso-symmetric)

lemma card-of-UNION-ordLeq-infinite:

assumes INF: $\neg finite B$ and LEQ-I: $|I| \leq o r$ and LEQ: $\forall i \in I. |A i| \leq o r$

shows $|\bigcup i \in I. A i| \leq o r$

proof (cases $I = \{\}$)

case False

have $|\bigcup i \in I. A i| \leq o |\Sigma I : A i|$

using card-of-UNION-Sigma by blast

moreover have $|\Sigma I : A i| \leq o |B|$

using assms card-of-Sigma-ordLeq-infinite by blast

ultimately show $?thesis$ using ordLeq-transitive by blast

qed (simp add: card-of-empty)

corollary card-of-UNION-ordLeq-infinite-Field:

assumes INF: $\neg finite (Field r)$ and r: Card-order r and

LEQ-I: $|I| \leq o r$ and LEQ: $\forall i \in I. |A i| \leq o r$

shows $|\bigcup i \in I. A i| \leq o r$

proof (cases $B = {}$)

let $?B = Field r$

have $1 : r = o |?B| \land |?B| = o r$

using r card-of-Field-ordIso ordIso-symmetric by blast

hence $|1| \leq o |?B| \forall i \in I. |A i| \leq o |?B|$

using LEQ-I LEQ ordLeq-ordIso-trans by blast+

hence $|\bigcup i \in I. A i| \leq o |?B|$ using INF LEQ

card-of-UNION-ordLeq-infinite by blast

thus $?thesis$ using 1 ordLeq-ordIso-trans by blast

qed

lemma card-of-Plus-infinite1:

assumes INF: $\neg finite A$ and LEQ: $|B| \leq o |A|$

shows $|A <+> B| = o |A|$

proof (cases $B = {}$)

case True

then show $?thesis$
by (simp add: card-of-Plus-empty1 card-of-Plus-empty2 ordIso-symmetric)
next
case False
let ?Inl = Inl:\'a \Rightarrow 'a + 'b let ?Inr = Inr:\'b \Rightarrow 'a + 'b
assume *: B ≠ \{\}
then obtain b1 where 1: b1 ∈ B by blast
show ?thesis
proof (cases B = \{b1\})
  case True
  have 2: bij-betw ?Inl A ((?Inl \ A)) unfolding bij-betw-def inj-on-def by auto
  hence 3: ~finite (?Inl \ A)
  using INF bij-betw-finite[of ?Inl A] by blast
  let ?A' = ?Inl \ A ∪ {?Inr b1}
  obtain g where bij-betw g (?Inl \ A) ?A'
    using 3 infinite-imp-bij-betw2[of ?Inl A] by auto
  moreover have ?A' = A <+> B using True by blast
  ultimately have bij-betw g (?Inl \ A) (A <+> B) by simp
  hence bij-betw (g o ?Inl) A (A <+> B)
    using 2 by (auto simp add: bij-betw-trans)
  thus ?thesis using card-of-ordIso ordIso-symmetric by blast
next
case False
with * 1 obtain b2 where 3: b1 ≠ b2 ∧ \{b1,b2\} ≤ B by fastforce
obtain f where inj-on f B ∧ f \ B ≤ A
  using LEQ card-of-ordLeq[of B] by fastforce
with 3 have f b1 ≠ f b2 ∧ \{f b1, f b2\} ≤ A
  unfolding inj-on-def by auto
with 3 have |A <+> B| ≤ o |A × B|
  by (auto simp add: card-of-Plus-Times)
moreover have |A × B| = o |A|
  using assms * by (simp add: card-of-Plus-infinite2)
ultimately have |A <+> B| ≤ o |A| using ordLeq-ordIso-trans by blast
thus ?thesis using card-of-Plus1 ordIso-iff-ordLeq by blast
qed
qed

lemma card-of-Plus-infinite2:
  assumes INF: ~finite A and LEQ: |B| ≤ o |A|
  shows |B <+> A| = o |A|
  using assms card-of-Plus-commute card-of-Plus-infinite1
  ordIso-equivalence by blast

lemma card-of-Plus-infinite:
  assumes INF: ~finite A and LEQ: |B| ≤ o |A|
  shows |A <+> B| = o |A| ∧ |B <+> A| = o |A|
  using assms by (auto simp: card-of-Plus-infinite1 card-of-Plus-infinite2)

corollary Card-order-Plus-infinite:
assumes \( \text{INF: } \neg \text{finite}(\text{Field } r) \) and \( \text{CARD: Card-order } r \) and
\( LEQ: \ p \leq o \ r \)
snews \( | (\text{Field } r) \leftrightarrow (\text{Field } p) | = o \ r \wedge | (\text{Field } p) \leftrightarrow (\text{Field } r) | = o \ r \)
proof
- have \( | Field r \leftrightarrow Field p | = o | Field r | \wedge | Field p \leftrightarrow Field r | = o | Field r | \)
  using assms by (simp add: card-of-Plus-infinite card-of-mono2)
thus ?thesis
  using assms card-of-Field-ordIso by (blast intro: ordIso-transitive)
qed

30.5 The cardinal \( \omega \) and the finite cardinals

The cardinal \( \omega \), of natural numbers, shall be the standard non-strict order relation on \text{nat}, that we abbreviate by \text{natLeq}. The finite cardinals shall be the restrictions of these relations to the numbers smaller than fixed numbers \( n \), that we abbreviate by \text{natLeq-on } n.

definition \( \text{natLeq} :: (\text{nat} \times \text{nat}) \) set \( \equiv \{(x,y). \ x \leq y\} \)
definition \( \text{natLess} :: (\text{nat} \times \text{nat}) \) set \( \equiv \{(x,y). \ x < y\} \)

abbreviation \( \text{natLeq-on} :: \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \) set
where \( \text{natLeq-on } n \equiv \{(x,y). \ x < n \wedge y < n \wedge x \leq y\} \)

lemma infinite-cartesian-product:
  assumes \( \neg \text{finite } A \) \( \neg \text{finite } B \)
  shows \( \neg \text{finite } (A \times B) \)
  using assms finite-cartesian-productD2 by auto

30.5.1 First as well-orders

lemma \( \text{Field-natLeq} \): Field \text{natLeq} = (\text{UNIV::nat} set)
  by (unfold Field-def natLeq-def, auto)

lemma \( \text{natLeq-Refl} \): \text{Refl } \text{natLeq}
  unfolding refl-on-def Field-def natLeq-def by auto

lemma \( \text{natLeq-trans} \): \text{trans } \text{natLeq}
  unfolding trans-def natLeq-def by auto

lemma \( \text{natLeq-Preorder} \): \text{Preorder } \text{natLeq}
  unfolding preorder-on-def
  by (auto simp add: natLeq-Refl natLeq-trans)

lemma \( \text{natLeq-antisym} \): \text{antisym } \text{natLeq}
  unfolding antisym-def natLeq-def by auto

lemma \( \text{natLeq-Partial-order} \): \text{Partial-order } \text{natLeq}
  unfolding partial-order-on-def
by (auto simp add: natLeq-Preorder natLeq-antisym)

lemma natLeq-Total: Total natLeq
  unfolding total-on-def natLeq-def by auto

lemma natLeq-Linear-order: Linear-order natLeq
  unfolding linear-order-on-def
  by (auto simp add: natLeq-Partial-order natLeq-Total)

lemma natLeq-natLess-Id: natLess = natLeq − Id
  unfolding natLeq-def natLess-def by auto

lemma natLeq-Well-order: Well-order natLeq
  unfolding well-order-on-def
  using natLeq-Linear-order wf-less natLeq-natLess-Id natLeq-def natLess-def by auto

lemma Field-natLeq-on: Field (natLeq-on n) = {x. x < n}
  unfolding Field-def by auto

lemma natLeq-underS-less: underS natLeq n = {x. x < n}
  unfolding underS-def natLeq-def by auto

lemma Restr-natLeq: Restr natLeq {x. x < n} = natLeq-on n
  unfolding natLeq-def by force

lemma Restr-natLeq2: Restr natLeq (underS natLeq n) = natLeq-on n
  by (auto simp add: Restr-natLeq natLeq-underS-less)

lemma natLeq-on-Well-order: Well-order(natLeq-on n)
  using Restr-natLeq[of n] natLeq-Well-order
  Well-order-Restr[of natLeq {x. x < n}] by auto

corollary natLeq-on-well-order-on: well-order-on {x. x < n} (natLeq-on n)
  using natLeq-on-Well-order Field-natLeq-on by auto

lemma natLeq-on-wo-rel: wo-rel(natLeq-on n)
  unfolding wo-rel-def using natLeq-on-Well-order .

30.5.2 Then as cardinals

lemma natLeq-Card-order: Card-order natLeq
  proof
    have natLeq-on n <o \{UNIV::nat set\} for n
    proof
      have finite(Field (natLeq-on n)) by (auto simp: Field-def)
      moreover have ~finite(UNIV::nat set) by auto
      ultimately show ?thesis
corollary card-of-Field-natLeq:
  \( |\text{Field natLeq}| = \circ \text{natLeq} \)
  using Field-natLeq natLeq-Card-order Card-order-iff-ordIso-card-of[of natLeq]
  ordIso-symmetric[of natLeq] by blast

corollary card-of-nat:
  \( |\text{UNIV::nat set}| = \circ \text{natLeq} \)
  using Field-natLeq card-of-Field-natLeq by auto

corollary infinite-iff-natLeq-ordLeq:
  \( \neg \text{finite } A = ( \text{natLeq} \leq \circ |A| ) \)
  ordIso-ordLeq-trans ordLeq-ordIso-trans ordIso-symmetric by blast

corollary finite-iff-ordLess-natLeq:
  \( \text{finite } A = ( |A| < \circ \text{natLeq}) \)
  using infinite-iff-natLeq-ordLeq not-ordLeq-iff-ordLess
  card-of-Well-order natLeq-Well-order by blast

30.6 The successor of a cardinal

First we define \( \text{isCardSuc } r r' \), the notion of \( r' \) being a successor cardinal of \( r \). Although the definition does not require \( r \) to be a cardinal, only this case will be meaningful.

definition isCardSuc :: 'a rel ⇒ 'a set rel ⇒ bool
  where isCardSuc r r' ≡
    Card-order r' ∧ r < o r' ∧
    (∀ (r'': 'a set rel). Card-order r'' ∧ r < o r'' → r' ≤ o r'')

Now we introduce the cardinal-successor operator \( \text{cardSuc} \), by picking some cardinal-order relation fulfilling \( \text{isCardSuc} \). Again, the picked item shall be proved unique up to order-isomorphism.

definition cardSuc :: 'a rel ⇒ 'a set rel
  where cardSuc r ≡ SOME r'. isCardSuc r r'

lemma exists-minim-Card-order:
  \[ R ≠ \{\}; ∀ r ∈ R. Card-order r \] ⇒ ∃ r ∈ R. ∀ r' ∈ R. r ≤ o r'
unfolding card-order-on-def using exists-minim-Well-order by blast

lemma exists-isCardSuc:
assumes Card-order r
shows ∃r′. isCardSuc r r′
proof -
  let ?R = {{r′::'a set rel}. Card-order r′ ∧ r <o r′}
  have |Pow(Field r)| ∈ ?R ∧ (∀r ∈ ?R. Card-order r) using assms
      by (simp add: card-of-Card-order Card-order-Pow)
  then obtain r where r ∈ ?R ∧ (∀r′ ∈ ?R. r ≤o r′)
      using exists-minim-Card-order[of ?R] by blast
  thus ?thesis unfolding isCardSuc-def by auto
qed

lemma cardSuc-isCardSuc:
assumes Card-order r
shows isCardSuc r (cardSuc r)
unfolding cardSuc-def using assms
by (simp add: exists-isCardSuc someI-ex)

lemma cardSuc-Card-order:
  Card-order r ⇒ Card-order(cardSuc r)
using cardSuc-isCardSuc unfolding isCardSuc-def by blast

lemma cardSuc-greater:
  Card-order r ⇒ r <o cardSuc r
using cardSuc-isCardSuc unfolding isCardSuc-def by blast

lemma cardSuc-ordLeq:
  Card-order r ⇒ r ≤o cardSuc r
using cardSuc-greater ordLeq-iff-ordLess-or-ordIso by blast

The minimality property of cardSuc originally present in its definition is
local to the type 'a set rel, i.e., that of cardSuc r:

lemma cardSuc-least-aux:
  [Card-order (r::'a rel); Card-order (r′::'a set rel); r <o r′] ⇒ cardSuc r ≤o r′
using cardSuc-isCardSuc unfolding isCardSuc-def by blast

But from this we can infer general minimality:

lemma cardSuc-least:
assumes CARD: Card-order r and CARD′: Card-order r′ and LESS: r <o r′
shows cardSuc r ≤o r′
proof -
  let ?p = cardSuc r
  have 0: Well-order ?p ∧ Well-order r′
      using assms cardSuc-Card-order unfolding card-order-on-def by blast
  { assume r′ <o ?p
    then obtain r″ where 1: Field r″ < Field ?p and 2: r′ =o r″ ∧ r″ <o ?p
      using internalize-ordLess[of r′ ?p] by blast
  }
have Card-order \( r' \) using CARD' Card-order-ordIso2 2 by blast
moreover have \( r < o r' \) using LESS 2 ordLess-ordIso-trans by blast
ultimately have \( ?p \leq o r' \) using cardSuc-least-aux CARD by blast
hence False using 2 not-ordLess-ordLeq by blast
}\)
thus \( \)thesis using 0 ordLess-or-ordLeq by blast
qed

lemma \( \)cardSuc-ordLess-ordLeq
\{ assumes CARD: Card-order \( r \) and CARD': Card-order \( r' \)
shows \( (r < o r') = (\text{cardSuc } r \leq o r') \)
proof
\begin{itemize}
  \item have Well-order \( r \) \( \land \) Well-order \( r' \)
    using assms unfolding card-order-on-def by auto
  \item moreover have Well-order(\( \text{cardSuc } r \) )
    using assms cardSuc-Card-order card-order-on-def by blast
  \item ultimately show \( \)thesis
    using assms cardSuc-ordLess-ordLeq by (blast dest: not-ordLeq-iff-ordLess)
\end{itemize}
qed (auto simp add: assms cardSuc-least)

lemma \( \)cardSuc-ordLeq-ordLess
\{ assumes CARD: Card-order \( r \) and CARD': Card-order \( r' \)
shows \( (r' < o \text{cardSuc } r) = (r' \leq o r) \)
proof
\{ have 0: Well-order \( r \) \( \land \) Well-order \( r' \) \( \land \) Well-order(\( \text{cardSuc } r \) ) \( \land \) Well-order(\( \text{cardSuc } r' \) )
  using assms by (simp add: card-order-on-well-order-on cardSuc-Card-order)
  thus \( \)thesis
  using ordIso-iff-ordLeq[of \( r \) \( r' \) ] ordIso-iff-ordLeq
  using cardSuc-mono-ordLeq[of \( r \) \( r' \) ] cardSuc-mono-ordLeq[of \( r' \) \( r \) ] assms by blast
\}
qed

lemma \( \)card-of-cardSuc-finite

finite(Field(cardSuc |A| )) = finite A

proof
  assume *: finite (Field (cardSuc |A| ))
  have θ: |Field(cardSuc |A| )| = o cardSuc |A|
    using card-of-Card-order cardSuc-Card-order card-of-Field-ordIso by blast
  hence |A| ≤ o |Field(cardSuc |A| )|
  thus finite A using * card-of-ordLeq-finite by blast
next
  assume finite A
  then have finite ( Field |Pow A| ) unfolding Field-card-of by simp
  moreover
  have cardSuc |A| ≤ o |Pow A|
    by (rule iffD1[OF cardSuc-ordLess-ordLeq card-of-Pow]) (simp-all add: card-of-Card-order)
  ultimately show finite (Field (cardSuc |A| ))
    by (blast intro: card-of-ordLeq-finite card-of-mono2)
qed

lemma cardSuc-finite:
  assumes Card-order r
  shows finite (Field (cardSuc r)) = finite (Field r)
proof −
  let ?A = Field r
  have |?A| = o r using assms by (simp add: card-of-Field-ordIso)
  hence cardSuc |?A| = o cardSuc r using assms
    by (simp add: card-of-Card-order cardSuc-invar-ordIso)
  moreover have |Field (cardSuc |?A| )| = o cardSuc |?A|
    by (simp add: card-of-card-order-on Field-card-of card-of-Field-ordIso card-Suc-Card-order)
  moreover
  { have |Field (cardSuc r) | = o cardSuc r
      using assms by (simp add: card-of-Field-ordIso cardSuc-Card-order)
      hence cardSuc r = o |Field (cardSuc r) |
        using ordIso-symmetric by blast
  }
  ultimately have |Field (cardSuc |?A| )| = o |Field (cardSuc r) |
    using ordIso-transitive by blast
  hence finite (Field (cardSuc |?A| ) ) = finite (Field (cardSuc r) )
    using card-of-ordIso-finite by blast
  thus ?thesis by (simp only: card-of-cardSuc-finite)
qed

lemma Field-cardSuc-not-empty:
  assumes Card-order r
  shows Field (cardSuc r) ≠ {}
proof
  assume Field(cardSuc r) = {} 
  then have |Field(cardSuc r)| ≤ o r using assms Card-order-empty[of r] by auto
then have cardSuc r ≤ o r using assms card-of-Field-ordIso
    cardSuc-Card-order ordIso-symmetric ordIso-ordLeq-trans by blast
then show False using cardSuc-greater not-ordLess-ordLeq assms by blast
qed

typedef 'a suc = Field (cardSuc |UNIV :: 'a set|)
    using Field-cardSuc-not-empty card-of-Card-order by blast

definition card-suc :: 'a rel ⇒ 'a suc rel where
card-suc ≡ λ. map-prod Abs-suc Abs-suc ' cardSuc |UNIV :: 'a set|

lemma Field-card-suc: Field (card-suc r) = UNIV
proof –
    let ?r = cardSuc |UNIV|
    let ?ar = λx. Abs-suc (Rep-suc x)
    have 1: ∀ P. (∀ x∈Field ?r. P x) = (∀ x. P (Rep-suc x)) using Rep-suc-induct
        Rep-suc by blast
    have 2: ∀ P. (∃ x∈Field ?r. P x) = (∃ x. P (Rep-suc x)) using Rep-suc-cases
        Rep-suc by blast
    have 3: ∀ A a b. (a, b) ∈ A → (Abs-suc a, Abs-suc b) ∈ map-prod Abs-suc Abs-suc ' A unfolding map-prod-def by auto
        unfolding Field-def Range.simps Domain.simps Un-iff by blast
    then have ∀ x. (∃ b. (Rep-suc x, Rep-suc b) ∈ ?r) ∨ (∃ a. (Rep-suc a, Rep-suc x) ∈ ?r) unfolding 1 2 .
        then have ∀ x. (∃ b. (?ar x, ?ar b) ∈ map-prod Abs-suc Abs-suc ' ?r) ∨ (∃ a. (?ar a, ?ar x) ∈ map-prod Abs-suc Abs-suc ' ?r) using 3 by fast
        then have ∀ x. (∃ b. (x, b) ∈ card-suc r) ∨ (∃ a. (a, x) ∈ card-suc r) unfolding card-suc-def Rep-suc-inverse .
        then show ?thesis unfolding Field-def Domain.simps Range.simps set-eq-iff
qed

lemma card-suc-alt: card-suc r = dir-image (cardSuc |UNIV :: 'a set| ) Abs-suc
    unfolding card-suc-def dir-image-def by auto

lemma cardSuc-Well-order: Card-order r ⇒ Well-order(cardSuc r)
    using cardSuc-Card-order unfolding card-order-on-def by blast

lemma cardSuc-ordIso-card-suc:
    assumes card-order r
    shows cardSuc r = o card-suc (r :: 'a rel)
proof –
    have cardSuc (r :: 'a rel) = o cardSuc |UNIV :: 'a set|
        using cardSuc-invar-ordIso THEN iffD2, OF - card-of-Card-order card-of-unique[OF assms] assms
        by (simp add: card-order-on-Card-order)
    also have cardSuc |UNIV :: 'a set| = o card-suc (r :: 'a rel)
        unfolding card-suc-alt
by (rule dir-image-ordIso) (simp-all add: inj-on-def Abs-suc-inject cardSuc-Well-order card-of-Card-order)
finally show ?thesis.
qed

lemma Card-order-card-suc: card-order r \implies Card-order (card-suc r)
  using cardSuc-Card-order[THEN Card-order-ordIso2[OF cardSuc-ordIso-card-suc]]
card-order-on-Card-order by blast

lemma card-order-card-suc: card-order r \implies card-order (card-suc r)
  using Card-order-card-suc arg-cong2[OF Field-card-suc refl, of card-order-on] by blast

lemma card-suc-greater: card-order r \implies r <o card-suc r
  using ordLess-ordIso-trans[OF cardSuc-greater cardSuc-ordIso-card-suc]
card-order-on-Card-order by blast

lemma card-of-Plus-ordLess-infinite:
  assumes INF: \neg finite C and LESS1: |A| <o |C| and LESS2: |B| <o |C|
  shows |A <+o B| <o |C|
proof(cases A = {} \vee B = {})
case True
  hence |A| =o |A <+o B| \vee |B| =o |A <+o B|
  using card-of-Plus-empty1 card-of-Plus-empty2 by blast
  hence |A <+o B| =o |A| \vee |A <+o B| =o |B|
  using ordIso-symmetric[of |A|] ordIso-symmetric[of |B|] by blast
  thus ?thesis using LESS1 LESS2
    ordIso-ordLess-trans[of |A <+o B| |A|]
    ordIso-ordLess-trans[of |A <+o B| |B|] by blast
next
case False
  have False if |C| \leq o |A <+o B|
  proof –
    have \$: \neg finite A \vee \neg finite B
      using that INF card-of-ordLeq-finite finite-Plus by blast
    consider |A| \leq o |B| \wedge |B| \leq o |A|
    using ordLeg-total [OF card-of-Well-order card-of-Well-order] by blast
    then show False
  proof cases
    case 1
      hence \neg finite B using \$ card-of-ordLeq-finite by blast
      hence |A <+o B| =o |B| using False 1
        by (auto simp add: card-of-Plus-infinite)
      thus False using LESS2 not-ordLess-ordLeg that ordLeg-ordIso-trans by blast
    next
case 2
      hence \neg finite A using \$ card-of-ordLeq-finite by blast
      hence |A <+o B| =o |A| using False 2
        by (auto simp add: card-of-Plus-infinite)
thus False using LESS1 not-ordLess-ordLeq that ordLeq-ordIso-trans by blast qed

lemma card-of-Plus-ordLess-infinite-Field:
assumes INF: ¬finite (Field r) and r: Card-order r and
LESS1: |A| < o r and LESS2: |B| < o r
shows |A <+> B| < o r
proof –
  let ?r′ = cardSuc r
  have 1: r =o |?r′| ∧ |?r′| =o r using assms
       by (simp add: cardSuc-Card-order cardSuc-finite)
  hence |A| < o |?r′| ∧ |B| < o |?r′| using LESS1 LESS2 ordLess-ordIso-trans by blast
  ultimately have |A <+> B| < o |?r′| using INF
       card-of-Plus-ordLess-infinite by blast
thus ?thesis using 1 ordLess-ordIso-trans by blast qed

lemma card-of-Plus-ordLeq-infinite-Field:
assumes r: ¬finite (Field r) and A: |A| ≤ o r and B: |B| ≤ o r
and c: Card-order r
shows |A <+> B| ≤ o r
proof –
  let ?r′ = cardSuc r
  have Card-order ?r′ ∧ ¬finite (Field ?r′) using assms
       by (simp add: cardSuc-Card-order cardSuc-finite)
  moreover have |A| < o ?r′ and |B| < o ?r′ using A B c
       by (auto simp: card-of-card-order-on Field-card-of cardSuc-ordLeq-ordLess)
  ultimately have |A <+> B| < o ?r′ using card-of-Plus-ordLess-infinite-Field by blast
thus ?thesis using c r
       by (simp add: card-of-card-order-on Field-card-of cardSuc-ordLeq-ordLess) qed

lemma card-of-Un-ordLeq-infinite-Field:
assumes C: ¬finite (Field r) and A: |A| ≤ o r and B: |B| ≤ o r
and Card-order r
shows |A Un B| ≤ o r
using assms card-of-Plus-ordLeq-infinite-Field card-of-Un-Plus-ordLeq
ordLeq-transitive by fast

lemma card-of-Un-ordLess-infinite:
assumes INF: ¬finite C and
LESS1: |A| < o |C| and LESS2: |B| < o |C|

shows $|A \cup B| < o |C|$

lemma card-of-Un-ordLess-infinite-Field:
assumes INF: ~finite (Field r) and r: Card-order r and
LESS1: |A| < o r and LESS2: |B| < o r
shows |A Un B| < o r
proof –
let ?C = Field r
have 1: r = o |?C| ∧ |?C| = o r using r card-of-Field-ordIso ordIso-symmetric by blast
hence |A| < o |?C| |B| < o |?C|
using LESS1 LESS2 ordLess-ordIso-trans by blast+
hence |A Un B| < o |?C| using INF
card-of-Un-ordLess-infinite by blast
thus ?thesis using 1 ordLess-ordIso-trans by blastqed

30.7 Regular cardinals
definition cofinal where
cofinal A r ≡ ∀ a ∈ Field r. ∃ b ∈ A. a ≠ b ∧ (a,b) ∈ r
definition regularCard where
regularCard r ≡ ∀ K. K ≤ Field r ∧ cofinal K r → |K| = o r
definition relChain where
relChain r As ≡ ∀ i j. (i,j) ∈ r → As i ≤ As j
lemma regularCard-UNION:
assumes r: Card-order r regularCard r
and As: relChain r As
and Bsub: B ≤ (⋃ i ∈ Field r. As i)
and cardB: |B| < o r
shows ∃ i ∈ Field r. B ≤ As i
proof –
let ?phi = λb j. j ∈ Field r ∧ b ∈ As j
have ∀ b∈B. ∃ j. ?phi b j using Bsub by blast
then obtain f where f: λb. b ∈ B → ?phi b (f b)
  using bchoice[of B ?phi] by blast
let ?K = f : B
{assume 1: ∃ i. i ∈ Field r → ¬ B ≤ As i
have 2: cofinal ?K r
  unfolding cofinal-def
proof (intro strip)
  fix i assume i: i ∈ Field r
  with 1 obtain b where b: b ∈ B ∧ b ∉ As i by blast
  hence i ≠ f b ∧ ¬ (f b,i) ∈ r}
using As f unfolding relChain-def by auto
hence i ≠ f b ∧ (i, f b) ∈ r using r
unfolding card-order-on-def well-order-on-def linear-order-on-def total-on-def using i f b by auto
with b show ∃ b ∈ f'B. i ≠ b ∧ (i, b) ∈ r using blast
qed
moreover have ?K ≤ Field r using f by blast
ultimately have |?K| = o r using r unfolding regularCard-def by blast
moreover
have |?K| < o r using cardB ordLeq-ordLess-trans card-of-image by blast
ultimately have False using not-ordLess-ordIso by blast
}
thus ?thesis by blast
qed

lemma infinite-cardSuc-regularCard:
assumes r-inf: ¬finite (Field r) and r-card: Card-order r
shows regularCard (cardSuc r)
proof –
let ?r' = cardSuc r
have r': Card-order ?r' ∨ p. Card-order p → (p ≤ o r) = (p < o ?r')
  using r-card by (auto simp: cardSuc-Card-order cardSuc-ordLeq-ordLess)
show ?thesis
  unfolding regularCard-def proof auto
  fix K assume 1: K ≤ Field ?r' and 2: cofinal K ?r'
hence |K| ≤ o |Field ?r'| by (simp only: card-of-mono1)
also have 22: |Field ?r'| = o ?r'
  using r' by (simp add: card-of-Field-ordIso[of ?r'])
finally have |K| ≤ o ?r'.
moreover
{ let ?L = ⋃ j ∈ K. underS ?r' j
  let ?J = Field r
  have rJ: r = o |?J|
    using r-card card-of-Field-ordIso ordIso-symmetric by blast
  assume |K| < o ?r'
hence |K| ≤ o r using r' card-of-Card-order[of K] by blast
hence |K| ≤ o |?J| using rJ ordLeq-ordIso-trans by blast
moreover
{ have ∀ j ∈ K. |underS ?r' j| < o ?r'
    using r' 1 by (auto simp: card-of-underS)
  hence ∀ j ∈ K. |underS ?r' j| ≤ o r
    using r' card-of-Card-order by blast
  hence ∀ j ∈ K. |underS ?r' j| ≤ o |?J|
    using rJ ordLeq-ordIso-trans by blast
  }
ultimately have |?L| ≤ o |?J|
  using r-inf card-of-UNION-ordLeq-infinite by blast
hence |?L| ≤ o r using rJ ordIso-symmetric ordLeq-ordIso-trans by blast
hence |?L| < o ?r' using r' card-of-Card-order by blast
}
moreover

\{ 
  have Field ?r' \leq |?L|
    using 2 unfolding underS-def cofinal-def by auto
  hence |Field ?r'| \leq o |?L| by (simp add: card-of-mono1)
  hence |?r'| \leq o |?L| by auto

  using 2 ordIso-ordLeq-trans ordIso-symmetric by blast
\}

ultimately have |?L| < o |?L| using ordLess-ordLeq-trans by blast

hence False using ordLess-irreflexive by blast

ultimately show |K| = o ?r' using ordLess-ordLeq-trans by blast

qed

proof

- let ?r' = cardSuc r

  have Card-order ?r' \land |B| < o ?r'
    using r cardB cardSuc-ordLeq-ordLess cardSuc-Card-order
    card-of-Card-order by blast

  moreover have regularCard ?r'
    using 2 assms by (simp add: infinite-cardSuc-regularCard)

  ultimately show ?thesis
    using As Bsub cardB regularCard-UNION by blast

qed

30.8 Others

lemma cardSuc-UNION:
  assumes r: Card-order r and -finite (Field r)
  and As: relChain (cardSuc r) As
  and Bsub: B \leq (\bigcup i \in Field (cardSuc r). As i)
  and cardB: |B| \leq o r
  shows \exists i \in Field (cardSuc r). B \leq As i

proof -

  let ?r' = cardSuc r

  have Card-order ?r' \land |B| < o ?r'
    using r cardB cardSuc-ordLeq-ordLess cardSuc-Card-order
    card-of-Card-order by blast

  moreover have regularCard ?r'
    using 2 assms by (simp add: infinite-cardSuc-regularCard)

  ultimately show ?thesis
    using As Bsub cardB regularCard-UNION by blast

qed

lemma card-of-Func-Times:
  \{|Func (A \times B) C| = o |Func A (Func B C)|\}

  unfolding card-of-ordIso[symmetric]

  using bij-betw-curr by blast

lemma card-of-Pow-Func:
  \{|Pow A| = o |Func A (UNIV::bool set)|\}

proof -

  define F where \[ abs-def \]: F A' a =
    (if a \in A then (if a \in A' then True else False) else undefined) for A' a

  have bij-betw F (Pow A) (Func A (UNIV::bool set))
    unfolding bij-betw-def inj-on-def proof (intro ballI impI conjI)

  fix A1 A2 assume A1 \in Pow A A2 \in Pow A F A1 = F A2
  thus A1 = A2 unfolding f-def Pow-def fun-eq-iff by (auto split: if-split-asm)
next
show \( F \cdot \text{Pow} \ A = \text{Func} \ A \ \text{UNIV} \)
proof safe
fix \( f \) assume \( f \in \text{Func} \ A \ (\text{UNIV}::\text{bool set}) \)
show \( f \in F \cdot \text{Pow} \ A \)
unfolding image_iff
proof
show \( f = F \ \{ a \in A. \ f a = \text{True} \} \)
using \( f \) unfolding Func_def F_def by force
qed auto
qed (unfold Func_def F_def, auto)
qed
thus \( \text{?thesis} \)
unfolding card-of-ordIso \[ \text{symmetric} \]
by blast
qed

lemma card-of-Func-UNIV:
\[ |\text{Func} \ (\text{UNIV}::'a set) \ (B::'b set)| = o \ [(f::'a \Rightarrow 'b. \text{range } f \subseteq B)] \]
proof
let \( ?F = \lambda f \ (a::'a). \ ((f a)::'b) \)
have bij_betw ?F \( \{ f. \text{range } f \subseteq B \} \) (Func UNIV B)
unfolding bij_betw_def inj_on_def
proof safe
fix \( h :: 'a \Rightarrow 'b \) assume \( h \in \text{Func} \ \text{UNIV} \ B \)
then obtain \( f \) where \( f \) : \( \forall a. \ h a = f a \) by blast
hence \( \text{range } f \subseteq B \) using \( h \) unfolding Func_def by auto
thus \( h \in (\lambda f \ a a \Rightarrow \text{range } f \subseteq B) \) using \( f \) by auto
qed (unfold Func_def fun_eq_iff, auto)
then show \( \text{?thesis} \)
by (blast intro: ordIso_symmetric card_of_ordIsoI)
qed

lemma Func-Times-Range:
\[ |\text{Func} \ A (B \times C)| = o |\text{Func} \ A (B \times C)| \ (is |?LHS| = o |?RHS|) \]
proof
let \( ?F = \lambda f. \ (\lambda x. \text{if } x \in A \text{ then } \text{fst} \ (fg x) \text{ else undefined,} \)
\( \lambda x. \text{if } x \in A \text{ then } \text{snd} \ (fg x) \text{ else undefined} \)
let \( ?G = \lambda(f, g) \ x. \text{if } x \in A \text{ then } (f x, g x) \text{ else undefined} \)
have bij_betw ?F ?LHS ?RHS unfolding bij_betw_def inj_on_def
proof (intro conjI implI ballI equalityI subsetI)
fix \( f \ g \) assume \( *: f \in \text{Func} \ (B \times C) \ g \in \text{Func} \ (B \times C) \ ?F \ f = \ ?F \ g \)
show \( f = g \)
proof
fix \( x \) from \( * \) have \( \text{fst} \ (f x) = \text{fst} \ (g x) \land \text{snd} \ (f x) = \text{snd} \ (g x) \)
by (cases \( x \in A \) ) (auto simp: Func_def fun_eq_iff split: if_splits)
then show \( f x = g x \) by (subst (1 2) surjective_pairing) simp
qed
next
fix \( f g \) assume \( f g \in \text{Func} \ A \ (B \times C) \)
thus \( f g \in ?F \cdot \text{Func} \ (B \times C) \)
by (intro image-eqI[of - - ?G fg]) (auto simp: Func-def)
qed (auto simp: Func-def fun-eq-iff)
thus ?thesis using card-of-ordIso by blast
qed

30.9 Regular vs. stable cardinals

definition stable :: 'a rel ⇒ bool
  where
    stable r ≡ ∀(A::'a set) (F :: 'a ⇒ 'a set).
    |A|<o r ∧ (∀a ∈ A. |F a|<o r)
     → |SIGMA a : A. F a|<o r

lemma regularCard-stable:
  assumes cr: Card-order r and ir: ¬finite (Field r) and reg: regularCard r
  shows stable r
unfolding stable-def proof safe
fix A :: 'a set and F :: 'a ⇒ 'a set assume A: |A|<o r and F: ∀a∈A. |F a|<o r
<o r
{assume r<o |Sigma A F|
hence |Field r|<o |Sigma A F| using card-of-Field-ordIso[OF cr] ordIso-ordLeq-trans
by blast
moreover have F: Field r ≠ {} using ir by auto
ultimately have ∃f. f ` Sigma A F = Field r using card-of-ordLeq2[OF F]
by blast
then obtain f where f: f ` Sigma A F = Field r by blast
have r: wo-rel r using cr unfolding card-order-on-def wo-rel-def by auto
{fix a assume a: a ∈ A
  define L where L = {(a,u) | u. u ∈ F a}
  have fl: f ` L ⊆ Field r using f a unfolding L-def by auto
  have bij-betw and {(a, u) | u. u ∈ F a} (F a)
    unfolding bij-betw-def inj-on-def by (auto simp: image-def)
  then have |L| =o |F a| unfolding L-def card-of-ordIso[|symmetric|] by blast
  hence |L|<o r using F a ordIso-ordLess-trans[of |L| |F a|] unfolding L-def
by auto
by auto
  hence ¬ cofinal (f ` L) r using reg fL unfolding regularCard-def
    by (force simp add: dest: notOrdLess-ordIso)
  then obtain k where k: k ∈ Field r and ∀ l ∈ L. ¬(f l ≠ k ∧ (k, f l) ∈ r)
    unfolding cofinal-def image-def by auto
  hence ∃ k ∈ Field r. ∀ l ∈ L. (f l, k) ∈ r
    using wo-rel.innotinIr[OF r - - (k ∈ Field r)] fL unfolding image-subset-iff
by fast
  hence ∃ k ∈ Field r. ∀ a ∈ F a. (f (a,u), k) ∈ r unfolding L-def by auto
}
  then have x: ∩a. a∈A → ∃ k. k ∈ Field r ∧ (∀u∈F a. (f (a, u), k) ∈ r)
    by blast
  obtain gg where ∩a. a∈A → gg a = (SOME k. k ∈ Field r ∧ (∀u∈F a. (f (a, u), k) ∈ r))
THEORY "BNF-Cardinal-Order-Relation"

(a, u, k) ∈ r) by simp

then have gg: ∀ a∈A. ∀ u∈F a. (f (a, u), gg a) ∈ r using some1-ex[OF x] by auto

obtain j0 where j0: j0 ∈ Field r using Fi by auto

define g where [abs-def]: g a = (if F a ≠ {} then gg a else j0) for a

have g: ∀ a∈A. ∀ u∈F a. (f (a, u), g a) ∈ r using gg unfolding g-def by auto

hence 1: Field r ⊆ (⋃ a∈A. under r (g a))

using j[symmetric] unfolding under-def image-def by auto

have gA: g A ⊆ Field r using gg j0 unfolding Field-def g-def by auto

moreover have cofinal (g ' A) r unfolding cofinal-def

proof safe

fix i assume i ∈ Field r

then obtain j where ij: (i, j) ∈ r i ≠ j using cr ir infinite-Card-order-limit

by fast

hence j ∈ Field r using card-order-on-def cr well-order-on-domain by fast

then obtain a where a: a ∈ A and j: (j, g a) ∈ r

using 1 unfolding under-def by auto

hence (i, g a) ∈ r using ij wo-rel.TRANS[OF r] unfolding trans-def by blast

moreover have i ≠ g a

using ij j r unfolding wo-rel-def unfolding well-order-on-def linear-order-on-def

partial-order-on-def antisym-def by auto

ultimately show ∃ j∈g ' A. i ≠ j ∧ (i, j) ∈ r using a by auto

qed

ultimately have |g ' A| = o r using reg unfolding regularCard-def by auto

moreover have |g ' A| ≤ o |A| using card-of-image by blast

ultimately have False using A using not-ordLess-ordIso ordLeq-ordLess-trans

by blast

{ }

thus |Sigma A F| < o r

using cr not-ordLess-iff-ordLeq using card-of-Well-order card-order-on-well-order-on

by blast

qed

lemma stable-regularCard:

assumes cr: Card-order r and ir: ¬ finite (Field r) and st: stable r

shows regularCard r

unfolding regularCard-def proof safe

fix K assume K: K ⊆ Field r and cof: cofinal K r

have |K| ≤ o r using K card-of-Field-ordIso card-of-mono1 cr ordLeq-ordIso-trans

by blast

moreover

{ assume Kr: |K| < o r

have x: ∃ a. a ∈ Field r ⇒ ∃ b. b ∈ K ∧ a ≠ b ∧ (a, b) ∈ r using cof

unfolding cofinal-def by blast

then obtain f where ∃ a. a ∈ Field r ⇒ f a = (SOME b. b ∈ K ∧ a ≠ b ∧ (a, b) ∈ r) by simp

then have v a∈Field r. f a ∈ K ∧ a ≠ f a ∧ (a, f a) ∈ r using some1-ex[OF
z] by simp
  hence Field r ⊆ (⋃ a ∈ K. underS r a) unfolding underS-def by auto
  hence r ≤o ⋃ a ∈ K. underS r a
  using cr Card-order-iff-ordLeq-card-of-mon1 ordLeq-transitive by blast
  also have ⋃ a ∈ K. underS r a ≤o |SIGMA a: K. underS r a| by (rule card-of-UNION-Sigma)
  also {have ∀ a ∈ K. underS r a <o r using K card-of-underS[OF cr] subsetD by auto
  hence |SIGMA a: K. underS r a| <o r using stKr unfolding stable-def by auto
  finally have r <o r .
  hence False using ordLess-irreflexive by blast
} ultimately show |K| =o r using ordLeq-iff-ordLess-or-ordIso by blast qed

lemma internalize-card-of-ordLess:
( |A| <o r ) = (∃ B < Field r. |A| =o |B| ∧ |B| <o r )
proof
  assume |A| <o r
  then obtain p where 1: Field p < Field r ∧ |A| =o p ∧ p <o r
  using internalize-ordLess[of |A| r] by blast
  hence Card-order p using card-of-Card-order Card-order-ordIso2 by blast
  hence |Field p| =o p using card-of-Field-ordIso by blast
  hence |A| =o |Field p| ∧ |Field p| <o r
  using 1 ordIso-equivalence ordIso-ordLess-trans by blast
  thus ∃ B < Field r. |A| =o |B| ∧ |B| <o r using 1 by blast
next
  assume ∃ B < Field r. |A| =o |B| ∧ |B| <o r
  thus |A| <o r using ordIso-ordLess-trans by blast
qed

lemma card-of-Sigma-cong1:
  assumes ∀ i ∈ I. |A i| =o |B i|
  shows |SIGMA i: I. A i| =o |SIGMA i: I. B i|
  using assms by (auto simp add: card-of-Sigma-mono1 ordIso-iff-ordLeq)

lemma card-of-Sigma-cong2:
  assumes bij-betw f (I::'i set) (J::'j set)
  shows |SIGMA i: I. (A::'j ⇒ 'a set) (f i)| =o |SIGMA j: J. A j|
proof
  let ⦧LEFT = SIGMA i: I. A (f i)
  let ⦧RIGHT = SIGMA j: J. A j
  define u where u ≡ λ(i::'i,a::'a). (f i,a)
  have bij-betw u ⦧LEFT ⦧RIGHT
    using assms unfolding u-def bij-betw-def inj-on-def by auto
THEORY "BNF-Cardinal-Order-Relation" 762

thus \( \text{thesis using card-of-ordIso by blast} \)

qed

lemma card-of-Sigma-cong:
assumes BIJ: bij-betw f I J and ISO: \( \forall j \in J. \; |A \cdot j| = o \cdot |B \cdot j| \)
sows \( |\Sigma g : I. \; A \cdot (g \cdot i)| = o \cdot |\Sigma g : J. \; B \cdot j| \)
proof –
have \( \forall i \in I. \; |A \cdot (f \cdot i)| = o \cdot |B \cdot (f \cdot i)| \)
using ISO BIJ unfolding bij-betw-def by blast
hence \(|\Sigma g : I. \; A \cdot (f \cdot i)| = o \cdot |\Sigma g : J. \; B \cdot j| \) by (rule card-of-Sigma-cong1)
moreover have \(|\Sigma g : J. \; B \cdot (f \cdot i)| = o \cdot |\Sigma g : I. \; A \cdot j| \)
using BIJ card-of-Sigma-cong2 by blast
ultimately show \( \text{thesis using ordIso-transitive by blast} \)

qed

lemma stable-elim:
assumes ST: stable r and A-LESS: \( |A| < o \cdot r \) and
F-LESS: \( \forall a, a \in A \Rightarrow |F \cdot a| < o \cdot r \)
sows \(|\Sigma g : A. \; F \cdot a| < o \cdot r \)
proof –
obtain A' where 1: \( |A'| < o \cdot r \)
using internalize-card-of-ordLess[of A r] A-LESS by blast
then obtain G where 2: \( \text{bij-betw G A' A} \)
using card-of-ordIso ordIso-symmetric by blast

\{ fix a assume a \in A \\
  hence \( \exists B', B' < Field \cdot r \wedge |F \cdot a| = o \cdot |B' \wedge |B'| < o \cdot r \)
  using internalize-card-of-ordLess[of F a r] F-LESS by blast \}
then obtain F' where
  \( \forall a \in A \cdot F' \cdot a < Field \cdot r \wedge |F \cdot a| = o \cdot |F' \cdot a| \wedge |F' \cdot a| < o \cdot r \)
  using behoice[of A \a a B'] B' < Field \cdot r \wedge |F \cdot a| = o \cdot |B' \wedge |B'| < o \cdot r \) by blast
hence 4: \( \forall a \in A \cdot F' \cdot a < Field \cdot r \wedge |F' \cdot a| < o \cdot r \) by auto
have 5: \( \forall a \in A \cdot |F' \cdot a| = o \cdot |F \cdot a| \) using temp ordIso-symmetric by auto

have \( \forall a' \in A' \cdot F' \cdot G \cdot a' < Field \cdot r \wedge |F' \cdot G \cdot a'| < o \cdot r \)
using 3 \# bij-betw-ball[of G A' A] by auto
hence \( |\Sigma g : A'. \; F' \cdot G \cdot a'| < o \cdot r \)
using ST 1 unfolding stable-def by auto
moreover have \( |\Sigma g : A'. \; F' \cdot G \cdot a'| = o \cdot |\Sigma g : A. \; F \cdot a| \)
using card-of-Sigma-cong[of G A' A F' F] 5 3 by blast
ultimately show \( \text{thesis using ordIso-symmetric ordIso-ordLess-trans by blast} \)

qed

lemma stable-natLeg: stable natLeq
proof(unsfold stable-def, safe)
  fix A :: 'a set and F :: 'a \Rightarrow 'a set
  assume \( |A| < o \cdot \text{natLeq} \) and \( \forall a \in A. \; |F \cdot a| < o \cdot \text{natLeq} \)
hence finite A ∧ (∀a ∈ A. finite(F a))
    by (auto simp add: finite-iff-ordLess-natLeq)

hence finite(Sigma A F) by (simp only: finite-Sigma[of A F])
thus |Sigma A F| <o natLeq
    by (auto simp add: finite-iff-ordLess-natLeq)

qed

corollary regularCard-natLeq: regularCard natLeq
  Field-natLeq by simp

lemma stable-ordIso1:
  assumes ST: stable r and ISO: r' =o r
  shows stable r'
proof(unfold stable-def, auto)
  fix A::'b set and F::'b ⇒ 'b set
  assume |A| <o r' and ∀ a ∈ A. |F a| <o r'
  hence (|A| <o r) ∧ (∀ a ∈ A. |F a| <o r)
  using ISO ordLess-ordIso-trans by blast
  hence |SIGMA a : A. F a| <o r using assms stable-elim by blast
  thus |SIGMA a : A. F a| <o r'
    using ISO ordIso-symmetric ordLess-ordIso-trans by blast
qed

lemma stable-UNION:
  assumes stable r and |A| <o r and ∏ a ∈ A. |F a| <o r
  shows |⋃ a ∈ A. F a| <o r
  using assms card-of-UNION-Sigma stable-elim ordLeq-ordLess-trans by blast

lemma card-of-UNION-ordLess-infinite:
  assumes stable |B| and |I| <o |B| and ∀ i ∈ I. |A i| <o |B|
  shows |⋃ i ∈ I. A i| <o |B|
  using assms stable-UNION by blast

lemma card-of-UNION-ordLess-infinite-Field:
  assumes ST: stable r and r: Card-order r and
  LEQ-I: |I| <o r and LEQ: ∀ i ∈ I. |A i| <o r
  shows |⋃ i ∈ I. A i| <o r
proof −
  let ?B = Field r
  have 1: r =o |?B| ∧ |?B| =o r using r card-of-Field-ordIso
    ordIso-symmetric by blast
  hence |I| <o |?B| ∀ i ∈ I. |A i| <o |?B|
    using LEQ-I LEQ ordLeq-ordIso-trans by blast+
  moreover have stable |?B| using stable-ordIso1 ST 1 by blast
  ultimately have |⋃ i ∈ I. A i| <o |?B| using LEQ
    card-of-UNION-ordLess-infinite by blast
  thus ?thesis using 1 ordLess-ordIso-trans by blast
qed
31 Cardinal Arithmetic as Needed by Bounded Natural Functors

theory BNF-Cardinal-Arithmetic
  imports BNF-Cardinal-Order-Relation
begin

lemma dir-image: \[ \forall x y. (f x = f y) = (x = y); \text{Card-order } r \implies r = o \text{ dir-image } r f \]
  by (rule dir-image-ordIso) (auto simp add: inj-on-def card-order-on-def)

lemma card-order-dir-image:
  assumes bij: bij f and co: card-order r
  shows card-order (dir-image r f)
proof -
  have Field (dir-image r f) = UNIV
    using assms card-order-on-Card-order[of UNIV r]
    unfolding bij-def dir-image-Field by auto
  moreover from bij have \[ \forall x y. (f x = f y) = (x = y) \]
    unfolding bij-def inj-on-def by auto
  with co have Card-order (dir-image r f)
    using card-order-on-Card-order Card-order-ordIso2[OF - dir-image]
    by blast
  ultimately show \(?thesis\) by auto
qed

lemma ordIso-refl: Card-order r \implies r = o r
  by (rule card-order-on-ordIso)

lemma ordLeq-refl: Card-order r \implies r \leq o r
  by (rule ordIso-imp-ordLeq, rule card-order-on-ordIso)

lemma card-of-ordIso-subst: A = B \implies |A| = o |B|
  by (simp only: ordIso-refl card-of-Card-order)

lemma Field-card-order: card-order r \implies Field r = UNIV
  using card-order-on-Card-order[of UNIV r] by simp

31.1 Zero

definition czero where
czero = card-of {}

lemma czero-ordIso: czero = o czero
  using card-of-empty-ordIso by (simp add: czero-def)
lemma card-of-ordIso-czero-iff-empty:
\[ |A| = o (\text{czero} :: 'b rel) \iff A = (\{\} :: 'a set) \]
unfolding czero-def by (rule iffI[OF card-of-empty2]) (auto simp: card-of-refl card-of-empty-ordIso)

abbreviation Cnotzero where
\[ \text{Cnotzero} (r :: 'a rel) \equiv \neg (r = o (\text{czero} :: 'a rel)) \land \text{Card-order } r \]

lemma Cnotzero-imp-not-empty: Cnotzero r \implies Field r \neq \{\}
unfolding Card-order-iff-ordIso-card-of czero-def by force

lemma czeroI:
\[ [\text{Card-order } r; \text{Field } r = \{\}] \implies r = o \text{ czero} \]
using Cnotzero-imp-not-empty ordIso-transitive[OF - czero-ordIso] by blast

lemma czeroE:
\[ r = o \text{ czero} \implies \text{Field } r = \{\} \]
unfolding czero-def
by (drule card-of-cong) (simp only: Field-card-of card-of-empty2)

lemma Cnotzero-mono:
\[ [\text{Cnotzero } r; \text{Card-order } q; r \leq o q] \implies \text{Cnotzero } q \]
by (force intro: czeroI dest: card-of-mono2 card-of-empty3 czeroE)

31.2 (In)finite cardinals

definition cinfinite where
\[ \text{cinfinite } r \equiv (\neg \text{finite } (\text{Field } r)) \]

abbreviation Cinfinite where
\[ \text{Cinfinite } r \equiv \text{cinfinite } r \land \text{Card-order } r \]

definition cfinite where
\[ \text{cfinite } r = \text{finite } (\text{Field } r) \]

abbreviation Cfinite where
\[ \text{Cfinite } r \equiv \text{cfinite } r \land \text{Card-order } r \]

lemma Cfinite-ordLess-Cinfinite:
\[ [\text{Cfinite } r; \text{Cinfinite } s] \implies r < o s \]
unfolding cfinite-def cinfinite-def
by (blast intro: finite-ordLess-infinite card-order-on-well-order-on)

lemmas natLeq-card-order = natLeq-Card-order[unfolded Field-natLeq]

lemma natLeq-cinfinite: cinfinite natLeq
unfolding cinfinite-def Field-natLeq by (rule infinite-UNIV-nat)
lemma natLeq-Cinfinite: Cinfinite natLeq
  using natLeq-cinfinite natLeq-Card-order by simp

lemma natLeq-ordLeq-cinfinite:
  assumes inf: Cinfinite r
  shows natLeq ≤ o r
proof −
  from inf have natLeq ≤ |Field r| unfolding cinfinite-def
  using infinite-iff-natLeq-ordLeq by blast
also from inf have |Field r| = o r by (simp add: card-of-unique ordIso-symmetric)
finally show ?thesis .
qed

lemma cinfinite-not-czero: cinfinite r ⇒ ¬ (r = o (czero :: 'a rel))
unfolding cinfinite-def by (cases Field r = {}) (auto dest: czeroE)

lemma Cinfinite-Cnotzero: Cinfinite r ⇒ Cnotzero r
using cinfinite-not-czero by auto

lemma Cinfinite-cong: [ r1 = o r2; Cinfinite r1 ] ⇒ Cinfinite r2
using Card-order-ordIso2[of r1 r2] unfolding cinfinite-def ordIso-iff-ordLeq
by (auto dest: card-of-ordLeq-infinite[of card-of-mono2])

lemma cinfinite-mono: [ r1 ≤ o r2; cinfinite r1 ] ⇒ cinfinite r2
unfolding cinfinite-def by (auto dest: card-of-ordLeq-infinite[of card-of-mono2])

lemma regularCard-ordIso:
  assumes k = o k' and Cinfinite k and regularCard k
  shows regularCard k'
proof −
  have stable k using assms cinfinite-def regularCard-stable by blast
  hence stable k' using assms stable-ordIso1 ordIso-symmetric by blast
  thus ?thesis using assms cinfinite-def stable-regularCard Cinfinite-cong by blast
qed

corollary card-of-UNION-ordLess-infinite-Field-regularCard:
  assumes regularCard r and Cinfinite r and |I| < o r and ∀ i ∈ I. |A i| < o r
  shows | ∪ i ∈ I. A i | < o r
using card-of-ORD-ordLess-infinite-Field regularCard-stable assms cinfinite-def
by blast

31.3 Binary sum

definition csum (infixr +c 65)
  where r1 +c r2 ≡ |Field r1 <+> Field r2|

lemma Field-csum: Field (r +c s) = Inl ' Field r ∪ Inr ' Field s
unfolding csum-def Field-card-of by auto
lemma Card-order-csum: Card-order \((r1 + c r2)\)
unfolding csum-def by (simp add: card-of-Card-order)

lemma csum-Cnotzero1: Cnotzero \(r1\) \(\Rightarrow\) Cnotzero \((r1 + c r2)\)
using Cnotzero-imp-not-empty
by (auto intro: card-of-Card-order simp: csum-def card-of-ordIso-czero-iff-empty)

lemma card-order-csum:
assumes card-order \(r1\) card-order \(r2\)
shows card-order \((r1 + c r2)\)
proof −
  have Field \(r1\) = UNIV Field \(r2\) = UNIV using assms card-order-on-Card-order
by auto
  thus \(?thesis\)
unfolding csum-def by (auto simp: card-of-card-order-on)
qed

lemma cinfinite-csum:
cinfinite \(r1\) \(\lor\) cinfinite \(r2\) \(\Rightarrow\) cinfinite \((r1 + c r2)\)
unfolding cinfinite-def csum-def by (auto simp: Field-card-of)

lemma Cinfinite-csum:
Cinfinite \(r1\) \(\lor\) Cinfinite \(r2\) \(\Rightarrow\) Cinfinite \((r1 + c r2)\)
using card-of-Card-order
by (auto simp: cinfinite-def csum-def Field-card-of)

lemma Cinfinite-csum1:
Cinfinite \(r1\) \(\Rightarrow\) Cinfinite \((r1 + c r2)\)
by (blast intro: Cinfinite-csum elim: )

lemma Cinfinite-csum-weak:
\([\text{Cinfinite } r1; \text{Cinfinite } r2]\) \(\Rightarrow\) Cinfinite \((r1 + c r2)\)
by (erule Cinfinite-csum1)

lemma csum-cong: \([p1 = o r1; p2 = o r2]\) \(\Rightarrow\) \(p1 + c p2 = o r1 + c r2\)
by (simp only: csum-def ordIso-Plus-cong)

lemma csum-cong1: \(p1 = o r1\) \(\Rightarrow\) \(p1 + c q = o r1 + c q\)
by (simp only: csum-def ordIso-Plus-cong1)

lemma csum-cong2: \(p2 = o r2\) \(\Rightarrow\) \(q + c p2 = o q + c r2\)
by (simp only: csum-def ordIso-Plus-cong2)

lemma csum-mono: \([p1 \leq o r1; p2 \leq o r2]\) \(\Rightarrow\) \(p1 + c p2 \leq o r1 + c r2\)
by (simp only: csum-def ordLeq-Plus mono)

lemma csum-mono1: \(p1 \leq o r1\) \(\Rightarrow\) \(p1 + c q \leq o r1 + c q\)
by (simp only: csum-def ordLeq-Plus mono1)

lemma csum-mono2: \(p2 \leq o r2\) \(\Rightarrow\) \(q + c p2 \leq o q + c r2\)
by (simp only: csum-def ordLeq-Plus mono2)
lemma ordLeq-csum1: Card-order p1 \implies p1 \leq n p1 + c p2
by (simp only: csum-def Card-order-Plus1)

lemma ordLeq-csum2: Card-order p2 \implies p2 \leq n p1 + c p2
by (simp only: csum-def Card-order-Plus2)

lemma csum-com: \( p1 + c p2 = o p2 + c p1 \)
by (simp only: csum-def card-of-Plus-commute)

lemma csum-assoc: \((p1 + c p2) + c p3 = o p1 + c p2 + c p3\)
by (simp only: csum-def Field-card-of card-of-Plus-assoc)

lemma Cfinite-csum: \([Cfinite r; Cfinite s] \implies Cfinite (r + c s)\)
unfolding cfinite-def csum-def Field-card-of using card-of-card-order-on by simp

lemma csum-csum: \((r1 + c r2) + c (r3 + c r4) = o r1 + c r2 + c (r3 + c r4)\)
proof –
  have \((r1 + c r2) + c (r3 + c r4) = o r1 + c r2 + c (r3 + c r4)\)
  by (rule csum-assoc)
  also have \(r1 + c r2 + c (r3 + c r4) = o r1 + c (r2 + c r3) + c r4\)
  by (intro csum-assoc csum-cong2 ordIso-symmetric)
  also have \(r1 + c (r2 + c r3) + c r4 = o r1 + (r2 + c r3) + c r4\)
  by (intro csum-cong1 csum-cong2)
  also have \(r1 + c r3 + c r2 + c r4 = o (r1 + c r3) + c (r2 + c r4)\)
  by (intro csum-assoc ordIso-symmetric)
also have \(r1 + c r3 + c r2 + c r4 = o (r1 + c r3) + c (r2 + c r4)\)
  by (intro csum-assoc ordIso-symmetric)
finally show \(?thesis\) .
qed

lemma Plus-csum: \(|A <+> B| = o |A| + c |B|\)
by (simp only: csum-def Field-card-of card-of-refl)

lemma Un-csum: \(|A \cup B| \leq o |A| + c |B|\)
using ordLeq-ordIso-trans[OF card-of-Un-Plus-ordLeq Plus-csum] by blast

31.4 One

definition cone where
cone = card-of \{(\)\}

lemma Card-order-cone: Card-order cone
unfolding cone-def by (rule card-of-Card-order)

lemma Cfinite-cone: Cfinite cone
unfolding cfinite-def by (simp add: Card-order-cone)
lemma cone-not-czero: ¬ (cone = o czero)
  unfolding czero-def cone-def ordIso-iff-ordLeq
  using card-of-empty3 empty-not-insert by blast

lemma cone-ordLeq-Cnotzero: Cnotzero r ⇒ cone ≤ o r
  unfolding cone-def by (rule Card-order-singl-ordLeq) (auto intro: czeroI)

31.5  Two

definition ctwo where
  ctwo = |UNIV :: bool set|

lemma Card-order-ctwo: Card-order ctwo
  unfolding ctwo-def by (rule card-of-Card-order)

lemma ctwo-not-czero: ¬ (ctwo = o czero)
  using card-of-empty3 [of UNIV :: bool set] ordIso-iff-ordLeq
  unfolding czero-def ctwo-def using UNIV-not-empty by auto

lemma ctwo-Cnotzero: Cnotzero ctwo
  by (simp add: ctwo-not-czero Card-order-ctwo)

31.6  Family sum

definition Csum where
  Csum r rs ≡ |SIGMA i : Field r. Field (rs i)|

syntax -Csum ::
  pttrn => ('a * 'a) set => 'b * 'b set => (('a * 'b) * ('a * 'b)) set
((3CSUM ::- ::) [0, 51, 10] 10)

translations
  CSUM i:r. rs == CONST Csum r (%i. rs)

lemma SIGMA-CSUM: |SIGMA i : I. As i| = (CSUM i : |I|. |As i|)
  by (auto simp: Csum-def Field-card-of)

31.7  Product

definition cprod (infixr "*c" 80) where
  r1 *c r2 = |Field r1 × Field r2|

lemma card-order-cprod:
  assumes card-order r1 card-order r2
  shows card-order (r1 *c r2)
proof -
  have Field r1 = UNIV Field r2 = UNIV
    using assms card-order-on-Card-order by auto
  thus ?thesis by (auto simp: cprod-def card-of-card-order-on)
qed

lemma Card-order-cprod: Card-order (r1 *c r2)
by (simp only: cprod-def Field-card-of card-of-card-order-on)

lemma cprod-mono1: p1 ≤o r1 ⟹ p1 *c q ≤o r1 *c q
by (simp only: cprod-def ordLeq-Times-mono1)

lemma cprod-mono2: p2 ≤o r2 ⟹ q *c p2 ≤o q *c r2
by (simp only: cprod-def ordLeq-Times-mono2)

lemma cprod-mono: [p1 ≤o r1; p2 ≤o r2] ⟹ p1 *c p2 ≤o r1 *c r2
by (rule ordLeq-transitive [OF cprod-mono1 cprod-mono2])

lemma ordLeq-cprod2: [Cnotzero p1; Card-order p2] ⟹ p2 ≤o p1 *c p2
unfolding cprod-def by (rule Card-order-Times2) (auto intro: czeroI)

lemma cinfinite-cprod: [cinfinite r1; cinfinite r2] ⟹ cinfinite (r1 *c r2)
by (simp add: cinfinite-def cprod-def Field-card-of infinite-cartesian-product)

lemma cinfinite-cprod2: [Cnotzero r1; Cinfinite r2] ⟹ cinfinite (r1 *c r2)
by (rule cinfinite-mono) (auto intro: ordLeq-cprod2)

lemma Cinfinite-cprod2: [Cnotzero r1; Cinfinite r2] ⟹ Cinfinite (r1 *c r2)
by (blast intro: cinfinite-cprod2 Card-order-cprod)

lemma cprod-cong: [p1 =o r1; p2 =o r2] ⟹ p1 *c p2 =o r1 *c r2
unfolding ordIso-iff-ordLeq by (blast intro: cprod-mono)

lemma cprod-cong1: [p1 =o r1] ⟹ p1 *c p2 =o r1 *c p2
unfolding ordIso-iff-ordLeq by (blast intro: cprod-mono1)

lemma cprod-cong2: p2 =o r2 ⟹ q *c p2 =o q *c r2
unfolding ordIso-iff-ordLeq by (blast intro: cprod-mono2)

lemma cprod-cong: p1 *c p2 =o p2 *c p1
by (simp only: cprod-def card-of-Times-commute)

lemma card-of-Csum-Times:
∀ i ∈ I. |A i| ≤o |B| ⟹ (CSUM i : |I|. |A i| ) ≤o |I| *c |B|
by (simp only: Csum-def card-of-card-order-on)

lemma card-of-Csum-Times':
assumes Card-order r ∨ i ∈ I. |A i| ≤o r
shows (CSUM i : |I|. |A i| ) ≤o |I| *c r
proof –
from assms(1) have *: r =o |Field r| by (simp add: card-of-unique)
with assms(2) have ∀ i ∈ I. |A i| ≤o |Field r| by (blast intro: ordLeq-ordIso-trans)
hence (CSUM i : |I|. |A i| ) ≤o |I| *c |Field r| by (simp only: card-of-Csum-Times)
also from * have \(|I| \times c |Field r| \leq o |I| \times c r\)
  by (simp only: Field-card-of card-of-refl cprod-def ordIso-imp-ordLeq)
finally show \(?thesis\).
qed

lemma cprod-csum-distrib1: \(c r1 \times c r2 + c r1 \times c r3 = o r1 \times c (r2 + c r3)\)
  unfolding csum-def cprod-def by (simp add: Field-card-of card-of-Times-Plus-distrib ordIso-symmetric)

lemma csum-absorb2': \([\text{Card-order } r2; \; r1 \leq o r2; \; \text{cinfinite } r1 \vee \text{cinfinite } r2] \implies\)
  \(r1 + c r2 = o r2\)
  unfolding csum-def
  using Card-order-Plus-infinite
  by (fastforce simp: cinfinite-def dest: cinfinite-mono)

lemma csum-absorb1':
  assumes card: \(\text{Card-order } r2\)
  and r12: \(r1 \leq o r2\) and cr12: \(\text{cinfinite } r1 \vee \text{cinfinite } r2\)
  shows \(r2 + c r1 = o r2\)
proof –
  have \(r1 + c r2 = o r2\)
    by (simp add: csum-absorb2' assms)
  then show \(?thesis\)
    by (blast intro: ordIso-transitive csum-com)
qed

lemma csum-mono-strict:

lemma regularCard-csum:
  assumes Cinfinite r Cinfinite s regularCard r regularCard s
  shows regularCard \((r + c s)\)
proof (cases \(r \leq s\))
  case True
  then show \(?thesis using regularCard-ordIso[of s] csum-absorb2'[THEN ordIso-symmetric]\)
    assms by auto
next
  case False
  have Well-order s Well-order r using assms card-order-on-well-order-on by auto
  then have \(s \leq o r\) using not-ordLeq-iff-ordLess False ordLess-imp-ordLeq by auto
  then show \(?thesis using regularCard-ordIso[of r] csum-absorb1'[THEN ordIso-symmetric]\)
    assms by auto
qed
assumes \( \text{Card-order}: \text{Card-order } r \text{ Card-order } q \)
and \( \text{Cinfinite}: \text{Cinfinite } r' \text{ Cinfinite } q' \)
and \( \text{less}: r < o r' q < o q' \)
shows \( r + c q < o r' + c q' \)

proof
have \( \text{Well-order}: \text{Well-order } r \text{ Well-order } q \text{ Well-order } r' \text{ Well-order } q' \)
using \( \text{card-order-on-well-order-on } \text{Card-order } \text{Cinfinite} \) by auto
show \( ?\text{thesis} \)
proof (cases \( \text{Cinfinite } r \))
case outer: True
then show \( ?\text{thesis} \)
proof (cases \( \text{Cinfinite } q \))
case inner: True
then show \( ?\text{thesis} \)
proof (cases \( r \leq o q \))
case True
then have \( r + c q = o q \) using \( \text{csum-absorb2 } \text{inner} \) by blast
then show \( ?\text{thesis} \)
using \( \text{ordIso-ordLess-trans } \text{ordLess-ordLeq-trans } \text{less } \text{Cinfinite } \text{ordLeq-csum2} \)
by blast
next
case False
then have \( q \leq o r \) using \( \text{not-ordLeq-iff-ordLess } \text{Well-order } \text{ordLess-imp-ordLeq} \)
by blast
then have \( r + c q = o r \) using \( \text{csum-absorb1 } \text{outer} \) by blast
then show \( ?\text{thesis} \)
using \( \text{ordIso-ordLess-trans } \text{ordLess-ordLeq-trans } \text{less } \text{Cinfinite } \text{ordLeq-csum1} \)
by blast
qed
next
case False
then have \( \text{Cfinite } q \) using \( \text{Card-order } \text{cinfinite-def } \text{cfinite-def} \) by blast
then have \( q \leq o r \) using \( \text{finite-ordLess-infinite } \text{cfinite-def } \text{cinfinite-def } \text{outer} \)
\( \text{Well-order } \text{ordLess-imp-ordLeq} \) by blast
then have \( r + c q = o r \) by (rule \( \text{csum-absorb1}[OF } \text{outer}])
then show \( ?\text{thesis} \) using \( \text{ordIso-ordLess-trans } \text{ordLess-ordLeq-trans } \text{less ordLeq-csum1 } \text{Cinfinite} \) by blast
qed
next
case False
then have \( \text{outer: } \text{Cfinite } r \) using \( \text{Card-order } \text{cinfinite-def } \text{cfinite-def} \) by blast
then show \( ?\text{thesis} \)
proof (cases \( \text{Cinfinite } q \))
case True
then have \( r \leq o q \) using \( \text{finite-ordLess-infinite } \text{cinfinite-def } \text{cfinite-def } \text{outer} \)
\( \text{Well-order } \text{ordLess-imp-ordLeq} \) by blast
then have \( r + c q = o q \) by (rule \( \text{csum-absorb2}[OF } \text{True}])
then show \( ?\text{thesis} \) using \( \text{ordIso-ordLess-trans } \text{ordLess-ordLeq-trans } \text{less ordLeq-csum1 } \text{Cinfinite} \) by blast
qed
dLeq-csum2 Cinfinite by blast

next
  case False
  then have \textit{Cfinite} \( q \) using \textit{Card-order cinfinite-def} \textit{cfinite-def} by blast
  then have \textit{Cfinite} \( r + c \) \( q \) using \textit{Cfinite-csum outer} by blast
  moreover have \textit{Cinfinite} \( r' + c \) \( q' \) using \textit{Cinfinite-csum1} \textit{Cinfinite} by blast
  ultimately show \textit{thesis} using \textit{Cfinite-ordLess-Cinfinite} by blast
  qed
  qed
  qed

31.8 Exponentiation

definition \textit{cexp} (infixr \( \wedge c \) 90) where
\( r1 \wedge c r2 \equiv |\text{Func} (\text{Field} r2) (\text{Field} r1)| \)

lemma \textit{Card-order-cexp}: \textit{Card-order} \( (r1 \wedge c r2) \)

unfolding \textit{cexp-def} by (rule \textit{card-of-Card-order})

lemma \textit{cexp-mono'}:
  assumes 1: \( p1 \leq o r1 \) and 2: \( p2 \leq o r2 \)
  and n: Field \( p2 = \{\} \Rightarrow \text{Field} r2 = \{\} \)
  shows \( p1 \wedge c p2 \leq o r1 \wedge c r2 \)

proof (cases Field \( p1 = \{\} \))
  case True
  hence Field \( p2 \neq \{\} \Rightarrow \text{Func} (\text{Field} p2) \{\} = \{\} \) unfolding \textit{Func-is-emp} by simp
  with True have Field \( p2 \neq \{\} \Rightarrow \text{Func} (\text{Field} p2) \{\} = \{\} \) unfolding \textit{Func-def} by (simp add: Field-card-of cexp-def)
  hence \( p1 \wedge c p2 \leq o \) cone unfolding \textit{cexp-def}.
  thus \textit{thesis}
  proof (cases Field \( p2 = \{\} \))
    case True
    with n have Field \( r2 = \{\} \).
    hence cone \( \leq o r1 \wedge c r2 \) unfolding cone-def cexp-def Func-def
    by (auto intro: card-of-ordLeqI where \( f=\lambda - - . \) undefined)
    thus \textit{thesis} using \( p1 \wedge c p2 \leq o \) cone ordLeq-transitive by auto
  next
    case False
    with True have Field \( (p1 \wedge c p2) = o \) czero
    unfolding card-of-ordIso-czero-iff-empty cexp-def Field-card-of Func-def by auto
    thus \textit{thesis unfolding} card-of-ordIso-czero-iff-empty Field-card-of Func-def by auto
  qed
next
  case False
  have 1: \( |\text{Field} p1| \leq o |\text{Field} r1| \) and 2: \( |\text{Field} p2| \leq o |\text{Field} r2| \)
using 1 2 by (auto simp: card-of-mono2)
obtain f1 where f1: f1 : Field r1 = Field p1
using 1 unfolding card-of-ordLeq2[OF False, symmetric] by auto
obtain f2 where f2: inj-on f2 (Field p2) f2 : Field p2 ⊆ Field r2
using 2 unfolding card-of-ordLeq[symmetric] by blast
have 0: Func-map (Field p2) f1 f2 : (Field (r1 'c r2)) = Field (p1 'c p2)
unfolding cexp-def Field-card-of using Func-map-surj[OF f1 f2 n, symmetric]

have 00: Field (p1 'c p2) ≠ {} unfolding cexp-def Field-card-of Func-is-emp
using False by simp
show ?thesis using 0 card-of-ordLeq2[OF 00] unfolding cexp-def Field-card-of by blast
qed

lemma cexp-mono:
assumes 1: p1 ≤ o r1 and 2: p2 ≤ o r2
and n: p2 = o czero ⇒ r2 = o czero and card: Card-order p2
shows p1 'c p2 ≤ o r1 'c r2
by (rule cexp-mono'[OF 1 2 czeroE[OF n[OF czeroI[OF card]]]])

lemma cexp-mono1:
assumes 1: p1 ≤ o r1 and q: Card-order q
shows p1 'c q ≤ o r1 'c q
using ordLeq-refl[OF q] by (rule cexp-mono[OF 1]) (auto simp: q)

lemma cexp-mono2':
assumes 2: p2 ≤ o r2 and q: Card-order q
and n: Field p2 = {} ⇒ Field r2 = {}
shows q 'c p2 ≤ o q 'c r2
using ordLeq-refl[OF q] by (rule cexp-mono'[OF - 2 n]) auto

lemma cexp-mono2:
assumes 2: p2 ≤ o r2 and q: Card-order q
and n: p2 = o czero ⇒ r2 = o czero and card: Card-order p2
shows q 'c p2 ≤ o q 'c r2
using ordLeq-refl[OF q] by (rule cexp-mono'[OF - 2 n card]) auto

lemma cexp-mono2-Cnotzero:
assumes p2 ≤ o r2 Card-order q Cnotzero p2
shows q 'c p2 ≤ o q 'c r2
using assms(3) czeroI by (blast intro: cexp-mono2'[OF assms(1,2)])

lemma cexp-cong:
assumes 1: p1 = o r1 and 2: p2 = o r2
and Cr: Card-order r2
and Cp: Card-order p2
shows p1 'c p2 = o r1 'c r2
proof –
obtain f where bij-betw f (Field p2) (Field r2)
using 2 card-of-ordIso[of Field p2 Field r2] card-of-cong by auto
hence 0: Field p2 = {} ↔ Field r2 = {} unfolding bij-betw-def by auto
have r: p2 =o czero ⇒ r2 =o czero
  and p: r2 =o czero ⇒ p2 =o czero
using 0 Cr Cp czeroE czeroI by auto
have r: p2 = o czero ⇒ r2 = o czero
  and p: r2 = o czero ⇒ p2 = o czero
using 0 Cr Cp cexp-mono[OF - - - Cp] cexp-mono[OF - - - Cr] by blast
qed

lemma cexp-cong1:
  assumes 1: p1 = o r1 and q: Card-order q
  shows p1 ^c q = o r1 ^c q
by (rule cexp-cong[OF 1 - q q]) (rule ordIso-refl[OF q])

lemma cexp-cong2:
  assumes 2: p2 = o r2 and q: Card-order q and p: Card-order p2
  shows q ^c p2 = o q ^c r2
by (rule cexp-cong[OF - 2]) (auto simp only: ordIso-refl Card-order-ordIso2)

lemma cexp-cone:
  assumes Card-order r
  shows r ^c cone = o r
proof –
  have r ^c cone = o |Field r|
    unfolding cexp-def cone-def Field-card-of Func-empty
    card-of-ordIso[symmetric] bij-betw-def Func-def inj-on-def image-def
    by (rule exI[of - λf. f ()]) auto
  also have r1 ^c (r3 *c r2) = o ?R
    using cprod-com r1 by (intro cexp-cong2, auto simp: Card-order-cprod)
  finally show ?thesis .
qed

lemma cexp-cprod:
  assumes r1: Card-order r1
  shows (r1 ^c r2) ^c r3 = o r1 ^c (r2 *c r3)
(is ?L =o ?R)
proof –
  have ?L = o r1 ^c (r3 *c r2)
    unfolding cprod-def cexp-def Field-card-of
    using card-of-ordIso-symmetric by (rule ordIso-symmetric)
  also have r1 ^c (r3 *c r2) = o ?R
    using cprod-com r1 by (intro cexp-cong2, auto simp: Card-order-cprod)
  finally show ?thesis .
qed

lemma cprod-infinite1': [Cinfinite r; Cnotzero p; p ≤ o r] ⇒ r *c p = o r
unfolding cprod-def cprod-def
by (rule Card-order-Times-infinite[THEN conjunct1]) (blast intro: czeroI)+

lemma cprod-infinite: Cinfinite r ⇒ r *c r = o r
using cprod-infinite1' Cinfinite-Cnotzero ordLeq-refl by blast

lemma cexp-cprod-ordLeq:
  assumes r1: Card-order r1 and r2: Cinfinite r2
  and r3: Cnotzero r3 \leq o r2
  shows (r1 \sim c r2) \sim c r3 = o r1 \sim c r2 (is ?L = o ?R)
proof –
  have ?L = o r1 \sim c (r2 \ast c r3) using cexp-cprod[OF r1] .
  also have r1 \sim c (r2 \ast c r3) = o ?R
  using assms by (fastforce simp: Card-order-cprod intro: cprod-infinite1' cexp-cong2)
finally show ?thesis .
qed

lemma Cnotzero-UNIV: Cnotzero |UNIV|
  by (auto simp: card-of-Card-order card-of-ordIso-czero-iff-empty)

lemma ordLess-ctwo-cexp:
  assumes Card-order r
  shows r <o ctwo \sim c r
proof –
  have r <o |Pow (Field r)| using assms by (rule Card-order-Pow)
  also have |Pow (Field r)| =o ctwo \sim c r
  unfolding ctwo-def cexp-def Field-card-of by (rule card-of-Pow-Func)
finally show ?thesis .
qed

lemma ordLeq-cexp1:
  assumes Cnotzero r Card-order q
  shows q \leq o q \sim c r
proof (cases q =o (czero :: 'a rel))
  case True thus ?thesis by (simp only: card-of-empty cexp-def czero-def ordIso-ordLeq-trans)
next
  case False
  have q =o q \sim c cone
    by (blast intro: assms ordIso-symmetric cexp-cone)
  also have q \sim c cone \leq o q \sim c r
    using assms
    by (intro cexp-mono2) (simp-all add: cone-ordLeq-Cnotzero cone-not-czero Card-order-cone)
finally show ?thesis .
qed

lemma ordLeq-cexp2:
  assumes ctwo \leq o q Card-order r
  shows r \leq o q \sim c r
proof (cases r =o (czero :: 'a rel))
  case True thus ?thesis by (simp only: card-of-empty cexp-def czero-def ordIso-ordLeq-trans)
next
case False
have \( r < o \operatorname{ctwo}^c r \)
  by (blast intro: assms ordLess-ctwo-cexp)
also have \( \operatorname{ctwo}^c r \leq o \operatorname{q}^c r \)
  by (blast intro: assms cezp-mono1)
finally show \(?thesis\) by (rule ordLess-imp-ordLeq)
qed

lemma \( \operatorname{cinfinite-cexp}: [ctwo \leq o \operatorname{q}; \operatorname{Cinfinite r}] \Rightarrow \operatorname{cinfinite} (q^c r) \)
by (rule cinfinite-mono[OF ordLeq-cexp2]) simp-all

lemma \( \operatorname{Cinfinite-cexp}: [ctwo \leq o \operatorname{q}; \operatorname{Cinfinite r}] \Rightarrow \operatorname{Cinfinite} (q^c r) \)
by (simp add: cinfinite-cexp Card-order-cexp)

lemma \( \operatorname{card-order-cexp}: \)
  assumes \( \operatorname{card-order r1 card-order r2} \)
  shows \( \operatorname{card-order} (r1^c r2) \)
proof 
  have Field \( r1 = \operatorname{UNIV} \)
  Field \( r2 = \operatorname{UNIV} \)
  using assms card-order-on-Card-order
  by auto
  thus \(?thesis\) unfolding cexp-def Func-def
  using card-of-card-order-on
  by simp
qed

lemma \( \operatorname{ctwo-ordLess-natLeq}: \operatorname{ctwo} < o \operatorname{natLeq} \)
  unfolding \( \operatorname{ctwo-def} \)
  using finite-UNIV natLeq-cinfinite natLeq-Card-order
  by (intro Cfinite-ordLess-Cinfinite)
  (auto simp: cfinite-def card-of-Card-order)

lemma \( \operatorname{ctwo-ordLess-Cinfinite}: \operatorname{Cinfinite r} \Rightarrow \operatorname{ctwo} < o r \)
  by (rule ordLess-ordLeq-trans[OF ctwo-ordLess-natLeq natLeq-ordLeq-cinfinite])

lemma \( \operatorname{ctwo-ordLeq-Cinfinite}: \)
  assumes \( \operatorname{Cinfinite r} \)
  shows \( \operatorname{ctwo} \leq o r \)
by (rule ordLess-imp-ordLeq[OF ctwo-ordLess-Cinfinite[OF assms]])

lemma \( \operatorname{Un-Cinfinite-bound}: [[|A| \leq o r; |B| \leq o r; \operatorname{Cinfinite r}] \Rightarrow |A \cup B| \leq o r \)
  by (auto simp add: cinfinite-def card-of-Un-ordLeq-infinite-Field)

lemma \( \operatorname{Un-Cinfinite-bound-strict}: [[|A| < o r; |B| < o r; \operatorname{Cinfinite r}] \Rightarrow |A \cup B| < o r \)
  by (auto simp add: cinfinite-def card-of-Un-ordLess-infinite-Field)

lemma \( \operatorname{UNION-Cinfinite-bound}: [[|I| \leq o r; \forall i \in I. |A i| \leq o r; \operatorname{Cinfinite r}] \Rightarrow \bigcup i \in I. A i \leq o r \)
  by (auto simp add: card-of-UNION-ordLeq-infinite-Field cinfinite-def)

lemma \( \operatorname{csum-cinfinite-bound}: \)
assumes \( p \leq \alpha \land q \leq \alpha \) Card-order \( p \) Card-order \( q \) Cinfinite \( r \)
shows \( p + c q \leq \alpha \)
proof

  have \(|\text{Field } p| \leq \alpha \land |\text{Field } q| \leq \alpha \)
  using assms card-of-least ordLeq-transitive unfolding card-order-on-def by blast+
  with assms show ?thesis unfolding cinfinite-def csum-def
    by (blast intro: card-of-Plus-ordLeq-infinite-Field)
qed

lemma cprod-cinfinite-bound:
assumes \( p \leq \alpha \land q \leq \alpha \) Card-order \( p \) Card-order \( q \) Cinfinite \( r \)
shows \( p * c q \leq \alpha \)
proof

  from assms(1-4) have \(|\text{Field } p| \leq \alpha \land |\text{Field } q| \leq \alpha \)
  unfolding card-order-on-def using card-of-least ordLeq-transitive by blast+
  with assms show ?thesis unfolding cinfinite-def cprod-def
    by (blast intro: card-of-Times-ordLeq-infinite-Field)
qed

lemma cprod-infinite2': \([\text{Cnotzero } r1; \text{Cinfinite } r2; r1 \leq \alpha r2] \implies r1 * c r2 = o\)
r2

  unfolding ordIso-iff-ordLeq
by (intro conjI cprod-cinfinite-bound ordLeq-cprod2 ordLeq-refl)
  (auto dest!: ordIso-imp-ordLeq not-ordLeq-ordLess simp: czero-def Card-order-empty)

lemma regularCard-cprod:
assumes Cinfinite \( r \) Cinfinite \( s \) regularCard \( r \) regularCard \( s \)
shows regularCard \((r * c s)\)
proof

  (cases \( r \leq \alpha s \))
  case True
  with assms Cinfinite-Cnotzero show ?thesis
    by (force intro: regularCard-ordIso ordIso-symmetric[OF cprod-infinite2'])
next
  case False
  have Well-order \( r \) Well-order \( s \) using assms card-order-on-well-order-on by auto
  then have \( s \leq \alpha r \) using not-ordLeq-iff-ordLess ordLess-imp-ordLeq False by blast
  with assms Cinfinite-Cnotzero show ?thesis
    by (force intro: regularCard-ordIso ordIso-symmetric[OF cprod-infinite1'])
qed

lemma cprod-csum-cexp:
r1 * c r2 \leq \alpha (r1 + c r2) \sim c ttwo

  unfolding cprod-def csum-def cexp-def ctwo-def Field-card-of
proof

  let \( \tilde{x} = \lambda(a, b). \%x. \text{if } x \text{ then Inl } a \text{ else Inr } b \)
  have inj-on \( \tilde{x} \) (Field r1 × Field r2) (is inj-on - ?LHS)
    by (auto simp: inj-on-def fun-eq-iff split: bool.split)
moreover
have \( \exists \ell \cdot \ell \subseteq \text{Func} (\text{UNIV} :: \text{bool set}) (\text{Field} r1 <+> \text{Field} r2) (\text{is} - \subseteq \ell \cdot \text{RHS}) \)
  by (auto simp: Func-def)
ultimately show \( |\ell \cdot \text{LHS}| \leq o |\ell \cdot \text{RHS}| \) using card-of-ordLeq by blast
qed

lemma Cfinite-cprod-Cinfinite: \( [Cfinite r; \text{Cinfinite} s] \Rightarrow r \cdot c s \leq o s \)
  by (intro cprod-cfinite-bound)
(auto intro: ordLeq-refl ordLess-imp-ordLeq[of Cfinite-ordLess-Cinfinite])

lemma cprod-cexp: \( (r \cdot c s) \cdot c t = o r \cdot c t \cdot s \cdot c t \)
unfolding cprod-def cexp-def Field-card-of by (rule Func-Times-Range)

lemma cprod-cexp-csum-cexp-Cinfinite:
  assumes \( t: \text{Cinfinite} t \)
shows \( (r \cdot c s) \cdot c t \leq o (r \cdot c s) \cdot c t \)
proof
  have \( (r \cdot c s) \cdot c t \leq o ((r \cdot c s) \cdot c \cdot \text{ctwo}) \cdot c t \)
    by (rule cexp-mono1[of cprod-csum-cexp conjunct2[of t]])
  also have \( ((r \cdot c s) \cdot c \cdot \text{ctwo}) \cdot c t = o (r \cdot c s) \cdot c (\cdot \text{ctwo} \cdot c t) \)
    by (rule cexp-cprod[of Card-order-csum])
  also have \( (r \cdot c s) \cdot c (\cdot \text{ctwo} \cdot c t) \cdot c \cdot \text{ctwo} = o (r \cdot c s) \cdot c t \)
    by (rule ordIso-symmetric[of cexp-cprod[of Card-order-csum]])
  also have \( ((r \cdot c s) \cdot c (\cdot \text{ctwo} \cdot c t)) \cdot c \cdot \text{ctwo} \cdot c t = o (r \cdot c s) \cdot c t \)
    by (rule cexp-cprod-ordLeq[of Card-order-csum t \cdot \text{ctwo} \cdot \text{Cnotzero} \cdot \text{ctwo} \cdot \text{ordLeq} \cdot \text{Cinfinite}[of t]])
  finally show \( ?\text{thesis} \).
qed

lemma Cfinite-cexp-Cinfinite:
  assumes \( s: \text{Cfinite} s \) and \( t: \text{Cinfinite} t \)
shows \( s \cdot c t \leq o \cdot \text{ctwo} \cdot c t \)
proof (cases \( s \cdot \leq o \cdot \text{ctwo} \))
case True thus \( ?\text{thesis} \) using \( t \) by (blast intro: cexp-mono1)
next
case False
hence \( \text{ctwo} \leq o \cdot s \) using ordLeq-total[of s \cdot \text{ctwo}] Card-order-ctwo s
  by (auto intro: card-order-on-swell-order-on)
  hence \( \text{Cnotzero} \cdot s \) using Cnotzero-mono[of \cdot \text{ctwo} \cdot \text{Cnotzero}] \( s \) by blast
  hence \( st: \text{Cnotzero} \cdot (\cdot s \cdot c t) \) by (intro Cinfinite-Cnotzero[of Cinfinite-cprod2])
(auto simp: t)
  have \( s \cdot c t \leq o (\cdot \text{ctwo} \cdot c s) \cdot c t \)
   using assms by (blast intro: cexp-mono1 ordLess-imp-ordLeq[of ordLess-ctwo-cexp])
  also have \( (\cdot \text{ctwo} \cdot c s) \cdot c t \cdot o \cdot \text{ctwo} \cdot c (\cdot s \cdot c t) \)
    by (blast intro: Card-order-ctwo cexp-cprod)
  also have \( \text{ctwo} \cdot c (\cdot s \cdot c t) \leq o \cdot \text{ctwo} \cdot c t \)
using assms st by (intro cexp-mono2-Cnotzero Cfinite-cprod-Cinfinite Card-order-ctwo)
finally show ?thesis .
qed

lemma csum-Cfinite-cexp-Cinfinite:
assumes r: Card-order r and s: Cfinite s and t: Cinfinite t
shows (r + c s) c t ≤ o (r + c ctwo) c t
proof (cases Cinfinite r)
case True
hence (r + c s) c t = o r c t using t by (blast intro: cexpc-cong1)
also have o r c t ≤ o (r + c ctwo) c t using t by (blast intro: cexp-mono1
ordLeq-csum2 Card-order-ctwo)
finally show ?thesis .
next
case False
with r have Cfinite r unfolding cinfinite-def cfinite-def by auto
hence Cfinite (r + c s) by (intro Cfinite-csum s)
hence (r + c s) c t ≤ o ctwo c t by (intro Cfinite-cexp-Cinfinite t)
also have ctwo c t ≤ o (r + c ctwo) c t using t
by (blast intro: cexp-mono1 ordLeq-csum2 Card-order-ctwo)
finally show ?thesis .
qed

lemma Cfinite-cardSuc: Cfinite r =⇒ Cfinite (cardSuc r)
by (simp add: cinfinite-def cardSuc-Card-order cardSuc-finite)

lemma cardSuc-UNION-Cinfinite:
assumes Cfinite r relChain (cardSuc r) As B ≤ (UN i ∈ Field (cardSuc r). As i) | B | ≤ o r
shows ∃ i ∈ Field (cardSuc r). B ≤ As i
using cardSuc-UNION assms unfolding cinfinite-def by blast

lemma Cfinite-card-suc: [ Cfinite r ; card-order r ] =⇒ Cfinite (card-suc r)

lemma card-suc-least: [[ card-order r ; Card-order s ; r <o s ] =⇒ card-suc r ≤o s
by (rule ordIso-ordLeq-trans[OF ordIso-symmetric[OF cardSuc-ordIso-card-suc]])
(auto intro!: cardSuc-least simp: card-order-on-Card-order)

lemma regularCard-cardSuc: Cfinite k =⇒ regularCard (cardSuc k)
by (rule infinite-cardSuc-regularCard) (auto simp: cinfinite-def)

lemma regularCard-card-suc: card-order r =⇒ Cfinite r =⇒ regularCard (card-suc r)
using cardSuc-ordIso-card-suc Cfinite-cardSuc regularCard-cardSuc regularCard-ordIso
32 Function Definition Base

theory Fun-Def-Base
imports Ctr-Sugar Set Wellfounded
begin

ML-file ⟨Tools/Function/function-lib.ML⟩
named-theorems termination-simp simplification rules for termination proofs
ML-file ⟨Tools/Function/function-common.ML⟩
ML-file ⟨Tools/Function/function-context-tree.ML⟩

attribute-setup fundef-cong =
  ⟨Attrib.add-del Function-Context-Tree.cong-add Function-Context-Tree.cong-del⟩
declaration of congruence rule for function definitions

ML-file ⟨Tools/Function/sum-tree.ML⟩

end

33 Definition of Bounded Natural Functors

theory BNF-Def
imports BNF-Cardinal-Arithmetic Fun-Def-Base
keywords
  print-bnfs :: diag and
  bnf :: thy-goal-defn
begin

lemma Collect-case-prodD: x ∈ Collect (case-prod A) ⇒ A (fst x) (snd x)
  by auto

inductive
  rel-sum :: (′a ⇒ ′c ⇒ bool) ⇒ (′b ⇒ ′d ⇒ bool) ⇒ ′a + ′b ⇒ ′c + ′d ⇒ bool
for R1 R2
where
  R1 a c ⇒ rel-sum R1 R2 (Inl a) (Inl c)
| R2 b d ⇒ rel-sum R1 R2 (Inr b) (Inr d)

definition
  rel-fun :: (′a ⇒ ′c ⇒ bool) ⇒ (′b ⇒ ′d ⇒ bool) ⇒ (′a ⇒ ′b) ⇒ (′c ⇒ ′d) ⇒ bool
where
  rel-fun A B = (λf g. ∀ x y. A x y ⇒ B (f x) (g y))

lemma rel-funI [intro]:
assumes $\forall x, y. A \ x \ y \Longrightarrow B \ (f \ x) \ (g \ y)$
shows rel-fun $A \ B \ f \ g$
using assms by (simp add: rel-fun-def)

lemma rel-funD:
assumes rel-fun $A \ B \ f \ g$ and $A \ x \ y$
shows $B \ (f \ x) \ (g \ y)$
using assms by (simp add: rel-fun-def)

lemma rel-fun-mono:
$[ \text{rel-fun} \ X \ A \ f \ g; \ \forall x, y. X \ x \ y \Longrightarrow X \ x \ y; \ \forall x, y. A \ x \ y \Longrightarrow B \ x \ y ] \Longrightarrow \text{rel-fun} \ Y \ B \ f \ g$
by (simp add: rel-fun-def)

lemma rel-fun-mono' [mono]:
$[ \forall x, y. X \ x \ y \Longrightarrow X \ x \ y; \ \forall x, y. A \ x \ y \Longrightarrow B \ x \ y ] \Longrightarrow \text{rel-fun} \ X \ A \ f \ g \Longrightarrow \text{rel-fun} \ Y \ B \ f \ g$
by (simp add: rel-fun-def)

definition rel-set :: $(\forall a \Rightarrow \forall b \Rightarrow \text{bool}) \Rightarrow (\forall a \Rightarrow \forall b \Rightarrow \text{bool})$
where rel-set $R = (\forall A, B. (\forall x \in A. \exists y \in B. R x y) \land (\forall y \in B. \exists x \in A. R x y))$

lemma rel-setI:
assumes $\forall x. x \in A \Longrightarrow \exists y \in B. R x y$
assumes $\forall y. y \in B \Longrightarrow \exists x \in A. R x y$
shows rel-set $R \ A \ B$
using assms unfolding rel-set-def by simp

lemma predicate2-transferD:
$[\text{rel-fun} \ R1 \ (\text{rel-fun} \ R2 \ (=)) \ P \ Q; a \in A; b \in B; A \subseteq \{(x, y). R1 \ x \ y\}; B \subseteq \{(x, y). R2 \ x \ y\}] \Longrightarrow P \ (\text{fst} \ a) \ (\text{fst} \ b) \longleftrightarrow Q \ (\text{snd} \ a) \ (\text{snd} \ b)$
unfolding rel-fun-def by (blast dest!: Collect-case-prodD)

definition collect where
collect $F \ x = (\bigcup f \in F. f \ x)$

lemma fstI: $x = (y, z) \Longrightarrow \text{fst} \ x = y$
by simp

lemma sndI: $x = (y, z) \Longrightarrow \text{snd} \ x = z$
by simp

lemma bijI': $\forall x, y. (f \ x = f \ y) = (x = y); \ \forall y. \exists x. y = f \ x] \Longrightarrow \text{bij} \ f$
unfolding bij-def inj-on-def by auto blast

definition Gr $A \ f = \{(a, f \ a) \mid a. \ a \in A\}$
definition  Grp A f = (λa b. b = f a ∧ a ∈ A)

definition  vimage2p where
  vimage2p f g R = (λx y. R (f x) (g y))

lemma  collect-comp: collect F o g = collect ((λf. f o g) ' F)
  by (rule ext) (simp add: collect-def)

definition  convol (⟨-/-⟩) where
  ⟨f, g⟩ ≡ λa. (f a, g a)

lemma  fst-convol: fst o ⟨f, g⟩ = f
  apply (rule ext)
  unfolding convol-def by simp

lemma  snd-convol: snd o ⟨f, g⟩ = g
  apply (rule ext)
  unfolding convol-def by simp

lemma  convol-mem-GrpI:
  x ∈ A ⇒ ⟨id, g⟩ x ∈ (Collect (case-prod (Grp A g)))
  unfolding convol-def Grp-def by auto

definition  csquare where
  csquare A f1 f2 p1 p2 ←→ (∀a ∈ A. f1 (p1 a) = f2 (p2 a))

lemma  eq-alt: (=) = Grp UNIV id
  unfolding Grp-def by auto

lemma  leq-conversepI: R = (=) ⇒ R ≤ R⁻¹⁻¹
  by auto

lemma  leq-OOI: R = (=) ⇒ R ≤ R OO R
  by auto

lemma  OO-Grp-alt: (Grp A f)⁻¹⁻¹ OO Grp A g = (λx y. \exists z. z ∈ A ∧ f z = x ∧
  g z = y)
  unfolding Grp-def by auto

lemma  Grp-UNIV-id: f = id ⇒ (Grp UNIV f)⁻¹⁻¹ OO Grp UNIV f = Grp
  UNIV f
  unfolding Grp-def by auto

lemma  Grp-UNIV-idI: x = y ⇒ Grp UNIV id x y
  unfolding Grp-def by auto

lemma  Grp-mono: A ≤ B ⇒ Grp A f ≤ Grp B f
  unfolding Grp-def by auto
THEORY “BNF-Def”

lemma GrpI: \[ f x = y; x \in A \] \implies Grp A f x y
  unfolding Grp-def by auto

lemma GrpE: Grp A f x y \implies (\[ f x = y; x \in A \] \implies R) \implies R
  unfolding Grp-def by auto

lemma Collect-case-prod-Grp-eqD: \[ z \in Collect (case-prod (Grp A f)) \] \implies (f \circ fst) z = snd z
  unfolding Grp-def comp-def by auto

lemma Collect-case-prod-Grp-in: \[ z \in Collect (case-prod (Grp A f)) \] \implies fst z \in A
  unfolding Grp-def comp-def by auto

definition pick-middlep P a c = (SOME b. P a b \land Q b c)

lemma pick-middlep:
  \( (P OO Q) a c \implies P a \ (pick-middlep P Q a c) \land Q \ (pick-middlep P Q a c) c \)
  unfolding pick-middlep-def by (rule someI-ex) auto

definition fstOp where
  \( \text{fstOp } P Q ac = (\text{fst } ac, \text{pick-middlep } P Q (\text{fst } ac) \ (\text{snd } ac)) \)

definition sndOp where
  \( \text{sndOp } P Q ac = (\text{pick-middlep } P Q (\text{fst } ac) \ (\text{snd } ac), \ (\text{snd } ac)) \)

lemma fstOp-in: \[ ac \in Collect (case-prod (P OO Q)) \] \implies \( \text{fstOp } P Q ac \in Collect \ (case-prod P) \)
  unfolding fstOp-def mem-Collect-eq
  by (subst (asm) surjective-pairing, unfold prod.case) (erule pick-middlep[THEN conjunct1])

lemma fst-fstOp:
  \( \text{fst } bc = (\text{fst } \circ \text{fstOp } P Q) bc \)
  unfolding comp-def fstOp-def by simp

lemma snd-sndOp: \( \text{snd } bc = (\text{snd } \circ \text{sndOp } P Q) bc \)
  unfolding comp-def sndOp-def by simp

lemma sndOp-in: \[ ac \in Collect (case-prod (P OO Q)) \] \implies \( \text{sndOp } P Q ac \in Collect \ (case-prod Q) \)
  unfolding sndOp-def mem-Collect-eq
  by (subst (asm) surjective-pairing, unfold prod.case) (erule pick-middlep[THEN conjunct2])

lemma csquare-fstOp-sndOp:
  \( \text{csquare } (\text{Collect } (f \ (P OO Q))) \ (\text{snd } (\text{fstOp } P Q) \ (\text{sndOp } P Q)) \)
  unfolding csquare-def fstOp-def sndOp-def using pick-middlep by simp

lemma snd-fst-flip:
  \( \text{snd } xy = (\text{fst } \circ (\% (x, y). \ (y, x))) xy \)
  by (simp split: prod.split)
lemma `fst-snd-flip: fst xy = (snd ∘ (%(x, y). (y, x))) xy`
  by (simp split: prod.split)

lemma `flip-pred: A ⊆ Collect (case-prod (R ^−1−1)) ⇒ (%(x, y). (y, x)) ' A ⊆ Collect (case-prod R)`
  by auto

lemma `predicate2-eqD: A = B ⇒ A a b ─→ B a b`
  by simp

lemma `case-sum-o-inj: case-sum f g ◦ Inl = f case-sum f g ◦ Inr = g`
  by auto

lemma `map-sum-o-inj: map-sum f g ◦ Inl = Inl ◦ f map-sum f g ◦ Inr = Inr ◦ g`
  by auto

lemma `card-order-csum-cone-cexp-def:
  card-order r ⇒ (|A1| + c cone) ^c r = |Func UNIV (Inl ' A1 ∪ {Inr ()})|
  unfolding cexp-def cone-def Field-csum Field-card-of by (auto dest: Field-card-order)

lemma `If-the-inv-into-in-Func:
  [ inj-on g C; C ⊆ B ∪ {x} ] ⇒ ((λi. if i ∈ g then the-inv-into C g i else x) ∈ Func UNIV (B ∪ {x})
  unfolding Func-def by (auto elim: the-inv-into-f-f)

lemma `the-inv-f-o-f-id:
  inj f ⇒ (the-inv f o f) z = id z`
  by (simp add: the-inv-f-f)

lemma `vimage2pI: R (f x) (g y) ⇒ vimage2p f g R x y`
  unfolding vimage2p-def .

lemma `rel-fun-iff-leq-vimage2p: (rel-fun R S) f g = (R ⊆ vimage2p f g S)`
  unfolding rel-fun-def vimage2p-def by auto

lemma `convol-image-vimage2p: (f o fst, g o snd) ' Collect (case-prod (vimage2p f g R)) ⊆ Collect (case-prod R)
  unfolding vimage2p-def convol-def by auto

lemma `vimage2p-Grp: vimage2p f g P = Grp UNIV f OO P OO (Grp UNIV g) ^−1−1`
  unfolding vimage2p-def Grp-def by auto

lemma `subst-Pair: P x y ⇒ a = (x, y) ⇒ P (fst a) (snd a)`
by simp

lemma comp-apply-eq: f (g x) = h (k x) \implies (f \circ g) x = (h \circ k) x
unfolding comp-apply by assumption

lemma refl-ge-eq: (\forall x. R x x) \implies (=) \leq R
by auto

lemma ge-eq-refl: (=) \leq R \implies R x x
by auto

lemma reflp-eq: reflp R = ((=) \leq R)
by (auto simp: reflp-def fun-eq-iff)

lemma transp-relcompp: transp r \iff r OO r \leq r
by (auto simp: transp-def)

lemma symp-conversep: symp R = (R^{-1}^{-1} \leq R)
by (auto simp: symp-def fun-eq-iff)

lemma diag-imp-eq-le: (\forall x. x \in A \implies R x x) \implies \forall x y. x \in A \implies y \in A \implies x = y \implies R x y
by blast

definition eq-onp :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool
where eq-onp R = (\lambda x y. R x \land x = y)

lemma eq-onp-Grp: eq-onp P = BNF-Def.Grp (Collect P) id
unfolding eq-onp-def Grp-def by auto

lemma eq-onp-to-eq: eq-onp P x y \implies x = y
by (simp add: eq-onp-def)

lemma eq-onp-top-eq-eq: eq-onp top = (=)
by (simp add: eq-onp-def)

lemma eq-onp-same-args: eq-onp P x x = P x
by (auto simp add: eq-onp-def)

lemma eq-onp-eqD: eq-onp P = Q \implies P x = Q x x
unfolding eq-onp-def by blast

lemma Ball-Collect: Ball A P = (A \subseteq (Collect P))
by auto

lemma eq-onp-mono0: \forall x\in A. P x \implies Q x \implies \forall x\in A. \forall y\in A. eq-onp P x y \implies eq-onp Q x y
unfolding eq-onp-def by auto
THEORY “BNF-Composition”

lemma eq-onp-True: eq-onp (\._. True) = (=)
  unfolding eq-onp-def by simp

lemma Ball-image-comp: Ball (f * A) g = Ball A (g \circ f)
  by auto

lemma rel-fun-Collect-case-prodD:
  rel-fun A B f g \implies X \subseteq Collect (case-prod A) \implies x \in X \implies B ((f \circ fst) x) ((g \\
  \circ snd) x)
  unfolding rel-fun-def by auto

lemma eq-onp-mono-iff: eq-onp P \leq eq-onp Q \iff P \leq Q
  unfolding eq-onp-def by auto

ML-file ⟨Tools/BNF/bnf-util.ML⟩
ML-file ⟨Tools/BNF/bnf-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-def-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-def.ML⟩

end

34 Composition of Bounded Natural Functors

theory BNF-Composition
imports BNF-Def
begin

lemma ssubst-mem: [t = s; s \in X] \implies t \in X
  by simp

lemma empty-natural: (\._. \{\}) \circ f = image g \circ (\._. \{\})
  by (rule ext) simp

lemma Cinfinite-gt-empty: Cinfinite r \implies |\{\}| <o r
  by (simp add: cinfinite-def finite-ordLess-infinite card-of-ordIso-finite Field-card-of \\
  card-of-well-order-on emptyI card-order-on-well-order-on)

lemma Union-natural: Union \circ image (image f) = image f \circ Union
  by (rule ext) (auto simp only: comp-apply)

lemma in-Union-o-assoc: x \in (Union \circ gset \circ gmap) A \implies x \in (Union \circ (gset \circ \
  gmap)) A
  by (unfold comp-assoc)

lemma regularCard-UNION-bound:
  assumes Cinfinite r regularCard r and |I| <o r \\bigwedge i. i \in I \implies |A i| <o r
  shows \bigcup i\in I. A i <o r
  using assms cinfinite-def regularCard-stable stable-UNION by blast
lemma comp-single-set-bd-strict:
assumes fbd: Cinfinite fbd regularCard fbd and
  gbd: Cinfinite gbd regularCard gbd and
  fset-bd: \(\forall x. |fset x| < o fbd\) and
  gset-bd: \(\forall x. |gset x| < o gbd\)
shows \(\bigcup (fset \cdot gset x) < o gbd * c fbd\)
proof (cases fbd < o gbd)
  case True
  then have \(|fset x| < o gbd for x\) using fset-bd ordLess-transitive by blast
  then have \(|\bigcup (fset \cdot gset x)| < o gbd\) using regularCard-UNION-bound[OF gbd gset-bd] by blast
  then have \(|\bigcup (fset \cdot gset x)| < o fbd * c gbd\) using ordLess-ordLeq-trans ordLeq-cprod2 gbd(1) fbd(1) cWeil-infinite-not-czero by blast
  then show \(?thesis using ordLess-ordIso-trans cprod-com by blast
next
  case False
  have Well-order fbd Well-order gbd using fbd(1) gbd(1) card-order-on-well-order-on by auto
  then have \(gbd \leq o fbd\) using not-ordLess-iff-ordLeq False by blast
  then have \(|gset x| < o fbd for x\) using gset-bd ordLess-ordLeq-trans by blast
  then have \(|\bigcup (fset \cdot gset x)| < o fbd\) using regularCard-UNION-bound[OF fbd fset-bd] by blast
  then show \(?thesis using ordLess-ordLeq-trans ordLeq-cprod2 gbd(1) fbd(1) cWeil-infinite-not-czero by blast
qed

lemma comp-single-set-bd:
assumes fbd: Card-order: Card-order fbd and
  fset-bd: \(\forall x. |fset x| \leq o fbd\) and
  gset-bd: \(\forall x. |gset x| \leq o gbd\)
shows \(|\bigcup (fset \cdot gset x)| \leq o gbd * c fbd\)
apply simp
apply (rule ordLeq-transitive)
apply (rule card-of-UNION-Sigma)
apply (rule subst SIGMA-CSUM)
apply (rule ordLeq-transitive)
apply (rule card-of-Csum-Times')
apply (rule fbd:Card-order)
apply (rule ballI)
apply (rule fset-bd)
apply (rule ordLeq-transitive)
apply (rule cprod-mono1)
apply (rule gset-bd)
apply (rule ordIso-imp-ordLeq)
apply (rule ordIso-refl)
apply (rule Card-order-cprod)
done
lemma csum-dup: cinfinite r ==> Card-order r ==> p + c p' = o r + c r ==> p + c p' = o r
apply (erule ordIso-transitive)
apply (frule csum-absorb2')
apply (erule ordLeq-refl)
by simp

lemma cprod-dup: cinfinite r ==> Card-order r ==> p * c p' = o r * c r ==> p * c p' = o r
apply (erule ordIso-transitive)
apply (rule cprod-infinite)
by simp

lemma Union-image-insert: \( \bigcup (f \cdot \text{insert} \ a \ \mathcal{B}) = f \ a \cup \bigcup (f \cdot \mathcal{B}) \)
by simp

lemma Union-image-empty: A \cup \bigcup (f \cdot \{\}) = A
by simp

lemma image-o-collect: collect ((\lambda f. \text{image} \ g \circ f) \cdot F) = \text{image} \ g \circ \text{collect} \ F
by (rule ext) (auto simp add: collect-def)

lemma conj-subset-def: \( A \subseteq \{x. \ P \ x \land Q \ x\} = (A \subseteq \{x. P \ x\} \land A \subseteq \{x. Q \ x\}) \)
by blast

lemma UN-image-subset: \( \bigcup (f \cdot g \ x) \subseteq X = (g \ x \subseteq \{x. f \ x \subseteq X\}) \)
by blast

lemma comp-set-bd-Union-o-collect: \( |\bigcup \bigcup ((\lambda f. f \ x \cdot X))| \leq o \ hbd \Longrightarrow |(\bigcup \circ \text{collect} \ X) \ x| \leq o \ hbd \)
by (unfold comp-apply collect-def) simp

lemma comp-set-bd-Union-o-collect-strict: \( |\bigcup \bigcup ((\lambda f. f \ x \cdot X))| < o \ hbd \Longrightarrow |(\bigcup \circ \text{collect} \ X) \ x| < o \ hbd \)
by (unfold comp-apply collect-def) simp

lemma Collect-inj: Collect P = Collect Q ==> P = Q
by blast

lemma Grp-fst-snd: (Grp (Collect (case-prod R)) f\text{st})^{-1}^{-1} OO Grp (Collect (case-prod R)) \text{snd} = R
unfolding Grp-def fun-eq-iff relcompp.simps by auto

lemma OO-Grp-cong: A = B ==> (Grp A f)^{-1}^{-1} OO Grp A g = (Grp B f)^{-1}^{-1} OO Grp B g
by (rule arg-cong)

lemma vimage2p-relcompp-mono: R OO S \leq T ==> vimage2p f g R OO vimage2p g h S \leq vimage2p f h T
unfolding \texttt{vimage2p-def} by \texttt{auto}

\textbf{lemma} \texttt{type-copy-map-cong0}: \( M (g \ x) = N (h \ x) \implies (f \circ M \circ g) \ x = (f \circ N \circ h) \ x \)
  by \texttt{auto}

\textbf{lemma} \texttt{type-copy-set-bd}: \((\\forall y. |S \ y| < o \ bd) \implies |(S \circ Rep) \ x| < o \ bd \)
  by \texttt{auto}

\textbf{lemma} \texttt{vimage2p-cong}: \( R = S \implies \text{vimage2p} \ f \ g \ R = \text{vimage2p} \ f \ g \ S \)
  by \texttt{simp}

\textbf{context}
\textit{fixes} \(\text{Rep} \ \text{Abs}\)
\textit{assumes} \texttt{type-copy: type-definition Rep Abs UNIV}
\textit{begin}

\textbf{lemma} \texttt{type-copy-map-id0}: \( M = id \implies Abs \circ M \circ Rep = id \)
  \textit{using} \texttt{type-definition.Rep-inverse[OF type-copy]} \texttt{by auto}

\textbf{lemma} \texttt{type-copy-map-comp0}: \( M = M1 \circ M2 \implies f \circ M \circ g = (f \circ M1 \circ Rep) \circ (\Abs \circ M2 \circ g) \)
  \textit{using} \texttt{type-definition.Abs-inverse[OF type-copy UNIV-I]} \texttt{by auto}

\textbf{lemma} \texttt{type-copy-set-map0}: \( S \circ M = \text{image} f \circ S' \implies (S \circ Rep) \circ (Abs \circ M \circ g) = \text{image} f \circ (S' \circ g) \)
  \textit{using} \texttt{type-definition.Abs-inverse[OF type-copy UNIV-I]} \texttt{by (auto simp: o-def fun-eq-iff)}

\textbf{lemma} \texttt{type-copy-wit}: \( x \in (S \circ Rep) \ (Abs \ y) \implies x \in S \ y \)
  \textit{using} \texttt{type-definition.Abs-inverse[OF type-copy UNIV-I]} \texttt{by auto}

\textbf{lemma} \texttt{type-copy-vimage2p-Grp-Rep}: \( \text{vimage2p} \ f \ Rep \ (\text{Grp} \ (\text{Collect} \ P) \ h) = \text{Grp} \ (\text{Collect} \ (\lambda x. \ P \ (f \ x))) \ (Abs \circ h \circ f) \)
  \textit{unfolding} \texttt{vimage2p-def Grp-def fun-eq-iff}

\textbf{lemma} \texttt{type-copy-vimage2p-Grp-Abs}:
\( \\forall h. \text{vimage2p} \ g \ Abs \ (\text{Grp} \ (\text{Collect} \ P) \ h) = \text{Grp} \ (\text{Collect} \ (\lambda x. \ P \ (g \ x))) \ (Rep \circ h \circ g) \)
  \textit{unfolding} \texttt{vimage2p-def Grp-def fun-eq-iff}
  \textit{by (auto simp: type-definition.Abs-inverse[OF type-copy UNIV-I]}
lemma type-copy-ex-RepI: (∃ b. F b) = (∃ b. F (Rep b))
proof safe
fix b assume F b
show ∃ b'. F (Rep b')
proof (rule exI)
from F b show F (Rep (Abs b)) using type-definition.Abs-inverse[OF type-copy]
by auto
qed
qed blast

lemma vimage2p-relcompp-converse:
vimage2p f g (R⁻¹⁻¹ OO S) = (vimage2p Rep f R)⁻¹⁻¹ OO vimage2p Rep g S
unfolding vimage2p-def relcompp.simps conversep.simps fun-eq-iff image-def
by (auto simp: type-copy-ex-RepI)

end

bnf DEADID: 'a
map: id :: 'a ⇒ 'a
bd: natLeq
rel: (=) :: 'a ⇒ 'a ⇒ bool
by (auto simp add: natLeq-card-order natLeq-cinfinite regularCard-natLeq)
definition id-bnf :: 'a ⇒ 'a where
id-bnf ≡ (λx. x)

lemma id-bnf-apply: id-bnf x = x
unfolding id-bnf-def by simp

bnf ID: 'a
map: id-bnf :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b
sets: λx. {x}
bd: natLeq
rel: id-bnf :: ('a ⇒ 'b ⇒ bool) ⇒ 'a ⇒ 'b ⇒ bool
pred: id-bnf :: ('a ⇒ bool) ⇒ 'a ⇒ bool
unfolding id-bnf-def
apply (auto simp: Grp-def fun-eq-iff relcompp.simps natLeq-card-order natLeq-cinfinite
regularCard-natLeq)
apply (rule finite-ordLess-infinite[OF natLeq-Well-order])
apply (auto simp add: Field-card-of Field-natLeq card-of-well-order-on)[3]
done

lemma type-definition-id-bnf-UNIV: type-definition id-bnf id-bnf UNIV
unfolding id-bnf-def by unfold-locales auto

ML-file ⟨Tools/BNF/bnf-comp-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-comp.ML⟩
hide-fact
DEADID.inj-map DEADID.inj-map-strong DEADID.map-comp DEADID.map-cong
DEADID.map-cong0
DEADID.map-cong-simp DEADID.map-id DEADID.map-id0 DEADID.map-ident
DEADID.map-transfer
DEADID.rel-Grp DEADID.rel-compp DEADID.rel-compp-Grp DEADID.rel-conversep
DEADID.rel-eq
DEADID.rel-flip DEADID.rel-map DEADID.rel-mono DEADID.rel-transfer
ID.inj-map ID.inj-map-strong ID.map-comp ID.map-cong ID.map-cong0 ID.map-cong-simp
ID.map-id
ID.map-id0 ID.map-ident ID.map-transfer ID.rel-Grp ID.rel-compp ID.rel-compp-Grp
ID.rel-conversep
ID.rel-eq ID.rel-flip ID.rel-map ID.rel-mono ID.rel-transfer ID.set-map ID.set-transfer

end

35 Registration of Basic Types as Bounded Natural Functors

theory Basic-BNFs
imports BNF-Def
begin

inductive-set setl :: ’a + ’b ⇒ ’a set for s :: ’a + ’b where
s = Inl x ⇒ x ∈ setl s

inductive-set setr :: ’a + ’b ⇒ ’b set for s :: ’a + ’b where
s = Inr x ⇒ x ∈ setr s

lemma sum-set-defs[code]:
setl = (λx. case x of Inl z ⇒ {z} | - ⇒ { })
setr = (λx. case x of Inr z ⇒ {z} | - ⇒ { })
by (auto simp: fun-eq-iff intro: setl.intros setr.intros elim: setl.cases setr.cases
split: sum.splits)

lemma rel-sum-simps[code, simp]:
rel-sum R1 R2 (Inl a1) (Inl b1) = R1 a1 b1
rel-sum R1 R2 (Inl a1) (Inr b2) = False
rel-sum R1 R2 (Inr a2) (Inl b1) = False
rel-sum R1 R2 (Inr a2) (Inr b2) = R2 a2 b2
by (auto intro: rel-sum.intros elim: rel-sum.cases)

inductive
pred-sum :: (’a ⇒ bool) ⇒ (’b ⇒ bool) ⇒ ’a + ’b ⇒ bool for P1 P2
where
P1 a ⇒ pred-sum P1 P2 (Inl a)
| P2 b ⇒ pred-sum P1 P2 (Inr b)
lemma pred-sum-inject[code, simp]:
  pred-sum P1 P2 (Inl a) \iff P1 a
  pred-sum P1 P2 (Inr b) \iff P2 b
  by (simp add: pred-sum.simps+)

bnf 'a + 'b
  map: map-sum
  sets: setl setr
  bd: natLeq
  wits: Inl Inr
  rel: rel-sum
  pred: pred-sum

proof –
  show map-sum id id = id by (rule map-sum.id)
next
  fix f1 :: 'o \Rightarrow 's and f2 :: 'p \Rightarrow 't
  and g1 :: 's \Rightarrow 'q and g2 :: 't \Rightarrow 'r
  show map-sum (g1 \circ f1) (g2 \circ f2) = map-sum g1 g2 \circ map-sum f1 f2
    by (rule map-sum.comp[symmetric])
next
  fix x and f1 :: 'o \Rightarrow 'q and f2 :: 'p \Rightarrow 'r
  assume a1: \\( \forall z. z \in \text{setl } x \Rightarrow f1 z = g1 z \)
  and a2: \\( \forall z. z \in \text{setr } z \Rightarrow f2 z = g2 z \)
  thus map-sum f1 f2 x = map-sum g1 g2 x
  proof (cases x)
    case Inl thus ?thesis using a1 by (clarsimp simp: sum-set-defs(1))
next
    case Inr thus ?thesis using a2 by (clarsimp simp: sum-set-defs(2))
  qed
next
  fix f1 :: 'o \Rightarrow 'q and f2 :: 'p \Rightarrow 'r
  show setl \circ map-sum f1 f2 = image f1 \circ setl
    by (rule ext, unfold o-apply) (simp add: sum-set-defs(1) split: sum.split)
next
  fix f1 :: 'o \Rightarrow 'q and f2 :: 'p \Rightarrow 'r
  show setr \circ map-sum f1 f2 = image f2 \circ setr
    by (rule ext, unfold o-apply) (simp add: sum-set-defs(2) split: sum.split)
next
  show card-order natLeq by (rule natLeq-card-order)
next
  show cinfinite natLeq by (rule natLeq-cinfinite)
next
  show regularCard natLeq by (rule regularCard-natLeq)
next
  fix x :: 'o + 'p
  show \(|\text{setl } x| < o \) natLeq
    apply (rule finite-iff-ordLess-natLeq[THEN iffD1])
    by (simp add: sum-set-defs(1) split: sum.split)
next
  fix x :: 'o + 'p
show setr z <o natLeq
apply (rule finite-iffLess-natLess[THEN iffD1])
by (simp add: sum-set-defs(2) split: sum.split)

next
fix R1 R2 S1 S2
show rel-sum R1 R2 OO rel-sum S1 S2 ≤ rel-sum (R1 OO S1) (R2 OO S2)
by (force elim: rel-sum.cases)

next
fix R S
show rel-sum R S = (λx y. ∃ z. (setl z ⊆ {x, y}. R x y) ∧ setr z ⊆ {x, y}. S x y) ∧
map-sum fst fst z = x ∧ map-sum snd snd z = y)
unfolding sum-set-defs relcompp.simps conversep.simps fun-eq-iff
by (fastforce elim: rel-sum.cases split: sum.splits)

qed (auto simp: sum-set-defs fun-eq-iff pred-sum.simps split: sum.splits)

inductive-set fsts :: 'a × 'b ⇒ 'a set
for p :: 'a × 'b where
fst p ∈ fsts p

inductive-set snds :: 'a × 'b ⇒ 'b set
for p :: 'a × 'b where
snd p ∈ snds p

lemma prod-set-defs[code]: fsts = (λp. {fst p}) snds = (λp. {snd p})
by (auto intro: fsts.intros snds.intros elim: fsts_cases snds_cases)

inductive rel-prod :: ('a ⇒ 'b ⇒ bool) ⇒ ('c ⇒ 'd ⇒ bool) ⇒ 'a × 'c ⇒ 'b × 'd ⇒ bool
for R1 R2
where
[R1 a b; R2 c d] ==> rel-prod R1 R2 (a, c) (b, d)

inductive pred-prod :: ('a ⇒ bool) ⇒ ('b ⇒ bool) ⇒ 'a × 'b ⇒ bool
for P1 P2
where
[P1 a; P2 b] ==> pred-prod P1 P2 (a, b)

lemma rel-prod-inject [code, simp]:
rel-prod R1 R2 (a, b) (c, d) ←→ R1 a c ∧ R2 b d
by (auto intro: rel-prod.intros elim: rel-prod_cases)

lemma pred-prod-inject [code, simp]:
pred-prod P1 P2 (a, b) ←→ P1 a ∧ P2 b
by (auto intro: pred-prod.intros elim: pred-prod_cases)

lemma rel-prod-conv:
rel-prod R1 R2 = (λ(a, b) (c, d). R1 a c ∧ R2 b d)
by force

definition
pred-fun :: ('a ⇒ bool) ⇒ ('b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ bool
where
pred-fun $A B = (\lambda f. \forall x. A x \to B (f x))$

lemma pred-funI: $(\forall x. A x \equiv B (f x)) \equiv pred-fun A B f$
unfolding pred-fun-def by simp

bnf 'a × 'b
map: map-prod
sets: fsts snds
bd: natLeq
rel: rel-prod
pred: pred-prod

proof (unfold prod-set-defs)
show map-prod id id = id by (rule map-prod.id)
next
fix $f_1 f_2 g_1 g_2$
show map-prod $(g_1 \circ f_1) (g_2 \circ f_2) = map-prod g_1 g_2 \circ map-prod f_1 f_2$
  by (rule map-prod.comp[symmetric])
next
fix $x f_1 f_2 g_1 g_2$
assume $\forall z. z \in \{fst x\} \equiv f_1 z = g_1 z \land z \in \{snd x\} \equiv f_2 z = g_2 z$
thus map-prod $f_1 f_2 x = map-prod g_1 g_2 x$ by (cases x) simp
next
fix $f_1 f_2$
show $(\lambda x. \{fst x\}) \circ map-prod f_1 f_2 = image f_1 \circ (\lambda x. \{fst x\})$
  by (rule ext, unfold o-apply) simp
next
fix $f_1 f_2$
show $(\lambda x. \{snd x\}) \circ map-prod f_1 f_2 = image f_2 \circ (\lambda x. \{snd x\})$
  by (rule ext, unfold o-apply) simp
next
show card-order natLeq by (rule natLeq-card-order)
next
show cinfinite natLeq by (rule natLeq-cinfinite)
next
show regularCard natLeq by (rule regularCard-natLeq)
next
fix $x$
show $|\{fst x\}| o natLeq$
  by (simp add: finite-iff-ordLess-natLeq[symmetric])
next
fix $x$
show $|\{snd x\}| o natLeq$
  by (simp add: finite-iff-ordLess-natLeq[symmetric])
next
fix $R_1 R_2 S_1 S_2$
show rel-prod $R_1 R_2 OO rel-prod S_1 S_2 \leq rel-prod (R_1 OO S_1) (R_2 OO S_2)$
  by auto
next
fix $R S$

show $\text{rel-prod } R S = (\lambda x y. \exists z. ((\text{fst } z) \subseteq \{(x, y). R x y\} \land (\text{snd } z) \subseteq \{(x, y). S x y\}) \land\n\text{map-prod } \text{fst} \text{ fst } z = x \land \text{map-prod } \text{snd} \text{ snd } z = y)$

unfolding $\text{prod-set-defs rel-prod-inject relcommp..simps conversep..simps fun-eq-iff}$

by auto

qed

lemma $\text{card-order-bd-fun: card-order } (\text{natLeq } + c \text{ card-suc } (|\text{UNIV}|))$

by (auto simp: card-order-csum natLeq-card-order card-order-card-suc card-of-card-order-on)

lemma $\text{Cinfinite-bd-fun: } \text{Cinfinite } (\text{natLeq } + c \text{ card-suc } (|\text{UNIV}|))$

by (auto simp: Cinfinite-csum natLeq-Cinfinite)

lemma $\text{regularCard-bd-fun: regularCard } (\text{natLeq } + c \text{ card-suc } (|\text{UNIV}|))$

(is $\text{regularCard } (- + c \text{ card-suc } ?U))$

proof (cases $\text{Cinfinite } ?U$)

  case True

  then show $\text{thesis}$

  by (intro regularCard-csum natLeq-Cinfinite Cinfinite-card-suc
      card-of-card-order-on regularCard-natLeq regularCard-card-suc)

next

  case False

  have $\text{card-suc } ?U \leq_o \text{natLeq}$

  unfolding $\text{cinfinite-def Field-card-of}$

  by (intro card-suc-least;

  then have $\text{natLeq } = o \text{natLeq } + c \text{ card-suc } ?U$

  using natLeq-Cinfinite csum-absorb1 ordIso-symmetric by blast

  then show $\text{thesis}$

  by (intro regularCard-ordIso[OF - natLeq-Cinfinite regularCard-natLeq])

qed

lemma $\text{ordLess-bd-fun: } |\text{UNIV}::'a set| < o \text{natLeq } + c \text{ card-suc } (|\text{UNIV}::'a set|)$

(is $- < o (- + c \text{ card-suc } (?U ::= 'a rel))))$

proof (cases $\text{Cinfinite } ?U$)

  case True

  have $\text{?U < o card-suc } ?U$ using card-of-card-order-on natLeq-card-order card-suc-greater

  by blast

  also have $\text{card-suc } ?U = o \text{natLeq } + c \text{ card-suc } ?U$ by (rule csum-absorb2[THEN ordIso-symmetric])

  (auto simp: True card-of-card-order-on intro!: Cinfinite-card-suc natLeq-ordLeq-Cinfinite)

  finally show $\text{thesis}$.

next

  case False

  then have $\text{?U < o natLeq}$

  by (auto simp: cinfinite-def Field-card-of card-of-card-order-on finite-iff-ordLess-natLeq[symmetric])

  then show $\text{thesis}$

  by (rule ordLess-ordLeq-trans[OF - natLeq-csum1[OF natLeq-Card-order]])
THEORY “BNF-Fixpoint-Base”

qed

bnf 'a ⇒ 'b
  map: (◦)
  sets: range
  bd: natLeq + c card-suc ( |UNIV::'a set| )
  rel: rel-fun (=)
  pred: pred-fun (λ-. True)

proof
  fix f show id o f = id f by simp
next
  fix f g show (◦) (g o f) = (◦) g o (◦) f
    unfolding comp-def[abs-def] ..
next
  fix x f g
  assume ⋀ z. z ∈ range x =⇒ f z = g z
  thus f o x = g o x by auto
next
  fix f show range o (◦) f = (◦) f o range
    by (auto simp add: fun-eq-iff)
next
  show card-order (natLeq + c card-suc ( |UNIV| ))
    by (rule card-order-bd-fun)
next
  show cinfinite (natLeq + c card-suc ( |UNIV| ))
    by (rule Cinfinite-bd-fun[THEN conjunct1])
next
  show regularCard (natLeq + c card-suc ( |UNIV| ))
    by (rule regularCard-bd-fun)
next
  fix f :: 'd ⇒ 'a
  show |range f| <o natLeq + c card-suc |UNIV :: 'd set|
    by (rule ordLeq-ordLess-trans[OF card-of-image ordLess-bd-fun])
next
  fix R S
  show rel-fun ( = ) R OO rel-fun ( = ) S ≤ rel-fun ( = ) (R OO S) by (auto simp: rel-fun-def)
next
  fix R
  show rel-fun ( = ) R = (λx y. ∃ z. range z ⊆ { (x, y). R x y } ∧ fst o z = x ∧ snd o z = y)
    unfolding rel-fun-def subset-iff by (force simp: fun-eq-iff[symmetric])
qed (auto simp: pred-fun-def)

end
36 Shared Fixpoint Operations on Bounded Natural Functors

theory BNF-Fixpoint-Base
imports BNF-Composition Basic-BNFs
begin

lemma conj-imp-eq-imp-imp: \((P \land Q \implies PROP R) \equiv (P \implies Q \implies PROP R)\)
  by standard simp-all

lemma predicate2D-conj: \(P \leq Q \land R \implies R \land (P x y \rightarrow Q x y)\)
  by blast

lemma eq-sym-Unity-conv: \((x = (() = ())) = x\)
  by blast

lemma case-unit-Unity: \((case u of () \Rightarrow f) = f\)
  by (cases u) (hypsubst, rule unit.case)

lemma case-prod-Pair-iden: \((case p of (x, y) \Rightarrow (x, y)) = p\)
  by simp

lemma unit-all-impI: \((P () = \Rightarrow Q ()) \Rightarrow \forall x. P x \rightarrow Q x\)
  by simp

lemma pointfree-idE: \(f \circ g = \text{id} = \Rightarrow f (g x) = x\)
  unfolding comp-def fun-eq-iff by simp

lemma o-bij:
  assumes gf: \(g \circ f = \text{id} \) and fg: \(f \circ g = \text{id}\)
  shows bij f
  unfolding bij-def inj-on-def surj-def proof safe
  fix a1 a2 assume f a1 = f a2
  hence \((f a1) = g (f a2)\) by simp
  thus a1 = a2 using gf unfolding fun-eq-iff by simp
next
  fix b
  have \(b = f (g b)\)
  using fg unfolding fun-eq-iff by simp
  thus \(\exists a. b = f a\) by blast
qed

lemma case-sum-step:
  case-sum \((\text{case-sum } f' g') g (\text{Inl } p) = \text{case-sum } f' g' p\)
  case-sum f \((\text{case-sum } f' g') (\text{Inr } p) = \text{case-sum } f' g' p\)
  by auto

lemma obj-one-pointE: \(\forall x. s = x \rightarrow P \implies P\)
  by blast
lemma type-copy-obj-one-point-absE:
assumes type-definition Rep Abs UNIV \(\forall x. s = Abs x \rightarrow P\)
shows \(P\)
proof
  fix x from assms show \(s = f x \rightarrow P\) by (cases x) auto
qed

lemma obj-sumE-f:
assumes \(\forall x. s = f (Inl x) \rightarrow P\) \(\forall x. s = f (Inr x) \rightarrow P\)
shows \(\forall x. s = f x \rightarrow P\)
proof
  case-sum \(f\) \(g\)
by auto

lemma case-sum-if:
  case-sum \(f\) \(g\) \((\text{if } p \text{ then } Inl x \text{ else } Inr y)\) = \((\text{if } p \text{ then } f x \text{ else } g y)\)
by simp

lemma prod-set-simps[simp]:
  \(\text{fsts} \ (x, y) = \{x\}\)
  \(\text{snds} \ (x, y) = \{y\}\)
unfolding prod-set-defs by simp+

lemma sum-set-simps[simp]:
  \(\text{setl} \ (\text{Inl} x) = \{x\}\)
  \(\text{setl} \ (\text{Inr} x) = \{\}\)
  \(\text{setr} \ (\text{Inl} x) = \{\}\)
  \(\text{setr} \ (\text{Inr} x) = \{x\}\)
unfolding sum-set-defs by simp+

lemma Inl-Inr-False: \((\text{Inl} x = \text{Inr} y)\) = False
by simp

lemma Inr-Inl-False: \((\text{Inr} x = \text{Inl} y)\) = False
by simp

lemma spec2: \(\forall x y. P x y \Rightarrow P x y\)
by blast

lemma rewriteR-comp-comp: \([g \circ h = r]\) \(\Rightarrow f \circ g \circ h = f \circ r\)
unfolding comp-def fun-eq-iff by auto

lemma rewriteR-comp-comp2: \([g \circ h = r1 \circ r2; f \circ r1 = l]\) \(\Rightarrow f \circ g \circ h = l \circ r2\)
unfolding comp-def fun-eq-iff by auto

lemma rewriteL-comp-comp: \([f \circ g = l]\) \(\Rightarrow f \circ (g \circ h) = l \circ h\)
unfolding comp-def fun-eq-iff by auto

lemma rewriteL-comp-comp2: \([f \circ g = l1 \circ l2; l2 \circ h = r]\) \(\Rightarrow f \circ (g \circ h) = l1\)
unfolding \texttt{comp-def fun-eq-iff} by \texttt{auto}

\textbf{lemma convol-o:} \( (f, g) \circ h = (f \circ h, g \circ h) \)

unfolding \texttt{convol-def} by \texttt{auto}

\textbf{lemma map-prod-o-convol:} \( \text{map-prod } h1 h2 \circ (f, g) = (h1 \circ f, h2 \circ g) \)

unfolding \texttt{convol-def} by \texttt{auto}

\textbf{lemma map-prod-o-convol-id:} \( \text{map-prod } f \text{id} \circ (\text{id}, g) \) \( x = (\text{id} \circ f, g) \) \( x \)

unfolding \texttt{map-prod-o-convol id-comp comp-id} ..

\textbf{lemma o-case-sum:} \( h \circ \text{case-sum } f g = \text{case-sum } (h \circ f) (h \circ g) \)

unfolding \texttt{convol-def} by \texttt{(auto split: sum.splits)}

\textbf{lemma case-sum-o-map-sum:} \( \text{case-sum } f g \circ \text{map-sum } h1 h2 = \text{case-sum } (f \circ h1) (g \circ h2) \)

unfolding \texttt{comp-def} by \texttt{(auto split: sum.splits)}

\textbf{lemma case-sum-o-map-sum-id:} \( \text{case-sum } \text{id } g \circ \text{map-sum } f \text{id} \) \( x = \text{case-sum } (f \circ \text{id}) g x \)

unfolding \texttt{case-sum-o-map-sum id-comp comp-id} ..

\textbf{lemma rel-fun-def-butlast:}

\texttt{rel-fun } R \texttt{(rel-fun S T) f g = } (\forall x y. R x y \rightarrow (\text{rel-fun S T) (f x) (g y)})

unfolding \texttt{rel-fun-def} ..

\textbf{lemma subst-eq-imp:} \( (\forall a b. a = b \rightarrow P a b) \equiv (\forall a. P a a) \)

by \texttt{auto}

\textbf{lemma eq-subset:} \( (\equiv) \leq (\lambda a b. P a b \lor a = b) \)

by \texttt{auto}

\textbf{lemma eq-le-Grp-id-iff:} \( ((\equiv) \leq \text{Grp } (\text{Collect } R) \text{id}) = (\text{All } R) \)

unfolding \texttt{Grp-def id-apply} by \texttt{blast}

\textbf{lemma Grp-id-mono-subst:} \( (\forall x y. \text{Grp } P \text{id} x y \Rightarrow \text{Grp } Q \text{id} (f x) (f y)) \equiv \\
(\forall x. x \in P \Rightarrow f x \in Q) \)

unfolding \texttt{Grp-def} by \texttt{rule auto}

\textbf{lemma vimage2p-mono:} \( \text{vimage2p } f g R x y \Rightarrow R \leq S \Rightarrow \text{vimage2p } f g S x y \)

unfolding \texttt{vimage2p-def} by \texttt{blast}

\textbf{lemma vimage2p-refl:} \( (\forall x. R x x) \Rightarrow \text{vimage2p } f f R x x \)

unfolding \texttt{vimage2p-def} by \texttt{auto}

\textbf{lemma}

\texttt{assumes type-definition Rep Abs UNIV}

\texttt{shows type-copy-Rep-o-Abs: Rep \circ Abs = id and type-copy-Abs-o-Rep: Abs \circ Rep}
unfolding fun-eq-iff comp-apply id-apply

lemma type-copy-map-comp0-undo:
  assumes type-definition Rep Abs UNIV
  type-definition Rep' Abs' UNIV
  type-definition Rep'' Abs'' UNIV
  shows Abs' 0 M 0 Rep'' = (Abs' 0 M 1 0 Rep) 0 (Abs 0 M2 0 Rep) 0 M1 0 M2 = M
  by (rule sgm) (auto simp: fun-eq-iff type-definition.Abs-inject OF assms
    UNIV-I UNIV-I)
  type-definition.Abs-inverse|OF assms|UNIV-I
  type-definition.Abs-inverse|OF assms 1|UNIV-I
  dest: spec[of - Abs'' x for x])

lemma vimage2p-id: vimage2p id id R = R
  unfolding vimage2p-def by auto

lemma vimage2p-comp: vimage2p (f1 0 f2) (g1 0 g2) = vimage2p f2 g2 0 vimage2p f1 g1
  unfolding fun-eq-iff vimage2p-def o-apply by simp

lemma vimage2p-rel-fun: rel-fun (vimage2p f g R) R f g
  unfolding rel-fun-def vimage2p-def by auto

lemma fun-cong-unused-0: f = (λx. g) 0 f (λx. 0) = g
  by (erule arg-cong)

lemma inj-on-convol-ident: inj-on (λx. (x, f x)) X
  unfolding inj-on-def by simp

lemma map-sum-if-distrib-then:
  ⋀ g e x y. map-sum f g (if e then Inl x else y) = (if e then Inl (f x) else map-sum f g y)
  ⋀ g e x y. map-sum f g (if e then Inl x else y) = (if e then Inl (g x) else map-sum f g y)
  by simp-all

lemma map-sum-if-distrib-else:
  ⋀ g e x y. map-sum f g (if e then x else Inl y) = (if e then map-sum f g x else
    Inl (f y))
  ⋀ g e x y. map-sum f g (if e then x else Inl y) = (if e then map-sum f g x else Inr (g y))
  by simp-all

lemma case-prod-app: case-prod f x y = case-prod (λl r. f l r y) x
  by (cases x) simp
lemma case-sum-map-sum: case-sum l r (map-sum f g x) = case-sum (l o f) (r o g) x by (cases x) simp-all


lemma case-prod-map-prod: case-prod h (map-prod f g x) = case-prod (λ l r. h (f l) (g r)) x by (cases x) simp-all

lemma case-prod-o-map-prod: case-prod f ◦ map-prod g1 g2 = case-prod (λ l r. f (g1 l) (g2 r)) unfolding comp-def by auto

lemma case-prod-transfer: (rel-fun (rel-fun A (rel-fun B C)) (rel-fun (rel-prod A B) C)) case-prod case-prod unfolding rel-fun-def by simp

lemma eq-ifI: (P → t = u1) ⇒ (¬ P → t = u2) ⇒ t = (if P then u1 else u2) by simp

lemma comp-transfer: rel-fun (rel-fun B C) (rel-fun (rel-fun A B) (rel-fun A C)) (◦) (◦) unfolding rel-fun-def by simp

lemma If-transfer: rel-fun (=) (rel-fun A (rel-fun A A)) If If unfolding rel-fun-def by simp


lemma Inl-transfer: rel-fun S (rel-sum S T) Inl Inl by auto

lemma Inr-transfer: rel-fun T (rel-sum S T) Inr Inr by auto

lemma Pair-transfer: rel-fun A (rel-fun B (rel-prod A B)) Pair Pair
unfolding rel-fun-def by simp

lemma eq-onp-live-step: x = y ⟹ eq-onp P a a ∧ x ⟷ P a ∧ y
  by (simp only: eq-onp_same-args)

lemma top-conj: top x ∨ P ⟷ P P ∨ top x ⟷ P
  by blast

lemma fst-convol': fst ((f, g) x) = f x
  using fst-convol unfolding convol-def by simp

lemma snd-convol': snd ((f, g) x) = g x
  using snd-convol unfolding convol-def by simp

lemma convol-expand-snd: fst ∘ f = g ⟹ ⟨g, snd ∘ f⟩ = f
  unfolding convol-def by auto

lemma convol-expand-snd':
  assumes (fst ∘ f = g)
  shows h = snd ∘ f ⟷ ⟨g, h⟩ = f
  proof
    from assms have *: ⟨g, snd ∘ f⟩ = f by (rule convol-expand-snd)
    then have h = snd ∘ f ⟷ h = snd ∘ ⟨g, snd ∘ f⟩ by simp
    moreover have .. ⟷ h = snd ∘ f by (simp add: snd-convol)
    moreover have .. ⟷ ⟨g, h⟩ = f by (subst (2) *[symmetric]) (auto simp: convol-def fun-eq-iff)
    ultimately show ?thesis by simp
  qed

lemma case-sum-expand-Inr-pointfree: f ∘ Inl = g ⟹ case-sum g (f ∘ Inr) = f
  by (auto split: sum.splits)

lemma case-sum-expand-Inr': f ∘ Inl = g ⟹ h = f ∘ Inr ⟷ case-sum g h = f
  by (rule iffI) (auto simp add: fun-eq-iff split: sum.splits)

lemma case-sum-expand-Inr: f ∘ Inl = g ⟹ f x = case-sum g (f ∘ Inr) x
  by (auto split: sum.splits)

lemma id-transfer: rel-fun A A id id
  unfolding rel-fun-def by simp

lemma fst-transfer: rel-fun (rel-prod A B) A fst
  unfolding rel-fun-def by simp

lemma snd-transfer: rel-fun (rel-prod A B) B snd
  unfolding rel-fun-def by simp

lemma convol-transfer:
  rel-fun (rel-fun R S) (rel-fun (rel-fun R T) (rel-fun R (rel-prod S T))) BNF-Def.convol
BNF-Def.convol

unfolding rel-fun-def convol-def by auto

lemma Let-const: Let x (\lambdax. c) = c
unfolding Let-def ..

ML-file ‹Tools/BNF/bnf-fp-util-tactics.ML›
ML-file ‹Tools/BNF/bnf-fp-util.ML›
ML-file ‹Tools/BNF/bnf-fp-def-sugar-tactics.ML›
ML-file ‹Tools/BNF/bnf-fp-def-sugar.ML›
ML-file ‹Tools/BNF/bnf-fp-n2m-tactics.ML›
ML-file ‹Tools/BNF/bnf-fp-n2m.ML›
ML-file ‹Tools/BNF/bnf-fp-n2m-sugar.ML›

end

37 Least Fixpoint (Datatype) Operation on Bounded Natural Functors

theory BNF-Least-Fixpoint
imports BNF-Fixpoint-Base
keywords
  datatype :: thy-defn and
  datatype-compat :: thy-defn
begin

lemma subset-emptyI: \(\forall x. x \in A \implies \text{False} \) \implies A \subseteq \{\}
by blast

lemma image-Collect-subsetI: \(\forall x. P x \implies f x \in B \) \implies f \{x. P x\} \subseteq B
by blast

lemma Collect-restrict: \{x. x \in X \land P x\} \subseteq X
by auto

lemma prop-restrict: \[\[x \in Z; Z \subseteq \{x. x \in X \land P x\}\]\] \implies P x
by auto

lemma underS-I: \[i \neq j; (i, j) \in R\] \implies i \in under\(S\) R j
unfolding underS-def by simp

lemma underS-E: i \in under\(S\) R j \implies i \neq j \land (i, j) \in R
unfolding underS-def by simp

lemma underS-Field: i \in under\(S\) R j \implies i \in Field R
unfolding underS-def Field-def by auto

lemma ex-bij-betw: |A| \leq o (r :: 'b rel) \implies \exists B :: 'b set. bij-betw f B A
by (subst (asm) internalize-card-of-ordLeq) (auto dest!: iffD2[OF card-of-ordIso ordIso-symmetric])  

lemma bij-betwI':
\[ \forall x \ y. \ [x \in X; \ y \in X] \implies (f \ x = f \ y) = (x = y); \]
\[ \forall x. \ x \in X \implies f \ x \in Y; \]
\[ \forall y. \ y \in Y \implies \exists x \in X. \ y = f \ x \] \implies bij-betw f X Y

unfolding bij-betw-def inj-on-def by blast

lemma surj-fun-eq:
assumes surj-on: \( f \cdot X = \text{UNIV} \) and eq-on:
\( \forall x \in X. \ (g_1 \circ f) \ x = (g_2 \circ f) \ x \)
shows \( g_1 = g_2 \)

proof (rule ext)
fix \( y \)
from surj-on obtain \( x \) where \( x \in X \) and \( y = f \ x \) by blast
thus \( g_1 \ y = g_2 \ y \) using eq-on by simp
qed

lemma Card-order-wo-rel: Card-order \( r \implies \text{wo-rel} \ r \)

unfolding wo-rel-def card-order-on-def by blast

lemma Cinfinite-limit: \( \forall x \in \text{Field} \ r; \ \text{Cinfinite} \ r \) \implies \( \exists y \in \text{Field} \ r. \ x \neq y \land (x, y) \in r \)

unfolding cinfinite-def by (auto simp add: infinite-Card-order-limit)

lemma Card-order-trans:
\[ \forall x \ y z. \ [\text{Card-order} \ r; \ x \neq y; \ (x, y) \in r; \ y \neq z; \ (y, z) \in r] \implies x \neq z \land (x, z) \in r \]

unfolding card-order-on-def well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def trans-def antisym-def by blast

lemma Cinfinite-limit2:
assumes \( x_1: \ x_1 \in \text{Field} \ r \) and \( x_2: \ x_2 \in \text{Field} \ r \) and \( r: \ \text{Cinfinite} \ r \)
shows \( \exists y \in \text{Field} \ r. \ (x_1 \neq y \land (x_1, y) \in r) \land (x_2 \neq y \land (x_2, y) \in r) \)

proof –
from \( r \) have trans: \( \text{trans} \ r \) and total: \( \text{Total} \ r \) and antisym: \( \text{antisym} \ r \)

unfolding card-order-on-def well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def trans-def antisym-def by auto

obtain \( y_1 \) where \( y_1: \ y_1 \in \text{Field} \ r \) \( x_1 \neq y_1 \) \( (x_1, y_1) \in r \)
using Cinfinite-limit[OF \( x_1 \ r \)] by blast

obtain \( y_2 \) where \( y_2: \ y_2 \in \text{Field} \ r \) \( x_2 \neq y_2 \) \( (x_2, y_2) \in r \)
using Cinfinite-limit[OF \( x_2 \ r \)] by blast

show \(?thesis \)

proof (cases \( y_1 = y_2 \))
  case True with \( y_1 \ y_2 \) show \(?thesis \) by blast
next
  case False
  with \( y_1(1) \) \( y_2(1) \) total have \( \forall y \in r. \ (y_1, y) \in r \)
  unfolding total-on-def by auto
  thus \(?thesis \)
proof
  assume \( *(y_1, y_2) \in r \) with \( \text{trans } y_1(3) \) have \( (x_1, y_2) \in r \) unfolding \text{trans-def} by blast
  with False \( y_1 y_2 \) * antisym show \( \text{thesis} \) by (cases \( x_1 = y_2 \)) (auto simp: antisym-def)
next
  assume \( *(y_2, y_1) \in r \) with \( \text{trans } y_2(3) \) have \( (x_2, y_1) \in r \) unfolding \text{trans-def} by blast
  with False \( y_1 y_2 \) * antisym show \( \text{thesis} \) by (cases \( x_2 = y_1 \)) (auto simp: antisym-def)
qed
qed
qed

lemma Cinfinite-limit-finite:
  \[ \text{finite } X; X \subseteq \text{Field } r; \text{Cinfinite } r \] \( \Rightarrow \exists y \in \text{Field } r. \forall x \in X. (x \neq y \land (x, y) \in r) \)
proof (induct \( X \) rule: finite-induct)
case empty thus \( ?case \) unfolding cinfinite-def using ex-in-conv[of \( \text{Field } r \)] finite.emptyI by auto
next
case (insert \( x \) \( X \) )
than obtain \( y \) where \( y \in \text{Field } r \forall x \in X. (x \neq y \land (x, y) \in r) \) by blast
then obtain \( z \) where \( z \in \text{Field } r x \neq z \land (x, z) \in r y \neq z \land (y, z) \in r \)
  using Cinfinite-limit2[of \( y(1) \) insert(5), of \( x \) ] insert(4) by blast
show \( ?case \)
  apply (intro bexI ballI)
  apply (erule insertE)
  apply hypsubst
  apply (rule \( z(2) \))
  using Card-order-trans[of \( y(5) \) \( \text{THEN conjunct2} \)] \( y(2) z(3) \)
  apply blast
  apply (rule \( z(1) \))
  done
qed

lemma insert-subsetI:
  \[ x \in A; X \subseteq A \] \( \Rightarrow \) insert \( x \) \( X \) \( \subseteq A \)
by auto

lemmas well-order-induct-imp = wo-rel.well-order-induct[of \( r \) \( \lambda x. x \in \text{Field } r \Rightarrow P \) \( x \) for \( r \) \( P \) ]

lemma meta-spec2:
  assumes \( \bigwedge x y. \text{PROP } P \) \( x \) \( y \)
  shows \( \text{PROP } P \) \( x \) \( y \)
by (rule assms)

lemma nchotomy-relcomppE:
  assumes \( \bigwedge y. \exists x. y = f x (r \text{ OO } s) a \) \( c \bigwedge b. r a (f b) \Rightarrow s (f b) c \Rightarrow P \)
shows $P$

proof (rule relcompp_cases[OF assms(2)], hypsubst)
  fix $b$ assume $r\ a\ b\ s\ b\ c$
  moreover from assms(1) obtain $b'$ where $b = f\ b'$ by blast
  ultimately show $P$ by (blast intro: assms(3))
qed

lemma predicate2D-vimage2p: $[R \leq \text{vimage2p} f\ g\ S; \ R\ x\ y] \Longrightarrow S\ (f\ x)\ (g\ y)$
  unfolding vimage2p_def by auto

lemma ssbst-Pair-rhs: $[(r, s) \in R; s' = s] \Longrightarrow (r, s') \in R$
  by (rule ssbst)

lemma all-mem-range1:
  $(\forall y. y \in \text{range}\ f \Longrightarrow P\ y) \equiv (\forall x. P\ (f\ x))$
  by (rule equal-intr_rule) fast+

lemma all-mem-range2:
  $(\forall fa\ y. fa \in \text{range}\ f \Longrightarrow y \in \text{range}\ fa \Longrightarrow P\ y) \equiv (\forall xa. P\ (f\ xa))$
  by (rule equal-intr_rule) fast+

lemma all-mem-range3:
  $(\forall fa\ fb\ y. fa \in \text{range}\ f \Longrightarrow fb \in \text{range}\ fa \Longrightarrow y \in \text{range}\ fb \Longrightarrow P\ y) \equiv (\forall xa\ xb. P\ (f\ xa\ xb))$
  by (rule equal-intr_rule) fast+

lemma all-mem-range4:
  $(\forall fa\ fb\ fc\ y. fa \in \text{range}\ f \Longrightarrow fb \in \text{range}\ fa \Longrightarrow fc \in \text{range}\ fb \Longrightarrow y \in \text{range}\ fc \Longrightarrow P\ y) \equiv (\forall xa\ xb\ xc. P\ (f\ xa\ xb\ xc))$
  by (rule equal-intr_rule) fast+

lemma all-mem-range5:
  $(\forall fa\ fb\ fc\ fd\ fe\ ff\ y. fa \in \text{range}\ f \Longrightarrow fb \in \text{range}\ fa \Longrightarrow fc \in \text{range}\ fb \Longrightarrow fd \in \text{range}\ fc \Longrightarrow y \in \text{range}\ fd \Longrightarrow P\ y) \equiv (\forall xa\ xb\ xc\ xd. P\ (f\ xa\ xb\ xc\ xd))$
  by (rule equal-intr_rule) fast+

lemma all-mem-range6:
  $(\forall fa\ fb\ fc\ fd\ fe\ ff\ y. fa \in \text{range}\ f \Longrightarrow fb \in \text{range}\ fa \Longrightarrow fc \in \text{range}\ fb \Longrightarrow fd \in \text{range}\ fc \Longrightarrow fe \in \text{range}\ fd \Longrightarrow ff \in \text{range}\ fe \Longrightarrow y \in \text{range}\ ff \Longrightarrow P\ y) \equiv (\forall xa\ xb\ xc\ xd\ xe. P\ (f\ xa\ xb\ xc\ xd\ xe\ xf))$
  by (rule equal-intr_rule) (fastforce, fast)

lemma all-mem-range7:
  $(\forall fa\ fb\ fc\ fd\ fe\ ff\ fg\ y. fa \in \text{range}\ f \Longrightarrow fb \in \text{range}\ fa \Longrightarrow fc \in \text{range}\ fb \Longrightarrow fd \in \text{range}\ fc \Longrightarrow$
\[ \begin{align*}
  \text{fe} \in \text{range } \text{fd} \implies \text{ff} \in \text{range } \text{fe} \implies \text{fg} \in \text{range } \text{ff} \implies y \in \text{range } \text{fg} \implies P \ y) \equiv \\
  (\forall x \ xa \ xb \ xc \ xd \ xe \ xf \ xg. \ P \ (f \ x \ xa \ xb \ xc \ xe \ xf \ xg))
\end{align*} \]

by (rule equal-intr-rule) (fastforce, fast)

**lemma** all-mem-range8:
\[ (\forall a \ fb \ fc \ fd \ ff \ fg \ fh \ y. \ a \in \text{range } f \implies fb \in \text{range } fa \implies fc \in \text{range } fb \implies fd \in \text{range } fc \implies ff \in \text{range } fc \implies fg \in \text{range } ff \implies fh \in \text{range } fg \implies y \in \text{range } fh \implies P \ y) \equiv \\
  (\forall x \ xa \ xb \ xc \ xd \ xe \ xf \ xg \ xh. \ P \ (f \ x \ xa \ xb \ xc \ xe \ xf \ xg \ xh)) \]

by (rule equal-intr-rule) (fastforce, fast)

**lemmas** all-mem-range = all-mem-range1 all-mem-range2 all-mem-range3 all-mem-range4 all-mem-range5 all-mem-range6 all-mem-range7 all-mem-range8

**lemma** pred-fun-True-id: NO-MATCH id \( p \implies \) pred-fun \((\lambda x. \ True) \ p \circ f \)

Unfolding fun, pred-map Unfolding comp-def id-def ..

ML-file ⟨Tools/BNF/bnf-lfp-util.ML⟩
ML-file ⟨Tools/BNF/bnf-lfp-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-lfp.ML⟩
ML-file ⟨Tools/BNF/bnf-lfp-compat.ML⟩
ML-file ⟨Tools/BNF/bnf-lfp-rec-sugar-more.ML⟩
ML-file ⟨Tools/BNF/bnf-lfp-size.ML⟩

ML-file ⟨Tools/datatype-simprocs.ML⟩

simproc-setup datatype-no-proper-subterm
\((x :: 'a :: size) = y) = \{K Datatype-Simprocs.no-proper-subterm-proc\}

end

38 Equivalence Relations in Higher-Order Set Theory

theory Eqv-Relations
  imports BNF-Least-Fixpoint
begin

38.1 Equivalence relations – set version

definition equiv :: 'a set ⇒ ('a × 'a) set ⇒ bool
  where equiv A r = refl-on A r ∨ sym r ∨ trans r

**lemma** equivI: refl-on A r ⇒ sym r ⇒ trans r ⇒ equiv A r

by (simp add: equiv-def)
lemma equivE:
    assumes equiv A r
    obtains refl-on A r and sym r and trans r
    using assms by (simp add: equiv-def)

Suppes, Theorem 70: $r$ is an equiv relation iff $r^{-1} O r = r$.

First half: $\text{equiv } A r \implies r^{-1} O r = r$.

lemma sym-trans-comp-subset: $\text{sym } r \implies \text{trans } r \implies r^{-1} O r \subseteq r$
    unfolding trans-def sym-def converse-unfold by blast

lemma refl-on-comp-subset: $\text{refl-on } A r \implies r \subseteq r^{-1} O r$
    unfolding refl-on-def by blast

lemma equiv-comp-eq: $\text{equiv } A r \implies r^{-1} O r = r$
    unfolding equiv-def by (iprover intro: sym-trans-comp-subset refl-on-comp-subset equalityI)

Second half.

lemma comp-equivI:
    assumes $r^{-1} O r = r$ Domain $r = A$
    shows $\text{equiv } A r$
    proof
        have $\forall x y. (x, y) \in r \implies (y, x) \in r$
            using assms by blast
        show $\forall x y. (x, y) \in r \implies (y, x) \in r$
            unfolding equiv-def refl-on-def sym-def trans-def
            using assms by (auto intro: *)
    qed

38.2 Equivalence classes

lemma equiv-class-subset: $\text{equiv } A r \implies (a, b) \in r \implies r''\{a\} \subseteq r''\{b\}$
    — lemma for the next result
    unfolding equiv-def trans-def sym-def by blast

theorem equiv-class-eq: $\text{equiv } A r \implies (a, b) \in r \implies r''\{a\} = r''\{b\}$
    by (intro equalityI equiv-class-subset; force simp add: equiv-def sym-def)

lemma equiv-class-self: $\text{equiv } A r \implies a \in A \implies a \in r''\{a\}$
    unfolding equiv-def refl-on-def by blast

lemma subset-equiv-class: $\text{equiv } A r \implies r''\{b\} \subseteq r''\{a\} \implies b \in A \implies (a, b) \in r$
    — lemma for the next result
    unfolding equiv-def refl-on-def by blast

lemma eq-equiv-class: $r''\{a\} = r''\{b\} \implies \text{equiv } A r \implies b \in A \implies (a, b) \in r$
    by (iprover intro: equalityD2 subset-equiv-class)
lemma equiv-class-nondisjoint: equiv A r \implies x \in (r''\{a\} \cap r''\{b\}) \implies (a, b) \in r
unfolding equiv-def trans-def sym-def by blast

lemma equiv-type: equiv A r \implies r \subseteq A \times A
unfolding equiv-def refl-on-def by blast

lemma equiv-class-eq-iff: equiv A r \implies (x, y) \in r \iff r''\{x\} = r''\{y\} \land x \in A \land y \in A
by (blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type)

lemma eq-equiv-class-iff: equiv A r \implies x \in A \land y \in A \implies r''\{x\} = r''\{y\} \iff (x, y) \in r
by (blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type)

lemma disjoint-equiv-class: equiv A r \implies \text{disjnt}\ (r''\{a\}) (r''\{b\}) \iff (a, b) \notin r
by (auto dest: equiv-class-self simp: equiv-class-eq-iff disjnt-def)

38.3 Quotients

definition quotient :: 'a set \Rightarrow ('a \times 'a) set \Rightarrow 'a set set (infixl "//") 90
where A // r = (\bigcup x \in A. \{r''\{x\}\}) — set of equiv classes

lemma quotientI: x \in A \implies r''\{x\} \in A // r
unfolding quotient-def by blast

lemma quotientE: X \in A // r \implies \(\forall x. X = r''\{x\} \implies x \in A \implies P\) \implies P
unfolding quotient-def by blast

lemma Union-quotient: equiv A r \implies \bigcup(A // r) = A
unfolding equiv-def refl-on-def quotient-def by blast

lemma quotient-disj: equiv A r \implies X \in A // r \implies Y \in A // r \implies X = Y \lor X \cap Y = \{\}
unfolding quotient-def equiv-def trans-def sym-def by blast

lemma quotient-eqI:
assumes equiv A r X \in A // r Y \in A // r and xy: x \in X y \in Y (x, y) \in r
shows X = Y
proof —
obtain a b where a \in A and a: X = r '' \{a\} and b \in A and b: Y = r '' \{b\}
using assms by (auto elim!: quotientE)
then have (a,b) \in r
using xy (equiv A r) unfolding equiv-def sym-def trans-def by blast
then show ?thesis
unfolding a b by (rule equiv-class-eq [OF (equiv A r)])
qed

lemma quotient-eq-iff:
assumes equiv A r X \in A // r Y \in A // r and xy: x \in X y \in Y
shows \( X = Y \iff (x, y) \in r \)

**proof**

assume \( L: X = Y \)

with \( \text{assms} \) show \( (x, y) \in r \)

unfolding equiv-def sym-def trans-def by (blast elim: quotientE)

next

assume \( \$: (x, y) \in r \) show \( X = Y \)

by (rule quotient-eqI) (use \( \$ \) assms in \( \langle\text{blast}\rangle\))

qed

**lemma** eq-equiv-class-iff2: \( \text{equiv A r} \implies x \in A \implies y \in A \implies \{x\}/r = \{y\}/r \)

\( \iff (x, y) \in r \)

by (simp add: quotient-def eq-equiv-class-iff)

**lemma** quotient-empty [simp]: \( \{\}/r = \{\} \)

by (simp add: quotient-def)

**lemma** quotient-is-empty [iff]: \( A/r = \{\} \iff A = \{\} \)

by (simp add: quotient-def)

**lemma** quotient-is-empty2 [iff]: \( \{\} = A/r \iff A = \{\} \)

by (simp add: quotient-def)

**lemma** singleton-quotient: \( \{x\}/r = \{r'' \{x\}\} \)

by (simp add: quotient-def)

**lemma** quotient-diff1: inj-on \( (\lambda a. \{a\}/r) \) \( A \)

\( \implies a \in A \implies (A - \{a\})/r = A/r - \{a\}/r \)

unfolding quotient-def inj-on-def by blast

**38.4** Refinement of one equivalence relation WRT another

**Lemma** refines-equiv-class-eq: \( R \subseteq S \implies \text{equiv A R} \implies \text{equiv A S} \implies R''(S''\{a\}) \)

\( = S''\{a\} \)

by (auto simp: equiv-class-eq-iff)

**lemma** refines-equiv-class-eq2: \( R \subseteq S \implies \text{equiv A R} \implies \text{equiv A S} \implies S''(R''\{a\}) \)

\( = S''\{a\} \)

by (auto simp: equiv-class-eq-iff)

**lemma** refines-equiv-image-eq: \( R \subseteq S \implies \text{equiv A R} \implies \text{equiv A S} \implies (\lambda X. S''X)''(A/R) = A/S \)

by (auto simp: quotient-def image-UN refines-equiv-class-eq2)

**lemma** finite-refines-finite:

\( \text{finite} (A//R) \implies R \subseteq S \implies \text{equiv A R} \implies \text{equiv A S} \implies \text{finite} (A//S) \)

by (erule finite-surj [where \( f = \lambda X. S''X\)] (simp add: refines-equiv-image-eq))

**lemma** finite-refines-card-le:
38.5 Defining unary operations upon equivalence classes

A congruence-preserving function.

**Definition** congruent :: (′a × ′a) set ⇒ (′a ⇒ ′b) ⇒ bool

where congruent r f ←→ (∀(y, z) ∈ r. f y = f z)

**Lemma** congruentI: (∀y z. (y, z) ∈ r =⇒ f y = f z) =⇒ congruent r f

by (auto simp add: congruent-def)

**Lemma** congruentD: congruent r f =⇒ (y, z) ∈ r =⇒ f y = f z

by (auto simp add: congruent-def)

**Abbreviation** RESPECTS :: (′a ⇒ ′b) ⇒ (′a × ′a) set ⇒ bool (infixr respects 80)

where f respects r ≡ congruent r f

**Lemma** UN-constant-eq: a ∈ A =⇒ ∀y ∈ A. f y = c =⇒ (∪y ∈ A. f y) = c

— lemma required to prove UN-equiv-class

by auto

**Lemma** UN-equiv-class:

assumes equiv A r f respects r a ∈ A

shows (∪x ∈ r``{a}. f x) = f a

— Conversion rule

**Proof**

have ‹∀x ∈ r``{a}. f x = f a›

using assms unfolding equiv-def congruent-def sym-def by blast

show ?thesis

by (iprover intro: assms UN-constant-eq [OF equiv-class-self ‹1›])

qed

**Lemma** UN-equiv-class-type:

assumes r: equiv A r f respects r and X: X ∈ A/\r and AB: \x. x ∈ A =⇒ f x ∈ B

shows (∪x ∈ X. f x) ∈ B

using assms unfolding quotient-def

by (auto simp: UN-equiv-class [OF r])

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; bcong could be \∀y. y ∈ A =⇒ f y ∈ B.

**Lemma** UN-equiv-class-inject:

assumes equiv A r f respects r
and $\forall x. (\bigcup y \in X. f x) = (\bigcup y \in Y. f y)$
and $X: X \in A//r$ and $Y: Y \in A//r$
and $fr: \forall x y. x \in A \longrightarrow y \in A \longrightarrow f x = f y \Longrightarrow (x, y) \in r$
shows $X = Y$
proof
obtain $a b$ where $a \in A$ and $a: X = r\{a\}$ and $b \in A$ and $b: Y = r\{b\}$
using assms by (auto elim!: quotientE)
then have $\bigcup \{f \cdot r\{a\}\} = f a \bigcup \{f \cdot r\{b\}\} = f b$
by (iprover intro: UN-equiv-class [OF \equiv A r] assms)+
then have $f a = f b$
using eq unfolding $a b$ by (iprover intro: trans sym)
then have $(a, b) \in r$
using fr $\forall a \in A. \forall b \in A$ by blast
then show $\text{thesis}$
  unfolding $a b$ by (rule equiv-class-eq [OF \equiv A r])
qed

38.6 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments.

definition congruent2 :: \((a \times 'a) set \Rightarrow \{b \times 'b\} set \Rightarrow \{a \Rightarrow \{b \Rightarrow 'c\} \Rightarrow 'b\} \Rightarrow 'a\)
where congruent2 $r1 r2 f \iff \forall (y1, z1) \in r1. \forall (y2, z2) \in r2. f y1 y2 = f z1 z2$

lemma congruent2I:
assumes $\forall y1 z1 y2 z2. (y1, z1) \in r1 \Longrightarrow (y2, z2) \in r2 \Longrightarrow f y1 y2 = f z1 z2$
shows congruent2 $r1 r2 f$
using assms by (auto simp add: congruent2-def)

lemma congruent2D: congruent2 $r1 r2 f \Longrightarrow (y1, z1) \in r1 \Longrightarrow (y2, z2) \in r2 \Longrightarrow f y1 y2 = f z1 z2$
by (auto simp add: congruent2-def)

Abbreviation for the common case where the relations are identical.

abbreviation RESPECTS2 :: \((a \Rightarrow \{a \Rightarrow 'b\} \Rightarrow \{a \times 'a\} set \Rightarrow 'b\) \Rightarrow 'a\) (infixr respects2 80)
where $f$ respects2 $r \equiv$ congruent2 $r r f$

lemma congruent2-implies-congruent:
equiv $A r1 \Longrightarrow$ congruent2 $r1 r2 f \Longrightarrow a \in A \Longrightarrow congruent r2 (f a)$
unfolding congruent-def congruent2-def equiv-def refl-on-def by blast

lemma congruent2-implies-congruent-UN:
assumes equiv $A1 r1 equiv A2 r2 congruent2 r1 r2 f a \in A2$
shows congruent $r1 (\lambda x1. \bigcup x2 \in r2\{a\}. f x1 x2)$
unfolding congruent-def
proof clarify
  fix $c d$

THEORY "Equiv-Relations"

assume \( cd : (c, d) \in r1 \)
then have \( c \in A1 \) \( d \in A1 \)
  using \( \langle \text{equiv } A1 \ r1 \rangle \) by (auto elim!: equiv-type \[\text{THEN subsetD, THEN SigmaE}2\])
moreover have \( f \ c \ a = f \ d \ a \)
  using assms \( cd \) unfolding congruent2-def equiv-def refl-on-def by blast
ultimately show \( \bigcup (f \ c \ ' r2'' \ \{a\}) = \bigcup (f \ d \ r2'' \ \{a\}) \)
  using assms by (simp add: UN-equiv-class congruent2-implies-congruent congruent2-implies-congruent-UN)
qed

lemma UN-equiv-class2:
  equiv A1 r1 \(\implies\) equiv A2 r2 \(\implies\) congruent2 r1 r2 f \(\implies\) a1 \(\in\) A1 \(\implies\) a2 \(\in\) A2
  \(\implies\) \(\bigcup x1 \in r1'' \{a1\} \cup x2 \in r2'' \{a2\} \cdot f x1 x2 = f a1 a2 \)
by (simp add: UN-equiv-class congruent2-implies-congruent congruent2-implies-congruent-UN)

lemma UN-equiv-class-type2:
  equiv A1 r1 \(\implies\) equiv A2 r2 \(\implies\) congruent2 r1 r2 f 
  \(\implies\) \(X1 \in A1 \implies X2 \in A2 \implies \bigcap x1 \in X1. \bigcup x2 \in X2. f x1 x2 \in B \)
unfolding quotient-def
by (blast intro: UN-equiv-class-type congruent2-implies-congruent congruent2-implies-congruent quotientI)

lemma UN-UN-split-split-eq:
  \(\bigcup (x1, x2) \in X \cup (y1, y2) \in Y \cdot A x1 x2 y1 y2 = \)
  \(\bigcup x \in X. \bigcup y \in Y. (\lambda(x1, x2). (\lambda(y1, y2). A x1 x2 y1 y2) y) x \)
— Allows a natural expression of binary operators,
— without explicit calls to split
by auto

lemma congruent2I:
  equiv A1 r1 \(\implies\) equiv A2 r2 
  \(\implies\) \(\lambda z w. w \in A2 \implies (y,z) \in r1 \implies f y w = f z w \)
  \(\implies\) \(\lambda z w. w \in A1 \implies (y,z) \in r2 \implies f w y = f w z \)
  \(\implies\) congruent2 r1 r2 f 
— Suggested by John Harrison – the two subproofs may be
— much simpler than the direct proof.
unfolding congruent2-def equiv-def refl-on-def
by (blast intro: trans)

lemma congruent2-commuteI:
assumes equivA: equiv A r
  and commute: \(\forall y z. y \in A \implies z \in A \implies f y z = f z y \)
  and congt: \(\forall y z w. w \in A \implies (y,z) \in r \implies f w y = f w z \)
shows f respects2 r
proof (rule congruent2I [OF equivA equivA])
note \(eqv = equivA \THEN equiv\text{-}type, \THEN subsetD, \THEN SigmaE2\)

show \(\forall y \ z \ w. \ [w \in A; (y, z) \in r] \implies f y w = f z w\)
by (iprover intro: commute \(\THEN\) trans sym cong elim: eqv)

show \(\forall y \ z \ w. \ [w \in A; (y, z) \in r] \implies f w y = f w z\)
by (iprover intro: cong elim: eqv)

qed

38.7 Quotients and finiteness

Suggested by Florian Kammüller

lemma finite-quotient:
assumes finite A r \(\subseteq A \times A\)
shows finite (\(\mathbb{A}/\mathbb{r}\))
— recall equiv \(\mathbb{A}/\mathbb{r}\) \(\implies \mathbb{r} \subseteq \mathbb{A} \times \mathbb{A}\)

proof —
have \(\mathbb{A}/\mathbb{r} \subseteq \text{Pow} \ A\)
using assms unfolding quotient-def by blast
moreover have finite (\(\text{Pow} \ A\))
using assms by simp
ultimately show \(?\text{thesis}\)
by (iprover intro: finite-subset)

qed

lemma finite-equiv-class:
finite A \(\implies r \subseteq A \times A \implies X \in A/\mathbb{r} \implies \) finite X
unfolding quotient-def
by (erule rev-finite-subset) blast

lemma equiv-imp-dvd-card:
assumes finite A equiv A r \(\forall X. X \in A/\mathbb{r} \implies k \text{dvd} \text{card} X\)
shows k dvd card A
proof (rule Union-quotient \(\THEN\) subst)
show k dvd card (\(\bigcup (A/\mathbb{r})\))
apply (rule dvd-partition)
using assms
by (auto simp: Union-quotient dest: quotient-disj)

qed (use assms in blast)

38.8 Projection

definition proj :: \('b \times 'a\) set \(\Rightarrow 'b \Rightarrow 'a\) set
where proj \(r \ x = r^{-1}\{x\}\)

lemma proj-preserves:
\(x \in A \implies proj r x \in A/\mathbb{r}\)
unfolding proj-def by (rule quotientI)

lemma proj-in-iff:
assumes equiv A r
shows proj \(r \ x \in A/\mathbb{r} \iff x \in A\)
(is \(?\text{lhs} \iff ?\text{rhs}\))
proof
  assume ?rhs
  then show ?lhs by (simp add: proj-preserves)
next
  assume ?lhs
  then show ?rhs
  unfolding proj-def quotient-def
  proof safe
    fix y
    assume y: y ∈ A and r "{x} = r "{y}
    moreover have y ∈ r "{y}
    using assms unfolding equiv-def refl-on-def by blast
    ultimately have (x, y) ∈ r by blast
    then show x ∈ A
    using assms unfolding equiv-def refl-on-def by blast
  qed
  qed

lemma proj-iff: equiv A r ⟹ {x, y} ⊆ A ⟹ proj r x = proj r y ↔ (x, y) ∈ r
by (simp add: proj-def eq-equiv-class-iff)

lemma proj-image: proj r ' A = A/r
  unfolding proj-def[abs-def] quotient-def by blast

lemma in-quotient-imp-non-empty: equiv A r ⟹ X ∈ A/r ⟹ X ≠ {}
  unfolding quotient-def using equiv-class-self by fast

lemma in-quotient-imp-in-rel: equiv A r ⟹ X ∈ A/r ⟹ {x, y} ⊆ X ⟹ (x, y) ∈ r
  using quotient-eq-iff[THEN iffD1] by fastforce

lemma in-quotient-imp-closed: equiv A r ⟹ X ∈ A/r ⟹ x ∈ X ⟹ (x, y) ∈ r ⟹ y ∈ X
  unfolding quotient-def equiv-def trans-def by blast

lemma in-quotient-imp-subset: equiv A r ⟹ X ∈ A/r ⟹ X ⊆ A
  using in-quotient-imp-in-rel equiv-type by fastforce

38.9 Equivalence relations – predicate version

Partial equivalences.

definition part-equivp :: ('a ⇒ 'a ⇒ bool) ⇒ bool
  where part-equivp R ⟷ (∃x. R x x) ∧ (∀x y. R x y ⟷ R x x ∧ R y y ∧ R x = R y)
  — John-Harrison-style characterization

lemma part-equivpI: ∃x. R x x ⟹ symp R ⟹ transp R ⟹ part-equivp R
by (auto simp add: part-equivp-def) (auto elim: sympE transpE)

lemma part-equivpE:
  assumes part-equivp R
  obtains x where R x x and symp R and transp R
proof –
  from assms have 1: ∃x. R x x
    and 2: ∀x y. R x y ↔ R x x ∧ R y y ∧ R x = R y
    unfolding part-equivp-def by blast+
  from 1 obtain x where R x x ..
moreover have symp R
proof (rule sympI)
  fix x y
  assume R x y
  with 2 [of x y] show R y x by auto
qed
moreover have transp R
proof (rule transpI)
  fix x y z
  assume R x y and R y z
  with 2 [of x y] 2 [of y z] show R x z by auto
qed
ultimately show thesis by (rule that)
qed

lemma part-equiv-refl-symp-transp: part-equivp R ⟷ (∃x. R x x) ∧ symp R ∧ transp R
by (auto intro: part-equivpI elim: part-equivpE)

lemma part-equiv-symp: part-equivp R ⟹ R x y ⟹ R y x
by (erule part-equivpE, erule sympE)

lemma part-equiv-transp: part-equivp R ⟹ R x y ⟹ R y z ⟹ R x z
by (erule part-equivpE, erule transpE)

lemma part-equiv-typedef: part-equivp R ⟹ ∃d. d ∈ {c. ∃x. R x x ∧ c = Collect (R x)}
by (auto elim: part-equivpE)

Total equivalences.
definition equivp :: ('a ⇒ 'a ⇒ bool) ⇒ bool
  where equivp R ⟷ (∀x y. R x y = (R x = R y)) — John-Harrison-style characterization

lemma equivpI: reflp R ⟹ symp R ⟹ transp R ⟹ equivp R
by (auto elim: reflpE sympE transpE simp add: equivp-def)

lemma equivpE:
  assumes equivp R
obtains reflp R and symp R and transp R using assms by (auto intro!: that reflpI sympI transpI simp add: equivp-def)

lemma equivp-implies-part-equivp: equivp R \implies part-equivp R
  by (auto intro!: part-equivpI elim: equivpE reflpE)

lemma equivp-equiv: equiv UNIV A \iff equivp (\lambda x y. (x, y) \in A)
  by (auto intro!: equivI equivpI [to-set] elim: equivE equivpE [to-set])

lemma equivp-reflp-symp-transp: equivp R \iff reflp R \land symp R \land transp R
  by (auto intro!: equivpI elim: equivpE)

lemma identity-equivp: equivp (=)
  by (auto intro!: equivpI reflpI sympI transpI)

lemma equivp-reflp: equivp R \implies R x x
  by (erule equivpE, erule reflpE)

lemma equivp-symp: equivp R \implies R x y \implies R y x
  by (erule equivpE, erule sympE)

lemma equivp-transp: equivp R \implies R x y \implies R y z \implies R x z
  by (erule equivpE, erule transpE)

lemma equivp-rtranclp: symp r \implies equivp r**
  by (intro equivpI reflpI sympI transpI)(auto dest: sympD[OF symp-rtranclp])

lemmas equivp-rtranclp-symclp [simp] = equivp-rtranclp[OF symp-on-symclp]

lemma equivp-vimage2p: equivp R \implies equivp (vimage2p f f R)
  by (auto simp add: equivp-def vimage2p-def dest: fun-cong)

lemma equivp-imp-transp: equivp R \implies transp R
  by (simp add: equivp-reflp-sym-clp-sym)

38.10 Equivalence closure

definition equivclp :: (‘a ⇒ ‘a ⇒ bool) ⇒ ‘a ⇒ ‘a ⇒ bool where
  equivclp r ≡ (symclp r)**

lemma transp-equivclp [simp]: transp (equivclp r)
  by (simp add: equivclp-def)

lemma reflp-equivclp [simp]: reflp (equivclp r)
  by (simp add: equivclp-def)

lemma symp-equivclp [simp]: symp (equivclp r)
  by (simp add: equivclp-def)
THEORY “Equiv-Relations” 819

lemma equivp-equivclp [simp]: equivp (equivclp r)
  by (simp add: equivpI)

lemma tranclp-equivclp [simp]: (equivclp r)++ = equivclp r
  by (simp add: equivclp-def)

lemma rtranclp-equivclp [simp]: (equivclp r)** = equivclp r
  by (simp add: equivclp-def)

lemma symclp-equivclp [simp]: symclp (equivclp r) = equivclp r
  by (simp add: equivclp-def symp-symclp-eq)

lemma equivclp-symclp [simp]: equivclp (symclp r) = equivclp r
  by (simp add: equivclp-def)

lemma equivclp-conversep [simp]: equivclp (conversep r) = equivclp r
  by (simp add: equivclp-def)

lemma equivclp-sym [sym]: equivclp r x y =⇒ equivclp r y x
  by (rule sympD [OF symp-equivclp])

lemma equivclp-OO-equivclp-le-equivclp: equivclp r OO equivclp r ≤ equivclp r
  by (rule transp-relcompp-less-eq transp-equivclp)

lemma rtranclp-le-equivclp: r∗∗ ≤ equivclp r
  unfolding equivclp-def by (rule rtranclp-mono) (simp add: symclp-pointfree)

lemma rtranclp-conversep-le-equivclp: r−1−1∗∗ ≤ equivclp r
  unfolding equivclp-def by (rule rtranclp-mono) (simp add: symclp-pointfree)

lemma symclp-rtranclp-le-equivclp: symclp r∗∗ ≤ equivclp r
  unfolding symclp-pointfree by (rule le-supI) (simp-all add: rtranclp-conversep [symmetric] rtranclp-le-equivclp rtranclp-conversep-le-equivclp)

lemma r-OO-conversep-into-equivclp:
  r∗∗ OO r−1−1∗∗ ≤ equivclp r

lemma equivclp-induct [consumes 1, case-names base step, induct pred: equivclp]:
  assumes a: equivclp r a b
  and cases: P a ∨ y z. equivclp r a y =⇒ r y z ∨ r z y =⇒ P y =⇒ P z
  shows P b
  using a unfolding equivclp-def
  by (induction rule: rtranclp-induct; fold equivclp-def; blast intro: cases elim: symclpE)

lemma converse-equivclp-induct [consumes 1, case-names base step]:
assumes major: equivclp r a b
   and cases: P b ∨ y z ∨ r z y =⇒ equivclp r z b =⇒ P z ⇒ P y
shows P a
using major unfolding equivclp-def
by (induction rule: converse-rtranclp-induct; fold equivclp-def; blast intro: cases elim: symclpE)

lemma equivclp-refl [simp]: equivclp r x x
by (rule reflD [OF reflp-equivclp])

lemma r-into-equivclp [intro]: r x y =⇒ equivclp r x y
unfolding equivclp-def by (blast intro: symclpI)

lemma converse-r-into-equivclp [intro]: r y x =⇒ equivclp r x y
unfolding equivclp-def by (blast intro: symclpI)

lemma rtranclp-into-equivclp: r∗∗x y =⇒ equivclp r x y
using rtranlcp-le-equivclp by blast

lemma converse-rtranclp-into-equivclp: r∗∗y x =⇒ equivclp r x y
by (blast intro: equivclp-sym rtranclp-into-equivclp)

lemma equivclp-into-equivclp: [ equivclp r a b; r b c ∨ r c b ] =⇒ equivclp r a c
unfolding equivclp-def by (erule rtranclp.rtrancl-into-rtrancl) (auto intro: symclpI)

lemma equivclp-trans [trans]: [ equivclp r a b; equivclp r b c ] =⇒ equivclp r a c
using equivclp-OO-equivclp-le-equivclp[of r] by blast

hide-const (open) proj

end
lemma xtor-rel: \( R (xtor x) (xtor y) = R x y \)
unfolding xtor-def by (rule refl)

lemma xtor-induct: \((\forall x. P (xtor x)) \Rightarrow P z\)
unfolding xtor-def by assumption

lemma xtor-xtor: \( xtor (xtor x) = x \)
unfolding xtor-def by (rule refl)

lemmas xtor-inject = xtor-rel[of (=)]

lemma xtor-rel-induct: \((\forall x y. vimage2p id-bnf id-bnf R x y \Rightarrow IR (xtor x) (xtor y)) \Rightarrow R \leq IR\)
unfolding xtor-def vimage2p-def id-bnf-def ..

lemma xtor-iff-xtor: \( u = xtor w \leftrightarrow xtor u = w \)
unfolding xtor-def ..

lemma Inl-def-alt: \( Inl \equiv (\lambda a. xtor (id-bnf (Inl a))) \)
unfolding xtor-def id-bnf-def by (rule reflexive)

lemma Inr-def-alt: \( Inr \equiv (\lambda a. xtor (id-bnf (Inr a))) \)
unfolding xtor-def id-bnf-def by (rule reflexive)

lemma Pair-def-alt: \( Pair \equiv (\lambda a b. xtor (id-bnf (a, b))) \)
unfolding xtor-def id-bnf-def by (rule reflexive)

definition ctor-rec :: 'a ⇒ 'a where
ctor-rec x = x

lemma ctor-rec: \( g = id \Rightarrow ctor-rec f (xtor x) = f ((id-bnf \circ g \circ id-bnf) x)\)
unfolding ctor-rec-def id-bnf-def xtor-def comp-def id-def by hypsubst (rule refl)

lemma ctor-rec-unique: \( g = id \Rightarrow f \circ xtor = s \circ (id-bnf \circ g \circ id-bnf) \Rightarrow f = ctor-rec s\)
unfolding ctor-rec-def id-bnf-def xtor-def comp-def id-def by hypsubst (rule refl)

lemma ctor-rec-def-alt: \( f = ctor-rec (f \circ id-bnf)\)
unfolding ctor-rec-def id-bnf-def comp-def by (rule refl)

lemma ctor-rec-o-map: \( ctor-rec f \circ g = ctor-rec (f \circ (id-bnf \circ g \circ id-bnf))\)
unfolding ctor-rec-def id-bnf-def comp-def by (rule refl)

lemma ctor-rec-transfer: \( rel-fun (rel-fun (vimage2p id-bnf id-bnf R) S) (rel-fun R S) ctor-rec ctor-rec\)
unfolding rel-fun-def vimage2p-def id-bnf-def ctor-rec-def by simp

lemma eq-fst-iff: \( a = fst p \leftrightarrow (\exists b. p = (a, b))\)
by (cases p) auto

lemma eq-snd-iff: b = snd p ←→ (∃ a. p = (a, b))
by (cases p) auto

lemma ex-neg-all-pos: ((∃ x. P x) ⇒ Q) ≡ (∀ x. P x ⇒ Q)
by standard blast+

lemma hypsubst-in-prems: (∀ x. P x = Q) ≡ (∀ x. P x = Q)
by standard blast+

lemma isl-map-sum:
isl (map-sum f g s) = isl s
by (cases s) simp-all

lemma map-sum-sel:
isl s =⇒ projl (map-sum f g s) = f (projl s)
¬ isl s =⇒ projr (map-sum f g s) = g (projr s)
by (cases s; simp)+

lemma set-sum-sel:
isl s =⇒ projl s ∈ setl s
¬ isl s =⇒ projr s ∈ setr s
by (cases s; auto intro: setl.intros setr.intros)+

lemma rel-sum-sel: rel-sum R1 R2 a b = (isl a = isl b ∧
(isl a =⇒ isl b =⇒ R1 (projl a) (projl b)) ∧
(¬ isl a =⇒ ¬ isl b =⇒ R2 (projr a) (projr b)))
by (cases a b rule: sum.exhaust[case-product sum.exhaust]) simp-all

lemma isl-transfer: rel-fun (rel-sum A B) (=) isl isl
unfolding rel-fun-def rel-sum-sel by simp

lemma rel-prod-sel: rel-prod R1 R2 p q = (R1 (fst p) (fst q) ∧ R2 (snd p) (snd q))
by (force simp: rel-prod.simps elim: rel-prod.cases)

ML-file ⟨Tools/BNF/bnf-lfp-basic-sugar.ML⟩
declare prod.size [no-atp]

hide-const (open) xtor ctor-rec

hide-fact (open)
xtor-def xtor-map xtor-set xtor-rel xtor-induct xtor-ctor xtor-inject ctor-rec-def
ctor-rec
ctor-rec-def-alt ctor-rec-o-map xtor-rel-induct Inl-def-alt Inr-def-alt Pair-def-alt
end
39  MESON Proof Method

de Morgan laws

\begin{itemize}
\item \textbf{lemma} \textit{not-conjD}: \( \neg(P \land Q) \implies \neg P \lor \neg Q \)
\item \textbf{and} \textit{not-disjD}: \( \neg(P \lor Q) \implies \neg P \land \neg Q \)
\item \textbf{and} \textit{not-notD}: \( \neg \neg P \implies P \)
\item \textbf{and} \textit{not-allD}: \( \forall x. \neg P(x) \implies \exists x. \neg P(x) \)
\item \textbf{by} \textit{fast+}
\end{itemize}

Removal of \( \implies \) and \( \iff \) (positive and negative occurrences)

\begin{itemize}
\item \textbf{lemma} \textit{imp-to-disjD}: \( P \implies Q \implies \neg P \lor Q \)
\item \textbf{and} \textit{not-impD}: \( \neg(P \implies Q) \implies P \land \neg Q \)
\item \textbf{and} \textit{iff-to-disjD}: \( P \iff Q \implies (\neg P \lor Q) \land (\neg Q \lor P) \)
\item \textbf{and} \textit{not-iffD}: \( \neg(P \iff Q) \implies (P \lor \neg Q) \land (\neg P \lor Q) \)
\item \( \textit{— Much more efficient than } P \land \neg Q \lor Q \land \neg P \textit{ for computing CNF} \)
\item \textbf{and} \textit{not-refl-disj-D}: \( x \neq x \lor P \implies P \)
\item \textbf{by} \textit{fast+}
\end{itemize}

39.2  Pulling out the existential quantifiers

Conjunction

\begin{itemize}
\item \textbf{lemma} \textit{conj-exD1}: \( \forall P \, Q. \, (\exists x. \, P(x)) \land Q \implies \exists x. \, P(x) \land Q \)
\item \textbf{and} \textit{conj-exD2}: \( \forall P \, Q. \, P \land (\exists x. \, Q(x)) \implies \exists x. \, P \land Q(x) \)
\item \textbf{by} \textit{fast+}
\end{itemize}

Disjunction

\begin{itemize}
\item \textbf{lemma} \textit{disj-exD}: \( \forall P \, Q. \, (\exists x. \, P(x)) \lor (\exists x. \, Q(x)) \implies \exists x. \, P(x) \lor Q(x) \)
\item \( \textit{— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!} \)
\item \( \textit{— With ex-Skolemization, makes fewer Skolem constants} \)
\item \textbf{and} \textit{disj-exD1}: \( \forall P \, Q. \, (\exists x. \, P(x)) \lor Q \implies \exists x. \, P(x) \lor Q \)
\item \textbf{and} \textit{disj-exD2}: \( \forall P \, Q. \, P \lor (\exists x. \, Q(x)) \implies \exists x. \, P \lor Q(x) \)
\item \textbf{by} \textit{fast+}
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \textit{disj-assoc}: \( (P \lor Q) \lor R \implies P \lor (Q \lor R) \)
\item \textbf{and} \textit{disj-comm}: \( P \lor Q \implies Q \lor P \)
\item \textbf{and} \textit{disj-FalseD1}: \( \text{False} \lor P \implies P \)
\item \textbf{and} \textit{disj-FalseD2}: \( P \lor \text{False} \implies P \)
\item \textbf{by} \textit{fast+}
\end{itemize}

Generation of contrapositives
Inserts negated disjunct after removing the negation; P is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

**Lemma** make-neg-rule: \( \neg P \lor Q \Rightarrow ((\neg P \Rightarrow P) \Rightarrow Q) \)

**Lemma** make-refined-neg-rule: \( \neg P \lor Q \Rightarrow (P \Rightarrow Q) \)

\( P \) should be a literal

**Lemma** make-pos-rule: \( P \lor Q \Rightarrow ((P \Rightarrow \neg P) \Rightarrow Q) \)

Versions of **make-neg-rule** and **make-pos-rule** that don’t insert new assumptions, for ordinary resolution.

**Lemma** make-neg-rule\(^\prime\) = make-refined-neg-rule

**Lemma** make-pos-rule\(^\prime\): \([ \neg P \lor Q; \neg P ] \Rightarrow Q\)

Generation of a goal clause – put away the final literal

**Lemma** make-neg-goal: \( \neg P \Rightarrow ((\neg P \Rightarrow P) \Rightarrow False) \)

**Lemma** make-pos-goal: \( P \Rightarrow ((P \Rightarrow \neg P) \Rightarrow False) \)

### 39.3 Lemmas for Forward Proof

There is a similarity to congruence rules. They are also useful in ordinary proofs.

**Lemma** conj-forward: \([P \land Q; P' \Rightarrow P; Q' \Rightarrow Q] \Rightarrow P \land Q\)

**Lemma** disj-forward: \([P \lor Q; P' \Rightarrow P; Q' \Rightarrow Q] \Rightarrow P \lor Q\)

**Lemma** imp-forward: \([P' \Rightarrow Q; P \Rightarrow P'; Q' \Rightarrow Q] \Rightarrow P \Rightarrow Q\)

**Lemma** imp-forward2: \([P' \Rightarrow Q; P \Rightarrow P'; P' \Rightarrow Q' \Rightarrow Q] \Rightarrow P \Rightarrow Q\)

**Lemma** disj-forward2: \([ P' \lor Q; P' \Rightarrow P; [Q; P \Rightarrow False] \Rightarrow Q] \Rightarrow P \lor Q\)

apply blast
done

**lemma** all-forward: \[ \forall x. P'(x); \ \forall x. P'(x) \implies P(x) \implies \forall x. P(x) \]
by blast

**lemma** ex-forward: \[ \exists x. P'(x); \ \forall x. P'(x) \implies P(x) \implies \exists x. P(x) \]
by blast

### 39.4 Clausification helper

**lemma** TruepropI: \( P \equiv Q \implies \text{Trueprop} P \equiv \text{Trueprop} Q \)
by simp

**lemma** ext-cong-neq: \( F g \neq F h \implies F g \neq F h \land (\exists x. g x \neq h x) \)
apply (erule contrapos-np)
apply clarsimp
apply (rule eq_cong [where \( f = F \)])
by auto

Combinator translation helpers

**definition** COMBI :: \( 'a \Rightarrow 'a \)
where
\[ \text{COMBI} P = P \]

**definition** COMBK :: \( 'a \Rightarrow 'b \Rightarrow 'a \)
where
\[ \text{COMBK} P Q = P \]

**definition** COMBB :: \( 'b \Rightarrow 'c \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'a \)
where
\[ \text{COMBB} P Q R = P (Q R) \]

**definition** COMBC :: \( 'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'a \Rightarrow 'c \Rightarrow 'b \Rightarrow 'a \)
where
\[ \text{COMBC} P Q R = P (Q R) \]

**definition** COMBS :: \( 'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'a \)
where
\[ \text{COMBS} P Q R = P (Q R) \]

**lemma** abs-S: \( \lambda x. (f x) (g x) \equiv \text{COMBS} f g \)
apply (rule eq_reflection)
apply (rule ext)
apply (simp add: COMBS_def)
done

**lemma** abs-I: \( \lambda x. x \equiv \text{COMBI} \)
apply (rule eq_reflection)
apply (rule ext)
apply (simp add: COMBI_def)
done

**lemma** abs-K: \( \lambda x. y \equiv \text{COMBK} y \)
apply (rule eq_reflection)
apply (rule ext)
apply (simp add: COMBK-def)
done

lemma abs-B: λx. a (g x) ≡ COMBB a g
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBB-def)
done

lemma abs-C: λx. (f x) b ≡ COMBC f b
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBC-def)
done

39.5 Skolemization helpers

definition skolem :: `'a ⇒ 'a where
skolem = (λx. x)

lemma skolem-COMBK-iff: P ←→ skolem (COMBK P (i::nat))
unfolding skolem-def COMBK-def by (rule refl)

lemmas skolem-COMBK-I = iffD1 [OF skolem-COMBK-iff]

39.6 Meson package

ML-file «Tools/Meson/meson.ML»
ML-file «Tools/Meson/meson-clausify.ML»
ML-file «Tools/Meson/meson-tactic.ML»
hide-const (open) COMBI COMBK COMBB COMBC COMBS skolem
hide-fact (open) not-conjD not-disjD not-notD not-allD not-exD imp-to-disjD
not-impD iff-to-disjD not-iffD not-refl-disj-D conj-exD1 conj-exD2 disj-exD
disj-exD1 disj-exD2 disj-assoc disj-comm disj-FalseD1 disj-FalseD2 TruepropI
ext-cong-neq COMBI-def COMBK-def COMBB-def COMBC-def COMBS-def
abs-I abs-K
abs-B abs-C abs-S skolem-def skolem-COMBK-iff skolem-COMBK-I
end

40 Automatic Theorem Provers (ATPs)

theory ATP
  imports Meson Hilbert-Choice
begin
40.1 ATP problems and proofs
ML-file ⟨Tools/ATP/atp-tilt.ML⟩
ML-file ⟨Tools/ATP/atp-problem.ML⟩
ML-file ⟨Tools/ATP/atp-proof.ML⟩
ML-file ⟨Tools/ATP/atp-proof-redirect.ML⟩

40.2 Higher-order reasoning helpers

definition fFalse :: bool where
  fFalse ←→ False

definition fTrue :: bool where
  fTrue ←→ True

definition fNot :: bool ⇒ bool where
  fNot P ←→ ¬ P

definition fComp :: (′a ⇒ bool) ⇒ ʻa ⇒ bool where
  fComp P = (λx. ¬ P x)

definition fconj :: bool ⇒ bool ⇒ bool where
  fconj P Q ←→ P ∧ Q

definition fdisj :: bool ⇒ bool ⇒ bool where
  fdisj P Q ←→ P ∨ Q

definition fimplies :: bool ⇒ bool ⇒ bool where
  fimplies P Q ←→ (P −→ Q)

definition fAll :: (′a ⇒ bool) ⇒ bool where
  fAll P ←→ All P

definition fEx :: (′a ⇒ bool) ⇒ bool where
  fEx P ←→ Ex P

definition fequal :: ʻa ⇒ ʻa ⇒ bool where
  fequal x y ←→ (x = y)

definition fChoice :: ʻa ⇒ bool ⇒ ʻa where
  fChoice ≡ Hilbert-Choice.Eps

lemma fTrue-ne-fFalse: fFalse ≠ fTrue
unfolding fFalse-def fTrue-def by simp

lemma fNot-table:
  fNot fFalse = fTrue
  fNot fTrue = fFalse
unfolding fFalse-def fTrue-def fNot-def by auto
lemma fconj-table:
fconj fFalse P = fFalse
fconj P fFalse = fFalse
fconj fTrue fTrue = fTrue
unfolding fFalse-def fTrue-def fconj-def by auto

lemma fdisj-table:
fdisj fTrue P = fTrue
fdisj P fTrue = fTrue
fdisj fFalse fFalse = fFalse
unfolding fFalse-def fTrue-def fdisj-def by auto

lemma fimplies-table:
fimplies P fTrue = fTrue
fimplies fFalse P = fTrue
fimplies fTrue fFalse = fFalse
unfolding fFalse-def fTrue-def fimplies-def by auto

lemma fAll-table:
Ex (fComp P) ∨ fAll P = fTrue
All P ∨ fAll P = fFalse
unfolding fFalse-def fTrue-def fComp-def fAll-def by auto

lemma fEx-table:
All (fComp P) ∨ fEx P = fTrue
Ex P ∨ fEx P = fFalse
unfolding fFalse-def fTrue-def fComp-def fEx-def by auto

lemma fequal-table:
fequal x x = fTrue
x = y ∨ fequal x y = fFalse
unfolding fFalse-def fTrue-def fequal-def by auto

lemma fNot-law:
fNot P ≠ P
unfolding fNot-def by auto

lemma fComp-law:
fComp P x ↔ ¬ P x
unfolding fComp-def ..

lemma fconj-laws:
fconj P P ↔ P
fconj P Q ↔ fconj Q P
fNot (fconj P Q) ↔ fdisj (fNot P) (fNot Q)
unfolding fNot-def fconj-def fdisj-def by auto

lemma fdisj-laws:
fdisj P P ↔ P
fdisj P Q ↔ fdisj Q P
\text{fNot} (fdisj P Q) ↔ fconj (fNot P) (fNot Q)
\text{unfolding fNot-def fconj-def fdisj-def by auto}

\text{lemma fimplies-laws:}
fimplies P Q ↔ fdisj (¬ P) Q
\text{fNot (fimplies P Q) ↔ fconj P (fNot Q)}
\text{unfolding fNot-def fconj-def fdisj-def by auto}

\text{lemma fAll-law:}
fNot (fAll R) ↔ fEx (fComp R)
\text{unfolding fNot-def fComp-def fAll-def fEx-def by auto}

\text{lemma fequal-laws:}
fequal x y = fequal y x
fequal x y = fFalse ∨ fequal y z = fFalse ∨ fequal x z = fTrue
\text{unfolding fFalse-def fTrue-def fequal-def by auto}

\text{lemma fChoice-iff-Ex: P (fChoice P) ↔ HOL.Ex P}
\text{unfolding fChoice-def by (fact some-eq-ex)}

We use the \textit{Ex} constant on the right-hand side of \textit{fChoice-iff-Ex} because we want to use the TPTP-native version if \textit{fChoice} is introduced in a logic that supports FOOL. In logics that don’t support it, it gets replaced by \textit{fEx} during processing. Notice that we cannot use \( \exists x. P x \), as existentials are not skolemized by the \textit{metis} proof method but \( \textit{Ex P} \) is eta-expanded if FOOL is supported.

40.3 Basic connection between ATPs and HOL

\text{ML-file <Tools/lambda-lifting.ML>}
\text{ML-file <Tools/monomorph.ML>}
\text{ML-file <Tools/ATP/atp-problem-generate.ML>}
\text{ML-file <Tools/ATP/atp-proof-reconstruct.ML>}

end

41 Metis Proof Method

\text{theory Metis}
\text{imports ATP}
\begin
context notes [[ML-catch-all]]
begin
ML-file ("~/src/Tools/Metis/mcis.ML")
end

41.1 Literal selection and lambda-lifting helpers

definition select :: 'a ⇒ 'a where
select = (λx. x)

lemma not-atomize: (∼A ⇒ False) ≡ Trueprop A
by (cut-tac atomize-not [of ∼A]) simp

lemma atomize-not-select: (A ⇒ select False) ≡ Trueprop (∼A)
unfolding select-def by (rule atomize-not)

lemma not-atomize-select: (∼A ⇒ select False) ≡ Trueprop A
unfolding select-def by (rule not-atomize)

lemma select-FalseI: False ⇒ select False
by simp

definition lambda :: 'a ⇒ 'a where
lambda = (λx. x)

lemma eq-lambdaI: x ≡ y ⇒ x ≡ lambda y
unfolding lambda-def by assumption

41.2 Metis package

ML-file (Tools/Metis/metis-generate.ML)
ML-file (Tools/Metis/metis-reconstruct.ML)
ML-file (Tools/Metis/metis-tactic.ML)

hide-const (open) select fFalse fTrue fNot fComp fconj fdisj fimplies fAll fEx fequal lambda
hide-fact (open) select-def not-atomize atomize-not-select not-atomize-select select-FalseI
fFalse-def fTrue-def fNot-def fComp-def fconj-def fdisj-def fimplies-def fAll-def fEx-def fequal-def
fTrue-ne-fFalse fNot-table fconj-table fdisj-table fimplies-table fAll-table fEx-table
fequal-table fAll-table fEx-table fNot-law fComp-law fconj-law fdisj-law fimplies-laws
fequal-laws fAll-law fEx-law lambda-def eq-lambdaI

end
42 Generic theorem transfer using relations

theory Transfer
imports Basic-BNF-LFPs Hilbert-Choice Metis
begin

42.1 Relator for function space

bundle lifting-syntax
begin

notation rel-fun (infixr ===> 55)
notation map-fun (infixr ---> 55)

context includes lifting-syntax
begin

lemma rel-funD2:
assumes rel-fun A B f g A x x
shows B (f x) (g x)
using assms by (rule rel-funD)

lemma rel-funE:
assumes rel-fun A B f g A x y
obtains B (f x) (g y)
using assms by (simp add: rel-fun-def)

lemmas rel-fun-eq = fun.rel-eq

lemma rel-fun-eq-rel:
shows rel-fun (=) R = (λf g. ∀ x. R (f x) (g x))
by (simp add: rel-fun-def)

42.2 Transfer method

Explicit tag for relation membership allows for backward proof methods.

definition Rel :: ('a ⇒ 'b ⇒ bool) ⇒ 'a ⇒ 'b ⇒ bool
where Rel r ≡ r

Handling of equality relations

definition is-equality :: ('a ⇒ 'a ⇒ bool) ⇒ bool
where is-equality R ≔ R = (=)

lemma is-equality-eq: is-equality (=)
unfolding is-equality-def by simp

Reverse implication for monotonicity rules

definition rev-implies where
rev-implies x y ≔ (y ⇒ x)
Handling of meta-logic connectives

definition transfer-forall where
    transfer-forall ≡ All

definition transfer-implies where
    transfer-implies ≡ (→)

definition transfer-bforall :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ bool
    where
        transfer-bforall ≡ (λx. P x) → Q x

lemma unfolding atomize-all transfer-forall-def ..

lemma transfer-implies-eq: (A ⇒ B) ≡ Trueprop (transfer-implies A B)
    unfolding atomize-imp transfer-implies-def ..

lemma transfer-bforall-unfold: Trueprop (transfer-bforall P (λx. Q x)) ≡ (λx. P x → Q x)
    unfolding transfer-bforall-def atomize-imp atomize-all ..

lemma transfer-start: [ [P; Rel (=) Q] ] ⇒ Q
    unfolding Rel-def by simp

lemma transfer-start': [ [P; Rel (→) Q] ] ⇒ Q
    unfolding Rel-def by simp

lemma transfer-prover-start: [ [x = x'; Rel R x y] ] ⇒ Rel R x y
    by simp

lemma untransfer-start: [ [Q; Rel (=) P Q] ] ⇒ P
    unfolding Rel-def by simp

lemma Rel-eq-refl: Rel (=) x x
    unfolding Rel-def ..

lemma Rel-app:
    assumes Rel (A =⇒ B) f g and Rel A x y
    shows Rel B (f x) (g y)
    using assms unfolding Rel-def rel-fun-def by fast

lemma Rel-abs:
    assumes (x y. Rel A x y =⇒ Rel B (f x) (g y))
    shows Rel (A =⇒ B) (λx. f x) (λy. g y)
    using assms unfolding Rel-def rel-fun-def by fast

42.3 Predicates on relations, i.e. “class constraints”

definition left-total :: ('a ⇒ 'b ⇒ bool) ⇒ bool
    where
        left-total R ≡ (∀ x. ∃ y. R x y)
THEORY "Transfer"

definition left-unique :: ('a ⇒ 'b ⇒ bool) ⇒ bool
  where left-unique R ←→ (∀ x y z. R x z −→ R y z −→ x = y)

definition right-total :: ('a ⇒ 'b ⇒ bool) ⇒ bool
  where right-total R ←→ (∀ y. ∃ x. R x y)

definition right-unique :: ('a ⇒ 'b ⇒ bool) ⇒ bool
  where right-unique R ←→ (∃ y. ∀ x. R x y ⇒ y = x)

definition bi-total :: ('a ⇒ 'b ⇒ bool) ⇒ bool
  where bi-total R ←→ (∀ x. ∃ y. R x y) ∧ (∀ y. ∃ x. R x y)

definition bi-unique :: ('a ⇒ 'b ⇒ bool) ⇒ bool
  where bi-unique R ←→ (∀ x y z. R x y −→ R x z −→ x = y) ∧ (∀ x y z. R x z −→ R y z −→ y = x)

lemma left-unique iff: left-unique R ←→ (∃ y. ∀ x. R x y)
unfolding Uniq-def left-unique-def by force

lemma left-unique I: (∀ x y z. [ A x z; A y z ] −→ x = y) −→ left-unique A
unfolding left-unique-def by blast

lemma left-unique D: [ left-unique A; A x z; A y z ] −→ x = y
unfolding left-unique-def by blast

lemma left-total I:
(∀ x. ∃ y. R x y) −→ left-total R
unfolding left-total-def by blast

lemma left-total E:
assumes left-total R
obtains (∀ x. ∃ y. R x y)
using assms unfolding left-total-def by blast

lemma bi-unique D r: [ bi-unique A; A x y; A x z ] −→ y = z
by(simp add: bi-unique-def)

lemma bi-unique D l: [ bi-unique A; A x y; A z y ] −→ x = z
by(simp add: bi-unique-def)

lemma bi-unique iff: bi-unique R ←→ (∃ y. ∀ x. R x y) ∧ (∀ z. ∃ x. R x z)
unfolding Uniq-def bi-unique-def by force

lemma right-unique iff: right-unique R ←→ (∃ x. ∀ z. R x z)
unfolding Uniq-def right-unique-def by force

lemma right-unique I: (∀ x y z. [ A x y; A x z ] −→ y = z) −→ right-unique A
unfolding right-unique-def by fast

lemma right-uniqueD: \[ \text{right-unique } A; \ A x y; \ A x z \] \implies y = z
unfolding right-unique-def by fast

lemma right-totalI: \( \forall y. \exists x. \ A x y \) \implies right-total A
by(simp add: right-total-def)

lemma right-totalE:
  assumes right-total A
  obtains x where A x y
using assms by(auto simp add: right-total-def)

lemma right-total-alt-def2:
  right-total R \iff \( (R \implies (\cdots \implies (\cdots) \implies \cdots)) \) All All (is ?lhs = ?rhs)
proof
  assume ?lhs then show ?rhs
    unfolding right-total-def rel-fun-def by blast
next
  assume §: ?rhs
  show ?lhs
    using § [unfolded rel-fun-def, rule-format, of \A x. True \A y. \exists x. R x y]
    unfolding right-total-def by blast
qed

lemma right-unique-alt-def2:
  right-unique R \iff \( (R \implies \cdots) \implies \cdots \implies \cdots \) (=) (=)
unfolding right-unique-def rel-fun-def by auto

lemma bi-total-alt-def2:
  bi-total R \iff \( (R \implies \cdots) \implies \cdots \implies \cdots \) (=) (=)
unfolding right-unique-def rel-fun-def by auto

lemma bi-unique-alt-def2:
  bi-unique R \iff \( (R \implies \cdots) \implies \cdots \implies \cdots \) (=) (=)
unfolding right-unique-def rel-fun-def by auto

lemma bi-unique-alt-def2:
  bi-unique R \iff \( (R \implies \cdots) \implies \cdots \implies \cdots \) (=) (=)
unfolding right-unique-def rel-fun-def by auto

lemma [simp]:
  shows left-unique-conversep: left-unique A\(^{-1-1}\) \iff right-unique A
  and right-unique-conversep: right-unique A\(^{-1-1}\) \iff left-unique A
by (auto simp add: left-unique-def right-unique-def)

lemma [simp]:
  shows left-total-conversep: \( left\text{-}total\ A \sim^{-1} \iff right\text{-}total\ A \)
  and right-total-conversep: \( right\text{-}total\ A \sim^{-1} \iff left\text{-}total\ A \)
  by (simp-all add: left-total-def right-total-def)

lemma bi-unique-conversep [simp]: \( bi\text{-}unique\ R \sim^{-1} = bi\text{-}unique\ R \)
  by (auto simp add: bi-unique-def)

lemma bi-total-conversep [simp]: \( bi\text{-}total\ R \sim^{-1} = bi\text{-}total\ R \)
  by (auto simp add: bi-total-def)

lemma right-unique-alt-def: \( right\text{-}unique\ R = (\text{conversep}\ R \sim\ O\ O\ R \leq (=)) \)
  unfolding right-unique-def by blast

lemma left-unique-alt-def: \( left\text{-}unique\ R = (R \sim\ O\ O\ (\text{conversep}\ R) \leq (=)) \)
  unfolding left-unique-def by blast

lemma right-total-alt-def: \( right\text{-}total\ R = (\text{conversep}\ R \sim\ O\ O\ R \geq (=)) \)
  unfolding right-total-def by blast

lemma left-total-alt-def: \( left\text{-}total\ R = (R \sim\ O\ O\ (\text{conversep}\ R) \geq (=)) \)
  unfolding left-total-def by blast

lemma bi-total-alt-def: \( bi\text{-}total\ A = (left\text{-}total\ A \land right\text{-}total\ A) \)
  unfolding bi-total-alt-def by blast

lemma bi-unique-alt-def: \( bi\text{-}unique\ A = (left\text{-}unique\ A \land right\text{-}unique\ A) \)
  unfolding bi-unique-alt-def by blast

lemma bi-totalI: \( left\text{-}total\ R \Rightarrow right\text{-}total\ R \Rightarrow bi\text{-}total\ R \)
  unfolding bi-total-alt-def ..

lemma bi-uniqueI: \( left\text{-}unique\ R \Rightarrow right\text{-}unique\ R \Rightarrow bi\text{-}unique\ R \)
  unfolding bi-unique-alt-def ..

end

lemma is-equality-lemma: \( (\forall R. \text{is-equality}\ R \Rightarrow PROP\ (P\ R)) \equiv PROP\ (P\ (=)) \)
  unfolding is-equality-def

proof (rule equal-intr-rule)
  show \( (\forall R. R = (=) \Rightarrow PROP\ P\ R) \Rightarrow PROP\ P\ (=) \)
    apply (drule meta-spec)
    apply (erule meta-mp [OF - refl])
  done
qed simp

lemma Domainp-lemma: \( (\forall R. \text{Domainp}\ T = R \Rightarrow PROP\ (P\ R)) \equiv PROP\ (P\ \text{(Domainp}\ T)) \)
proof (rule equal-intr-rule)
  show (∀ R. Domainp T = R ⟷ PROP P R) ⟷ PROP P (Domainp T)
  apply (erule meta-spec)
  apply (erule meta-mp [OF - refl])
  done
qed simp

ML-file ‹Tools/Transfer/transfer.ML›
declare refl [transfer-rule]

hide-const (open) Rel

context includes lifting-syntax
begin

Handling of domains

lemma Domainp-iff: Domainp T x ⟷ (∃ y. T x y)
  by auto

lemma Domainp-refl[transfer-domain-rule]:
  Domainp T = Domainp T ..

lemma Domain-eq-top[transfer-domain-rule]: Domainp (=) = top by auto

lemma Domain-pred-fun-eq[relator-domain]:
  assumes left-unique T
  shows Domainp (T ===> S) = pred-fun (Domainp T) (Domainp S) 
  (is ⌧lhs = ⌧rhs)
  proof (intro ext iffI)
  fix x
  assume ⌧lhs x
  then show ⌧rhs x
    using assms unfolding rel-fun-def pred-fun-def by blast
next
  fix x
  assume ⌧rhs x
  then have ∃ g. ∀ y xa. T xa y ⟶ S (x xa) (g y)
    using assms unfolding Domainp-iff left-unique-def pred-fun-def
    by (intro choice) blast
  then show ⌧lhs x
    by blast
qed

Properties are preserved by relation composition.

lemma OO-def: R OO S = (λx z. ∀ y. R x y ∧ S y z)
  by auto

lemma bi-total-OO: [bi-total A; bi-total B] ⟷ bi-total (A OO B)
  unfolding bi-total-def OO-def by fast
THEORY "Transfer"

lemma bi-unique-OO: \[[\text{bi-unique } A; \text{bi-unique } B]\] \implies \text{bi-unique } (A OO B)
   unfolding bi-unique-def OO-def by blast

lemma right-total-OO:
   \[[\text{right-total } A; \text{right-total } B]\] \implies \text{right-total } (A OO B)
   unfolding right-total-def OO-def by fast

lemma right-unique-OO:
   \[[\text{right-unique } A; \text{right-unique } B]\] \implies \text{right-unique } (A OO B)
   unfolding right-unique-def OO-def by fast

lemma left-total-OO: \text{left-total } R \implies \text{left-total } S \implies \text{left-total } (R OO S)
   unfolding left-total-def OO-def by fast

lemma left-unique-OO: \text{left-unique } R \implies \text{left-unique } S \implies \text{left-unique } (R OO S)
   unfolding left-unique-def OO-def by blast

42.4 Properties of relators

lemma left-total-eq[transfer-rule]: \text{left-total } (=)
   unfolding left-total-def by blast

lemma left-unique-eq[transfer-rule]: \text{left-unique } (=)
   unfolding left-unique-def by blast

lemma right-total-eq [transfer-rule]: \text{right-total } (=)
   unfolding right-total-def by simp

lemma right-unique-eq [transfer-rule]: \text{right-unique } (=)
   unfolding right-unique-def by simp

lemma bi-total-eq[transfer-rule]: \text{bi-total } (=)
   unfolding bi-total-def by simp

lemma bi-unique-eq[transfer-rule]: \text{bi-unique } (=)
   unfolding bi-unique-def by simp

lemma left-total-fun[transfer-rule]:
   assumes \text{left-unique } A \text{ left-total } B
   shows \text{left-total } (A ===> B)
   unfolding left-total-def
proof
  fix f
  show \text{Ex } ((A ===> B) f)
     unfolding rel-fun-def
proof (intro exI strip)
  fix x y
  assume A: A x y
have \((\text{THE } x. A x y) = x\)
  using \(A\) assms by (simp add: left-unique-def the-equality)
then show \(B (f x) (\text{SOME } z. B (f (\text{THE } x. A x y))) z\)
  using assms by (force simp: left-total-def intro: someI-ex)
qed

lemma left-unique-fun [transfer-rule]:
  \(\text{[left-total } A; \text{left-unique } B] \implies \text{left-unique } (A \Longrightarrow B)\)
unfolding left-total-def left-unique-def rel-fun-def
by (clarify, rule ext, fast)

lemma right-total-fun [transfer-rule]:
  assumes right-unique \(A\) right-total \(B\)
shows right-total \((A \Longrightarrow B)\)
unfolding right-total-def
proof
  fix \(g\)
  show \(\exists x. (A \Longrightarrow B) x g\)
    unfolding rel-fun-def
  proof (intro exI strip)
    fix \(x y\)
    assume \(A: A x y\)
    have \((\text{THE } y. A x y) = y\)
      using \(A\) assms by (simp add: right-unique-def the-equality)
    then show \(B \text{ SOME } z. B z (g (\text{THE } y. A x y))) (g y)\)
      using assms by (force simp: right-total-def intro: someI-ex)
  qed
qed

lemma right-unique-fun [transfer-rule]:
  \(\text{[right-total } A; \text{right-unique } B] \implies \text{right-unique } (A \Longrightarrow B)\)
unfolding right-total-def right-unique-def rel-fun-def
by (clarify, rule ext, fast)

lemma bi-total-fun [transfer-rule]:
  \(\text{[bi-unique } A; \text{bi-total } B] \implies \text{bi-total } (A \Longrightarrow B)\)
unfolding bi-unique-all-def bi-total-all-def
by (blast intro: right-total-fun left-total-fun)

lemma bi-unique-fun [transfer-rule]:
  \(\text{[bi-total } A; \text{bi-unique } B] \implies \text{bi-unique } (A \Longrightarrow B)\)
unfolding bi-unique-all-def bi-total-all-def
by (blast intro: right-unique-fun left-unique-fun)

end

lemma if-conn:
  \((\text{if } P \land Q \text{ then } t \text{ else } e) = (\text{if } P \text{ then } Q \text{ then } t \text{ else } e \text{ else } e)\)
(if $P \lor Q$ then $t$ else $e$) = (if $P$ then $t$ else if $Q$ then $t$ else $e$) 
(if $P \rightarrow Q$ then $t$ else $e$) = (if $P$ then if $Q$ then $t$ else $e$ else $t$) 
(if $\neg P$ then $t$ else $e$) = (if $P$ then $e$ else $t$) 
by auto

ML-file ⟨Tools/Transfer/transfer-bnf.ML⟩
ML-file ⟨Tools/BNF/bnf-fp-rec-sugar-transfer.ML⟩

declare pred-fun-def [simp]
declare rel-fun-eq [relator-eq]
declare fun. Domainp-rel [relator-domain def]

42.5 Transfer rules

context includes lifting-syntax
begin

lemma Domainp-forall-transfer [transfer-rule]:
assumes right-total $A$
shows $((A ===> (=)) ===> (=))$
  (transfer-bforall (Domainp $A$)) transfer-forall
using assms unfolding right-total-def
unfolding transfer-forall-def transfer-bforall-def rel-fun-def Domainp-iff
by fast

Transfer rules using implication instead of equality on booleans.

lemma transfer-forall-transfer [transfer-rule]:
bi-total $A \Rightarrow ((A ===> (=)) ===> (=))$ transfer-forall transfer-forall
right-total $A \Rightarrow ((A ===> (=)) ===> implies)$ transfer-forall transfer-forall
right-total $A \Rightarrow ((A ===> implies) ===> implies)$ transfer-forall transfer-forall
bi-total $A \Rightarrow ((A ===> (=)) ===> rev-implies)$ transfer-forall transfer-forall
bi-total $A \Rightarrow ((A ===> rev-implies) ===> rev-implies)$ transfer-forall transfer-forall
unfolding transfer-forall-def rev-implies-def rel-fun-def right-total-def bi-total-def
by fast+

lemma transfer-implies-transfer [transfer-rule]:
$((=) ===> (=)) ===> (=)$ transfer-implies transfer-implies
(rev-implies ===> implies) ===> implies transfer-implies transfer-implies
(rev-implies ===> (=) ===> implies) transfer-implies transfer-implies
((=) ===> implies ===> implies) transfer-implies transfer-implies
((=) ===> (=) ===> implies) transfer-implies transfer-implies
(implies ===> rev-implies ===> rev-implies) transfer-implies transfer-implies
(implies ===> (=) ===> rev-implies) transfer-implies transfer-implies
(implies ===> rev-implies ===> (=)) transfer-implies transfer-implies
((=) ===> rev-implies ===> rev-implies) transfer-implies transfer-implies
unfolding transfer-implies-def rev-implies-def rel-fun-def by auto

lemma eq-imp-transfer [transfer-rule]:
right-unique A \implies (A \implies A (\rightarrow\rightarrow) (\rightarrow)) (\rightarrow) (\rightarrow)
unfolding right-unique-alt-def2.

Transfer rules using equality.

lemma left-unique-transfer [transfer-rule]:
assumes right-total A
assumes right-total B
assumes bi-unique A
shows ((A \implies B \implies (\rightarrow)) \implies (\rightarrow)) left-unique left-unique
using assms unfolding left-unique-def right-total-def bi-unique-def rel-fun-def by metis

lemma eq-transfer [transfer-rule]:
assumes bi-unique A
shows (A \implies A (\rightarrow) (\rightarrow) (\rightarrow)) (\rightarrow) (\rightarrow)
using assms unfolding bi-unique-def rel-fun-def by auto

lemma right-total-Ex-transfer [transfer-rule]:
assumes right-total A
shows ((A \implies (\rightarrow)) \implies (\rightarrow)) (\rightarrow) (\rightarrow) Ex
using assms unfolding right-total-def Bex-def rel-fun-def Domainp-iff by fast

lemma right-total-All-transfer [transfer-rule]:
assumes right-total A
shows ((A \implies (\rightarrow)) \implies (\rightarrow)) (\rightarrow) (\rightarrow) All
using assms unfolding right-total-def Ball-def rel-fun-def Domainp-iff by fast

context
includes lifting-syntax
begin

lemma right-total-fun-eq-transfer:
assumes [transfer-rule]: right-total A bi-unique B
shows ((A \implies B) \implies (A \implies B) \implies (\rightarrow)) (\rightarrow) (\rightarrow) (\rightarrow) (\rightarrow) (\rightarrow)
unfolding fun-eq-iff
by transfer-prover

end

lemma All-transfer [transfer-rule]:
assumes bi-total A
shows ((A \implies (\rightarrow)) \implies (\rightarrow)) All All
using assms unfolding bi-total-def rel-fun-def by fast
lemma Ex-transfer [transfer-rule]:
  assumes bi-total A
  shows \((A \Longrightarrow (=)) \Longrightarrow (=)\) Ex Ex
  using assms unfolding bi-total-def rel-fun-def by fast

lemma Ex1-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-unique A bi-total A
  shows \((A \Longrightarrow (=)) \Longrightarrow (=)\) Ex1 Ex1
  unfolding Ex1-def by transfer-prover

declare If-transfer [transfer-rule]

lemma Let-transfer [transfer-rule]: \((A \Longrightarrow (A \Longrightarrow B)) \Longrightarrow B\) Let Let
  unfolding rel-fun-def by simp

declare id-transfer [transfer-rule]

declare comp-transfer [transfer-rule]

lemma curry-transfer [transfer-rule]:
  \(((\text{rel-prod } A B \Longrightarrow C) \Longrightarrow A \Longrightarrow B \Longrightarrow C)\) curry curry
  unfolding curry-def by transfer-prover

lemma fun-upd-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows \((A \Longrightarrow B) \Longrightarrow A \Longrightarrow B \Longrightarrow A \Longrightarrow B)\) fun-upd fun-upd
  unfolding fun-upd-def by transfer-prover

lemma case-nat-transfer [transfer-rule]:
  \((A \Longrightarrow ((=) \Longrightarrow A) \Longrightarrow A) \Longrightarrow (=) \Longrightarrow A)\) case-nat case-nat
  unfolding rel-fun-def by (simp split: nat.split)

lemma rec-nat-transfer [transfer-rule]:
  \((A \Longrightarrow ((=) \Longrightarrow A \Longrightarrow A) \Longrightarrow (=) \Longrightarrow A)\) rec-nat rec-nat
  unfolding rel-fun-def
  apply safe
  subgoal for - - - - - n
    by (induction n) simp-all
  done

lemma funpow-transfer [transfer-rule]:
  \(((=) \Longrightarrow (A \Longrightarrow A) \Longrightarrow (A \Longrightarrow A))\) compow compow
  unfolding funpow-def by transfer-prover

lemma mono-transfer[transfer-rule]:
  assumes [transfer-rule]: bi-total A
  assumes [transfer-rule]: \((A \Longrightarrow A \Longrightarrow (=))\) \((\leq)\) \((\leq)\)
assumes \{ transfer-rule \}: \( B \Rightarrow \Rightarrow B \Rightarrow \Rightarrow (=) \) \( \leq \) \( \leq \)
shows \( (A \Rightarrow \Rightarrow B) \Rightarrow \Rightarrow (=) \) mono mono
unfolding mono-def by transfer-prover

lemma right-total-relcompp-transfer[transfer-rule]:
assumes \{ transfer-rule \}: right-total B
shows \( (A \Rightarrow \Rightarrow B \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow (B \Rightarrow \Rightarrow C \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow A \Rightarrow \Rightarrow C \Rightarrow \Rightarrow (=) \)
\( \lambda\exists R S x z. y \in \text{Collect}(\text{Domainp} \ B). R x y \wedge S y z \) \( \text{(OO)} \)
unfolding OO-def by transfer-prover

lemma relcompp-transfer[transfer-rule]:
assumes \{ transfer-rule \}: bi-total B
shows \( (A \Rightarrow \Rightarrow B \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow (B \Rightarrow \Rightarrow C \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow A \Rightarrow \Rightarrow C \Rightarrow \Rightarrow (=) \) \( \text{(OO)} \) \( \text{(OO)} \)
unfolding OO-def by transfer-prover

lemma right-total-Domainp-transfer[transfer-rule]:
assumes \{ transfer-rule \}: right-total B
shows \( (A \Rightarrow \Rightarrow B \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow A \Rightarrow \Rightarrow (=) \) \( \lambda T x. \exists y \in \text{Collect}(\text{Domainp} \ B). T x y \) \( \text{Domainp} \)
apply\( (\text{subst}(2) \ \text{Domainp-iff}[\text{abs-def}]) \) by transfer-prover

lemma Domainp-transfer[transfer-rule]:
assumes \{ transfer-rule \}: bi-total B
shows \( (A \Rightarrow \Rightarrow B \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow A \Rightarrow \Rightarrow (=) \) \( \text{Domainp} \ \text{Domainp} \)
unfolding Domainp-iff by transfer-prover

lemma reflp-transfer[transfer-rule]:
bi-total A \( \Rightarrow ((A \Rightarrow \Rightarrow A \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow (=)) \) reflp reflp
right-total A \( \Rightarrow ((A \Rightarrow \Rightarrow A \Rightarrow \Rightarrow \text{implies}) \Rightarrow \Rightarrow \text{implies}) \) reflp reflp
right-total A \( \Rightarrow ((A \Rightarrow \Rightarrow A \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow \text{implies}) \) reflp reflp
bi-total A \( \Rightarrow ((A \Rightarrow \Rightarrow A \Rightarrow \Rightarrow \text{rev-implies}) \Rightarrow \Rightarrow \text{rev-implies}) \) reflp reflp
bi-total A \( \Rightarrow ((A \Rightarrow \Rightarrow A \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow \text{rev-implies}) \) reflp reflp
unfolding reflp-def rev-implies-def bi-total-def right-total-def rel-fun-def
by fast+

lemma right-unique-transfer [transfer-rule]:
\[ \text{right-total} A; \text{right-total} B; \text{bi-unique} B \]
\( \Rightarrow ((A \Rightarrow \Rightarrow B \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow \text{implies}) \) right-unique right-unique
unfolding right-unique-def right-total-def bi-unique-def rel-fun-def
by metis

lemma left-total-parametric [transfer-rule]:
assumes \{ transfer-rule \}: bi-total A bi-total B
shows \( (A \Rightarrow \Rightarrow B \Rightarrow \Rightarrow (=)) \Rightarrow \Rightarrow (=) \) left-total left-total
unfolding left-total-def by transfer-prover

lemma right-total-parametric [transfer-rule]:
assumes [transfer-rule]: bi-total A bi-total B
shows ((A ===> B ===> (=)) ===> (=)) right-total right-total
unfolding right-total-def by transfer-prover

lemma left-unique-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-unique A bi-total A bi-total B
  shows ((A ===> B ===> (=)) ===> (=)) left-unique left-unique
unfolding left-unique-def by transfer-prover

lemma prod-pred-parametric [transfer-rule]:
  ((A ===> (=)) ===> (B ===> (=)) ===> rel-prod A B ===> (=))
pred-prod pred-prod
unfolding prod.pod-set Basic-BNFs.fstsp-def Basic-BNFs sndsp-def fstp.simps sndsp.simps
by simp transfer-prover

lemma apfst-parametric [transfer-rule]:
  ((A ===> B) ===> rel-prod A C ===> rel-prod B C) apfst apfst
unfolding apfst-def by transfer-prover

lemma rel-fun-eq-eq-onp: ((=) ===> eq-onp P) = eq-onp (λf. ∀x. P(f x))
unfolding eq-onp-def rel-fun-def by auto

lemma rtranclp-parametric [transfer-rule]:
  assumes bi-unique A bi-total A
  shows ((A ===> A ===> (=)) ===> A ===> A ===> (=)) rtranclp
rtranclp
proof (rule rel-funI jffI)+
fix R :: 'a ⇒ 'a ⇒ bool and R' x y x' y'
assume R: (A ===> A ===> (=)) R R' and A x x'
{
  assume R" x y A y y'
  thus R"" x' y'
proof (induction arbitrary: y')
    case base
    with ⟨bi-unique A⟩ ⟨A x x'⟩ have x' = y' by (rule bi-uniqueDr)
    thus ?case by simp
next
  case (step y z z')
  from bi-total A obtain y' where A y y' unfolding bi-total-def by blast
hence $R^{**} x' y'$ by (rule step.IH)
moreover from $R (\langle A x y' \rangle \triangleleft \langle A z z' \rangle \triangleleft R y z)$$
have $R' y' z'$ by (auto dest: rel-funD)
ultimately show $\{\text{case ..}$$
qed
next
assume $R^{**} x' y' A y'$
thus $R^{**} x y$
proof (induction arbitrary: $y$)
  case base
  with $\langle \text{bi-unique } A \rangle \langle A x x' \rangle$ have $x = y$ by (rule bi-uniqueDl)
  thus $\{\text{case by simp}$$
next
  case (\text{step } y' z' z)
  from $\langle \text{bi-total } A \rangle$ obtain $y$ where $A y y'$ unfolding bi-total-def by blast
  hence $R^{**} x y$ by (rule step.IH)
  moreover from $R (\langle A y y' \rangle \triangleleft \langle A z z' \rangle \triangleleft R' y' z')$$
  have $R y z$ by (auto dest: rel-funD)
  ultimately show $\{\text{case ..}$$
  qed
}\)
 qed

lemma right-unique-parametric [transfer-rule]:
assumes [transfer-rule]: $\text{bi-total } A \text{ bi-unique } B \text{ bi-total } B$
shows $\langle A \Rightarrow> B \Rightarrow> (\Rightarrow) \Rightarrow> (\Rightarrow) \Rightarrow> (\Rightarrow) \Rightarrow> \text{right-unique right-unique}$
unfolding right-unique-def by transfer-prover

lemma map-fun-parametric [transfer-rule]:
$\langle A \Rightarrow> B \Rightarrow> (\Rightarrow) \Rightarrow> (\Rightarrow) \Rightarrow> (\Rightarrow) \Rightarrow> A \Rightarrow> D \Rightarrow> D \Rightarrow> \text{map-fun map-fun}$
unfolding map-fun-def by transfer-prover

end

42.6 of-bool and of-nat

context
  includes lifting-syntax
begin

lemma transfer-rule-of-bool:
  $\langle \langle \equiv \equiv \rangle \Rightarrow> \text{of-bool} \Rightarrow> \text{of-bool} \rangle$
  if [transfer-rule]: $\langle 0 \equiv 0 \rangle \Rightarrow> 1 \Rightarrow> 1$
  for $R :: \langle \text{a::zero-neq-one } \Rightarrow> \text{b::zero-neq-one } \Rightarrow> \text{bool} \rangle$ (infix $\equiv$ 50)
  unfolding of-bool-def by transfer-prover

lemma transfer-rule-of-nat:
  $\langle \equiv \Rightarrow> \text{of-nat} \Rightarrow> \text{of-nat} \rangle$
if [transfer-rule]: \langle 0 \equiv 0 \rangle \langle 1 \equiv 1 \rangle
\langle (\equiv) \Longrightarrow (\equiv) \rangle \langle (\equiv) \rangle (\langle + \rangle (\langle + \rangle)
for R :: \langle 'a::semiring-1 \Rightarrow 'b::semiring-1 \Rightarrow bool \rangle \ (infix (\equiv) 50)
unfolding of-nat-def by transfer-prover
end
end

43 Lifting package

theory Lifting
imports Equiv-Relations Transfer
keywords
  parametric and
  print-quot-maps print-quotients :: diag and
  lift-definition :: thy-goal-defn and
  setup-lifting lifting-forget lifting-update :: thy-decl
begin

43.1 Function map

context includes lifting-syntax
begin

lemma map-fun-id:
  (id \Longrightarrow id) = id
by (simp add: fun-eq-iff)

43.2 Quotient Predicate

definition
  Quotient R Abs Rep T \iff
  (\forall a. Abs (Rep a) = a) \land
  (\forall a. R (Rep a) (Rep a)) \land
  (\forall r s. R r s \Longrightarrow R r r \land R s s \land Abs r = Abs s) \land
  T = (\lambda x y. R x x \land Abs x = y)

lemma QuotientI:
assumes \langle a. Abs (Rep a) = a \rangle
and \langle a. R (Rep a) (Rep a) \rangle
and \langle r s. R r s \Longrightarrow R r r \land R s s \land Abs r = Abs s \rangle
and T = (\lambda x y. R x x \land Abs x = y)
shows Quotient R Abs Rep T
using assms unfolding Quotient-def by blast

context

fixes R Abs Rep T
assumes a: Quotient R Abs Rep T
begin

lemma Quotient-abs-rep: Abs (Rep a) = a
  using a unfolding Quotient-def
  by simp

lemma Quotient-rep-reflp: R (Rep a) (Rep a)
  using a unfolding Quotient-def
  by blast

lemma Quotient-rel:
  R r r ∧ R s s ∧ Abs r = Abs s ⟷ R r s — orientation does not loop on rewriting
  using a unfolding Quotient-def
  by blast

lemma Quotient-cr-rel: T = (λx y. R x x ∧ Abs x = y)
  using a unfolding Quotient-def
  by blast

lemma Quotient-refl1: R r s ⟷ R r r
  using a unfolding Quotient-def
  by fast

lemma Quotient-refl2: R r s ⟷ R s s
  using a unfolding Quotient-def
  by fast

lemma Quotient-rel-rep: R (Rep a) (Rep b) ⟷ a = b
  using a unfolding Quotient-def
  by metis

lemma Quotient-rep-abs: R r r ⟷ R (Rep (Abs r)) r
  using a unfolding Quotient-def
  by blast

lemma Quotient-rep-abs-eq: R t t ⟷ R ≤ (=) ⟷ Rep (Abs t) = t
  using a unfolding Quotient-def
  by blast

lemma Quotient-rep-abs-fold-unmap:
  assumes x' ≡ Abs x and R x x and Rep x' ≡ Rep' x'
  shows R (Rep' x') x
  proof —
   have R (Rep x') x using assms(1-2) Quotient-rep-abs by auto
   then show  "thesis" using assms(3) by simp
   qed

lemma Quotient-Rep-eq:
  assumes x' ≡ Abs x
shows \(\text{Rep } x' \equiv \text{Rep } x'\)
by simp

lemma Quotient-rel-abs: \(R \ r \ s \implies \text{Abs } r = \text{Abs } s\)
  using a unfolding Quotient-def
by blast

lemma Quotient-rel-abs2:
  assumes \(R (\text{Rep } x) \ y\)
  shows \(x = \text{Abs } y\)
proof –
  from assms have \(\text{Abs } (\text{Rep } x) = \text{Abs } y\)
  by (auto intro: Quotient-rel-abs)
  then show \(?\text{thesis using } \text{assms}(1)\)
  by (simp add: Quotient-abs-rep)
qed

lemma Quotient-symp: symp \(R\)
  using a unfolding Quotient-def using sympI
  by (metis (full-types))

lemma Quotient-transp: transp \(R\)
  using a unfolding Quotient-def using transpI
  by (metis (full-types))

lemma Quotient-part-equivp: part-equivp \(R\)
  by (metis Quotient-rep-reflp Quotient-symp Quotient-transp part-equivpI)

end

lemma identity-quotient: Quotient \(=\) \(\text{id} \ \text{id} =\)
unfolding Quotient-def by simp

TODO: Use one of these alternatives as the real definition.

lemma Quotient-alt-def:
  Quotient \(R\) \(\text{Abs} \ \text{Rep} \ T \longleftrightarrow\)
  \((\forall a \ b. \ T \ a \ b \implies \text{Abs } a = b) \land\)
  \((\forall b. \ T (\text{Rep } b) \ b) \land\)
  \((\forall x \ y. \ R \ x \ y \longleftrightarrow T x (\text{Abs } x) \land T y (\text{Abs } y) \land \text{Abs } x = \text{Abs } y)\)
apply safe
apply (simp (no-asm-use) only: Quotient-def, fast)
apply (simp (no-asm-use) only: Quotient-def, fast)
apply (simp (no-asm-use) only: Quotient-def, fast)
apply (simp (no-asm-use) only: Quotient-def, fast)
apply (simp (no-asm-use) only: Quotient-def, fast)
apply (simp (no-asm-use) only: Quotient-def, fast)
apply (rule QuotientI)
apply simp
apply metis
apply simp
apply (rule ext, rule ext, metis)
done
lemma Quotient-alt-def2:
Quotient R Abs Rep T \leftrightarrow 
(\forall a b. T a b \rightarrow Abs a = b) \land
(\forall b. T (Rep b) b) \land
(\forall x y. R x y \leftrightarrow T x (Abs y) \land T y (Abs x))
unfolding Quotient-alt-def by (safe, metis+)

lemma Quotient-alt-def3:
Quotient R Abs Rep T \leftrightarrow 
(\forall a b. T a b \rightarrow Abs a = b) \land
(\exists b. T (Rep b) b) \land
(\forall x y. R x y \leftrightarrow (\exists z. T x z \land T y z))
unfolding Quotient-alt-def2 by (safe, metis+)

lemma Quotient-alt-def4:
Quotient R Abs Rep T \leftrightarrow 
(\forall a b. T a b \rightarrow Abs a = b) \land
(\forall b. T (Rep b) b) \land
R = T OO conversep T
unfolding Quotient-alt-def3 fun-eq-iff by auto

lemma Quotient-alt-def5:
Quotient R Abs Rep T \leftrightarrow 
T \leq BNF-Def.Grp UNIV Abs \land BNF-Def.Grp UNIV Rep \leq T^{-1} \land R = T OO T^{-1}
unfolding Quotient-alt-def4 Grp-def by blast

lemma fun-quotient:
assumes 1: Quotient R1 abs1 rep1 T1
assumes 2: Quotient R2 abs2 rep2 T2
shows Quotient (R1 ===> R2) (rep1 ===> abs2) (abs1 ===> rep2) (T1 ===> T2)
using assms unfolding Quotient-alt-def2
unfolding rel-fun-def fun-eq-iff map-fun-apply
by (safe, metis+)

lemma apply-rsp:
fixes f g :: 'a \Rightarrow 'c
assumes q: Quotient R1 Abs1 Rep1 T1
and a: (R1 ===> R2) f g R1 x y
shows R2 (f x) (g y)
using a by (auto elim: rel-funE)

lemma apply-rsp':
assumes a: (R1 ===> R2) f g R1 x y
shows R2 (f x) (g y)
using a by (auto elim: rel-funE)

lemma apply-rsp'':
assumes Quotient R Abs Rep T
and (R ===> S) f f
shows S (f (Rep x)) (f (Rep x))
proof –
from assms(1) have R (Rep x) (Rep x) by (rule Quotient-rep-reflp)
then show ?thesis using assms(2) by (auto intro: apply-rsp')
qed

43.3 Quotient composition

lemma Quotient-compose:
  assumes 1: Quotient R1 Abs1 Rep1 T1
  assumes 2: Quotient R2 Abs2 Rep2 T2
  shows  Quotient (T1 OO R2 OO conversep T1) (Abs2 ∘ Abs1) (Rep1 ∘ Rep2)
           (T1 OO T2)
  using assms unfolding Quotient-alt-def4 by fastforce

lemma equivp-reflp2:
equivp R ⇒ reflp R
by (erule equivpE)

43.4 Respects predicate

definition Respects :: (′a ⇒ ′a ⇒ bool) ⇒ ′a set
where Respects R = {x. R x x}

lemma in-respects: x ∈ Respects R ↔ R x x
unfolding Respects-def by simp

lemma UNIV-typedef-to-Quotient:
  assumes type-definition Rep Abs UNIV
  and T-def: T ≡ (λx y. x = Rep y)
  shows Quotient (=) Abs Rep T
proof –
  interpret type-definition Rep Abs UNIV by fact
  from Abs-inject Rep-inverse Abs-inverse T-def show ?thesis
  by (fastforce intro!: QuotientI fun-eq-iff)
qed

lemma UNIV-typedef-to-equivp:
  fixes Abs :: ′a ⇒ ′b
  and Rep :: ′b ⇒ ′a
  assumes type-definition Rep Abs (UNIV::′a set)
  shows equivp ((=) ::′a⇒′a⇒bool)
by (rule identity-equivp)

lemma typedef-to-Quotient:
  assumes type-definition Rep Abs S
  and T-def: T ≡ (λx y. x = Rep y)
  shows Quotient (eq-onp (λx. x ∈ S)) Abs Rep T
proof –
  interpret type-definition Rep Abs S by fact
  from Rep Abs-inject Rep-inverse Abs-inverse T-def show ?thesis
by (auto intro: QuotientI simp: eq-onp-def fun-eq-iff)
qed

lemma typedef-to-part-equivp:
  assumes type-definition Rep Abs S
  shows part-equivp (eq-onp (λx. x ∈ S))
proof (intro part-equivpI)
  interpret type-definition Rep Abs S by fact
  show ∃x. eq-onp (λx. x ∈ S) x x using Rep by (auto simp: eq-onp-def)
next
  show symp (eq-onp (λx. x ∈ S)) by (auto intro: sympI simp: eq-onp-def)
next
  show transp (eq-onp (λx. x ∈ S)) by (auto intro: transpI simp: eq-onp-def)
qed

lemma open-typedef-to-Quotient:
  assumes type-definition Rep Abs {x. P x}
  and T-def: T ≡ (λx y. x = Rep y)
  shows Quotient (eq-onp P) Abs Rep T
  using typedef-to-Quotient [OF assms] by simp

lemma open-typedef-to-part-equivp:
  assumes type-definition Rep Abs {x. P x}
  shows part-equivp (eq-onp P)
  using typedef-to-part-equivp [OF assms] by simp

lemma type-definition-Quotient-not-empty: Quotient (eq-onp P) Abs Rep T =⇒ ∃x. P x
  unfolding eq-onp-def by (drule Quotient-rep-reflp) blast

lemma type-definition-Quotient-not-empty-witness: Quotient (eq-onp P) Abs Rep T =⇒ P (Rep undefined)
  unfolding eq-onp-def by (drule Quotient-rep-reflp) blast

Generating transfer rules for quotients.

context
  fixes R Abs Rep T
  assumes I: Quotient R Abs Rep T
begin

lemma Quotient-right-unique: right-unique T
  using I unfolding Quotient-alt-def right-unique-def by metis

lemma Quotient-right-total: right-total T
  using I unfolding Quotient-alt-def right-total-def by metis

lemma Quotient-rel-eq-transfer: (T ==⇒ T ==⇒ (=)) R (=)
  using I unfolding Quotient-alt-def rel-fun-def by simp
 lemma Quotient-abs-induct:
  assumes \( \forall y. Ry \implies P(Absy) \) shows \( Px \)
  using 1 assms unfolding Quotient-def by metis

end

Generating transfer rules for total quotients.

context
  fixes \( R Abs Rep T \)
  assumes 1: Quotient \( R Abs Rep T \) and 2: reflp \( R \)
begin

lemma Quotient-left-total: \( \text{left-total } T \)
  using 1 2 unfolding Quotient-alt-def left-total-def reflp-def by auto

lemma Quotient-bi-total: \( \text{bi-total } T \)
  using 1 2 unfolding Quotient-alt-def bi-total-def reflp-def by auto

lemma Quotient-id-abs-transfer: \((=) \Rightarrow T) (\lambda x. x) Abs
  using 1 2 unfolding Quotient-alt-def reflp-def rel-fun-def by simp

lemma Quotient-total-abs-induct: \( \forall y. P(Absy) \) \implies Px
  using 1 2 unfolding Quotient-alt-def reflp-def by metis

lemma Quotient-total-abs-eq-iff: Abs \( x \) = Abs \( y \) \iff \( R x y \)
  using Quotient-rel [OF 1] 2 unfolding reflp-def by simp

end

Generating transfer rules for a type defined with typedef.

context
  fixes \( Rep Abs A T \)
  assumes type: type-definition \( Rep Abs A \)
  assumes T-def: \( T \equiv (\lambda (x::'a) (y::'b). x = Rep y) \)
begin

lemma typedef-left-unique: \( \text{left-unique } T \)
  unfolding left-unique-def T-def
  by (simp add: type-definition.Rep-inject [OF type])

lemma typedef-bi-unique: \( \text{bi-unique } T \)
  unfolding bi-unique-def T-def
  by (simp add: type-definition.Rep-inject [OF type])

lemma typedef-right-unique: \( \text{right-unique } T \)
  using T-def type Quotient-right-unique typedef-to-Quotient
  by blast
lemma typedef-right-total: right-total T
  using T-def type Quotient-right-total typedef-to-Quotient
  by blast

lemma typedef-rep-transfer: (T ===> (=)) (λx. x) Rep
  unfolding rel-fun-def T-def by simp
end

Generating the correspondence rule for a constant defined with lift-definition.

lemma Quotient-to-transfer:
  assumes Quotient R Abs Rep T
  and R c c'
  shows T c c'
  using assms by (auto dest: Quotient-cr-rel)

Proving reflexivity

lemma Quotient-to-left-total:
  assumes q: Quotient R Abs Rep T
  and r-R: reflp R
  shows left-total T
  using r-R Quotient-cr-rel[OF q] unfolding left-total-def by (auto elim: reflpE)

lemma Quotient-composition-ge-eq:
  assumes left-total T
  assumes R ≥ (=)
  shows (T OO R OO T)^−1^−1) ≥ (=)
  using assms unfolding left-total-def by fast

lemma Quotient-composition-le-eq:
  assumes left-unique T
  assumes R ≤ (=)
  shows (T OO R OO T)^−1^−1) ≤ (=)
  using assms unfolding left-unique-def by blast

lemma eq-onp-le-eq:
  eq-onp P ≤ (=) unfolding eq-onp-def by blast

lemma reflp-ge-eq:
  reflp R ⇒ R ≥ (=) unfolding reflp-def by blast

Proving a parametrized correspondence relation

definition POS :: (′a ⇒ ′b ⇒ bool) ⇒ (′a ⇒ ′b ⇒ bool) ⇒ bool where
  POS A B ≡ A ≤ B

definition NEG :: (′a ⇒ ′b ⇒ bool) ⇒ (′a ⇒ ′b ⇒ bool) ⇒ bool where
  NEG A B ≡ B ≤ A

lemma pos-OO-eq:
shows $\text{POS} \ (A \ O O \ (=)) \ A$
unfolding $\text{POS-def \ OO-def \ by \ blast}$

lemma $\text{pos-eq-OO}$:
shows $\text{POS} \ ((=) \ O O \ A) \ A$
unfolding $\text{POS-def \ OO-def \ by \ blast}$

lemma $\text{neg-OO-eq}$:
shows $\text{NEG} \ (A \ O O \ (=)) \ A$
unfolding $\text{NEG-def \ OO-def \ by \ auto}$

lemma $\text{neg-eq-OO}$:
shows $\text{NEG} \ ((=) \ O O \ A) \ A$
unfolding $\text{NEG-def \ OO-def \ by \ auto}$

lemma $\text{POS-trans}$:
assumes $\text{POS} \ A \ B$
assumes $\text{POS} \ B \ C$
shows $\text{POS} \ A \ C$
using $\text{assms \ unfolding \ POS-def \ by \ auto}$

lemma $\text{NEG-trans}$:
assumes $\text{NEG} \ A \ B$
assumes $\text{NEG} \ B \ C$
shows $\text{NEG} \ A \ C$
using $\text{assms \ unfolding \ NEG-def \ by \ auto}$

lemma $\text{POS-NEG}$:
$\text{POS} \ A \ B \equiv \text{NEG} \ B \ A$
unfolding $\text{POS-def \ NEG-def \ by \ auto}$

lemma $\text{NEG-POS}$:
$\text{NEG} \ A \ B \equiv \text{POS} \ B \ A$
unfolding $\text{POS-def \ NEG-def \ by \ auto}$

lemma $\text{POS-per-rule}$:
assumes $\text{POS} \ (A \ O O \ B) \ C$
shows $\text{POS} \ (A \ O O \ B \ O O \ X) \ (C \ O O \ X)$
using $\text{assms \ unfolding \ POS-def \ OO-def \ by \ blast}$

lemma $\text{NEG-per-rule}$:
assumes $\text{NEG} \ (A \ O O \ B) \ C$
shows $\text{NEG} \ (A \ O O \ B \ O O \ X) \ (C \ O O \ X)$
using $\text{assms \ unfolding \ NEG-def \ OO-def \ by \ blast}$

lemma $\text{POS-apply}$:
assumes $\text{POS} \ R \ R'$
assumes $R \ f \ g$
shows $R' \ f \ g$
using assms unfolding POS-def by auto

Proving a parametrized correspondence relation

lemma fun-mono:
  assumes A ≥ C
  assumes B ≤ D
  shows (A ===> B) ≤ (C ===> D)
using assms unfolding rel-fun-def by blast

lemma pos-fun-distr: ((R ===> S) OO (R' ===> S')) ≤ ((R OO R') ===> (S OO S'))
unfolding OO-def rel-fun-def by blast

lemma functional-relation: right-unique R ===> left-total R ===> ∀x. ∃!y. R x y
unfolding right-unique-def left-total-def by blast

lemma functional-converse-relation: left-unique R ===> right-total R ===> ∀y. ∃!x. R x y
unfolding left-unique-def right-total-def by blast

lemma neg-fun-distr1:
  assumes 1: left-unique R right-total R
  assumes 2: right-unique R' left-total R'
  shows (R OO R' ===> S OO S') ≤ ((R ===> S) OO (R' ===> S'))
  unfolding rel-fun-def OO-def
  apply clarify
  apply (subst all-comm)
  apply (subst all-conj-distrib[symmetric])
  apply (intro choice)
  by metis

lemma neg-fun-distr2:
  assumes 1: right-unique R' left-total R'
  assumes 2: left-unique S' right-total S'
  shows (R OO R' ===> S OO S') ≤ ((R ===> S) OO (R' ===> S'))
  unfolding rel-fun-def OO-def
  apply clarify
  apply (subst all-comm)
  apply (subst all-conj-distrib[symmetric])
  apply (intro choice)
  by metis

43.5 Domains

lemma composed-equiv-rel-eq-onp:
  assumes left-unique R
  assumes (R ===> (=)) P P'
assumes Domainp R = P''
shows (R OO eq-onp P' OO R'^{-1}) = eq-onp (inf P'' P)
using assms unfolding OO-def conversep-iff Domainp-iff[abs-def] left-unique-def
rel-fun-def eq-onp-def
fun-eq-iff by blast

lemma composed-equiv-rel-eq-onp:
  assumes left-unique R
  assumes Domainp R = P
  shows (R OO (=) OO R'^{-1}) = eq-onp P
using assms unfolding OO-def conversep-iff Domainp-iff[abs-def] left-unique-def
eq-onp-def
fun-eq-iff is-equality-def by metis

lemma pcr-Domainp-par-left-total:
  assumes Domainp B = P
  assumes left-total A
  shows Domainp (A OO B) = P'
using assms unfolding Domainp-iff[abs-def] OO-def bi-unique-def left-total-def rel-fun-def
by (fast intro: fun-eq-iff)

lemma pcr-Domainp-par:
  assumes Domainp B = P2
  assumes Domainp A = P1
  assumes (A ===> (=)) P' P2
  shows Domainp (A OO B) = (inf P1 P2')
using assms unfolding rel-fun-def Domainp-iff[abs-def] OO-def
by (fast intro: fun-eq-iff)

definition rel-pred-comp :: ('a => 'b => bool) => ('b => bool) => 'a => bool
where rel-pred-comp R P ≡ λx. ∃ y. R x y ∧ P y

lemma pcr-Domainp:
  assumes Domainp B = P
  shows Domainp (A OO B) = (λx. ∃ y. A x y ∧ P y)
using assms by blast

lemma pcr-Domainp-total:
  assumes left-total B
  assumes Domainp A = P
  shows Domainp (A OO B) = P
using assms unfolding left-total-def
by fast

lemma Quotient-to-Domainp:
  assumes Quotient R Abs Rep T
  shows Domainp T = (λx. R x x)
lemma eq-onp-to-Domainp:
  assumes Quotient (eq-onp P) Abs Rep T
  shows Domainp T = P
  by (simp add: eq-onp-def Domainp-iff Quotient-cr-rel OF assms)

end

lemma right-total-UNIV-transfer:
  assumes right-total A
  shows (rel-set A) (Collect (Domainp A)) UNIV
  using assms unfolding right-total-def rel-set-def Domainp-iff by blast

43.6 ML setup
ML-file ⟨Tools/Lifting/lifting-util.ML⟩

named-theorems relator-eq-onp
  theorems that a relator of an eq-onp is an eq-onp of the corresponding predicate
ML-file ⟨Tools/Lifting/lifting-info.ML⟩

declare fun-quotient[quot-map]
declare fun-mono[relator-mono]
lemmas [relator-distr] = pos-fun-distr neg-fun-distr1 neg-fun-distr2

ML-file ⟨Tools/Lifting/lifting-bnf.ML⟩
ML-file ⟨Tools/Lifting/lifting-term.ML⟩
ML-file ⟨Tools/Lifting/lifting-def.ML⟩
ML-file ⟨Tools/Lifting/lifting-setup.ML⟩
ML-file ⟨Tools/Lifting/lifting-def-code-dt.ML⟩

lemma pred-prod-beta: pred-prod P Q xy ⟷ P (fst xy) ∧ Q (snd xy)
  by (cases xy) simp

lemma pred-prod-split: P (pred-prod Q R xy) ⟷ (∀ x y. xy = (x, y) ⟹ P (Q x ∧ R y))
  by (cases xy) simp

hide-const (open) POS NEG

end

44 Definition of Quotient Types

theory Quotient
imports Lifting
Basic definition for equivalence relations that are represented by predicates.

Composition of Relations

abbreviation
rel-conj :: ('a ⇒ 'b ⇒ bool) ⇒ ('b ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'b ⇒ bool (infixr OOO 75)

where
r1 OOO r2 ≡ r1 OO r2 OO r1

lemma eq-comp-r:
shows ((=) OOO R) = R
by (auto simp add: fun-eq-iff)

context includes lifting-syntax
begin

44.1 Quotient Predicate

definition
Quotient3 R Abs Rep = (\forall a. Abs (Rep a) = a) \land (\forall a. R (Rep a) (Rep a)) \land (\forall r s. R r s \iff R r r \land R s s \land Abs r = Abs s)

lemma Quotient3I:
assumes \(\forall a. Abs (Rep a) = a\)
and \(\forall a. R (Rep a) (Rep a)\)
and \(\forall r s. R r s \iff R r r \land R s s \land Abs r = Abs s\)
shows Quotient3 R Abs Rep
using assms unfolding Quotient3-def by blast

context
fixes R Abs Rep
assumes a: Quotient3 R Abs Rep
begin

lemma Quotient3-abs-rep:
Abs (Rep a) = a
using a
unfolding Quotient3-def
by simp

lemma Quotient3-reflp:
THEORY “Quotient”

\[
R \ (\text{Rep } a) \ (\text{Rep } a)
\]
\[
\text{using } a
\]
\[
\text{unfolding Quotient3-def}
\]
\[
\text{by blast}
\]

\textbf{lemma Quotient3-rel:}
\[
R \ r \ r \land R \ s \ s \land \text{Abs } r = \text{Abs } s \iff R \ r \ s \quad \text{— orientation does not loop on rewriting}
\]
\[
\text{using } a
\]
\[
\text{unfolding Quotient3-def}
\]
\[
\text{by blast}
\]

\textbf{lemma Quotient3-refl1:}
\[
R \ r \ s \implies R \ r \ r
\]
\[
\text{using a unfolding Quotient3-def}
\]
\[
\text{by fast}
\]

\textbf{lemma Quotient3-refl2:}
\[
R \ r \ s \implies R \ s \ s
\]
\[
\text{using a unfolding Quotient3-def}
\]
\[
\text{by fast}
\]

\textbf{lemma Quotient3-rel-rep:}
\[
R \ (\text{Rep } a) \ (\text{Rep } b) \iff a = b
\]
\[
\text{using } a
\]
\[
\text{unfolding Quotient3-def}
\]
\[
\text{by metis}
\]

\textbf{lemma Quotient3-rep-abs:}
\[
R \ r \ r \implies R \ (\text{Rep } (\text{Abs } r)) \ r
\]
\[
\text{using a unfolding Quotient3-def}
\]
\[
\text{by blast}
\]

\textbf{lemma Quotient3-rel-abs:}
\[
R \ r \ s \implies \text{Abs } r = \text{Abs } s
\]
\[
\text{using a unfolding Quotient3-def}
\]
\[
\text{by blast}
\]

\textbf{lemma Quotient3-symp:}
\[
\text{symp } R
\]
\[
\text{using a unfolding Quotient3-def using sympI by metis}
\]

\textbf{lemma Quotient3-transp:}
\[
\text{transp } R
\]
\[
\text{using a unfolding Quotient3-def using transpI by (metis (full-types))}
\]

\textbf{lemma Quotient3-part-eqv:}
\[
\text{part-equivp } R
\]
\[
\text{by (metis Quotient3-rep-reflp Quotient3-symp Quotient3-transp part-eqvI)}
\]
lemma abs-o-rep:
  Abs ◦ Rep = id
unfolding fun-eq-iff
by (simp add: Quotient3-abs-rep)

lemma equals-rsp:
  assumes b: R xa xb R ya yb
  shows R xa ya = R xb yb
using b Quotient3-symp Quotient3-transp
by (blast elim: sympE transpE)

lemma rep-abs-rsp:
  assumes b: R x1 x2
  shows R x1 (Rep (Abs x2)) for R x1 x2
using b Quotient3-rel Quotient3-abs-rep Quotient3-rep-reflp
by metis

lemma rep-abs-rsp-left:
  assumes b: R x1 x2
  shows R Rep (Abs x1) x2 for R x1 x2
using b Quotient3-rel Quotient3-abs-rep Quotient3-rep-reflp
by metis

end

lemma identity-quotient3:
  Quotient3 (=) id id
unfolding Quotient3-def id-def
by blast

lemma fun-quotient3:
  assumes q1: Quotient3 R1 abs1 rep1
  and q2: Quotient3 R2 abs2 rep2
  shows Quotient3 (R1 ===> R2) (rep1 ===> rep2)
proof -
  have (rep1 ===> rep2) ((abs1 ===> rep2) a) = a for a
    using q1 q2 by (simp add: Quotient3-def fun-eq-iff)
moreover
  have (R1 ===> R2) ((abs1 ===> rep2) a) (abs1 ===> rep2) a) for a
    by (rule rel-funI)
      (use q1 q2 Quotient3-rel-abs [of R1 abs1 rep1] Quotient3-rel-rep [of R2 abs2 rep2])
    in (simp (no_asm) add: Quotient3-def, simp)
moreover
  have (R1 ===> R2) r s = (R1 ===> R2) r r ∧ (R1 ===> R2) s s ∧
    (rep1 ===> abs2) r = (rep1 ===> abs2) s) for r s
    using Quotient3-part-equivp[OF q1] Quotient3-part-equivp[OF q2]
by (metis (full-types) part-equivp-def)
moreover have  \((R1 ===> R2) \ r \ s \implies (R1 ===> R2) \ s \ s\) unfolding
rel-fun-def
using Quotient3-part-equivp[OF q1] Quotient3-part-equivp[OF q2]
by (metis (full-types) part-equivp-def)
moreover have  \((R1 ===> R2) \ r \ s \implies (\text{rep1} ===> \text{abs2}) \ r \equal\ (\text{rep1} ===> \text{abs2}) \ s\)
by (auto simp add: rel-fun-def fun-eq-iff
(use q1 q2 in \langle unfold Quotient3-def, metis \rangle)
moreover have \((R1 ===> R2) \ r \ r \land (R1 ===> R2) \ s \ s \land (\text{rep1} ===> \text{abs2}) \ r \equal\ (\text{rep1} ===> \text{abs2}) \ s\) \implies (R1 ===> R2) \ r \ s
by (auto simp add: rel-fun-def fun-eq-iff
(use q1 q2 in \langle unfold Quotient3-def, metis map-fun-apply \rangle)
ultimately show \(?thesis by blast
qed
ultimately show \(?thesis by (intro Quotient3I) (assumption+)
qed

lemma lambda-prs:
assumes q1: Quotient3 R1 Abs1 Rep1
and q2: Quotient3 R2 Abs2 Rep2
shows \((\text{Rep1} ===> \text{Abs2}) \ (\lambda x. \text{Rep2} (f (\text{Abs1} \ x))) \equal\ (\lambda x. f x)\)
unfolding fun-eq-iff
using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]
by simp

lemma lambda-prs1:
assumes q1: Quotient3 R1 Abs1 Rep1
and q2: Quotient3 R2 Abs2 Rep2
shows \((\text{Rep1} ===> \text{Abs2}) \ (\lambda x. (\text{Abs1} ===> \text{Rep2}) f x) \equal\ (\lambda x. f x)\)
unfolding fun-eq-iff
using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]
by simp

In the following theorem R1 can be instantiated with anything, but we know some of the types of the Rep and Abs functions; so by solving Quotient assumptions we can get a unique R1 that will be provable; which is why we need to use apply-rsp and not the primed version

lemma apply-rspQ3:
fixes f g::'a \Rightarrow 'c
assumes q: Quotient3 R1 Abs1 Rep1
and a: (R1 ===> R2) f g R1 x y
shows R2 (f x) (g y)
using a by (auto elim: rel-funE)

lemma apply-rspQ3":
assumes Quotient3 R Abs Rep
and (R ===> S) f f
shows S (f (Rep x)) (f (Rep x))
proof –
from assms(1) have R (Rep x) (Rep x) by (rule Quotient3-rep-reflp)
then show ?thesis using assms(2) by (auto intro: apply-rsp')
qed

44.2 lemmas for regularisation of ball and bex

lemma ball-reg-eqv:
fixes P :: 'a ⇒ bool
assumes a: equivp R
shows Ball (Respects R) P = (All P)
using a
unfolding equivp-def
by (auto simp add: in-respects)

lemma bex-reg-eqv:
fixes P :: 'a ⇒ bool
assumes a: equivp R
shows Bex (Respects R) P = (Ex P)
using a
unfolding equivp-def
by (auto simp add: in-respects)

lemma ball-reg-right:
assumes a: ∀x. x ∈ R ⇒ P x ⇒ Q x
shows All P ⇒ Ball R Q
using a by fast

lemma bex-reg-left:
assumes a: ∀x. x ∈ R ⇒ Q x ⇒ P x
shows Bex R Q ⇒ Ex P
using a by fast

lemma ball-reg-left:
assumes a: equivp R
shows (∀x. (Q x ⇒ P x)) ⇒ Ball (Respects R) Q ⇒ All P
using a by (metis equivp-reflp in-respects)

lemma bex-reg-right:
assumes a: equivp R
shows (∀x. (Q x ⇒ P x)) ⇒ Ex Q ⇒ Bex (Respects R) P
using a by (metis equivp-reflp in-respects)

lemma ball-reg-eqv-range:
fixes P::'a ⇒ bool
and x:'a
assumes a: equivp R2
shows (Ball (Respects (R1 ==⇒ R2)) (λf. P (f x)) = All (λf. P (f x)))
proof (intro allI iffI)
THEORY “Quotient”

fix f
assume \( \forall f \in \text{Respects } (R1 \implies R2). P(f x) \)
moreover have \((\lambda y. f x) \in \text{Respects } (R1 \implies R2)\)
  using a equivp-reflp-symp-transp[of R2]
  by (auto simp add: in-respects rel-fun-def elim: equivpE reflpE)
ultimately show \( P(f x) \)
  by auto
qed auto

lemma bex-reg-eqv-range:
assumes a: equivp R2
shows \( (\text{Bex } (\text{Respects } (R1 \implies R2)) (\lambda f. P(f x)))) = \text{Ex } (\lambda f. P(f x)) \)
proof –
{ fix f
  assume P(f x)
  have \((\lambda y. f x) \in \text{Respects } (R1 \implies R2)\)
    using a equivp-reflp-symp-transp[of R2]
    by (auto simp add: Respects-def in-respects rel-fun-def elim: equivpE reflpE)
}
then show ?thesis
  by auto
qed

lemma all-reg:
assumes a: \( \forall x :: 'a. (P x \implies Q x) \)
and b: All P
shows All Q
using a b by fast

lemma ex-reg:
assumes a: \( \forall x :: 'a. (P x \implies Q x) \)
and b: Ex P
shows Ex Q
using a b by fast

lemma ball-reg:
assumes a: \( \forall x :: 'a. (x \in R \implies P x \implies Q x) \)
and b: Ball R P
shows Ball R Q
using a b by fast

lemma bex-reg:
assumes a: \( \forall x :: 'a. (x \in R \implies P x \implies Q x) \)
and b: Bex R P
shows Bex R Q
using a b by fast
lemma ball-all-comm:
  assumes \( \forall y. (\forall x \in P. A x y) \longrightarrow (\forall x. B x y) \)
  shows \( (\forall x \in P. \forall y. A x y) \longrightarrow (\forall x. \forall y. B x y) \)
  using assms by auto

lemma bex-ex-comm:
  assumes \( (\exists y. \exists x. A x y) \longrightarrow (\exists y. \exists x \in P. B x y) \)
  shows \( (\exists x. \exists y. A x y) \longrightarrow (\exists x \in P. \exists y. B x y) \)
  using assms by auto

44.3 Bounded abstraction

definition
  \( Babs :: \text{a set} \Rightarrow (\text{a} \Rightarrow (\text{b} \Rightarrow \text{a} \Rightarrow \text{b})) \Rightarrow (\text{a} \Rightarrow \text{b} \Rightarrow \text{a} \Rightarrow \text{b}) \)
  where \( x \in p \Rightarrow Babs p m x = m x \)

lemma babs-rsp:
  assumes \( q: \text{Quotient3 R1 Abs1 Rep1} \)
  and \( a: (R1 \Longrightarrow R2) f g \)
  shows \( (R1 \Longrightarrow R2) (Babs (\text{Respects R1}) f) (Babs (\text{Respects R1}) g) \)
proof
  fix \( x y \)
  assume \( R1 x y \)
  then have \( x \in \text{Respects R1} \land y \in \text{Respects R1} \)
    unfolding \text{in-respects rel-fun-def} using \text{Quotient3-rel[OF q]} by metis
  then show \( R2 (Babs (\text{Respects R1}) f) (Babs (\text{Respects R1}) g) \)
    using \( \langle R1 x y, a \rangle \) by (simp add: \text{Babs-def rel-fun-def})
qed

lemma babs-prs:
  assumes \( q1: \text{Quotient3 R1 Abs1 Rep1} \)
  and \( q2: \text{Quotient3 R2 Abs2 Rep2} \)
  shows \( ((\text{Rep1} \Longrightarrow \text{Abs2}) (Babs (\text{Respects R1}) ((\text{Abs1} \Longrightarrow \text{Rep2}) f)) = f \)
proof
  { fix \( x \)
    have \( \text{Rep1} x \in \text{Respects R1} \)
      by (simp add: \text{in-respects Quotient3-rel-rep[OF q1]})
    then have \( \text{Abs2} (Babs (\text{Respects R1}) ((\text{Abs1} \Longrightarrow \text{Rep2}) f) (\text{Rep1} x)) = f \)
      by (simp add: \text{Babs-def Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]})
  }
  then show \( \forall x \bullet \text{thesis} \)
    by force
qed

lemma babs-simp:
  assumes \( q: \text{Quotient3 R1 Abs Rep} \)
  shows \( ((R1 \Longrightarrow R2) (Babs (\text{Respects R1}) f) (Babs (\text{Respects R1}) g)) = ((R1 \Longrightarrow R2) (Babs (\text{Respects R1}) f) (Babs (\text{Respects R1}) g)) \)
THEORY "Quotient"

\[ \text{R} \xrightarrow{\text{\(=\)}} R2 \) \quad \text{f} \quad \text{g} \]

(is \( ?\text{lhs} = ?\text{rhs} \))

proof
assume \( ?\text{lhs} \)
then show \( ?\text{rhs} \)
  unfolding rel-fun-def by (metis Babs-def in-respects Quotient3-rel[OF q])
qed (simp add: babs-rsp[OF q])

If a user proves that a particular functional relation is an equivalence, this
may be useful in regularising

lemma babs-reg-eqv:
shows \( \text{equivp} \text{ R} \implies \text{Babs} \quad \text{Respects} \text{ R} \quad \text{P} = \text{P} \)
by (simp add: fun-eq-iff Babs-def in-respects equivp-reflp)

lemma ball-rsp:
assumes \( a: (\text{R} \xrightarrow{\text{\(=\)}} \text{\text{(\(=\))}}) \quad \text{f} \quad \text{g} \)
shows \( \text{Ball} \quad \text{Respects} \text{ R} \quad \text{f} = \text{Ball} \quad \text{Respects} \text{ R} \quad \text{g} \)
using a by (auto simp add: Ball-def in-respects elim: rel-funE)

lemma bex-rsp:
assumes \( a: (\text{R} \xrightarrow{\text{\(=\)}} \text{\text{(\(=\))}}) \quad \text{f} \quad \text{g} \)
shows \( (\text{Bex} \quad \text{Respects} \text{ R} \quad \text{f} = \text{Bex} \quad \text{Respects} \text{ R} \quad \text{g}) \)
using a by (auto simp add: Bex-def in-respects elim: rel-funE)

lemma bex1-rsp:
assumes \( a: (\text{R} \xrightarrow{\text{\(=\)}} \text{\text{(\(=\))}}) \quad \text{f} \quad \text{g} \)
shows \( (\text{Ex1} \quad (\lambda x. x \in \text{Respects} \quad \text{R} \land f \quad x) = \text{Ex1} \quad (\lambda x. x \in \text{Respects} \quad \text{R} \land g \quad x)) \)
using a by (auto elim: rel-funE simp add: Ex1-def in-respects)

Two lemmas needed for cleaning of quantifiers

lemma all-prs:
assumes \( a: \text{Quotient3} \quad \text{R} \quad \text{absf} \quad \text{repf} \)
shows \( \text{Ball} \quad \text{Respects} \text{ R} \quad ((\text{absf} \quad \rightarrow \quad \text{id}) \quad f) = \text{All} \quad f \)
using a unfolding Quotient3-def Ball-def in-respects id-apply comp-def map-fun-def
by metis

lemma ex-prs:
assumes \( a: \text{Quotient3} \quad \text{R} \quad \text{absf} \quad \text{repf} \)
shows \( \text{Bex} \quad \text{Respects} \text{ R} \quad ((\text{absf} \quad \rightarrow \quad \text{id}) \quad f) = \text{Ex} \quad f \)
using a unfolding Quotient3-def Bex-def in-respects id-apply comp-def map-fun-def
by metis

44.4 Bex1-rel quantifier

definition
\( \text{Bex1-rel} :: (\text{\('a \Rightarrow \text{\('a} \Rightarrow \text{bool} \))} \Rightarrow (\text{\('a} \Rightarrow \text{bool} \)) \Rightarrow \text{bool} \)
where
THEORY “Quotient”

\[ \text{Bex1-rel } R \ P \iff (\exists x \in \text{Respects } R. \ P x) \land (\forall x \in \text{Respects } R. \forall y \in \text{Respects } R. \ ((P x \land P y) \rightarrow (R x y))) \]

**Lemma bex1-rel-aux:**
\[
[\forall xa ya. \ R xa ya \rightarrow x xa = y ya; \text{Bex1-rel } R x] \Rightarrow \text{Bex1-rel } R y
\]

*unfolding* \text{Bex1-rel-def}  
*by* (\text{metis in-respects})

**Lemma bex1-rel-aux2:**
\[
[\forall xa ya. \ R xa ya \rightarrow x xa = y ya; \text{Bex1-rel } R y] \Rightarrow \text{Bex1-rel } R x
\]

*unfolding* \text{Bex1-rel-def}  
*by* (\text{metis in-respects})

**Lemma bex1-rel-rsp:**

**assumes** \( a : \text{Quotient3 } R \text{ absf repf} \)

**shows** \((R \Longrightarrow \text{=} \Longrightarrow \text{=} (\text{Bex1-rel } R) (\text{Bex1-rel } R)) \)

*unfolding* \text{rel-fun-def}  
*by* (\text{metis bex1-rel-aux bex1-rel-aux2})

**Lemma ex1-prs:**

**assumes** \( Q : \text{Quotient3 } R \text{ absf repf} \)

**shows** \((\text{Ex1 } f = \text{Ex1 } f) \)

*using* \text{assms}  
*apply* (\text{auto simp add: Bex1-rel-def Respects-def})

*by* (\text{metis (full-types Quotient3-def)})

**Lemma bex1-bexeq-reg:**

**shows** \((\exists ! x \in \text{Respects } R. \ P x) \rightarrow (\text{Bex1-rel } R (\lambda x. P x)) \)

*by* (\text{metis (full-types) Bex1-rel-def in-respects})

**Lemma bex1-bexeq-reg-eqv:**

**assumes** \( a : \text{equivp } R \)

**shows** \((\exists ! x. P x) \rightarrow \text{Bex1-rel } R P \)

*using* \text{equivp-reflp[OF a]}  
*by* (\text{metis (full-types) Bex1-rel-def in-respects})

### 44.5 Various respects and preserve lemmas

**Lemma quot-rel-rsp:**

**assumes** \( a : \text{Quotient3 } R \text{ Abs Rep} \)

**shows** \((R \Longrightarrow \text{=} \Longrightarrow \text{=} (\text{Bex1-rel } R) R) \)

*apply* (\text{rule rel-funI})  
*by* (\text{meson assms equals-rsp})

**Lemma o-prs:**

**assumes** \( q1 : \text{Quotient3 } R1 \text{ Abs1 Rep1} \)

and \( q2 : \text{Quotient3 } R2 \text{ Abs2 Rep2} \)

and \( q3 : \text{Quotient3 } R3 \text{ Abs3 Rep3} \)

**shows** \((\text{Abs2} \Longrightarrow \text{Rep3}) \Longrightarrow (\text{Abs1} \Longrightarrow \text{Rep2}) \Longrightarrow (\text{Rep1} \Longrightarrow \text{Rep3}) \)
Abs3)) (o) = (o)

and (id ---> (Abs1 ---> id) ---> Rep1 ---> id) (o) = (o)

using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2] Quotient3-abs-rep[OF q3]

by (simp-all add: fun-eq-iff)

lemma o-rsp:

((R2 ===> R3) ===> (R1 ===> R2) ===> (R1 ===> R3)) (o) (o)

((=) ===> (R1 ===> (=)) ===> R1 ===> (=)) (o) (o)

by (force elim: rel-funE)+

lemma cond-prs:

assumes a: Quotient3 R absf repf

shows absf (if a then repf b else repf c) = (if a then b else c)

using a unfolding Quotient3-def by auto

lemma if-prs:

assumes q: Quotient3 R Abs Rep

shows (id ---> Rep ---> Rep ---> Abs) If = If

using Quotient3-abs-rep[OF q]

by (auto simp add: fun-eq-iff)

lemma if-rsp:

assumes q: Quotient3 R Abs Rep

shows ((=) ===> R ===> R ===> R) If If

by force

lemma let-prs:

assumes q1: Quotient3 R1 Abs1 Rep1

and q2: Quotient3 R2 Abs2 Rep2

shows (Rep2 ---> (Abs2 ---> Rep1) ---> Abs1) Let = Let

using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]

by (auto simp add: fun-eq-iff)

lemma let-rsp:

shows (R1 ===> (R1 ===> R2) ===> R2) Let Let

by (force elim: rel-funE)

lemma id-rsp:

shows (R ===> R) id id

by auto

lemma id-prs:

assumes a: Quotient3 R Abs Rep

shows (Rep ---> Abs) id = id

by (simp add: fun-eq-iff Quotient3-abs-rep [OF a])

end
locale quot-type =
  fixes R :: 'a ⇒ 'a ⇒ bool
  and Abs :: 'a set ⇒ 'b
  and Rep :: 'b ⇒ 'a set
assumes equivp: part-equivp R
and rep-prop: ∀y. ∃x. R x x ∧ Rep y = Collect (R x)
and rep-inverse: ∀x. Abs (Rep x) = x
and abs-inverse: ∀x. Abs (Rep x) = x
and rep-inject: ∀x y. (Rep x = Rep y) = (x = y)
begin

definition abs :: 'a ⇒ 'b where abs x = Abs (Collect (R x))
definition rep :: 'b ⇒ 'a where rep a = (SOME x. x ∈ Rep a)

lemma some-collect:
assumes R r r
shows R (SOME x. x ∈ Collect (R r)) = R r
apply simp
by (metis assms exE-some equivp [simplified part-equivp-def])

lemma Quotient: shows Quotient3 R abs rep
unfolding Quotient3-def abs-def rep-def
proof (intro conjI allI)
  fix a r s
  show: R r r =⇒ R s s =⇒ Abs (Collect (R r)) = Abs (Collect (R s)) ⇔ (R r) = R s
  proof −
assume $R \equiv r \land R \equiv s$

then have $\text{Abs}(\text{Collect}(R \equiv r)) = \text{Abs}(\text{Collect}(R \equiv s))$ by (metis abs-inverse)

also have $\text{Collect}(R \equiv r) = \text{Collect}(R \equiv s)$ by (rule iffI) simp-all

finally show $\text{Abs}(\text{Collect}(R \equiv r)) = \text{Abs}(\text{Collect}(R \equiv s))$ ←→ $R \equiv r = R \equiv s$ by simp

qed

end

44.6 Quotient composition

lemma OOO-quotient3:
fixes $R_1 :: 'a \Rightarrow 'a \Rightarrow \text{bool}$
fixes $Abs1 :: 'a \Rightarrow 'b$ and $Rep1 :: 'b \Rightarrow 'a$
fixes $Abs2 :: 'b \Rightarrow 'c$ and $Rep2 :: 'c \Rightarrow 'b$
fixes $R_2' :: 'a \Rightarrow 'b \Rightarrow \text{bool}$

assumes $R_1: \text{Quotient3} R_1 Abs1 Rep1$
assumes $R_2: \text{Quotient3} R_2 Abs2 Rep2$

assumes $Abs1: \forall x y. R_2' x y \Longrightarrow R_1 x x \Longrightarrow R_1 y y \Longrightarrow R_2 (Abs1 x) (Abs1 y)$
assumes $Rep1: \forall x y. R_2 x y \Longrightarrow R_2' (Rep1 x) (Rep1 y)$

shows $\text{Quotient3} (R_1 OO R_2' OO R_1) (Abs2 \circ Abs1) (Rep1 \circ Rep2)$

proof –

have $*: (R_1 OO R_2') r r \land (R_1 OO R_2') s s \land (Abs2 \circ Abs1) r = (Abs2 \circ Abs1) s$ ←→ $(R_1 OO R_2') r s$ for $r s$

proof (intro iffI conjI; clarify)

show $(R_1 OO R_2') r s$

if $r: R_1 r a R_2' a b R_1 b r$ and $s: R_1 s c R_2' c d R_1 d s$ for $a b c d$

proof –

have $R_1 r (Rep1 (Abs1 r))$

using $r \text{ Quotient3-refl1} R_1 \text{ rep-abs-rsp}$ by fastforce

moreover have $R_2' (Rep1 (Abs1 r)) (Rep1 (Abs1 s))$

using that

apply simp

apply (metis (full-types) $Rep1 Abs1 \text{ Quotient3-rel} R_2 \text{ Quotient3-refl1}$ [OF $R_1$] \text{ Quotient3-refl2} [OF $R_1$] \text{ Quotient3-rel-abs} [OF $R_1$])

done

moreover have $R_1 (Rep1 (Abs1 s)) s$

by (metis $s \text{ Quotient3-rel} R_1 \text{ rep-abs-rsp-left}$)


ultimately show thesis
  by (metis relcomppI)
qed

next
  fix x y
  assume xy: R1 r x R2' x y R1 y s
  then have R2 (Abs1 x) (Abs1 y)
    by (iprover dest: Abs1 elim: Quotient3-refl1 [OF R1] Quotient3-refl2 [OF R1])
  then have R2' (Rep1 (Abs1 x)) (Rep1 (Abs1 x)) R2' (Rep1 (Abs1 y)) (Rep1 (Abs1 y))
    by (simp-all add: Quotient3-refl1 [OF R2] Quotient3-refl2 [OF R2] Rep1)
  with (R1 r x) (R1 y s) show (R1 OOO R2') r r (R1 OOO R2') s s
    by (metis (full-types) Quotient3-def R1 refl reflp relcompp relcompI)+
  show (Abs2 o Abs1) r = (Abs2 o Abs1) s
    using xy by simp (metis (full-types) Abs1 Quotient3-rel R1 R2)
qed

lemma OOO-eq-quotient3:
  fixes R1 :: 'a ⇒ 'a ⇒ bool
  fixes Abs1 :: 'a ⇒ 'b and Rep1 :: 'b ⇒ 'a
  fixes Abs2 :: 'b ⇒ 'c and Rep2 :: 'c ⇒ 'b
  assumes R1: Quotient3 R1 Abs1 Rep1
  assumes R2: Quotient3 (=) Abs2 Rep2
  shows Quotient3 (R1 OOO (=)) (Abs2 o Abs1) (Rep1 o Rep2)
using assms
by (rule OOO-quotient3) auto

44.7 Quotient3 to Quotient

lemma Quotient3-to-Quotient:
  assumes Quotient3 R Abs Rep
  and T ≡ λx y. R x x ∧ Abs x = y
  shows Quotient R Abs Rep T
using assms unfolding Quotient3-def by (intro QuotientI) blast+

lemma Quotient3-to-Quotient-equivp:
  assumes q: Quotient3 R Abs Rep
  and T-def: T ≡ λx y. Abs x = y
  and eR: equivp R
  shows Quotient R Abs Rep T
proof (intro QuotientI)
fix \( a \)
show \( \text{Abs}(\text{Rep } a) = a \) using \( q \) by (rule Quotient3-abs-rep)
next
fix \( a \)
show \( R(\text{Rep } a)(\text{Rep } a) \) using \( q \) by (rule Quotient3-rep-reflp)
next
fix \( r s \)
show \( R r s = (R r r \land R s s \land \text{Abs } r = \text{Abs } s) \) using \( q \) by (rule Quotient3-rel[symmetric])
next
show \( T = (\lambda x y. R x x \land \text{Abs } x = y) \) using T-def equivp-reflp[OF eR] by simp
qed

44.8 ML setup

Auxiliary data for the quotient package

**named-theorems** quot-equiv equivalence relation theorems
and quot-respect respectfulness theorems
and quot-preserve preservation theorems
and id-simps identity simp rules for maps
and quot-thm quotient theorems

**ML-file** ⟨Tools/Quotient/quotient-info.ML⟩

**declare** [[mapQ3 fun = (rel-fun, fun-quotient3)]]

**lemmas** [quot-thm] = fun-quotient3
**lemmas** [quot-respect] = quot-rel-rsp if-rsp o-rsp let-rsp id-rsp
**lemmas** [quot-preserve] = if-prs o-prs let-prs id-prs
**lemmas** [quot-equiv] = identity-equivp

Lemmas about simplifying id’s.

**lemmas** [id-simps] =
  id-def[symmetric]
  map-fun-id
  id-apply
  id-o
  o-id
  eq-comp-r
  vimage-id

Translation functions for the lifting process.

**ML-file** ⟨Tools/Quotient/quotient-term.ML⟩

Definitions of the quotient types.

**ML-file** ⟨Tools/Quotient/quotient-type.ML⟩

Definitions for quotient constants.

**ML-file** ⟨Tools/Quotient/quotient-def.ML⟩
An auxiliary constant for recording some information about the lifted theorem in a tactic.

**definition**

Quot-True :: 'a ⇒ bool

**where**

Quot-True x ←→ True

**lemma**

shows QT-all: Quot-True (All P) ⇒ Quot-True P

and QT-ex: Quot-True (Ex P) ⇒ Quot-True P

and QT-ex1: Quot-True (Ex1 P) ⇒ Quot-True P

and QT-lam: Quot-True (λx. P x) ⇒ (∀x. Quot-True (P x))

and QT-ext: (∀x. Quot-True (a x) ⇒ f x = g x) ⇒ (Quot-True a ⇒ f = g)

by (simp-all add: Quot-True-def ext)

**lemma** QT-imp: Quot-True a ≡ Quot-True b

by (simp add: Quot-True-def)

**context includes** lifting-syntax

**begin**

Tactics for proving the lifted theorems

**ML-file** 〈Tools/Quotient/quotient-tacs.ML〉

**end**

**44.9  Methods / Interface**

**method-setup** lifting =

〈Attrib.thms >> (fn thms => fn ctxt =>

SIMPLE-METHOD' (Quotient-Tacs.lift-tac ctxt [] thms))〉

〈lift theorems to quotient types〉

**method-setup** lifting-setup =

〈Attrib.thm >> (fn thm => fn ctxt =>

SIMPLE-METHOD' (Quotient-Tacs.lift-procedure-tac ctxt [] thm))〉

〈set up the three goals for the quotient lifting procedure〉

**method-setup** descending =

〈Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.descend-tac ctxt []))〉

〈descend theorems to the raw level〉

**method-setup** descending-setup =

〈Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.descend-procedure-tac ctxt []))〉

〈set up the three goals for the decending theorems〉
THEORY "Quotient"

**method-setup partiality-descending** =
\<Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.partiality-descend-tac ctxt []))\>
\<descend theorems to the raw level\>

**method-setup partiality-descending-setup** =
\<Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.partiality-descend-procedure-tac ctxt []))\>
\<set up the three goals for the decending theorems\>

**method-setup regularize** =
\<Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.regularize-tac ctxt))\>
\<prove the regularization goals from the quotient lifting procedure\>

**method-setup injection** =
\<Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.all-injection-tac ctxt))\>
\<prove the rep/abs injection goals from the quotient lifting procedure\>

**method-setup cleaning** =
\<Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.clean-tac ctxt))\>
\<prove the cleaning goals from the quotient lifting procedure\>

**attribute-setup quot-lifted** =
\<Scan.succeed Quotient-Tacs.lifted-attrib\>
\<lift theorems to quotient types\>

**no-notation**
\rel-conj \(\text{(infixr} \ 75)\)

45  **Lifting of BNFs**

**lemma** sum-insert-Inl-unit: \(x \in A \Rightarrow (\forall y. x = \text{Inr } y \Rightarrow \text{Inr } y \in B) \Rightarrow x \in \text{insert } (\text{Inl } ()) B\)
by (cases x) (simp-all)

**lemma** lift-sum-unit-vimage-commute:
\(\text{insert } (\text{Inl } ()) (\text{Inr } f \cdot A) = \text{map-sum } \text{id } f \cdot \text{insert } (\text{Inl } ()) (\text{Inr } f \cdot A)\)
by (auto simp: map-sum-def split: sum.splits)

**lemma** insert-Inl-int-map-sum-unit: \(\text{insert } (\text{Inl } ()) \cap \text{range } (\text{map-sum } \text{id } f) \neq \{\}\)
by (auto simp: map-sum-def split: sum.splits)

**lemma** image-map-sum-unit-subset:
\(A \subseteq \text{insert } (\text{Inl } ()) (\text{Inr } f \cdot B) \Rightarrow \text{map-sum } \text{id } f \cdot A \subseteq \text{insert } (\text{Inl } ()) (\text{Inr } f \cdot B)\)
by auto
lemma subset-lift-sum-unitD: A ⊆ insert (Inl ()) (Inr B) ⟹ Inr x ∈ A ⟹ x ∈ B
  unfolding insert-def by auto

lemma UNIV-sum-unit-conv: insert (Inl ()) (range Inr) = UNIV
  unfolding UNIV-sum UNIV-unit image-insert image-empty Un-insert-left sup-bot.left-neutral..

lemma subset-vimage-image-subset: A ⊆ f −' B ⟹ f ' A ⊆ B
  by auto

lemma relcompp-mem-Grp-neq-bot: A ∩ range f ≠ {} ⟹ (λx y. x ∈ A ∧ y ∈ A) OO (Grp UNIV f)⁻¹⁻¹ ≠ bot
  unfolding Grp-def relcompp-apply fun-eq-iff by blast

lemma rel-sum-eq2-nonempty: rel-sum (=) A OO rel-sum (=) B ≠ bot
  by (auto simp: fun-eq-iff relcompp-apply intro: exI[of - Inl -])

lemma hypsubst: A = B ⟹ x ∈ B ⟹ (x ∈ A ⟹ P) ⟹ P by simp
lemma Quotient-crel-quotient: Quotient R Abs Rep T \implies equivp R \implies T \equiv (\lambda x y. Abs x = y)
  by (drule Quotient-cr-rel) (auto simp: fun-eq-iff equivp-reflp intro: eq-reflection)

lemma Quotient-crel-typedef: Quotient (eq-onp P) Abs Rep T \implies T \equiv (\lambda x y. x = Rep y)
  unfolding Quotient-def
  by (auto 0 4 simp: fun-eq-iff eq-onp-def intro: symmetric) (rule Quotient-crel-typedef)

lemma Quotient-crel-typecopy: Quotient (\equiv) Abs Rep T \implies T \equiv (\lambda x y. x = Rep y)
  by (subst (asm) eq-onp-True [symmetric]) (rule Quotient-crel-typedef)

lemma equivp-add-relconj: assumes equiv: equivp R equivp R' and le: S OO T OO U OO T OO U OO R OO STU OO R' \leq R OO STU OO R' shows R OO S OO T OO U OO R' \leq R OO STU OO R'
proof
  have trans: R OO R \leq R R' OO R' \leq R' using equiv unfolding equivp-reflp-symp-transp transp-relcompp
    by blast+
  have R OO S OO T OO U OO R' = R OO (S OO T OO U) OO R'
    unfolding relcompp-assoc ..
  also have \ldots \leq R OO (R OO STU OO R') OO R'
    by (intro le relcompp-mono order-refl)
  also have \ldots \leq (R OO R) OO STU OO (R' OO R')
    unfolding relcompp-assoc ..
  also have \ldots \leq R OO STU OO R'
    by (intro trans relcompp-mono order-refl)
  finally show ?thesis.
  qed

lemma Grp-conversep-eq-onp: ((BNF-Def.Grp UNIV f)^{-1-1} OO BNF-Def.Grp UNIV f) = eq-onp (\lambda x. x \in range f)
  by (auto simp: fun-eq-iff Grp-def eq-onp-def image-iff)

lemma Grp-conversep-nonempty: (BNF-Def.Grp UNIV f)^{-1-1} OO BNF-Def.Grp UNIV f \neq bot
  by (auto simp: fun-eq-iff Grp-def)

lemma relcomppI2: r a b \implies s b c \implies t c d \implies (r OO s OO t) a d
  by (auto)

lemma rel-conj-eq-onp: equivp R \implies rel-conj R (eq-onp P) \leq R
  by (auto simp: eq-onp-def transp-def equivp-def)

lemma Quotient-Quotient3: Quotient R Abs Rep T \implies Quotient3 R Abs Rep
  unfolding Quotient-def Quotient3-def by blast

lemma Quotient-reflp-imp-equivp: Quotient R Abs Rep T \implies reflp R \implies equivp R
using Quotient-symp Quotient-transp equivpI by blast

lemma Quotient-eq-onp-typedef:  
Quotient (eq-onp P) Abs Rep cr \Longrightarrow type-definition Rep Abs \{ x. P x \}
unfolding Quotient-def eq-onp-def  
by unfold-locales auto

lemma Quotient-eq-onp-type-copy:  
Quotient (=) Abs Rep cr \Longrightarrow type-definition Rep Abs UNIV
unfolding Quotient-def eq-onp-def  
by unfold-locales auto

ML-file (Tools/BNF/bnf-lift.ML)

hide-fact  
sum-insert-Inl-unit lift-sum-unit-vimage-commute insert-Inl-int-map-sum-unit
image-map-sum-unit-subset subset-lift-sum-unitD UNIV-sum-unit-conv subset-vimage-image-subset
relcompp-mem-Grp-neq-bot comp-projr-Inr in-rel-sum-in-image-projr subset-rel-sumI
relcompp-eq-Grp-neq-bot rel-fun-rel-OO1 rel-fun-rel-OO2 rel-sum-eq2-nonempty
rel-sum-eq3-nonempty
hypsubst equivp-add-relconj Grp-conversep-eq-onp Grp-conversep-nonempty rel-
compp12 rel-conj-eq-onp
Quotient-reflp-imp-equivp Quotient-Quotient3

eend

46 Binary Numerals

theory Num
  imports BNF-Least-Fixpoint Transfer
begin

46.1 The num type

datatype num = One | Bit0 num | Bit1 num

Increment function for type num

primrec inc :: num \Rightarrow num  
where
  inc One = Bit0 One  
| inc (Bit0 x) = Bit1 x  
| inc (Bit1 x) = Bit0 (inc x)

Converting between type num and type nat

primrec nat-of-num :: num \Rightarrow nat  
where
  nat-of-num One = Suc 0  
| nat-of-num (Bit0 x) = nat-of-num x + nat-of-num x
\texttt{THEORY “Num”}

> \texttt{nat-of-num (Bit1 x) = Suc (nat-of-num x + nat-of-num x)}

\texttt{primrec num-of-nat :: nat \Rightarrow num}
\begin{itemize}
  \item where
  \begin{itemize}
    \item \texttt{num-of-nat 0 = One}
    \item \texttt{num-of-nat (Suc n) = (if 0 < n then inc (num-of-nat n) else One)}
  \end{itemize}
\end{itemize}

\texttt{lemma \texttt{nat-of-num-pos}: 0 < nat-of-num x}
\begin{itemize}
  \item by (induct x) simp-all
\end{itemize}

\texttt{lemma \texttt{nat-of-num-neq-0}: nat-of-num x \neq 0}
\begin{itemize}
  \item by (induct x) simp-all
\end{itemize}

\texttt{lemma \texttt{nat-of-num-inc}: nat-of-num (inc x) = Suc (nat-of-num x)}
\begin{itemize}
  \item by (induct x) simp-all
\end{itemize}

\texttt{lemma \texttt{nat-of-num-double}: 0 < n \Rightarrow num-of-nat \ (n + n) = Bit0 \ (num-of-nat n)}
\begin{itemize}
  \item by (induct n) simp-all
\end{itemize}

Type \texttt{num} is isomorphic to the strictly positive natural numbers.

\texttt{lemma \texttt{nat-of-num-inverse}: num-of-nat \ (nat-of-num x) = x}
\begin{itemize}
  \item by (induct x) \ (simp-all add: num-of-nat-double nat-of-num-pos)
\end{itemize}

\texttt{lemma \texttt{num-of-nat-inverse}: 0 < n \Rightarrow nat-of-num \ (num-of-nat n) = n}
\begin{itemize}
  \item by (induct n) \ (simp-all add: nat-of-num-inc)
\end{itemize}

\texttt{lemma \texttt{num-eq-iff}: x = y \iff nat-of-num x = nat-of-num y}
\begin{itemize}
  \item apply safe
  \item apply (drule arg-cong [where \texttt{f=num-of-nat}])
  \item apply (simp add: nat-of-num-inverse)
  \item done
\end{itemize}

\texttt{lemma \texttt{num-induct [case-names One inc]}:}
\begin{itemize}
  \item fixes \texttt{P :: num \Rightarrow bool}
  \item assumes One: \texttt{P One}
  \item and inc: \texttt{\forall x. P x \Rightarrow P (inc x)}
  \item shows \texttt{P x}
\end{itemize}

\texttt{proof –}
\begin{itemize}
  \item obtain \texttt{n where n: Suc n = nat-of-num x}
  \item by (cases \texttt{nat-of-num x}) \ (simp-all add: \texttt{nat-of-num-neq-0})
  \item have \texttt{P (num-of-nat (Suc n))}
  \item proof (induct \texttt{n})
  \item case \texttt{0}
  \item from \texttt{One show \ ?case by simp}
  \item next
  \item case \texttt{(Suc n)}
  \item then have \texttt{P (inc (num-of-nat (Suc n)))} \ by \ (rule inc)
  \item then show \texttt{P (num-of-nat (Suc (Suc n)))} \ by \ simp
  \item qed
\end{itemize}
with n show P x
  by (simp add: nat-of-num-inverse)
qed

From now on, there are two possible models for num: as positive naturals (rule num-induct) and as digit representation (rules num.induct, num.cases).

46.2 Numeral operations

instantiation num :: {plus, times, linorder}
begin

definition [code del]: 
  m + n = num-of-nat (nat-of-num m + nat-of-num n)

definition [code del]: 
  m * n = num-of-nat (nat-of-num m * nat-of-num n)

definition [code del]: 
  m ≤ n <-> nat-of-num m ≤ nat-of-num n

definition [code del]: 
  m < n <-> nat-of-num m < nat-of-num n

instance
  by standard (auto simp add: less-num-def less-eq-num-def num-eq-iff)
end

lemma nat-of-num-add: nat-of-num (x + y) = nat-of-num x + nat-of-num y
  unfolding plus-num-def
  by (intro num-of-nat-inverse add-pos-pos nat-of-num-pos)

lemma nat-of-num-mult: nat-of-num (x * y) = nat-of-num x * nat-of-num y
  unfolding times-num-def
  by (intro num-of-nat-inverse mult-pos-pos nat-of-num-pos)

lemma add-num-simps [simp, code]:
  One + One = Bit0 One
  One + Bit0 n = Bit1 n
  One + Bit1 n = Bit0 (n + One)
  Bit0 m + One = Bit1 m
  Bit0 m + Bit0 n = Bit0 (m + n)
  Bit0 m + Bit1 n = Bit1 (m + n)
  Bit1 m + One = Bit0 (m + One)
  Bit1 m + Bit0 n = Bit1 (m + n)
  Bit1 m + Bit1 n = Bit0 (m + n + One)
  by (simp-all add: num-eq-iff nat-of-num-add)

lemma mult-num-simps [simp, code]:
  m * One = m
  One * n = n
  Bit0 m * Bit0 n = Bit0 (Bit0 (m * n))
THEORY “Num” 878

\[\begin{align*}
\text{Bit0} m \cdot \text{Bit1} n &= \text{Bit0} (m \cdot \text{Bit1} n) \\
\text{Bit1} m \cdot \text{Bit0} n &= \text{Bit0} (\text{Bit1} m \cdot n) \\
\text{Bit1} m \cdot \text{Bit1} n &= \text{Bit1} (m + n + \text{Bit0} (m \cdot n))
\end{align*}\]

by (simp-all add: num-eq-iff nat-of-num-add nat-of-num-mult distrib-right distrib-left)

lemma eq-num-simps:
One = One \iff True
One = Bit0 n \iff False
Bit0 m = One \iff False
Bit1 m = One \iff False
Bit0 m = Bit0 n \iff m = n
Bit0 m = Bit1 n \iff False
Bit1 m = Bit0 n \iff False
Bit1 m = Bit1 n \iff m = n

by simp-all

lemma le-num-simps [simp, code]:
One \leq n \iff True
\text{Bit0} m \leq One \iff False
\text{Bit1} m \leq One \iff False
\text{Bit0} m \leq \text{Bit0} n \iff m \leq n
\text{Bit0} m \leq \text{Bit1} n \iff m \leq n
\text{Bit1} m \leq \text{Bit1} n \iff m \leq n
\text{Bit1} m \leq \text{Bit0} n \iff m < n

using nat-of-num-pos [of n] nat-of-num-pos [of m]
by (auto simp add: less-eq-num-def less-num-def)

lemma less-num-simps [simp, code]:
m < One \iff False
One < Bit0 n \iff True
One < Bit1 n \iff True
\text{Bit0} m < \text{Bit0} n \iff m < n
\text{Bit0} m < \text{Bit1} n \iff m \leq n
\text{Bit1} m < \text{Bit1} n \iff m < n
\text{Bit1} m < \text{Bit0} n \iff m < n

using nat-of-num-pos [of n] nat-of-num-pos [of m]
by (auto simp add: less-eq-num-def less-num-def)

lemma le-num-One-iff: \boxed{x \leq \text{num.} One \iff x = \text{num.} One}
by (simp add: antisym-conv)

Rules using One and inc as constructors.

lemma add-One: \boxed{x + One = \text{inc} x}
by (simp add: num-eq-iff nat-of-num-add nat-of-num-inc)

lemma add-One-commute: \boxed{One + n = n + One}
by (induct n) simp-all
lemma add-inc: \( x + \text{inc} \ y = \text{inc} \ (x + y) \)
by (simp add: num-eq-iff nat-of-num-add nat-of-num-inc)

lemma mult-inc: \( x \times \text{inc} \ y = x \times y + x \)

The \text{num-of-
at} conversion.
lemma num-of-
at-One: \( n \leq 1 \implies \text{num-of-
at} \ n = \text{One} \)
by (cases \( n \)) simp-all

lemma num-of-
at-plus-
distrib:
\( 0 < m \implies 0 < n \implies \text{num-of-
at} \ (m + n) = \text{num-of-
at} \ m + \text{num-of-
at} \ n \)
by (induct \( n \)) (auto simp add: add-One add-One-commute add-inc)

A double-and-decrement function.
primrec BitM :: \( \text{num} \Rightarrow \text{num} \)
where
\( \text{BitM} \ \text{One} = \text{One} \)
| \( \text{BitM} \ (\text{Bit0} \ n) = \text{Bit1} \ (\text{BitM} \ n) \)
| \( \text{BitM} \ (\text{Bit1} \ n) = \text{Bit1} \ (\text{Bit0} \ n) \)

lemma BitM-plus-
one: \( \text{BitM} \ n + \text{One} = \text{Bit0} \ n \)
by (induct \( n \)) simp-all

lemma one-plus-
BitM: \( \text{One} + \text{BitM} \ n = \text{Bit0} \ n \)
unfolding add-One-commute BitM-plus-
one .

lemma BitM-inc-eq:
\( \langle \text{Num.BitM} \ (\text{Num.inc} \ n) = \text{Num.Bit1} \ n \rangle \)
by (induction \( n \)) simp-all

lemma inc-
BitM-eq:
\( \langle \text{Num.inc} \ (\text{Num.BitM} \ n) = \text{num.Bit0} \ n \rangle \)
by (simp add: BitM-plus-
one[symmetric] add-One)

Squaring and exponentiation.
primrec sqr :: \( \text{num} \Rightarrow \text{num} \)
where
\( \text{sqr} \ \text{One} = \text{One} \)
| \( \text{sqr} \ (\text{Bit0} \ n) = \text{Bit0} \ (\text{Bit0} \ (\text{sqr} \ n)) \)
| \( \text{sqr} \ (\text{Bit1} \ n) = \text{Bit1} \ (\text{Bit0} \ (\text{sqr} \ n + n)) \)

primrec pow :: \( \text{num} \Rightarrow \text{num} \Rightarrow \text{num} \)
where
\( \text{pow} \ x \ \text{One} = x \)
| \( \text{pow} \ x \ (\text{Bit0} \ y) = \text{sqr} \ (\text{pow} \ x \ y) \)
| \( \text{pow} \ x \ (\text{Bit1} \ y) = \text{sqr} \ (\text{pow} \ x \ y) \times x \)
\textbf{46.3 Binary numerals}

We embed binary representations into a generic algebraic structure using \texttt{numeral}.

\begin{verbatim}
class numeral = one + semigroup-add begin

primrec numeral :: num ⇒ 'a
where
  numeral-One: numeral One = 1
  | numeral-Bit0: numeral (Bit0 n) = numeral n + numeral n
  | numeral-Bit1: numeral (Bit1 n) = numeral n + numeral n + 1

lemma numeral-code [code]:
  numeral One = 1
  numeral (Bit0 n) = (let m = numeral n in m + m)
  numeral (Bit1 n) = (let m = numeral n in m + m + 1)
  by (simp-all add: Let-def)

lemma one-plus-numeral-commute: 1 + numeral x = numeral x + 1
proof (induct x)
  case One
  then show ?case by simp
next
  case Bit0
  then show ?case by (simp add: add.assoc [symmetric]) (simp add: add.assoc)
  next
  case Bit1
  then show ?case by (simp add: add.assoc [symmetric]) (simp add: add.assoc)
qed

lemma numeral-inc: numeral (inc x) = numeral x + 1
proof (induct x)
  case One
  then show ?case by simp
next
  case Bit0
  then show ?case by simp
next

end
\end{verbatim}
case (Bit1 x)
  have numeral x + (1 + numeral x) + 1 = numeral x + (numeral x + 1) + 1
    by (simp only; one-plus-numeral-commute)
  with Bit1 show ?case
    by (simp add; add.assoc)
qed

declare numeral.simps [simp del]

abbreviation Numeral1 ≡ numeral One
declare numeral-One [code-post]
end

Numeral syntax.
syntax
  -Numeral :: num-const ⇒ 'a (-)

ML-file 〈Tools/numeral.ML〉

parse-translation ₣
  let
    fun numeral-tr [(c as Const (syntax-const (-constrain), -))] $ t $ u =
      c $ numeral-tr [t] $ u
    | numeral-tr [Const (num, -)] =
      (Numeral.mk-number-syntax o #value o Lexicon.read-num) num
    | numeral-tr ts = raise TERM (numeral-tr, ts);
  in [(syntax-const (-Numeral), K numeral-tr)] end

typed-print-translation ₣
  let
    fun num-tr' ctxt T [n] =
      let
        val k = Numeral.dest-num-syntax n;
        val t' =
          Syntax.const syntax-const (-Numeral) $
          Syntax.free (string-of-int k);
      in
        (case T of
          Type (type-name (fun), [-, T']) =>>
          if Printer.type-emphasis ctxt T' then
            Syntax.const syntax-const (-constrain) $ t' $
            Syntax-Phases.term-of-typ ctxt T'
          else t'
          | - => if T = dummyT then t' else raise Match)
      end;
  in
46.4 Class-specific numeral rules

numeral is a morphism.

46.4.1 Structures with addition: class numeral

context numeral
begin

lemma numeral-add: numeral \( m + n \) = numeral \( m \) + numeral \( n \)
  by (induct \( n \) rule: numeral-induct)
  (simp-all only: numeral-One add-One add-inc numeral-inc add assoc)

lemma numeral-plus-numeral: numeral \( m + n \) = numeral \( (m + n) \)
  by (rule numeral-add [symmetric])

lemma numeral-plus-one: numeral \( n + 1 \) = numeral \( (n + One) \)
  using numeral-add [of \( n \) One] by (simp add: numeral-One)

lemma one-plus-numeral: \( 1 + n \) = numeral \( (One + n) \)
  using numeral-add [of One \( n \)] by (simp add: numeral-One)

lemma one-add-one: \( 1 + 1 \) = 2
  using numeral-add [of One One] by (simp add: numeral-One)

lemmas add-numeral-special =
  numeral-plus-one one-plus-numeral one-add-one
end

46.4.2 Structures with negation: class neg-numeral

class neg-numeral = numeral + group-add
begin

lemma uminus-numeral-One: \( - \) numeral \( 1 \) = \( - 1 \)
  by (simp add: numeral-One)

Numerals form an abelian subgroup.

inductive is-num :: \( 'a \Rightarrow bool \)
where
  is-num 1
| is-num \( x \) \( \Rightarrow \) is-num \( (- x) \)
| is-num \( x \) \( \Rightarrow \) is-num \( y \) \( \Rightarrow \) is-num \( (x + y) \)
lemma \textit{is-num-numeral}: is-num \ (\textit{numeral} \ k)
by (induct \ k) \ (simp-all \ add: \ \textit{numeral}.\textit{simps} \ is-num.\textit{intros})

lemma \textit{is-num-add-commute}: is-num \ x \implies is-num \ y \implies x + y = y + x
proof (induction \ x \ rule: \ is-num.\textit{induct})
case 1
then show \ ?case
proof (induction \ y \ rule: \ is-num.\textit{induct})
case 1
then show \ ?case \ by simp
next
case (2 \ y)
then have \ y + (1 + - y) + y = y + (- y + 1) + y
by (simp \ add: \ \textit{add}.\textit{assoc})
then have \ y + (1 + - y) = y + (- y + 1)
by simp
then show \ ?case
by (rule \ add-left-imp-eq[of \ y])
next
case (3 \ x \ y)
then have \ 1 + (x + y) = x + 1 + y
by (simp \ add: \ \textit{add}.\textit{assoc} \ [\textit{symmetric}])
then show \ ?case \ using 3
by (simp \ add: \ \textit{add}.\textit{assoc})
qed
next
case (2 \ x)
then have \ x + (- x + y) + x = x + (y + - x) + x
by (simp \ add: \ \textit{add}.\textit{assoc})
then have \ x + (- x + y) = x + (y + - x)
by simp
then show \ ?case
by (rule \ add-left-imp-eq[of \ x])
next
case (3 \ x \ z)
moreover have \ x + (y + z) = (x + y) + z
by (simp \ add: \ \textit{add}.\textit{assoc}[\textit{symmetric}])
ultimately show \ ?case
by (simp \ add: \ \textit{add}.\textit{assoc})
qed

lemma \textit{is-num-add-left-commute}: is-num \ x \implies is-num \ y \implies x + (y + z) = y + (x + z)
by (simp \ only: \ \textit{add}.\textit{assoc} \ [\textit{symmetric}] \ is-num-add-commute)

lemmas \textit{is-num-normalize} =
\textit{add}.\textit{assoc} \ is-num-add-commute \ is-num-add-left-commute
is-num.\textit{intros} \ is-num-numeral
\textit{minus}.\textit{add}
definition dbl :: 'a ⇒ 'a
where dbl x = x + x

definition dbl-inc :: 'a ⇒ 'a
where dbl-inc x = x + x + 1

definition dbl-dec :: 'a ⇒ 'a
where dbl-dec x = x + x - 1

definition sub :: num ⇒ num ⇒ 'a
where sub k l = numeral k - numeral l

lemma numeral-BitM: numeral (BitM n) = numeral (Bit0 n) - 1
by (simp only: BitM-plus-one [symmetric] numeral-add numeral-One eq-diff-eq)

lemma sub-inc-One-eq:
  Num.sub (Num.inc n) Num.One = numeral n
by (simp-all add: sub-def diff-eq-eq numeral-inc numeral One eq-diff-eq)

lemma dbl-simps [simp]:
  dbl (- numeral k) = - dbl (numeral k)
  dbl 0 = 0
  dbl 1 = 2
  dbl (- 1) = - 2
  dbl (numeral k) = numeral (Bit0 k)
by (simp-all add: dbl-def numeral.simps minus-add)

lemma dbl-inc-simps [simp]:
  dbl-inc (- numeral k) = - dbl-dec (numeral k)
  dbl-inc 0 = 1
  dbl-inc 1 = 3
  dbl-inc (- 1) = - 1
  dbl-inc (numeral k) = numeral (Bit1 k)
by (simp-all add: dbl-inc-def dbl-dec-def numeral.simps numeral-BitM is-num-normalize algebra-simps
del: add-uminus-conv-diff)

lemma dbl-dec-simps [simp]:
  dbl-dec (- numeral k) = - dbl-inc (numeral k)
  dbl-dec 0 = - 1
  dbl-dec 1 = 1
  dbl-dec (- 1) = - 3
  dbl-dec (numeral k) = numeral (BitM k)
by (simp-all add: dbl-dec-def dbl-inc-def numeral.simps numeral-BitM is-num-normalize)

lemma sub-num-simps [simp]:
  sub One One = 0
  sub One (Bit0 l) = - numeral (BitM l)
THEORY “Num”

sub One (Bit1 l) = − numeral (Bit0 l)
sub (Bit0 k) One = numeral (BitM k)
sub (Bit1 k) One = numeral (Bit0 k)
sub (Bit0 k) (Bit0 l) = dbl (sub k l)
sub (Bit0 k) (Bit1 l) = dbl-dec (sub k l)
sub (Bit1 k) (Bit0 l) = dbl-inc (sub k l)
sub (Bit1 k) (Bit1 l) = dbl (sub k l)

by (simp-all add: dbl-def dbl-dec-def dbl-inc-def sub-def numeral
numeral-BitM is-num-normalize del: add-uminus-conv-diff add: diff-conv-add-uminus)

lemma add-neg-numeral-simps:
numeral m + − numeral n = sub m n
− numeral m + numeral n = sub n m
− numeral m + − numeral n = − (numeral m + numeral n)

by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize
del: add-uminus-cone-diff add: diff-conv-add-uminus)

lemma add-neg-numeral-special:
1 + − numeral m = sub One m
− numeral m + 1 = sub One m
numeral m + − 1 = sub m One
− 1 + numeral n = sub n One
− 1 + − numeral n = − numeral (inc n)
− numeral m + − 1 = − numeral (inc m)
1 + − 1 = 0
− 1 + 1 = 0
− 1 + − 1 = − 2

by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize right-minus
numeral-inc
del: add-uminus-cone-diff add: diff-conv-add-uminus)

lemma diff-numeral-simps:
umeral m − numeral n = sub m n
numeral m − − numeral n = numeral (m + n)
− numeral m − numeral n = − numeral (m + n)
− numeral m − − numeral n = sub n m

by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize
del: add-uminus-cone-diff add: diff-conv-add-uminus)

lemma diff-numeral-special:
1 − numeral n = sub One n
numeral m − 1 = sub m One
1 − − numeral n = numeral (One + n)
− numeral m − 1 = − numeral (m + One)
− 1 − numeral n = − numeral (inc n)
numeral m − − 1 = numeral (inc m)
− 1 − − numeral n = sub n One
− numeral m − − 1 = sub One m
1 − 1 = 0
\[
-1 - 1 = -2 \\
1 - (-1) = 2 \\
-1 - (-1) = 0
\]

by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize numeral-inc
del: add-uminus-one-diff add: diff-one-add-uminus)

end

46.4.3 Structures with multiplication: class \texttt{semiring-numeral}

\texttt{class semiring-numeral = semiring + monoid-mult}

\begin{itemize}
\item \texttt{subclass numeral ..}
\item \texttt{lemma numeral-mul: numeral (m * n) = numeral m * numeral n}
  by (induct n rule: numeral-induct)
  (simp-all add: numeral-one mul-inc numeral-inc numeral-add distrib-left)
\item \texttt{lemma numeral-times-numeral: numeral m * numeral n = numeral (m * n)}
  by (rule numeral-mult [symmetric])
\item \texttt{lemma mul-2: 2 * z = z + z}
  by (simp add: one-add-one [symmetric] distrib-right)
\item \texttt{lemma mul-2-right: z * 2 = z + z}
  by (simp add: one-add-one [symmetric] distrib-left)
\item \texttt{lemma left-add-twice:}
  \[a + (a + b) = 2 * a + b\]
  by (simp add: mul-2 ac-simps)
\item \texttt{lemma numeral-Bit0-eq-double:}
  \[\langle\text{numeral (num.Bit0 n)} = 2 * \text{numeral n}\rangle\]
  by (simp add: mul-2) (simp add: numeral-Bit0)
\item \texttt{lemma numeral-Bit1-eq-inc-double:}
  \[\langle\text{numeral (num.Bit1 n)} = 2 * \text{numeral n} + 1\rangle\]
  by (simp add: mul-2) (simp add: numeral-Bit1)
\end{itemize}

end

46.4.4 Structures with a zero: class \texttt{semiring-1}

\texttt{context semiring-1}

\begin{itemize}
\item \texttt{subclass semiring-numeral ..}
\item \texttt{lemma of-nat-numeral [simp]: of-nat (numeral n) = numeral n}
\end{itemize}

end
THEORY “Num”

by (induct n) (simp-all only: numeral.simps numeral-class.numeral.simps of-nat-add of-nat-1)

end

lemma nat-of-num-numeral [code-abbrev]: nat-of-num = numeral
proof
fix n
have numeral n = nat-of-num n
  by (induct n) (simp-all add: numeral.simps)
then show nat-of-num n = numeral n
  by simp
qed

lemma nat-of-num-code [code]:
nat-of-num One = 1
nat-of-num (Bit0 n) = (let m = nat-of-num n in m + m)
nat-of-num (Bit1 n) = (let m = nat-of-num n in Suc (m + m))
by (simp-all add: Let-def)

46.4.5 Equality: class semiring-char-0

context semiring-char-0
begin

lemma numeral-eq-iff: numeral m = numeral n ←→ m = n
  by (simp only: of-nat-numeral [symmetric] nat-of-num-numeral [symmetric]
    of-nat-eq-iff num-eq-iff)

lemma numeral-eq-one-iff: numeral n = 1 ←→ n = One
  by (rule numeral-eq-iff [of n One, unfolded numeral-One])

lemma one-eq-numeral-iff: 1 = numeral n ←→ One = n
  by (rule numeral-eq-iff [of One n, unfolded numeral-One])

lemma numeral-neq-zero: numeral n ≠ 0

lemma zero-neq-numeral: 0 ≠ numeral n
  unfolding eq-commute [of 0] by (rule numeral-neq-zero)

lemmas eq-numeral-simps [simp] =
  numeral-eq-iff
  numeral-eq-one-iff
  one-eq-numeral-iff
  numeral-neq-zero
  zero-neq-numeral
end
46.4.6  **Comparisons:** class `linordered-nonzero-semiring`

```
context `linordered-nonzero-semiring`
begin

lemma numeral-le-iff: numeral m ≤ numeral n ↔ m ≤ n
proof
  have of-nat (numeral m) ≤ of-nat (numeral n) ↔ m ≤ n
    by (simp only: less-eq-num-def nat-of-num-numeral of-nat-le-iff)
  then show ?thesis by simp
qed

lemma one-le-numeral: 1 ≤ numeral n
  using numeral-le-iff [of num.One n] by (simp add: numeral-One)

lemma numeral-le-one-iff: numeral n ≤ 1 ↔ n ≤ num.One
  using numeral-le-iff [of n num.One] by (simp add: numeral-One)

lemma numeral-less-iff: numeral m < numeral n ↔ m < n
proof
  have of-nat (numeral m) < of-nat (numeral n) ↔ m < n
    unfolding less-num-def nat-of-num-numeral of-nat-less-iff ..
  then show ?thesis by simp
qed

lemma not-numeral-less-one: ¬ numeral n < 1
  using numeral-less-iff [of n num.One] by (simp add: numeral-One)

lemma one-less-numeral-iff: 1 < numeral n ↔ num.One < n
  using numeral-less-iff [of num.One n] by (simp add: numeral-One)

lemma zero-le-numeral: 0 ≤ numeral n
  using dual-order.trans one-le-numeral zero-le-one by blast

lemma zero-less-numeral: 0 < numeral n
  using less-linear not-numeral-less-one order.strict-trans zero-less-one by blast

lemma not-numeral-le-zero: ¬ numeral n ≤ 0
  by (simp add: not-le zero-less-numeral)

lemma not-numeral-less-zero: ¬ numeral n < 0
  by (simp add: not-less zero-le-numeral)

lemmas le-numeral-extra =
  zero-le-one not-one-le-zero
  order-refl [of 0] order-refl [of 1]

lemmas less-numeral-extra =
  zero-less-one not-one-less-zero
  less-irrefl [of 0] less-irrefl [of 1]
```
lemmas le-numeral-simps [simp] =
numeral-le-iff
one-le-numeral
numeral-le-one-iff
zero-le-numeral
not-numeral-le-zero

lemmas less-numeral-simps [simp] =
numeral-less-iff
one-less-numeral-iff
not-numeral-less-one
zero-less-numeral
not-numeral-less-zero

lemma min-0-1 [simp]:
fixes min' :: 'a ⇒ 'a ⇒ 'a
defines min' ≡ min
shows
  min' 0 1 = 0
  min' 1 0 = 0
  min' 0 (numeral x) = 0
  min' (numeral x) 0 = 0
  min' 1 (numeral x) = 1
  min' (numeral x) 1 = 1
by (simp-all add: min'-def min-def le-num-One-iff)

lemma max-0-1 [simp]:
fixes max' :: 'a ⇒ 'a ⇒ 'a
defines max' ≡ max
shows
  max' 0 1 = 1
  max' 1 0 = 1
  max' 0 (numeral x) = numeral x
  max' (numeral x) 0 = numeral x
  max' 1 (numeral x) = numeral x
  max' (numeral x) 1 = numeral x
by (simp-all add: max'-def max-def le-num-One-iff)
end

Unfold min and max on numerals.

lemmas max-number-of [simp] =
  max-def [of numeral u numeral v]
  max-def [of numeral u - numeral v]
  max-def [of - numeral u numeral v]
  max-def [of - numeral u - numeral v] for u v

lemmas min-number-of [simp] =
46.4.7 Multiplication and negation: class ring-1

context ring-1

subclass neg-numeral ..

lemma mult-neg-numeral-simps:
  − numeral m * − numeral n = numeral (m * n)
  − numeral m * numeral n = − numeral (m * n)
  numeral m * − numeral n = − numeral (m * n)
  by (simp-all only; mult-minus-left mult-minus-right minus-minus numeral-mult)

lemma mult-minus1 [simp]: − 1 * z = − z
  by (simp add: numeral.simps)

lemma mult-minus1-right [simp]: z * − 1 = − z
  by (simp add: numeral.simps)

lemma minus-sub-one-diff-one [simp]:
  − (sub m One − 1) = − numeral m
proof –
  have (sub m One + 1 = numeral m)
    by (simp flip: eq-diff-eq add: diff-numeral-special)
  then have (− (sub m One + 1) = − numeral m)
    by simp
  then show ?thesis
    by simp
qed

end

46.4.8 Equality using iszero for rings with non-zero characteristic

context ring-1

begin

definition iszero :: 'a ⇒ bool
  where iszero z ←→ z = 0

lemma iszero-0 [simp]: iszero 0
  by (simp add: iszero-def)

lemma not-iszero-1 [simp]: ¬ iszero 1
  by (simp add: iszero-def)
lemma not-iszero-Numeral1: ¬ iszero Numeral1
by (simp add: numeral-One)

lemma not-iszero-neg-1 [simp]: ¬ iszero (− 1)
by (simp add: iszero-def)

lemma not-iszero-neg-Numeral1: ¬ iszero (− Numeral1)
by (simp add: numeral-One)

lemma iszero-neg-numeral [simp]: iszero (− numeral w) ←→ iszero (numeral w)
unfolding iszero-def by (rule neg-equal-0-iff-equal)

lemma eq-iff-iszero-diff: x = y ←→ iszero (x − y)
unfolding iszero-def by (rule eq-iff-diff-eq-0)

The eq-numeral-iff-iszero lemmas are not declared [simp] by default, because
for rings of characteristic zero, better simp rules are possible. For a type like
integers mod n, type-instantiated versions of these rules should be added to
the simplifier, along with a type-specific rule for deciding propositions of the
form iszero (numeral w).

bh: Maybe it would not be so bad to just declare these as simp rules anyway?
I should test whether these rules take precedence over the ring-char-0 rules
in the simplifier.

lemma eq-numeral-iff-iszero:
  numeral x = numeral y ←→ iszero (sub x y)
  numeral x = − numeral y ←→ iszero (numeral (x + y))
− numeral x = numeral y ←→ iszero (numeral (x + y))
− numeral x = − numeral y ←→ iszero (sub y x)
  numeral x = 1 ←→ iszero (sub x One)
  1 = numeral y ←→ iszero (sub One y)
− numeral x = 1 ←→ iszero (numeral (x + One))
  1 = − numeral y ←→ iszero (numeral (One + y))
  numeral x = 0 ←→ iszero (numeral x)
  0 = numeral y ←→ iszero (numeral y)
− numeral x = 0 ←→ iszero (numeral x)
  0 = − numeral y ←→ iszero (numeral y)
unfolding eq-iff-iszero-diff diff-numeral-simps diff-numeral-special
by simp-all

end

46.4.9 Equality and negation: class ring-char-0

context ring-char-0
begin

lemma not-iszero-numeral [simp]: ¬ iszero (numeral w)

by (simp add: iszero-def)

lemma neg-numeral-eq-iff: \(-\text{numeral } m = - \text{numeral } n \iff m = n\)
  by simp

lemma numeral-neq-neg-numeral: \(\text{numeral } m \neq - \text{numeral } n\)
  by (simp add: eq.neg_iff_add_eq_0 numeral.add numeral_plus)

lemma neg-numeral-neq-numeral: \(- \text{numeral } m \neq \text{numeral } n\)
  by (rule numeral-neq-neg-numeral [symmetric])

lemma zero-neq-neg-numeral: \(0 \neq - \text{numeral } n\)
  by simp

lemma neg-numeral-neq-zero: \(- \text{numeral } n \neq 0\)
  by simp

lemma one-neq-neg-numeral: \(1 \neq - \text{numeral } n\)
  using numeral-neq-neg-numeral [of One n] by (simp add: numeral-One)

lemma neg-numeral-neq-one: \(- \text{numeral } n \neq 1\)
  using neg-numeral-neq-numeral [of n One] by (simp add: numeral-One)

lemma numeral-neq-neg-one: \(\text{numeral } n \neq - 1\)
  using numeral-neq-neg-numeral [of n One] by (simp add: numeral-One)

lemma neg-one-neq-numeral iff: \(- 1 = - \text{numeral } n \iff n = \text{One}\)
  using numeral-neq-neg-numeral [of One n] by (auto simp add: numeral-One)

lemma numeral-eq-one-neq-numeral iff: \(\text{numeral } n = - 1 \iff n = \text{One}\)
  using numeral-neq-neg-numeral [of n One] by (auto simp add: numeral-One)

lemma neg-one-neq-zero: \(- 1 \neq 0\)
  by simp

lemma zero-neq-neg-one: \(0 \neq - 1\)
  by simp

lemma one-neq-neg-one: \(1 \neq - 1\)
  using numeral-neq-neg-numeral [of One One] by (simp only: numeral-One not-False-eq-True)

lemma one-neq-neg-one: \(1 \neq - 1\)
  using numeral-neq-neg-numeral [of One One] by (simp only: numeral-One not-False-eq-True)

lemmas eq-neq-numeral-simps [simp] =
  numeral_neq_neg_numeral
numeral-neq-neg-numeral neg-numeral-neq-numeral one-neq-neg-numeral neg-numeral-neq-one
zero-neq-neg-numeral neg-numeral-neq-zero neg-one-neq-neg-numeral numeral-neq-neg-one
neg-one-eq-numeral-iff numeral-eq-neg-one-iff
neg-one-neq-zero zero-neq-neg-one
neg-one-neq-one one-neq-neg-one

end

46.4.10 Structures with negation and order: class linordered-idom

context linordered-idom

begin

subclass ring-char-0 ..

lemma neg-numeral-le-iff: − numeral m ≤ − numeral n ↔ n ≤ m
by (simp only: neg-iff-le numeral-le-iff)

lemma neg-numeral-less-iff: − numeral m < − numeral n ↔ n < m
by (simp only: neg-less-iff-less numeral-less-iff)

lemma neg-numeral-less-zero: − numeral n < 0
by (simp only: neg-less-0-iff-less zero-less-numeral)

lemma neg-numeral-le-zero: − numeral n ≤ 0
by (simp only: neg-0-iff-le zero-le-numeral)

lemma not-zero-less-neg-numeral: ¬ 0 < − numeral n
by (simp only: not-less neg-numeral-le-zero)

lemma not-zero-le-neg-numeral: ¬ 0 ≤ − numeral n
by (simp only: not-le neg-numeral-less-zero)

lemma neg-numeral-less-numeral: − numeral m < numeral n
using neg-numeral-less-zero less-trans

lemma neg-numeral-le-numeral: − numeral m ≤ numeral n
by (simp only: less-imp-le numeral-le-numeral)

lemma not-numeral-less-neg-numeral: ¬ numeral m < − numeral n
by (simp only: not-less neg-numeral-le-numeral)

lemma not-numeral-le-neg-numeral: ¬ numeral m ≤ − numeral n
by (simp only: not-le neg-numeral-less-numeral)

lemma neg-numeral-less-one: − numeral m < 1
by (rule neg-numeral-less-numeral [of m One, unfolded numeral-One])
lemma neg-numeral-le-one: \(-\) numeral \(m\) \(\leq 1\)
  by (rule neg-numeral-le-numeral [of \(m\) One, unfolded numeral-One])

lemma not-one-less-neg-numeral: \(-1 < -\) numeral \(m\)
  by (simp only: not-less neg-numeral-le-one)

lemma not-one-le-neg-numeral: \(-1 \leq -\) numeral \(m\)
  by (simp only: not-le neg-numeral-less-one)

lemma not-numeral-less-neg-one: \(\text{numeral} \ m < -1\)
  using not-numeral-less-neg-numeral [of \(m\) One] by (simp add: numeral-One)

lemma neg-one-less-numeral: \(-1 < \text{numeral} \ m\)
  using neg-numeral-less-numeral [of One \(m\)] by (simp add: numeral-One)

lemma neg-one-le-numeral: \(-1 \leq \text{numeral} \ m\)
  using neg-numeral-le-numeral [of One \(m\)] by (simp add: numeral-One)

lemma neg-numeral-less-neg-one-iff: \(-\text{numeral} \ m < -1\) \(\iff\) \(m \neq 1\)
  by (cases \(m\)) simp-all

lemma neg-numeral-le-neg-one: \(-\text{numeral} \ m \leq -1\)
  by simp

lemma not-neg-one-less-neg-numeral: \(-1 < -\text{numeral} \ m\)
  by simp

lemma not-neg-one-le-neg-numeral-iff: \(-1 \leq -\text{numeral} \ m\) \(\iff\) \(m \neq 1\)
  by (cases \(m\)) simp-all

lemma sub-non-negative: \(\text{sub} \ n \ m \geq 0\) \(\iff\) \(n \geq m\)
  by (simp only: sub-def le-diff-eq) simp

lemma sub-positive: \(\text{sub} \ n \ m > 0\) \(\iff\) \(n > m\)
  by (simp only: sub-def less-diff-eq) simp

lemma sub-non-positive: \(\text{sub} \ n \ m \leq 0\) \(\iff\) \(n \leq m\)
  by (simp only: sub-def diff-le-eq) simp

lemma sub-negative: \(\text{sub} \ n \ m < 0\) \(\iff\) \(n < m\)
  by (simp only: sub-def diff-less-eq) simp

lemmas le-neg-numeral-simps [simp] =
  neg-numeral-le-iff
  neg-numeral-le-numeral not-numeral-le-neg-numeral
neg-numeral-le-zero not-zero-le-neg-numeral
neg-numeral-le-one not-one-le-neg-numeral
neg-one-le-numeral not-numeral-le-neg-one
neg-numeral-le-neg-one not-neg-one-le-neg-numeral-iff

lemma le-minus-one-simps [simp]:
\[-1 \leq 0\]
\[-1 \leq 1\]
\[-0 \leq -1\]
\[-1 \leq -1\]
by simp-all

lemmas less-neg-numeral-simps [simp] =
neg-numeral-less-iff
neg-numeral-less-numeral not-numeral-less-neg-numeral
neg-numeral-less-zero not-zero-less-neg-numeral
neg-numeral-less-one not-one-less-neg-numeral
neg-one-less-numeral not-numeral-less-neg-one
neg-numeral-less-neg-one-iff not-neg-one-less-neg-numeral

lemma less-minus-one-simps [simp]:
\[-1 < 0\]
\[-1 < 1\]
\[-0 < -1\]
\[-1 < -1\]
by (simp-all add: less-le)

lemma abs-numeral [simp]: \(|\text{numeral } n| = \text{numeral } n\)
by simp

lemma abs-neg-numeral [simp]: \(|-\text{numeral } n| = \text{numeral } n\)
by (simp only: abs-minus-cancel abs-numeral)

lemma abs-neg-one [simp]: \(|-1| = 1\)
by simp

end

46.4.11 Natural numbers

lemma numeral-num-of-nat:
\(|\text{numeral } (\text{num-of-nat } n)| = n \text{ if } n > 0\)
using that \text{nat-of-num-numeral num-of-nat-inverse} by simp

lemma Suc-1 [simp]: Suc 1 = 2
unfolding Suc-eq-plus1 by (rule one-add-one)

lemma Suc-numeral [simp]: Suc (numeral n) = numeral (n + One)
unfolding Suc-eq-plus1 by (rule numeral-plus-one)
definition pred-numeral :: num ⇒ nat
    where pred-numeral k = numeral k - 1

declare [[code drop: pred-numeral]]

lemma numeral-eq-Suc: numeral k = Suc (pred-numeral k)
    by (simp add: pred-numeral-def)

lemma eval-nat-numeral:
    numeral One = Suc 0
    numeral (Bit0 n) = Suc (numeral (BitM n))
    numeral (Bit1 n) = Suc (numeral (Bit0 n))
    by (simp-all add: numeral.simps BitM-plus-one)

lemma pred-numeral-simps [simp]:
    pred-numeral One = 0
    pred-numeral (Bit0 k) = numeral (BitM k)
    pred-numeral (Bit1 k) = numeral (Bit0 k)
    by (simp-all only: pred-numeral-def eval-nat-numeral diff-Suc-Suc diff-0)

lemma pred-numeral-inc [simp]:
    pred-numeral (Num.inc k) = numeral k
    by (simp only: pred-numeral-def numeral-inc diff-add-inverse2)

lemma numeral-2-eq-2: 2 = Suc (Suc 0)
    by (simp add: eval-nat-numeral)

lemma numeral-3-eq-3: 3 = Suc (Suc (Suc 0))
    by (simp add: eval-nat-numeral)

lemma numeral-1-eq-Suc-0: Numeral1 = Suc 0
    by (simp only: numeral-One One-nat-def)

lemma Suc-nat-number-of-add: Suc (numeral v + n) = numeral (v + One) + n
    by simp

lemma numerals: Numeral1 = (1::nat) 2 = Suc (Suc 0)
    by (rule numeral-One) (rule numeral-2-eq-2)

lemmas numeral-nat = eval-nat-numeral BitM.simps One-nat-def

Comparisons involving Suc.

lemma eq-numeral-Suc [simp]: numeral k = Suc n ←→ pred-numeral k = n
    by (simp add: numeral-eq-Suc)

lemma Suc-eq-numeral [simp]: Suc n = numeral k ←→ n = pred-numeral k
    by (simp add: numeral-eq-Suc)
lemma less-numeral-Suc [simp]: numeral k < Suc n ↔ pred-numeral k < n
by (simp add: numeral-eq-Suc)

lemma less-Suc-numeral [simp]: Suc n < numeral k ↔ n < pred-numeral k
by (simp add: numeral-eq-Suc)

lemma le-numeral-Suc [simp]: numeral k ≤ Suc n ↔ pred-numeral k ≤ n
by (simp add: numeral-eq-Suc)

lemma le-Suc-numeral [simp]: Suc n ≤ numeral k ↔ n ≤ pred-numeral k
by (simp add: numeral-eq-Suc)

lemma diff-Suc-numeral [simp]: Suc n − numeral k = n − pred-numeral k
by (simp add: numeral-eq-Suc)

lemma diff-numeral-Suc [simp]: numeral k − Suc n = pred-numeral k − n
by (simp add: numeral-eq-Suc)

lemma max-Suc-numeral [simp]: max (Suc n) (numeral k) = Suc (max n (pred-numeral k))
by (simp add: numeral-eq-Suc)

lemma max-numeral-Suc [simp]: max (numeral k) (Suc n) = Suc (max (pred-numeral k) n)
by (simp add: numeral-eq-Suc)

lemma min-Suc-numeral [simp]: min (Suc n) (numeral k) = Suc (min n (pred-numeral k))
by (simp add: numeral-eq-Suc)

lemma min-numeral-Suc [simp]: min (numeral k) (Suc n) = Suc (min (pred-numeral k) n)
by (simp add: numeral-eq-Suc)

For case-nat and rec-nat.

lemma case-nat-numeral [simp]: case-nat a f (numeral v) = (let pv = pred-numeral v in f pv)
by (simp add: numeral-eq-Suc)

lemma case-nat-add-eq-if [simp]:
    case-nat a f ((numeral v) + n) = (let pv = pred-numeral v in f (pv + n))
by (simp add: numeral-eq-Suc)

lemma rec-nat-numeral [simp]:
    rec-nat a f (numeral v) = (let pv = pred-numeral v in f pv (rec-nat a pv))
by (simp add: numeral-eq-Suc Let-def)

lemma rec-nat-add-eq-if [simp]:
    rec-nat a f (numeral v + n) = (let pv = pred-numeral v in f (pv + n) (rec-nat a
f (pv + n))
  by (simp add: numeral-eq-Suc Let-def)

Case analysis on n < (2::'a).

lemma less-2-cases: n < 2 \implies n = 0 \lor n = Suc 0
  by (auto simp add: numeral-2-eq-2)

lemma less-2-cases-iff: n < 2 \iff n = 0 \lor n = Suc 0
  by (auto simp add: numeral-2-eq-2)

Removal of Small Numerals: 0, 1 and (in additive positions) 2.

bh: Are these rules really a good idea? LCP: well, it already happens for 0 and 1!

lemma add-2-eq-Suc [simp]: 2 + n = Suc (Suc n)
  by simp

lemma add-2-eq-Suc' [simp]: n + 2 = Suc (Suc n)
  by simp

Can be used to eliminate long strings of Sucs, but not by default.

lemma Suc3-eq-add-3: Suc (Suc (Suc n)) = 3 + n
  by simp

lemmas nat-1-add-1 = one-add-one [where 'a=nat]

context semiring-numeral
begin

lemma numeral-add-unfold-funpow:
  numeral k + a = ((+) 1 ^^^ numeral k) a
proof (rule sym, induction k arbitrary: a)
  case One
  then show ?case
    by (simp add: numeral-One Num.numeral-One)
next
  case (Bit0 k)
  then show ?case
    by (simp add: numeral-Bit0 Num.numeral-Bit0 ac-simps funpow-add)
next
  case (Bit1 k)
  then show ?case
    by (simp add: numeral-Bit1 Num.numeral-Bit1 ac-simps funpow-add)
qed

end

context semiring-1
begin
lemma numeral-unfold-funpow:
\[\text{numeral } k = ((+ \, 1) \uparrow \text{ numeral } k) \, 0\]
using numeral-add-unfold-funpow [of \(k\, 0\)] by simp
end

context
includes lifting-syntax
begin

lemma transfer-rule-numeral:
\[\text{((=) \implies \(R\)) numeral numeral}\]
if [transfer-rule]: \(\langle R \, 0, 0\rangle, \langle R \, 1, 1\rangle\)
\(\langle R \implies R \implies R\rangle \, (+) \, (+)\)
for \(R::\langle\text{semiring-numeral, monoid-add}\rangle \Rightarrow \langle\text{bool}\rangle\)
\[\text{proof -- have } \langle(=) \implies \(R\)\rangle \, (\lambda k. \, ((+ \, 1) \uparrow \text{ numeral } k) \, 0) \, (\lambda k. \, ((+ \, 1) \uparrow \text{ numeral } k) \, 0)\]
by transfer-prover
moreover have \(\langle\text{numeral} = (\lambda k. \, ((+ \, 'a) \uparrow \text{ numeral } k) \, 0)\rangle\)
using numeral-add-unfold-funpow [where \(?a = 'a\), of \(- 0\)]
by (simp add: fun-eq-iff)
moreover have \(\langle\text{numeral} = (\lambda k. \, ((+ \, 'b) \uparrow \text{ numeral } k) \, 0)\rangle\)
using numeral-add-unfold-funpow [where \(?a = 'b\), of \(- 0\)]
by (simp add: fun-eq-iff)
ultimately show \(?\text{thesis}\)
by simp
qed
end

46.5 Particular lemmas concerning \(2::'a\)

context linordered-field
begin

subclass field-char-0 ..

lemma half-gt-zero-iff: \(0 < a / 2 \iff 0 < a\)
by (auto simp add: field-simps)

lemma half-gt-zero [simp]: \(0 < a \Longrightarrow 0 < a / 2\)
by (simp add: half-gt-zero-iff)
end
46.6  Numeral equations as default simplification rules
declare (in numeral) numeral-One [simp]
declare (in numeral) numeral-plus-numeral [simp]
declare (in numeral) add-numeral-special [simp]
declare (in neg-numeral) add-neg-numeral-simps [simp]
declare (in neg-numeral) add-neg-numeral-special [simp]
declare (in neg-numeral) diff-numeral-simps [simp]
declare (in neg-numeral) diff-numeral-special [simp]
declare (in semiring-numeral) numeral-times-numeral [simp]
declare (in ring-1) mult-neg-numeral-simps [simp]

46.6.1 Special Simplification for Constants
These distributive laws move literals inside sums and differences.
lemmas distrib-right-numeral [simp] = distrib-right [of - - numeral v] for v
lemmas distrib-left-numeral [simp] = distrib-left [of numeral v] for v
lemmas left-diff-distrib-numeral [simp] = left-diff-distrib [of - - numeral v] for v
lemmas right-diff-distrib-numeral [simp] = right-diff-distrib [of numeral v] for v

These are actually for fields, like real
lemmas zero-less-divide-iff-numeral [simp, no-atp] = zero-less-divide-iff [of numeral w] for w
lemmas divide-less-0-iff-numeral [simp, no-atp] = divide-less-0-iff [of numeral w] for w
lemmas zero-le-divide-iff-numeral [simp, no-atp] = zero-le-divide-iff [of numeral w] for w
lemmas divide-le-0-iff-numeral [simp, no-atp] = divide-le-0-iff [of numeral w] for w

Replaces inverse #nn by 1/#nn. It looks strange, but then other simprocs simplify the quotient.
lemmas inverse-eq-divide-numeral [simp] = inverse-eq-divide [of numeral w] for w
lemmas inverse-eq-divide-neg-numeral [simp] = inverse-eq-divide [of - numeral w] for w

These laws simplify inequalities, moving unary minus from a term into the literal.
lemmas equation-minus-iff-numeral [no-atp] = equation-minus-iff [of numeral v] for v
lemmas minus-equation-iff-numeral [no-atp] = minus-equation-iff [of - numeral v] for v
lemmas le-minus-iff-numeral [no-atp] = le-minus-iff [of numeral v] for v
lemmas minus-le-iff-numeral [no-atp] = 
  minus-le-iff [of - numeral v] for v

lemmas less-minus-iff-numeral [no-atp] = 
  less-minus-iff [of numeral v] for v

lemmas minus-less-iff-numeral [no-atp] = 
  minus-less-iff [of - numeral v] for v

Cancellation of constant factors in comparisons (< and ≤)

lemmas mult-less-cancel-left-numeral [simp, no-atp] = 
  mult-less-cancel-left [of numeral v] for v
lemmas mult-less-cancel-right-numeral [simp, no-atp] = 
  mult-less-cancel-right [of - numeral v] for v
lemmas mult-le-cancel-left-numeral [simp, no-atp] = 
  mult-le-cancel-left [of numeral v] for v
lemmas mult-le-cancel-right-numeral [simp, no-atp] = 
  mult-le-cancel-right [of - numeral v] for v

Multiplying out constant divisors in comparisons (<, ≤ and =)

named-theorems divide-const-simps simplication rules to simplify comparisons involving constant divisors

lemmas le-divide-eq-numeral1 [simp,divide-const-simps] = 
  pos-le-divide-eq [of numeral w, OF zero-less-numeral]
  neg-le-divide-eq [of - numeral w, OF neg-numeral-less-zero] for w

lemmas divide-le-eq-numeral1 [simp,divide-const-simps] = 
  pos-divide-le-eq [of numeral w, OF zero-less-numeral]
  neg-divide-le-eq [of - numeral w, OF neg-numeral-less-zero] for w

lemmas less-divide-eq-numeral1 [simp,divide-const-simps] = 
  pos-less-divide-eq [of numeral w, OF zero-less-numeral]
  neg-less-divide-eq [of - numeral w, OF neg-numeral-less-zero] for w

lemmas divide-less-eq-numeral1 [simp,divide-const-simps] = 
  pos-divide-less-eq [of numeral w, OF zero-less-numeral]
  neg-divide-less-eq [of - numeral w, OF neg-numeral-less-zero] for w

lemmas eq-divide-eq-numeral1 [simp,divide-const-simps] = 
  eq-divide-eq [of - - numeral w]
  eq-divide-eq [of - - - numeral w] for w

lemmas divide-eq-eq-numeral1 [simp,divide-const-simps] = 
  divide-eq-eq [of - numeral w]
  divide-eq-eq [of - - numeral w] for w
46.6.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

**lemmas** le-divide-eq-numeral [divide-const-simps] =
le-divide-eq [of numeral w] for w

**lemmas** divide-le-eq-numeral [divide-const-simps] =
divide-le-eq [of - numeral w] for w

**lemmas** less-divide-eq-numeral [divide-const-simps] =
less-divide-eq [of numeral w] for w

**lemmas** divide-less-eq-numeral [divide-const-simps] =
divide-less-eq [of - numeral w] for w

**lemmas** eq-divide-eq-numeral [divide-const-simps] =
less-divide-eq [of - numeral w] for w

**lemmas** divide-eq-eq-numeral [divide-const-simps] =
divide-eq-eq [of - numeral w] for w

Not good as automatic simprules because they cause case splits.

**lemmas** [divide-const-simps] =
le-divide-eq-1 divide-le-eq-1 less-divide-eq-1 divide-less-eq-1

46.7 Setting up simprocs

**lemma** mult-numeral-1: Numeral1 * a = a
  for a :: 'a::semiring-numeral
  by simp

**lemma** mult-numeral-1-right: a * Numeral1 = a
  for a :: 'a::semiring-numeral
  by simp

**lemma** divide-numeral-1: a / Numeral1 = a
  for a :: 'a::field
  by simp

**lemma** inverse-numeral-1: inverse Numeral1 = (Numeral1::'a::division-ring)
  by simp

Theorem lists for the cancellation simprocs. The use of a binary numeral
for 1 reduces the number of special cases.

**Lemma** mult-1s-semiring-numeral:

\[ \text{Numeral1} \times a = a \]
\[ a \times \text{Numeral1} = a \]

for \( a :: a \text{:semiring-numeral} \)

by simp-all

**Lemma** mult-1s-ring-1:

\[ -\text{Numeral1} \times b = -b \]
\[ b \times -\text{Numeral1} = -b \]

for \( b :: a \text{:ring-1} \)

by simp-all

**Lemmas** mult-1s = mult-1s-semiring-numeral mult-1s-ring-1

**Setup**

\[
\text{Reorient-Proc.add}
\]
\[
\text{Reorient-Proc.proc}
\]

46.7.1 Simplification of arithmetic operations on integer constants

**Lemmas** arith-special =

add-numeral-special add-neg-numeral-special
diff-numeral-special

**Lemmas** arith-extra-simps =

numeral-plus-numeral add-neg-numeral-simps add-0-left add-0-right

minus-zero
diff-numeral-simps diff-0 diff-0-right

numeral-times-numeral multi-neg-numeral-simps

multi-zero-left multi-zero-right

abs-numeral abs-neg-numeral

For making a minimal simpset, one must include these default simprules. Also include simp-thms.

**Lemmas** arith-simps =

add-num-simps multi-num-simps sub-num-simps

BitM.simps dbl-simps dbl-inc-simps dbl-dec-simps

abs-zero abs-one arith-extra-simps

**Lemmas** more-arith-simps =
Simplification of relational operations.

**lemmas** rel-simps =
- le-num-simps
- less-num-simps
- eq-num-simps
- le-numeral-simps
- less-numeral-simps
- le-neg-numeral-simps
- less-neg-numeral-simps
- le-minus-one-simps
- le-numeral-extra
- less-numeral-extra
- eq-numeral-simps
- eq-neg-numeral-simps
- eq-numeral-extra

**lemma** Let-numeral [simp]: Let (numeral v) f = f (numeral v)
- Unfold all lets involving constants
**unfolding** Let-def ..

**lemma** Let-neg-numeral [simp]: Let (− numeral v) f = f (− numeral v)
- Unfold all lets involving constants
**unfolding** Let-def ..

**declaration**

```plaintext
declaration (let
    fun number-of ctxt T n =
        if not (Sign.of_sort (Proof_Context.theory_of ctxt) (T, sort numeral))
        then raise CTERM (number-of, [])
        else Numeral.mk_cnumber (Thm.ctyp_of ctxt T) n;
    in
    K (Lin_Arith.set_number_of number_of
        #> Lin_Arith.add_simps
        @[thms arith-simps more-arith-simps rel-simps pred-numeral-simps
            arith-special numeral-One of-nat-simps aminus-numeral-One
            Suc-numeral Let-numeral Let-neg-numeral Let-0 Let-1
            le-Suc-numeral le-numeral-Suc less-Suc-numeral less-numeral-Suc
            Suc-eq-numeral eq-numeral-Suc mult-Suc Suc-right of-nat-numeral])
    end)
```

**46.7.2** Simplification of arithmetic when nested to the right

**lemma** add-numeral-left [simp]: numeral v + (numeral w + z) = (numeral(v + w) + z)
by (simp-all add: add.assoc [symmetric])

lemma add-neg-numeral-left [simp]:
numeral v + (− numeral w + y) = (sub v w + y)
− numeral v + (numeral w + y) = (sub w v + y)
− numeral v + (− numeral w + y) = (− numeral(v + w) + y)
by (simp-all add: add.assoc [symmetric])

lemma mult-numeral-left-semiring-numeral:
numeral v * (numeral w * z) = (numeral(v * w) * z :: 'a::semiring-numeral)
by (simp add: mult.assoc [symmetric])

lemma mult-numeral-left-ring-1:
− numeral v * (numeral w * y) = (− numeral(v * w) * y :: 'a::ring-1)
numeral v * (− numeral w * y) = (− numeral(v * w) * y :: 'a::ring-1)
− numeral v * (− numeral w * y) = (numeral(v * w) * y :: 'a::ring-1)
by (simp-all add: mult.assoc [symmetric])

lemmas mult-numeral-left [simp] =
mult-numeral-left-semiring-numeral
mult-numeral-left-ring-1

hide-const (open) One Bit0 Bit1 BitM inc pow sqr sub dbl dbl-inc dbl-dec

46.8 Code module namespace

code-identifier
code-module Num → (SML) Arith and (OCaml) Arith and (Haskell) Arith

46.9 Printing of evaluated natural numbers as numerals

lemma [code-post]:
Suc 0 = 1
Suc 1 = 2
Suc (numeral n) = numeral (Num.inc n)
by (simp-all add: numeral-inc)

lemmas [code-post] = Num.inc.simps

46.10 More on auxiliary conversion

context semiring-1
begin

lemma numeral-num-of-nat-unfold:
<numeral (num-of-nat n) = (if n = 0 then 1 else of-nat n)>
by (induction n) (simp-all add: numeral-inc ac-simps)

lemma num-of-nat-numeral-eq [simp]:
<num-of-nat (numeral q) = q>
proof (induction q)
  case One
  then show ?case
    by simp
next
  case (Bit0 q)
  then have num-of-nat (numeral (num.Bit0 q)) = num-of-nat (numeral q + numeral q)
    by (simp only: Num.numeral-Bit0 Num.numeral-add)
  also have \ldots = num.Bit0 (num-of-nat (numeral q))
    by (rule num-of-nat-double) simp
  finally show ?case
    using Bit0.IH by simp
next
  case (Bit1 q)
  then have num-of-nat (numeral (num.Bit1 q)) = num-of-nat (numeral q + numeral q + 1)
    by (simp only: Num.numeral-Bit1 Num.numeral-add)
  also have \ldots = num-of-nat (numeral q + numeral q) + num-of-nat 1
    by (rule num-of-nat-plus-distrib) auto
  also have \ldots = num.Bit0 (num-of-nat (numeral q)) + num-of-nat 1
    by (subst num-of-nat-double) auto
  finally show ?case
    using Bit1.IH by simp
qed

end

end

47 Exponentiation

theory Power
  imports Num
begin

47.1 Powers for Arbitrary Monoids

class power = one + times
begin

primrec power :: 'a ⇒ nat ⇒ 'a (infixr ^ 80)
  where
    power-0: a ^ 0 = 1
    | power-Suc: a ^ Suc n = a * a ^ n

notation (latex output)
  power ((\cdot) [1000] 1000)
THEORY “Power”

Special syntax for squares.

abbreviation power2 :: 'a ⇒ 'a ((^2) [1000] 999)
  where x^2 ≡ x *^ 2

end

context
  includes lifting-syntax
begin

lemma power-transfer [transfer-rule]:
  ⟨R ===> (=) ===> R (↑) (↑),
  if [transfer-rule]: ⟨R I I⟩
  ⟨R ===> R ===> R) (↑) (↑),
  for R :: 'a::power ⇒ 'b::power ⇒ bool;
by (simp only: power-def [abs-def] transfer-prover)

end

context monoid-mult
begin

subclass power .

lemma power-one [simp]: 1 ^ n = 1
  by (induct n) simp-all

lemma power-one-right [simp]: a ^ 1 = a
  by simp

lemma power-Suc0-right [simp]: a ^ Suc 0 = a
  by simp

lemma power-commutes: a ^ n * a = a * a ^ n
  by (induct n) (simp-all add: mult.assoc)

lemma power-Suc2: a ^ Suc n = a ^ n * a
  by (simp add: power-commutes)

lemma power-add: a ^ (m + n) = a ^ m * a ^ n
  by (induct m) (simp-all add: algebra-simps)

lemma power-mult: a ^ (m * n) = (a ^ m) ^ n
  by (induct n) (simp-all add: power-add)

lemma power-even-eq: a ^ (2 * n) = (a ^ n) ^ 2
  by (subst mult.commute) (simp add: power-mult)

lemma power-odd-eq: a ^ Suc (2*n) = a * (a ^ n) ^ 2
by (simp add: power-even-eq)

lemma power-numeral-even: $z ^ \text{numeral (Num.Bit0 w)} = (\text{let } w = z ^ \text{(numeral w)} \text{ in } w * w)$
  by (simp only: numeral-Bit0 power-add Let-def)

lemma power-numeral-odd: $z ^ \text{numeral (Num.Bit1 w)} = (\text{let } w = z ^ \text{(numeral w)} \text{ in } z * w * w)$
  by (simp only: numeral-Bit1 One-nat-def add-Suc-right add-0-right power-Suc power-add Let-def mult.assoc)

lemma power2-eq-square: $a^2 = a * a$
  by (simp add: numeral-2-eq-2)

lemma power3-eq-cube: $a^3 = a * a * a$
  by (simp add: numeral-3-eq-3 mult.assoc)

lemma power4-eq-xxxx: $x^4 = x * x * x * x$
  by (simp add: mult.assoc power-numeral-even)

lemma power-numeral-reduce: $x ^ \text{numeral n} = x * x ^ \text{pred-numeral n}$
  by (simp add: numeral-eq-Suc)

lemma funpow-times-power: $(\text{times } x ^ \text{f x}) = \text{times } (x ^ f x)$
proof (induct f x arbitrary: f)
  case 0
  then show ?case by (simp add: fun-eq-iff)
next
  case (Suc n)
  define g where $g \text{ x} = f \text{ x} - 1$ for x
  with Suc have $n = g \text{ x}$ by simp
  with Suc have times x ^ g x = times (x ^ f x) by simp
  moreover from Suc g-def have $f x = g x + 1$ by simp
  ultimately show ?case
    by (simp add: power-add funpow-add fun-eq-iff mult.assoc)
qed

lemma power-commuting-commutes:
  assumes $x * y = y * x$
  shows $x ^ n * y = y * x ^ n$
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have $x ^ \text{Suc n} * y = x ^ \text{n * y * x}$
    by (subst power-Suc2) (simp add: assms ac-simps)
  also have $\ldots = y * x ^ \text{Suc n}$
    by (simp only: Suc power-Suc2) (simp add: ac-simps)
finally show ?case.
qed

lemma power-minus-mult: \(0 < n \implies a^{(n - 1)} * a = a^n\)
  by (simp add: power-commutes split: nat-diff-split)

lemma left-right-inverse-power:
  assumes \(x * y = 1\)
  shows \(x^n * y^n = 1\)
proof (induct n)
case (Suc n)
  moreover have \(x^{Suc n} * y^{Suc n} = x^n * (x * y) * y^n\)
    by (simp add: power-Suc2 [symmetric] mult.assoc [symmetric])
ultimately show ?case by (simp add: assms)
qed simp

end

category comm-monoid-mult
begin

lemma power-mult-distrib [algebra-simps, algebra-split-simps, field-simps, field-split-simps, divide-simps]:
  \((a * b)^n = (a^n) * (b^n)\)
by (induction n) (simp-all add: ac-simps)

end

Extract constant factors from powers.
declare power-mult-distrib [where \(a = \text{numeral } w\) for \(w\), simp]
declare power-mult-distrib [where \(b = \text{numeral } w\) for \(w\), simp]

lemma power-add-numeral [simp]: \(a^{\text{numeral } m} * a^{\text{numeral } n} = a^{\text{numeral } (m + n)}\)
  for \(a :: 'a::monoid-mult\)
by (simp add: power-add [symmetric])

lemma power-add-numeral2 [simp]: \(a^{\text{numeral } m} * (a^{\text{numeral } n} * b) = a^{\text{numeral } (m + n)} * b\)
  for \(a :: 'a::monoid-mult\)
by (simp add: mult.assoc [symmetric])

lemma power-mult-numeral [simp]: \((a^{\text{numeral } m})^{\text{numeral } n} = a^{\text{numeral } (m * n)}\)
  for \(a :: 'a::monoid-mult\)
by (simp only: numeral-mul power-mult)

end
lemma numeral-sqr: numeral (Num.sqr k) = numeral k * numeral k
  by (simp only: sqr-conv-mult numeral-mult)

lemma numeral-pow: numeral (Num.pow k l) = numeral k ^ numeral l
  by (induct l)
    (simp-all only: numeral-class.numeral.simps pow.simps
     numeral-sqr numeral-mult power-add power-one-right)

lemma power-numeral [simp]: numeral k ^ numeral l = numeral (Num.pow k l)
  by (rule numeral-pow [symmetric])

end

context semiring-1
begin

lemma of-nat-power [simp]: of-nat (m ^ n) = of-nat m ^ n
  by (induct n) simp-all

lemma zero-power: 0 < n =⇒ 0 ^ n = 0
  by (cases n) simp-all

lemma power-zero-numeral [simp]: 0 ^ numeral k = 0
  by (simp add: numeral-eq-Suc)

lemma zero-power2: 0 ^ 2 = 0
  by (rule power-zero-numeral)

lemma one-power2: 1 ^ 2 = 1
  by (rule power-one)

lemma power-0-Suc [simp]: 0 ^ Suc n = 0
  by simp

It looks plausible as a simprule, but its effect can be strange.

lemma power-0-left: 0 ^ n = (if n = 0 then 1 else 0)
  by (cases n) simp-all

end

context semiring-char-0 begin

lemma numeral-power-eq-of-nat-cancel-iff [simp]:
  numeral x ^ n = of-nat y =⇒ numeral x ^ n = y
  using of-nat-eq-iff by fastforce

lemma real-of-nat-eq-numeral-power-cancel-iff [simp]:
  of-nat y = numeral x ^ n =⇒ y = numeral x ^ n
using numeral-power-eq-of-nat-cancel-iff[of x n y] by (metis (mono-tags))

lemma of-nat-eq-of-nat-power-cancel-iff[simp]: (of-nat b) ~ w = of-nat x ⟷ b ~ w = x 
  by (metis of-nat-power of-nat-eq-iff)

lemma of-nat-power-eq-of-nat-cancel-iff[simp]: of-nat x = (of-nat b) ~ w ⟷ x = b ~ w 
  by (metis of-nat-eq-of-nat-power-cancel-iff)

end

context comm-semiring-1
begin

The divides relation.

lemma le-imp-power-dvd:
  assumes m ≤ n 
  shows a ~ m dvd a ~ n 
proof
  from assms have a ~ n = a ~ (m + (n − m)) by simp 
  also have ... = a ~ m * a ~ (n − m) by (rule power-add) 
  finally show a ~ n = a ~ m * a ~ (n − m) . 
qed

lemma power-le-dvd: a ~ n dvd b ⟷ m ≤ n ⟷ a ~ m dvd b 
  by (rule dvd-trans [OF le-imp-power-dvd])

lemma dvd-power-same: x dvd y ⟷ x ~ n dvd y ~ n 
  by (induct n) (auto simp add: mult-dvd-mono)

lemma dvd-power-le: x dvd y ⟷ m ≥ n ⟷ x ~ n dvd y ~ m 
  by (rule power-le-dvd [OF dvd-power-same])

lemma dvd-power [simp]:
  fixes n :: nat
  assumes n > 0 ∨ x = 1 
  shows x dvd (x ~ n) 
  using assms
proof
  assume 0 < n 
  then have x ~ n = x ~ Suc (n − 1) by simp 
  then show x dvd (x ~ n) by simp
next
  assume x = 1 
  then show x dvd (x ~ n) by simp
qed

end
context semiring-1-no-zero-divisors
begin

subclass power .

lemma power-eq-0-iff [simp]: \( a ^ n = 0 \leftrightarrow a = 0 \land n > 0 \)
  by (induct n) auto

lemma power-not-zero: \( a \neq 0 \implies a ^ n \neq 0 \)
  by (induct n) auto

lemma zero-eq-power2 [simp]: \( a^2 = 0 \leftrightarrow a = 0 \)
  unfolding power2-eq-square by simp

end

context ring-1
begin

lemma power-minus: \((- a) ^ n = (- 1) ^ n \cdot a ^ n\)
  proof (induct n)
    case 0
    show ?case by simp
  next
    case (Suc n)
    then show ?case
      by (simp del: power-Suc add: power-Suc2 mult.assoc)
  qed

lemma power-minus': NO-MATCH \( I \ x = (- x) ^ n = (- 1) ^ n \cdot x ^ n \)
  by (rule power-minus)

lemma power-minus-Bit0: \((- x) ^ \text{numeral (Num.Bit0 k)} = x ^ \text{numeral (Num.Bit0 k)}\)
  by (induct k, simp-all-only: numeral-class.numeral.simps power-add
      power-one-right mult-minus-left mult-minus-right minus-minus)

lemma power-minus-Bit1: \((- x) ^ \text{numeral (Num.Bit1 k)} = - (x ^ \text{numeral (Num.Bit1 k)})\)
  by (simp only: eval-nat-numeral3 power-Suc power-minus-Bit0 mult-minus-left)

lemma power2-minus [simp]: \((- a)^2 = a^2\)
  by (fact power-minus-Bit0)

lemma power-minus1-even [simp]: \((- 1) ^ (2\cdot n) = 1\)
  proof (induct n)
    case 0
    show ?case by simp
  next
    case (Suc n)
    show ?case
      by (simp del: power-Suc add: power-Suc2 mult.assoc)
  qed
THEORY "Power"

next
  case (Suc n)
  then show ?case by (simp add: power-add power2-eq-square)
qed

lemma power-minus1-odd: \((-1)^{\text{Suc} (2*n)} = -1\)
  by simp

lemma power-minus-even [simp]: \((-a)^{2*n} = a^{(2*n)}\)
  by (simp add: power-minus [of a])

end

context ring-1-no-zero-divisors
begin

lemma power2-eq-1-iff: \(a^2 = 1 \iff a = 1 \lor a = -1\)
  using square-eq-1-iff [of a] by (simp add: power2-eq-square)

end

context idom
begin

lemma power2-eq-iff: \(x^2 = y^2 \iff x = y \lor x = -y\)
  unfolding power2-eq-square by (rule square-eq-iff)

end

context semidom-divide
begin

lemma power-diff:
  \(a^{m-n} = (a^m) \div (a^n)\) if \(a \neq 0\) and \(n \leq m\)

proof -
  define q where \(q = m - n\)
  with \(n \leq m\) have \(m = q + n\) by simp
  with \(a \neq 0\) q-def show \(?thesis\)
    by (simp add: power-add)
qed

end

context algebraic-semidom
begin

lemma div-power: \(b \div a \Longrightarrow (a \div b)^{\text{Suc} n} = a^{\text{Suc} n} \div b^{\text{Suc} n}\)
  by (induct n) (simp-all add: div-mult-if-dvd dvd-power-same)
lemma is-unit-power-iff: \( \text{is-unit} \ (a \ ^n) \iff \text{is-unit} \ a \ \lor \ n = 0 \)
by (induct n) (auto simp add: is-unit-mult-iff)

lemma dvd-power-iff:
assumes \( x \ \neq \ 0 \)
shows \( x \ ^m \ \text{dvd} \ x \ ^n \iff \text{is-unit} \ x \ \lor \ m \ \leq \ n \)
proof
assume \( \ast: \ x \ ^m \ \text{dvd} \ x \ ^n \)
{ 
assume \( m > n \)
note \( \ast \)
also have \( x \ ^n = x \ ^n \ast 1 \) by simp
also from \( m > n \) have \( m = n + (m - n) \) by simp
also have \( x \ \ldots = x \ ^n \ast x \ ^{(m - n)} \) by (rule power-add)
finally have \( x \ ^{(m - n)} \ \text{dvd} \ 1 \)
using assms by (subst (asm) dvd-times-left-cancel-iff) simp-all
with \( m > n \) have \( \text{is-unit} \ x \) by (simp add: is-unit-power-iff)
} 
thus \( \text{is-unit} \ x \ \lor \ m \ \leq \ n \) by force
qed (auto intro: unit-imp-dvd simp: is-unit-power-iff le-imp-power-dvd)

end

context normalization-semidom-multiplicative
begin

lemma normalize-power: normalize \((a \ ^n)\) = normalize \((a \ ^n)\)
by (induct n) (simp-all add: normalize-mult)

lemma unit-factor-power: unit-factor \((a \ ^n)\) = unit-factor \((a \ ^n)\)
by (induct n) (simp-all add: unit-factor-mult)

end

context division-ring
begin

Perhaps these should be simprules.

lemma power-inverse [field-simps, field-split-simps, divide-simps]: inverse \((a \ ^n)\) = inverse \((a \ ^n)\)
proof (cases a = 0)
case True
then show \( \ast \)thesis by (simp add: power-0-left)
next
case False
then have inverse \((a \ ^n)\) = inverse \((a \ ^n)\)
by (induct n) (simp-all add: nonzero-inverse-mult-distrib power-commutes)
then show \( \ast \)thesis by simp

end
qed

lemma power-one-over [field-simps, field-split-simps, divide-simps]: \((1 / a)^n = 1 / a^n\)
  using power-inverse [of a] by (simp add: divide-inverse)

end

context field
begin

lemma power-divide [field-simps, field-split-simps, divide-simps]: \((a / b)^n = a^n / b^n\)
  by (induct n) simp-all

end

47.2 Exponentiation on ordered types

context linordered-semidom
begin

lemma zero-less-power [simp]: \(0 < a \Rightarrow 0 < a^n\)
  by (induct n) simp-all

lemma zero-le-power [simp]: \(0 \leq a \Rightarrow 0 \leq a^n\)
  by (induct n) simp-all

lemma power-mono: \(a \leq b \Rightarrow 0 \leq a \leq b^n\)
  by (induct n) (auto intro: mult-mono order-trans [of 0 a b])

lemma one-le-power [simp]: \(1 \leq a \Rightarrow 1 \leq a^n\)
  using power-mono [of 1 a n] by simp

lemma power-le-one: \(0 \leq a \Rightarrow a \leq 1 \Rightarrow a^n \leq 1\)
  using power-mono [of a 1 n] by simp

lemma power-gt1-lemma:
  assumes \(gt1\): \(1 < a\)
  shows \(1 < a * a^n\)
  proof −
    from \(gt1\) have \(0 \leq a\)
      by (fact order-trans [OF zero-le-one less-imp-le])
    from \(gt1\) have \(1 * 1 < a * 1\) by simp
    also from \(gt1\) have \(\ldots \leq a * a^n\)
      by (simp only: mult-mono \(\langle 0 \leq a \rangle\) one-le-power order-less-imp-le zero-le-one order-refl)
    finally show \(?thesis\) by simp
  qed
lemma power-gt1: \(1 < a \Rightarrow 1 < a \cdot \text{Suc} n\)
by (simp add: power-gt1-lemma)

lemma one-less-power [simp]: \(1 < a \Rightarrow 0 < n \Rightarrow 1 < a^n\)
by (cases n) (simp-all add: power-gt1-lemma)

lemma power-le-imp-le-exp:
assumes gt1: \(1 < a\)
shows \(a \cdot m \leq a \cdot n \Rightarrow m \leq n\)
proof (induct m arbitrary: n)
  case 0
  show ?case by simp
next
  case (Suc m)
  show ?case (cases n)
    case 0
    with Suc have \(a \cdot a \cdot m \leq 1\) by simp
    with gt1 show ?thesis
      by (force simp only: power-gt1-lemma not-less [symmetric])
  next
  case (Suc n)
    with Suc.prems Suc.hyps show ?thesis
      by (force dest: mult-left-le-imp-le simp add: less-trans [OF zero-less-one gt1])
  qed
qed

lemma of-nat-zero-less-power-iff [simp]: \(\text{of-nat } x^n > 0 \iff x > 0 \lor n = 0\)
by (induct n) auto

Surely we can strengthen this? It holds for \(0 < a < 1\) too.

lemma power-inject-exp [simp]:
\[ \langle a \cdot m = a \cdot n \iff m = n \rangle \text{ if } (1 < a)\]
using that by (force simp add: order-class.order.antisym power-le-imp-le-exp)

Can relax the first premise to \((0::'a) < a\) in the case of the natural numbers.

lemma power-less-imp-le-exp: \(1 < a \Rightarrow a \cdot m < a \cdot n \Rightarrow m < n\)
by (simp add: order-less-le [of m n] less-le [of a·m a·n] power-le-imp-le-exp)

lemma power-strict-mono: \(a < b \Rightarrow 0 \leq a \Rightarrow 0 < n \Rightarrow a \cdot n < b \cdot n\)
proof (induct n)
  case 0
  then show ?case by simp
next
case (Suc n)
then show ?case
  by (cases n = 0) (auto simp: mult-strict-mono le-less-trans [of 0 a b])
qed
lemma power-mono-iff [simp]:
  shows \([a \geq 0; b \geq 0; n > 0]\) \(\Rightarrow a \uparrow n \leq b \uparrow n \iff a \leq b\)
  using power-mono [of a b] power-strict-mono [of b a] not-le by auto

Lemma for power-strict-decreasing

lemma power-Suc-less: \(0 < a \Rightarrow a < 1 \Rightarrow a \cdot a \uparrow n < a \uparrow n\)
  using (induct n) (auto simp: mult-strict-left-mono)

lemma power-strict-decreasing: \(n < N \Rightarrow 0 < a \Rightarrow a < 1 \Rightarrow a \uparrow N < a \uparrow n\)
  proof (induction N)
    case 0
    then show ?case by simp
  next
    case (Suc N)
    then show ?case
      using mult-strict-mono [of a 1 a \uparrow N a \uparrow n]
      by (auto simp add: power-Suc-less less-Suc-eq)
  qed

Proof resembles that of power-strict-decreasing.

lemma power-decreasing: \(n \leq N \Rightarrow 0 \leq a \Rightarrow a \leq 1 \Rightarrow a \uparrow N \leq a \uparrow n\)
  proof (induction N)
    case 0
    then show ?case by simp
  next
    case (Suc N)
    then show ?case
      using mult-mono [of a 1 a \uparrow N a \uparrow n]
      by (auto simp add: le-Suc-eq)
  qed

lemma power-decreasing-iff [simp]: \([0 < b; b < 1]\) \(\Rightarrow b \uparrow m \leq b \uparrow n \iff n \leq m\)
  using power-strict-decreasing [of m n b]
  by (auto intro: power-decreasing ccontr)

lemma power-strict-decreasing-iff [simp]: \([0 < b; b < 1]\) \(\Rightarrow b \uparrow m < b \uparrow n \iff n < m\)
  using power-decreasing-iff [of b m n] unfolding le-less
  by (auto dest: power-strict-decreasing le-neg-implies-less)

lemma power-Suc-less-one: \(0 < a \Rightarrow a < 1 \Rightarrow a \uparrow \text{Suc } n < 1\)
  using power-strict-decreasing [of 0 Suc n a] by simp

Proof again resembles that of power-strict-decreasing.

lemma power-increasing: \(n \leq N \Rightarrow 1 \leq a \Rightarrow a \uparrow n \leq a \uparrow N\)
  proof (induct N)
    case 0

...
then show \(?\)case by simp

next
  case (Suc \(N\))
  then show \(?\)case
    using mult-mono[of \(1\ a\ a\ ^n\ a\ ^N\)]
    by (auto simp add: le-Suc-eq order-trans [OF zero-le-one])

qed

Lemma for power-strict-increasing.

lemma power-less-power-Suc: \(1 < a\ \Rightarrow\ a\ ^n < a\ *\ a\ ^n\)
  by (induct \(n\)) (auto simp: mult-strict-left-mono less-trans [OF zero-less-one])

lemma power-strict-increasing: \(n < N\ \Rightarrow\ 1 < a\ \Rightarrow\ a\ ^n < a\ ^N\)
  proof (induct \(N\))
    case 0
    then show \(?\)case by simp
  next
    case (Suc \(N\))
    then show \(?\)case
      using mult-strict-mono[of \(1\ a\ a\ ^n\ a\ ^N\)]
      by (auto simp add: power-less-power-Suc less-Suc-eq less-trans [OF zero-less-one] less-imp-le)

qed

lemma power-increasing-iff [simp]: \(1 < b\ \Rightarrow\ b\ ^x \leq b\ ^y\ \iff\ x \leq y\)
  by (blast intro: power-le-imp-le-exp power-increasing less-imp-le)

lemma power-strict-increasing-iff [simp]: \(1 < b\ \Rightarrow\ b\ ^x < b\ ^y\ \iff\ x < y\)
  by (blast intro: power-less-imp-less-exp power-strict-increasing)

lemma power-le-imp-le-base:
  assumes le: \(a\ ^n \leq b\ ^n\)
  and \(0 \leq b\)
  shows \(a\ \leq b\)
  proof (rule contrapos-pp)
    assume \(~\)thesis
    then have \(b < a\) by (simp only: linorder-not-le)
    then have \(b\ ^n \leq a\ ^n\)
      by (simp only: assms(2) power-strict-mono)
    with le show False
      by (simp add: linorder-not-less [symmetric])
  qed

lemma power-less-imp-less-base:
  assumes less: \(a\ ^n < b\ ^n\)
  assumes nonneg: \(0 \leq b\)
  shows \(a < b\)
  proof (rule contrapos-pp [OF less])
    assume \(~\)thesis
  qed
then have $b \leq a$ by (simp only: linorder-not-less)
from this nonneg have $b^n \leq a^n$ by (rule power-mono)
then show $a^n < b^n$ by (simp only: linorder-not-less)
qed

lemma power-inject-base: $a^{Suc\ n} = b^{Suc\ n} \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a = b$
by (blast intro: power-le-imp-base order.antisym eq-refl sym)

lemma power-eq-imp-eq-base: $a^n = b^n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 < n \Longrightarrow a = b$
by (cases n) (simp-all del: power-Suc, rule power-inject-base)

lemma power-eq-iff-eq-base: $0 < n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 < n \Longrightarrow a = b$
by (cases n) (simp-all del: power-Suc, rule power-inject-base)

lemma power2-le-imp-le: $x^2 \leq y^2 \Longrightarrow 0 \leq y \Longrightarrow x \leq y$
unfolding numeral-2-eq-2 by (rule power-le-imp-le-base)

lemma power2-less-imp-less: $x^2 < y^2 \Longrightarrow 0 \leq y \Longrightarrow x < y$
by (rule power-less-imp-less-base)

lemma power2-eq-imp-eq: $x^2 = y^2 \Longrightarrow 0 \leq x \Longrightarrow 0 \leq y \Longrightarrow x = y$
unfolding numeral-2-eq-2 by (erule (2) power-eq-imp-eq-base) simp

lemma power-Suc-le-self: $0 \leq a \Longrightarrow a \leq 1 \Longrightarrow a^{Suc\ n} \leq a$
using power-decreasing [of 1 Suc n a] by simp

lemma power2-eq-iff-nonneg [simp]:
assumes $0 \leq x \leq y$
shows $(x^2 = y^2) \leftrightarrow x = y$
using assms power2-eq-imp-eq by blast

lemma of-nat-less-numeral-power-cancel-iff [simp]:
of-nat $x < \text{numeral } i^n \leftrightarrow x < \text{numeral } i^n$
using of-nat-less-iff [of x numeral i ^ n, unfolded of-nat-numeral of-nat-power] .

lemma of-nat-le-numeral-power-cancel-iff [simp]:
of-nat $x \leq \text{numeral } i^n \leftrightarrow x \leq \text{numeral } i^n$

lemma numeral-power-less-of-nat-cancel-iff [simp]:
umeral $i^n < \text{of-nat } x \leftrightarrow \text{numeral } i^n < x$
using of-nat-less-iff [of numeral i ^ n x, unfolded of-nat-numeral of-nat-power] .

lemma numeral-power-le-of-nat-cancel-iff [simp]:
umeral $i^n \leq \text{of-nat } x \leftrightarrow \text{numeral } i^n \leq x$
lemma of-nat-le-of-nat-power-cancel-iff[simp]: (of-nat b) \^ w \leq of-nat x \iff b \^ w \leq x
  by (metis of-nat-le-iff of-nat-power)

lemma of-nat-power-le-of-nat-cancel-iff[simp]: of-nat x \leq (of-nat b) \^ w \iff x \leq b \^ w
  by (metis of-nat-le-iff of-nat-power)

lemma of-nat-less-of-nat-power-cancel-iff[simp]: (of-nat b) \^ w < of-nat x \iff b \^ w < x
  by (metis of-nat-less-iff of-nat-power)

lemma of-nat-power-less-of-nat-cancel-iff[simp]: of-nat x < (of-nat b) \^ w \iff x < b \^ w
  by (metis of-nat-less-iff of-nat-power)

lemma power2-nonneg-ge-1-iff:
  assumes x \geq 0
  shows x \^ 2 \geq 1 \iff x \geq 1
  using assms by (auto intro: power2-le-imp-le)

lemma power2-nonneg-gt-1-iff:
  assumes x \geq 0
  shows x \^ 2 > 1 \iff x > 1
  using assms by (auto intro: power-less-imp-base)

end

Some nat-specific lemmas:

lemma mono-ge2-power-minus-self:
  assumes k \geq 2
  shows mono (\lambda m. k \^ m - m)
  unfolding mono-iff-le-Suc
  proof
    fix n
    have k \^ n < k \^ Suc n using power-strict-increasing-iff[of k n Suc n] assms by linarith
    thus k \^ n - n \leq k \^ Suc n - Suc n by linarith
  qed

lemma self-le-ge2-pow[simp]:
  assumes k \geq 2
  shows m \leq k \^ m
  proof (induction m)
    case 0 show ?case by simp
  next
    case (Suc m)
    hence Suc m \leq Suc (k \^ m) by simp
    also have \ldots \leq k \^ m + k \^ m using one-le-power[of k m] assms by linarith
    also have \ldots \leq k \* k \^ m by (metis mult-2 mult-le-mono1[OF assms])
    finally show ?case by simp
  end
qed

lemma diff-le-diff-pow [simp]:
assumes \( k \geq 2 \)
shows \( m - n \leq k \, ^m - k \, ^n \)
proof (cases \( n \leq m \))
  case True
  thus ?thesis
  using monoD [OF mono-ge2-power-minus-self [OF assms] True] self-le-ge2-pow [OF assms, of m]
    by (simp add: le-diff-conv le-diff-conv2)
qed auto

context linordered-ring-strict
begin

lemma sum-squares-eq-zero-iff:
\( x \times x + y \times y = 0 \) \iff \( x = 0 \) \& \( y = 0 \)
by (simp add: add-nonneg-eq-0-iff)

lemma sum-squares-le-zero-iff:
\( x \times x + y \times y \leq 0 \) \iff \( x = 0 \) \& \( y = 0 \)
by (simp add: le-less not-sum-squares-lt-zero sum-squares-eq-zero-iff)

lemma sum-squares-gt-zero-iff:
\( 0 < x \times x + y \times y \) \iff \( x \neq 0 \) \lor \( y \neq 0 \)
by (simp add: not-le [symmetric] sum-squares-le-zero-iff)

end

context linordered-idom
begin

lemma zero-le-power2 [simp]: \( 0 \leq a^2 \)
by (simp add: power2-eq-square)

lemma zero-less-power2 [simp]: \( 0 < a^2 \) \iff \( a \neq 0 \)
by (force simp add: power2-eq-square zero-less-mult-iff linorder-neq-iff)

lemma power2-less-0 [simp]: \( \neg a^2 < 0 \)
by (force simp add: power2-eq-square mult-less-0-iff)

lemma power-abs: \(|a \, ^n| = |a| \, ^n \) -- FIXME simp?
by (induct n) (simp-all add: abs-mult)

lemma power-sgn [simp]: \( \sgn (a \, ^n) = \sgn a \, ^n \)
by (induct n) (simp-all add: sgn-mult)

lemma abs-power-minus [simp]: \(|(\neg a) \, ^n| = |a \, ^n|\)
by (simp add: power-abs)

lemma zero-less-power-abs-iff [simp]: \( 0 < |a| \, ^n \) \iff \( a \neq 0 \) \lor \( n = 0 \)

proof (induct n)
  case 0
  show ?case by simp
next
  case Suc
  then show ?case by (auto simp: zero-less-mult-iff)

lemma zero-le-power-abs [simp]: 0 ≤ |a| ^ n
  by (rule zero-le-power [OF abs-ge-zero])

lemma power2-less-eq-zero-iff [simp]: a^2 ≤ 0 ⟷ a = 0
  by (simp add: le-less)

lemma abs-power2 [simp]: |a|^2 = a^2
  by (simp add: power2-eq-square)

lemma power2-abs [simp]: |a|^2 = a^2
  by (simp add: power2-eq-square)

lemma odd-power-less-zero: a < 0 ⟹ a ^ Suc (2 * n) < 0
proof (induct n)
  case 0
  then show ?case
next
  case (Suc n)
  have a ^ Suc (2 * Suc n) = (a*a) * a ^ Suc(2*n)
    by (simp add: ac-simps power-add power2-eq-square)
  then show ?case
    by (simp del: power-Suc add: Suc mult-less-0-iff mult-neg-neg)

lemma odd-0-le-power-imp-0-le: 0 ≤ a ^ Suc (2 * n) ⟹ 0 ≤ a
  using odd-power-less-zero [of a n]
  by (force simp add: linorder-not-less [symmetric])

lemma zero-le-even-power [simp]: 0 ≤ a ^ (2 * n)
proof (induct n)
  case 0
  show ?case by simp
next
  case (Suc n)
  have a ^ (2 * Suc n) = (a*a) * a ^ (2*n)
    by (simp add: ac-simps power-add power2-eq-square)
  then show ?case
    by (simp add: Suc zero-le-mult-iff)

lemma sum-power2-ge-zero: 0 ≤ x^2 + y^2
by (intro add-nonneg-nonneg zero-le-power2)

lemma not-sum-power2-lt-zero: \(\neg x^2 + y^2 < 0\)
  unfolding not-less by (rule sum-power2-ge-zero)

lemma sum-power2-eq-zero-iff: \(x^2 + y^2 = 0 \iff x = 0 \land y = 0\)
  unfolding power2-eq-square by (simp add: add-nonneg-eq-0-iff)

lemma sum-power2-le-zero-iff: \(x^2 + y^2 \leq 0 \iff x = 0 \land y = 0\)
  by (simp add: le-less sum-power2-eq-zero-iff not-sum-power2-lt-zero)

lemma sum-power2-gt-zero-iff: \(0 < x^2 + y^2 \iff x \neq 0 \lor y \neq 0\)
  unfolding not-le [symmetric] by (simp add: sum-power2-le-zero-iff)

lemma abs-le-square-iff: \(|x| \leq |y| \iff x^2 \leq y^2\)
  (is \(\text{?lhs} \iff \text{?rhs}\))
  proof
    assume \(\text{?lhs}\)
    then have \(|x|^2 \leq |y|^2\) by (rule power-mono) simp
    then show \(\text{?rhs}\) by simp
  next
    assume \(\text{?rhs}\)
    then show \(\text{?lhs}\)
      by (auto intro!: power2-le-imp-le [OF - abs-ge-zero])
  qed

lemma power2-le-iff-abs-le:
  \(y \geq 0 \implies x^2 \leq y^2 \iff |x| \leq y\)
  by (metis abs-le-square-iff abs-of-nonneg)

lemma abs-square-le-1: \(x^2 \leq 1 \iff |x| \leq 1\)
  using abs-le-square-iff [of x 1] by simp

lemma abs-square-eq-1: \(x^2 = 1 \iff |x| = 1\)
  by (auto simp add: abs-if power2-eq-1-iff)

lemma abs-square-less-1: \(x^2 < 1 \iff |x| < 1\)
  using abs-square-eq-1 [of x] abs-square-le-1 [of x] by (auto simp add: le-less)

lemma square-le-1:
  assumes \(-1 \leq x\)
  shows \(x^2 \leq 1\)
  using assms
    by (metis add.inverse-inverse linear mult-le-one neg-equal-0-iff-equal neg-le-iff-le
         power2-eq-square power-minus-Bit0)

diary
47.3 Miscellaneous rules

context linordered-semidom
begin

lemma self-le-power: \( 1 \leq a \implies 0 < n \implies a \leq a^n \)
using power-increasing [of \( 1 \) \( n \) \( a \)] power-one-right [of \( a \)] by auto

lemma power-le-one-iff: \( 0 \leq a \implies a^n \leq 1 \iff (n = 0 \vee a \leq 1) \)
by (metis (mono-tags) gr0I nle-le one-le-power power-le-one self-le-power power-0)

end

lemma power2-ge-1-iff: \( x^2 \geq 1 \iff x \geq 1 \vee x \leq (-1 :: 'a :: linordered-idom) \)
using abs-le-square-iff [of \( 1 \) \( x \)] by (auto simp: abs_if split: if-splits)

lemma (in power) power-eq-if: \( p^m = (\text{if } m=0 \text{ then } 1 \text{ else } p \cdot (p^\sim (m - 1))) \)
unfolding One-nat-def by (cases \( m \)) simp-all

lemma (in comm-semiring-1) power2-sum: \( (x + y)^2 = x^2 + y^2 + 2 \cdot x \cdot y \)
by (simp add: algebra-simps power2-eq-square mult-2-right)

context comm-ring-1
begin

lemma power2-diff: \( (x - y)^2 = x^2 + y^2 - 2 \cdot x \cdot y \)
by (simp add: algebra-simps power2-eq-square mult-2-right)

lemma power2-commute: \( (x - y)^2 = (y - x)^2 \)
by (simp add: algebra-simps power2-eq-square)

lemma minus-power-mult-self: \((- a)^n \cdot (- a)^n = a^{2 \cdot n} \)
by (simp add: power-mul-distrib [symmetric])
  (simp add: power2-eq-square [symmetric] power-mul [symmetric])

lemma minus-one-mult-self [simp]: \((- 1)^n \cdot (- 1)^n = 1 \)
using minus-power-mult-self [of \( 1 \) \( n \)] by simp

lemma left-minus-one-mult-self [simp]: \((- 1)^n \cdot ((- 1)^n \cdot a) = a \)
by (simp add: mult.assoc [symmetric])

end
Simprules for comparisons where common factors can be cancelled.

**lemmas** zero-compare-simps =
add-strict-increasing add-strict-increasing2 add-increasing
zero-le-mult-iff zero-le-divide-iff
zero-less-mult-iff zero-less-divide-iff
mult-le-0-iff divide-le-0-iff
mult-less-0-iff divide-less-0-iff
zero-le-power2 power2-less-0

### 47.4 Exponentiation for the Natural Numbers

**lemma** nat-one-le-power [simp]: Suc 0 ≤ i ≤ Suc 0 ≤ i ^ n
by (rule one-le-power [of i n, unfolded One-nat-def])

**lemma** nat-zero-less-power-iff [simp]: x ^ n > 0 ←→ x > 0 ∨ n = 0
for x :: nat
by (induct n) auto

**lemma** nat-power-eq-Suc-0-iff [simp]: x ^ m = Suc 0 ←→ m = 0 ∨ x = Suc 0
by (induct m) auto

**lemma** power-Suc-0 [simp]: Suc 0 ^ n = Suc 0
by simp

Valid for the naturals, but what if 0 < i < 1? Premises cannot be weakened:
consider the case where i = 0, m = 1 and n = 0.

**lemma** nat-power-less-imp-less:
fixes i :: nat
assumes nonneg: 0 < i
assumes less: i ^ m < i ^ n
shows m < n
proof (cases i = 1)
case True
with less power-one [where 'a = nat] show ?thesis by simp
next
case False
with nonneg have 1 < i by auto
from power-strict-increasing-iff [OF this] less show ?thesis ..
qed

**lemma** power-gt-expt: n > Suc 0 ⇒ n ^ k > k
by (induction k) (auto simp: less-trans-Suc n-less-m-mult-n)

**lemma** less-exp [simp]:
(n < 2 ^ n)
by (simp add: power-gt-expt)

**lemma** power-dvd-imp-le:
fixes i :: nat
THEORY “Power”

assumes \( i \vdash m \text{ dvd } i \vdash n \) \( 1 < i \)
shows \( m \leq n \)

by (auto intro: power-le-imp-le-exp [OF \( 1 < i \) dvd-imp-le])

lemma \( \text{dvd-power-iff-le} \):

fixes \( k :: \text{nat} \)
shows \( 2 \leq k \implies (k \vdash m \text{ dvd } k \vdash n \iff m \leq n) \)

using \( \text{le-imp-power-dvd} \) \( \text{power-dvd-imp-le} \) by force

lemma \( \text{power2-nat-le-eq-le} \):

\( m^2 \leq n^2 \iff m \leq n \)

for \( m n :: \text{nat} \)
by (auto intro: power2-le-imp-le power-mono)

lemma \( \text{power2-nat-le-imp-le} \):

fixes \( m n :: \text{nat} \)
assumes \( m^2 \leq n \)
shows \( m \leq n \)

proof (cases \( m \))
  case 0
  then show \( \text{thesis} \) by simp

next
  case (Suc \( k \))
  show \( \text{thesis} \)
  proof (rule ccontr)
    assume \( \neg \text{thesis} \)
    then have \( n < m \) by simp
    with assms Suc show False
    by (simp add: power2-eq-square)
  qed

qed

lemma \( \text{ex-power-ivl1} \):

fixes \( b k :: \text{nat} \)
assumes \( b \geq 2 \)
shows \( k \geq 1 \implies \exists n. b^n \leq k \land k < b^{(n+1)} \) (is \( - \implies \exists n. ?P k n \))

proof (induction \( k \))
  case 0 thus \( ?\text{case} \) by simp

next
  case (Suc \( k \))
  show \( ?\text{case} \)
  proof (cases)
    assume \( k=0 \)
    hence \( ?P (\text{Suc } k) \) 0 using assms by simp
    thus \( ?\text{case} \) ..

next
  assume \( k \neq 0 \)
  with Suc obtain \( n \) where \( \text{IH: } ?P k n \) by auto
  show \( ?\text{case} \)
  proof (cases \( k = b^{(n+1)} - 1 \))
    case True
    hence \( ?P (\text{Suc } k) (n+1) \) using assms
by (simp add: power-less-power-Suc)
thus ?thesis ..
next
case False
hence ?P (Suc k) n using IH by auto
thus ?thesis ..
qed
qed
qed

lemma ex-power-ivl2: fixes b k :: nat assumes b ≥ 2 k ≥ 2
shows ∃n. b n < k ∧ k ≤ b (n+1)
proof –
have 1 ≤ k − 1 using assms(2) by arith
from ex-power-ivl1[OF assms(1) this]
obtain n where b n ≤ k − 1 ∧ k − 1 < b (n+1) ..
hence b n < k ∧ k ≤ b (n+1) using assms by auto
thus ?thesis ..
qed

47.4.1 Cardinality of the Powerset

lemma card-UNIV-bool [simp]: card (UNIV :: bool set) = 2
  unfolding UNIV-bool by simp

lemma card-Pow: finite A =⇒ card (Pow A) = 2 ^ card A
proof (induct rule: finite-induct)
case empty
show ?case by simp
next
case (insert x A)
from "x ∉ A" have disjoint: Pow A ∩ insert x ' Pow A = { } by blast
from "x ∉ A" have inj-on: inj-on (insert x) (Pow A)
  unfolding inj-on-def by auto

have card (Pow (insert x A)) = card (Pow A ∪ insert x ' Pow A)
  by (simp only: Pow-insert)
also have ... = card (Pow A) + card (insert x ' Pow A)
  by (rule card-Un-disjoint) (use finite A disjoint in simp-all)
also from inj-on have card (insert x ' Pow A) = card (Pow A)
  by (rule card-image)
also have ... + ... = 2 * ... by (simp add: mult-2)
also from insert(3) have ... = 2 ^ Suc (card A) by simp
also from insert(1,2) have Suc (card A) = card (insert x A)
  by (rule card-insert-disjoint [symmetric])
finally show ?case .
qed
47.5 Code generator tweak

code-identifier

code-module Power → (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

48 Big sum and product over finite (non-empty) sets

theory Groups-Big
imports Power Equiv-Relations
begin

48.1 Generic monoid operation over a set

locale comm-monoid-set = comm-monoid
begin

48.1.1 Standard sum or product indexed by a finite set

interpretation comp-fun-commute f
by standard (simp add: fun-eq-iff left-commute)

interpretation comp?: comp-fun-commute f ∘ g
by (fact comp-comp-fun-commute)

definition F :: ('b ⇒ 'a) ⇒ 'b set ⇒ 'a
where eq-fold: F g A = Finite-Set.fold (f ∘ g) 1 A

lemma infinite [simp]: ¬ finite A ⇒ F g A = 1
by (simp add: eq-fold)

lemma empty [simp]: F g {} = 1
by (simp add: eq-fold)

lemma insert [simp]: finite A ⇒ x ∉ A ⇒ F g (insert x A) = g x ∗ F g A
by (simp add: eq-fold)

lemma remove:
  assumes finite A and x ∈ A
  shows F g A = g x ∗ F g (A − {x})
proof –
  from ⟨x ∈ A⟩ obtain B where B: A = insert x B and x ∉ B
  by (auto dest: mk-disjoint-insert)
  moreover from ⟨finite A: B have finite B by simp
ultimately show ?thesis by simp
qed
lemma insert-remove: finite A \implies F g (insert x A) = g x \ast F g (A \setminus \{x\})
by (cases x \in A) (simp-all add: remove insert-absorb)

lemma insert-if: finite A \implies F g (insert x A) = (if x \in A then F g A else g x \ast F g A)
by (cases x \in A) (simp-all add: insert-absorb)

lemma neutral: \forall x \in A. g x = 1 \implies F g A = 1
by (induct A rule: infinite-finite-induct) simp-all

lemma neutral-const [simp]: F (\lambda -. 1) A = 1
by (simp add: neutral)

lemma union-inter:
assumes finite A and finite B
shows F g (A \cup B) \ast F g (A \cap B) = F g A \ast F g B
— The reversed orientation looks more natural, but LOOPS as a simprule!
using assms
proof (induct A)
case empty
then show ?case by simp
next
case (insert x A)
then show ?case
  by (auto simp: insert-absorb Int-insert-left commute[of - g x] assoc left-commute)
qed

corollary union-inter-neutral:
assumes finite A and finite B
  and \forall x \in A \cap B. g x = 1
shows F g (A \cup B) = F g A \ast F g B
using assms by (simp add: union-inter [symmetric] neutral)

corollary union-disjoint:
assumes finite A and finite B
assumes A \cap B = {}
shows F g (A \cup B) = F g A \ast F g B
using assms by (simp add: union-inter-neutral)

lemma union-diff2:
assumes finite A and finite B
shows F g (A \cup B) = F g (A \setminus B) \ast F g (B \setminus A) \ast F g (A \cap B)
proof —
  have A \cup B = A \setminus B \cup (B \setminus A) \cup A \cap B
    by auto
  with assms show ?thesis
    by simp (subst union-disjoint, auto)+
qed
THEORY “Groups-Big”

lemma subset-diff:
assumes $B \subseteq A$ and finite $A$
shows $F \circ g A = F \circ g (A - B) \ast F \circ g B$

proof –
from assms have finite $(A - B)$ by auto
moreover from assms have finite $B$ by (rule finite-subset)
moreover from assms have $(A - B) \cap B = \emptyset$ by auto
ultimately have $F \circ g (A - B \cup B) = F \circ g (A - B) \ast F \circ g B$ by (rule union-disjoint)
moreover from assms have $A \cup B = A$ by auto
ultimately show \( ?thesis \) by simp
qed

lemma Int-Diff:
assumes finite $A$
shows $F \circ g A = F \circ g (A \cap B) \ast F \circ g (A - B)$
by (subst subset-diff [where $B = A - B$]) (auto simp: Diff-Diff-Int assms)

lemma setdiff-irrelevant:
assumes finite $A$
shows $F \circ g (A - \{x. g x = z\}) = F \circ g A$
using assms by (induct $A$ rule: infinite-finite-induct)
next
case empty
then show \( ?case \) by simp
next
case (insert a $A$)
then show \( ?case \) by fastforce
qed
with that show thesis by blast
qed

lemma reindex:
assumes inj-on $h$ $A$
shows $F \circ g (h \circ A) = F \circ (g \circ h) A$

proof (cases finite $A$)
case True
with assms show \( ?thesis \)
  by (simp add: eq-fold fold-image comp-assoc)
next
case False
with 

assms have \( \neg \text{finite} \ (h \cdot A) \) by (blast dest: finite-imageD)

with False show \(?thesis\) by simp

qed

lemma cong [fundef-cong]:
assumes \( A = B \)
assumes \( g-h: \ \forall x. \ x \in B \implies g \ x = h \ x \)
shows \( F \ g \ A = F \ h \ B \)
using \( g-h \) unfolding \( A = B \)
by (induct B rule: infinite-finite-induct) auto

lemma cong-simp [cong]:
[ \( A = B \); \( \forall x. \ x \in B \implies g \ x = h \ x \) ] \( \implies F (\lambda x. \ g \ x) \ A = F (\lambda x. \ h \ x) \ B \)
by (rule cong) (simp-all add: simp-implies-def)

lemma reindex-cong:
assumes inj-on \( l \) \( B \)
assumes \( A = l \cdot B \)
assumes \( \forall x. \ x \in B \implies g (l \ x) = h \ x \)
shows \( F \ g \ A = F \ h \ B \)
using assms by (simp add: reindex)

lemma image-eq:
assumes inj-on \( g \) \( A \)
shows \( F (\lambda x. \ g \ x) (g \cdot A) = F g \ A \)
using assms reindex-cong by fastforce

lemma UNION-disjoint:
assumes finite \( I \) and \( \forall i \in I. \ \text{finite} \ (A \ i) \)
and \( \forall i \in I. \ \forall j \in I. \ i \neq j \implies A \ i \cap A \ j = \{\} \)
shows \( F \ g \ (\bigcup (A \cdot I)) = F (\lambda x. \ F g (A \ x)) \ I \)
using assms
proof (induction rule: finite-induct)
case \( \text{insert} \ i \ I \)
then have \( \forall j \in I. \ j \neq i \)
by blast
with \( \text{insert.prems} \) have \( A \ i \cap \bigcup (A \cdot I) = \{\} \)
by blast
with \( \text{insert show} \) \(?case\)
by (simp add: union-disjoint)
qed auto

lemma Union-disjoint:
assumes \( \forall A \in C. \ \text{finite} \ A \ \forall A \in C. \ \forall B \in C. \ A \neq B \implies A \cap B = \{\} \)
shows \( F g (\bigcup C) = (F \circ F) g C \)
proof (cases \text{finite} \ C)
case True
from UNION-disjoint \( [OF \ this \ assms] \) show \(?thesis\) by simp
next
case False
then show ?thesis by (auto dest: finite-UnionD intro: infinite)
qed

lemma distrib: 
\[ F(\lambda x. g x \cdot h x) A = F g A \cdot F h A \]
by (induct A rule: infinite-finite-induct) (simp-all add: assoc commute left-commute)

lemma Sigma:
assumes finite A \(\forall x \in A. \) finite \((B x)\)
shows \[ F(\lambda x. F(g x) (B x)) A = F(\lambda x. F(\lambda(x, y). g x y)) (\bigcup y \in B x. \{(x, y)\}) \]
proof (subst UNION-disjoint)
  show \[ F(g x) (B x) = F(\lambda(x, y). g x y) (\bigcup y \in B x. \{(x, y)\}) \]
  if \(x \in A\) for \(x\)
  using that assms by (simp add: UNION-disjoint)
qed

lemma related:
assumes Re: \(R 1 1\)
and Rop: \(\forall x1 y1 x2 y2. R x1 x2 \land R y1 y2 \rightarrow R(x1 \cdot y1) (x2 \cdot y2)\)
and fin: finite \(S\)
and R-h-g: \(\forall x \in S. R(h x) (g x)\)
shows \(R(F h S) (F g S)\)
using fin by (rule finite-subset-induct) (use assms in auto)

lemma mono-neutral-cong-left:
assumes finite \(T\)
and \(S \subseteq T\)
and \(\forall i \in T - S. h i = 1\)
and \(\forall x. x \in S \implies g x = h x\)
shows \(F g S = F h T\)
proof -
  have eq: \(T = S \cup (T - S)\) using \(S \subseteq T\) by blast
  have d: \(S \cap (T - S) = \{\}\) using \(S \subseteq T\) by blast
  from \(\langle S, T, S \subseteq T \rangle\) have f: finite \(S\) finite \((T - S)\)
  by (auto intro: finite-subset)
  show ?thesis using assms(4)
  by (simp add: union-disjoint \[OF f d, unfolded eq \[symmetric]\] neutral \[OF assms(3)]\)
qed

lemma mono-neutral-cong-right:
finite \(T\) \(\implies S \subseteq T \implies \forall i \in T - S. g i = 1 \implies (\forall x. x \in S \implies g x = h x) \implies F g T = F h S\)
by (auto intro: mono-neutral-cong-left [symmetric])

lemma mono-neutral-left: finite \( T \implies S \subseteq T \implies \forall i \in T - S. \ g \ i = 1 \implies F \ g \ S = F \ g \ T \)
by (blast intro: mono-neutral-cong-left)

lemma mono-neutral-right: finite \( T \implies S \subseteq T \implies \forall i \in T - S. \ g \ i = 1 \implies F \ g \ T = F \ g \ S \)
by (blast intro: mono-neutral-left [symmetric])

lemma mono-neutral-cong:
assumes [simp]: finite \( T \) finite \( S \)
and *: \( \forall i. \ i \in T - S \implies h \ i = 1 \) \( \forall i. \ i \in S - T \implies g \ i = 1 \)
shows \( F \ g \ S = F \ h \ T \)
proof -
have \( F \ g \ S = F \ g \ (S \cap T) \)
  by (rule mono-neutral-right) (auto intro: *)
also have . . . = \( F \ h \ (S \cap T) \) using refl gh by (rule cong)
also have . . . = \( F \ h \ T \)
  by (rule mono-neutral-left) (auto intro: *)
finally show ?thesis .
qed

lemma reindex-bij-betw: bij-betw \( h \ S \ T \implies F \ (\lambda x. \ g \ (h \ x)) \) \( S = F \ g \ T \)
by (auto simp: bij-betw-def reindex)

lemma reindex-bij-witness:
assumes witness:
\( \forall a. \ a \in S \implies i \ (j \ a) = a \)
\( \forall a. \ a \in S \implies j \ a \in T \)
\( \forall b. \ b \in T \implies j \ (i \ b) = b \)
\( \forall b. \ b \in T \implies i \ b \in S \)
assumes eq:
\( \forall a. \ a \in S \implies h \ (j \ a) = g \ a \)
shows \( F \ g \ S = F \ h \ T \)
proof -
have \( bij-betw \ j \ S \ T \)
  using bij-betw-byWitness [where \( A=S \) and \( f=j \) and \( f'=i \) and \( A'=T \)] witness
by auto
moreover have \( F \ g \ S = F \ (\lambda x. \ h \ (j \ x)) \) \( S \)
  by (intro cong) (auto simp: eq)
ultimately show ?thesis
  by (simp add: reindex-bij-betw)
qed

lemma reindex-bij-betw-not-neutral:
assumes fin: finite \( S' \) finite \( T' \)
assumes bij: bij-betw \( h \ (S - S') \) \( T - T' \)
assumes \( nn \):
\[
\forall a. a \in S' \implies g (h a) = z
\]
\[
\forall b. b \in T' \implies g b = z
\]
shows \( F \) \((\lambda x. g (h x))\) \( S = F g T \)
proof –
have [simp]: finite \( S \leftarrow\) finite \( T \)
using bij-betw-finite[OF bij] fin by auto
show ?thesis
proof (cases finite \( S \))
case True
with \( nn \) have \( F \) \((\lambda x. g (h x))\) \( S = F g T \)
by (intro mono-neutral-cong-right) auto
also have \( \ldots = F g (T - T') \)
using bij by (rule reindex-bij-betw)
also have \( \ldots = F g T \)
using nn \( \langle \text{finite } S \rangle \) by (intro mono-neutral-cong-left) auto
finally show ?thesis .
next
case False
then show ?thesis by simp
qed

lemma reindex-nontrivial:
assumes finite \( A \)
and \( nz \): \( \forall x y. x \in A \implies y \in A \implies x \neq y \implies h x = h y \implies g (h x) = 1 \)
shows \( F g (\lambda ' A) = F (g \circ h) A \)
proof (subst reindex-bij-betw-not-neutral [symmetric])
show bij-betw \((A - \{ x \in A. (g \circ h) x = 1 \})\) \((\lambda ' A - \lambda ' (x \in A. (g \circ h) x = 1))\)
using nz by (auto intro!: inj-onI simp: bij-betw-def)
qed (use \( \langle \text{finite } A \rangle \) in auto)

lemma reindex-bij-witness-not-neutral:
assumes fin: finite \( S' \) finite \( T' \)
assumes witness:
\[
\forall a. a \in S - S' \implies i (j a) = a
\]
\[
\forall a. a \in S - S' \implies j a \in T - T'
\]
\[
\forall b. b \in T - T' \implies j (i b) = b
\]
\[
\forall b. b \in T - T' \implies i b \in S - S'
\]
assumes \( nn \):
\[
\forall a. a \in S' \implies g a = z
\]
\[
\forall b. b \in T' \implies h b = z
\]
assumes eq:
\[
\forall a. a \in S \implies h (j a) = g a
\]
shows \( F g S = F h T \)
proof –
have bij: bij-betw \((S - (S' \cap S))\) \((T - (T' \cap T))\)
using witness by (intro bij-betw-byWitness[where \( f' = i \)]) auto
have F-eq: F g S = F (λx. h (j x)) S
  by (intro cong) (auto simp: eq)
show ?thesis
  unfolding F-eq using fin nn eq
  by (intro reindex-bij-betw-not-neutral[OF - bij]) auto
qed

lemma delta-remove:
  assumes fS: finite S
  shows F (λk. if k = a then b k else c k) S = (if a ∈ S then b a ∗ F c (S−{a})
  else F c (S−{a})
proof
  let ?f = (λk. if k = a then b k else c k)
  show ?thesis
    proof (cases a ∈ S)
      case False
      then have ∀k∈S. ?f k = c k by simp
      with False show ?thesis by simp
    next
      case True
      let ?A = S−{a}
      let ?B = {a}
      from True have eq: S = ?A ∪ ?B by blast
      have dj: ?A ∩ ?B = {} by simp
      from fS have fAB: finite ?A finite ?B by auto
      have F ?f S = F ?f ?A ∗ F ?f ?B
        using union-disjoint [OF fAB dj, of ?f, unfolded eq [symmetric]] by simp
      with True show ?thesis
        using comm-monoid-set.remove comm-monoid-set-axioms fS by fastforce
    qed
qed

lemma delta [simp]:
  assumes fS: finite S
  shows F (λk. if k = a then b k else 1) S = (if a ∈ S then b a else 1)
  by (simp add: delta-remove [OF assms])

lemma delta' [simp]:
  assumes fin: finite S
  shows F (λk. if a = k then b k else 1) S = (if a ∈ S then b a else 1)
  using delta [OF fin, of a b, symmetric] by (auto intro: cong)

lemma If-cases:
  fixes P :: 'b ⇒ bool and g h :: 'b ⇒ 'a
  assumes fin: finite A
  shows F (λx. if P x then h x else g x) A = F h (A ∩ {x. P x}) ∗ F g (A ∩ − {x. P x})
proof
  have a: A = A ∩ {x. P x} ∪ A ∩ − {x. P x} (A ∩ {x. P x}) ∩ (A ∩ − {x. P x})
THEORY “Groups-Big”  

\begin{align*}
\text{let } \lambda x. \text{if } P x \text{ then } h x \text{ else } g x \\
\text{by (simp add: Sigma)}
\end{align*}

\begin{align*}
\text{lemma } \text{cartesian-product}' : \\
F g (A \times B) = F (\lambda x. F (\lambda y. g (x,y)) B) A
\end{align*}

\begin{align*}
\text{lemma } \text{inter-restrict}: \\
\text{assumes } \text{finite } A \\
\text{shows } F g (A \cap B) = F (\lambda x. \text{if } x \in B \text{ then } g x \text{ else } 1) A \\
\text{proof} \\
\text{let } \lambda x. \text{if } x \in A \cap B \text{ then } g x \text{ else } 1 \\
\text{have } \forall i \in A \cap B. (\text{if } i \in A \cap B \text{ then } g i \text{ else } 1) = 1 \text{ by simp} \\
\text{moreover have } A \cap B \subseteq A \text{ by blast} \\
\text{ultimately have } F ?g (A \cap B) = F ?g A \\
\text{using } F ?g (A \cap B) = F ?g A \\
\text{then show } \text{?thesis by simp} \end{align*}
lemma inter-filter:
  \[ \text{finite } A \implies F \{ x \in A \mid P x \} = F (\lambda x. \text{if } P x \text{ then } g x \text{ else } 1) \ A \]
by (simp add: inter-restrict [symmetric, of A \{ x \mid P x \} g, simplified mem-Collect-eq]
Int-def)

lemma Union-comp:
  assumes \( \forall A \in B. \text{finite } A \)
  and \( \bigwedge A1 A2 x. A1 \in B \implies A2 \in B \implies A1 \neq A2 \implies x \in A1 \implies x \in A2 \)
  \( \implies g x = 1 \)
  shows \( F \ g \ (\bigcup B) = (F \circ F) \ g \ B \)
  using assms
  proof (induct B rule: infinite-finite-induct)
  case (infinite A)
  then have \( \neg \text{finite } (\bigcup A) \)
  by (blast dest: finite-UnionD)
  with infinite show ?case by simp
  next
  case empty
  then show ?case by simp
  next
  case (insert A B)
  then have \( \text{finite } A \text{ finite } B \text{ finite } (\bigcup B) \ A \notin B \)
  and \( H: F \ g \ (\bigcup B) = (F \circ F) \ g \ B \)
  by auto
  then have \( F \ g \ (A \cup \bigcup B) = F \ g \ A \ast F \ g \ (\bigcup B) \)
  by (simp add: union-inter-neutral)
  with \( \text{finite } B; \ A \notin B \) show ?case
  by (simp add: H)
  qed

lemma swap: \( F (\lambda i. F (g i) \ B) A = F (\lambda i. F (\lambda j. g i j) A) B \)
unfolding cartesian-product
  by (rule reindex-bij-witness [where \( i = \lambda(i, j). (j, i) \) and \( j = \lambda(i, j). (j, i) \)]
auto

lemma swap-restrict:
  \( \text{finite } A \implies \text{finite } B \implies F (\lambda x. F (g x) \{ y. y \in B \land R x y \}) A = F (\lambda y. F (\lambda x. g x y) \{ x. x \in A \land R x y \}) B \)
  by (simp add: inter-filter) (rule swap)

lemma image-gen:
  assumes fin: \( \text{finite } S \)
  shows \( F \ h \ S = F (\lambda y. F \ h \ { x. x \in S \land g x = y } ) \ (g ' S) \)
  proof
  have \( \{ y. y \in g'S \land g x = y \} = \{ g x \} \) if \( x \in S \) for \( x \)
  using that by auto
  then have \( F \ h \ S = F (\lambda x. F (\lambda y. h x) \{ y. y \in g'S \land g x = y \}) S \)
  by simp
  also have \ldots = \ldots by simp
by (rule swap-restrict [OF fin finite-imageI [OF fin]])

finally show \( ?\text{thesis} \).

dqed

lemma group:

assumes \( \text{fS}: \text{finite } S \text{ and } \text{fT}: \text{finite } T \text{ and } \text{fST}: g ^{\ast} \subseteq T \)

shows \( F (\lambda y. F \ h \ \{x. \ x \in S \land g \ x = y\}) \ T = F \ h \ S \)

unfolding image-gen[\( \text{OF fS, of h g} \)]

by (auto intro: neutral mono-neutral-right[\( \text{OF fT fST} \)])

lemma Plus:

fixes \( A :: \text{'}b \text{ set and } B :: \text{'}c \text{ set} \)

assumes \( \text{fin: finite } A \text{ finite } B \)

shows \( F \ g \ (A <\leftrightarrow> B) = F \ (g \circ \text{Inl}) \ A \ast F \ (g \circ \text{Inr}) \ B \)

proof –

have \( A <\leftrightarrow> B = \text{Inl} ^{\ast} A \cup \text{Inr} ^{\ast} B \text{ by auto} \)

moreover from \( \text{fin have } \text{finite } (\text{Inl} ^{\ast} A) \text{ finite } (\text{Inr} ^{\ast} B) \text{ by auto} \)

moreover have \( \text{Inl} ^{\ast} A \cap \text{Inr} ^{\ast} B = \{\} \text{ by auto} \)

moreover have \( \text{inj-on } \text{Inl } A \text{ inj-on } \text{Inr } B \text{ by } (\text{auto intro: inj-on}) \)

ultimately show \( ?\text{thesis} \)

using \( \text{fin by (simp add: union-disjoint reindex)} \)

dqed

lemma same-carrier:

assumes \( \text{finite } C \)

assumes \( \text{subset: } A \subseteq C \ B \subseteq C \)

assumes \( \text{trivial: } \forall a. \ a \in C - A \implies g \ a = 1 \ \forall b. \ b \in C - B \implies h \ b = 1 \)

shows \( F \ g \ A = F \ h \ B \iff F \ g \ C = F \ h \ C \)

proof –

have \( \text{finite } A \text{ and } \text{finite } B \text{ and } \text{finite } (C - A) \text{ and } \text{finite } (C - B) \)

using \( \langle\text{finite } C\rangle \text{ subset by (auto elim: finite-subset)} \)

from \( \text{subset have [simp]: } A - (C - A) = A \text{ by auto} \)

from \( \text{subset have [simp]: } B - (C - B) = B \text{ by auto} \)

from \( \text{subset have } C = A \cup (C - A) \text{ by auto} \)

then have \( F \ g \ C = F \ g \ (A \cup (C - A)) \text{ by simp} \)

also have \( \ldots = F \ g \ (A - (C - A)) \ast F \ g \ (C - A - A) \ast F \ g \ (A \cap (C - A)) \)

using \( \langle\text{finite } A\rangle \text{ } \langle\text{finite } (C - A)\rangle \text{ by (simp only: union-diff2)} \)

finally have \( *: F \ g \ C = F \ g \ A \text{ using trivial by simp} \)

from \( \text{subset have } C = B \cup (C - B) \text{ by auto} \)

then have \( F \ h \ C = F \ h \ (B \cup (C - B)) \text{ by simp} \)

also have \( \ldots = F \ h \ (B - (C - B)) \ast F \ h \ (C - B - B) \ast F \ h \ (B \cap (C - B)) \)

using \( \langle\text{finite } B\rangle \text{ } \langle\text{finite } (C - B)\rangle \text{ by (simp only: union-diff2)} \)

finally have \( F \ h \ C = F \ h \ B \)

using trivial by simp

with \( * \text{ show } ?\text{thesis by simp} \)

dqed

lemma same-carrier1:

assumes \( \text{finite } C \)
assumes subset: \( A \subseteq C \) \( B \subseteq C \)
assumes trivial: \( \forall a. a \in C - A \implies g a = 1 \) \( \land b. b \in C - B \implies h b = 1 \)
assumes \( F g C = F h C \)
shows \( F g A = F h B \)
using assms same-carrier \([\text{of } C A B]\) by simp

lemma eq-general:
assumes \( B: \forall y. y \in B \implies \exists ! x. x \in A \land h x = y \) and \( A: \forall x. x \in A \implies h x \in B \land \gamma(h x) = \varphi x \)
shows \( F \varphi A = F \gamma B \)
proof –
  have eq: \( B = h \cdot A \)
    by (auto dest: assms)
  have h: inj-on h A
    using assms by (blast intro: inj-onI)
  have \( F \varphi A = F (\gamma \circ h) A \)
    using A by auto
  also have \( \ldots = F \gamma B \)
    by (simp add: eq reindex h)
  finally show \(?thesis\).
qed

lemma eq-general-inverses:
assumes \( B: \forall y. y \in B \implies k y \in A \land h(k y) = y \) and \( A: \forall x. x \in A \implies h x \in B \land k(h x) = x \land \gamma(h x) = \varphi x \)
shows \( F \varphi A = F \gamma B \)
proof (rule eq-general [where \( \gamma \cdot h \)])
  have \( \gamma B = \{ i \in I. x i \neq 1 \} \cup \{ i \in I. y i \neq 1 \} \)
    using left-neutral by force+
finally show \(?thesis\).

48.1.2 HOL Light variant: sum/product indexed by the non-neutral subset

NB only a subset of the properties above are proved

definition \( G :: \{ a \Rightarrow b \Rightarrow \text{set} \} \Rightarrow \text{a} \)
  where \( G p I \equiv \text{if finite } \{ x \in I. p x \neq 1 \} \text{ then } F p \{ x \in I. p x \neq 1 \} \text{ else } 1 \)

lemma finite-Collect-op:
  shows \([\text{finite } \{ i \in I. x i \neq 1 \}; \text{finite } \{ i \in I. y i \neq 1 \}] \implies \text{finite } \{ i \in I. x i \ast y i \}\neq 1\]
  apply (rule finite-subset [where \( B = \{ i \in I. x i \neq 1 \} \cup \{ i \in I. y i \neq 1 \}\}])
  using left-neutral by force+

lemma empty'[simp]: \( G p \{ \} = 1 \)
  by (auto simp: G-def)

lemma eq-sum [simp]: \( \text{finite } I \implies G p I = F p I \)
  by (auto simp: G-def intro: mono-neutral-cong-left)

lemma insert'[simp]:
  assumes finite \( \{ x \in I. p x \neq 1 \} \)
shows \( G \ p \ (\text{insert} \ i \ I) = (\text{if} \ i \in I \ \text{then} \ G \ p \ I \ \text{else} \ p \ i \ast G \ p \ I) \)

proof –
  have \( \{ x. \ x = i \land p \ x \neq 1 \ \lor \ x \in I \land p \ x \neq 1 \} = (\text{if} \ p \ i = 1 \ \text{then} \ \{ x \in I. \ p \ x \neq 1 \}) \)
    by auto
  then show \(?thesis\)
    using \(\text{assms by (simp add: G-def conj-disj-distribR insert-absorb)}\)
  qed

lemma \(\text{distrib-triv}'\):
  assumes \(\text{finite} \ I\)
  shows \(G \ (\lambda i. \ g \ i \ast h \ i) \ I = G \ g \ I \ast G \ h \ I\)
  by \((\text{simp add: assms local.distrib})\)

lemma \(\text{non-neutral}'\): \(G \ g \ \{ x \in I. \ g \ x \neq 1 \} = G \ g \ I\)
  by \((\text{simp add: G-def})\)

lemma \(\text{distrib}''\):
  assumes \(\text{finite} \ \{ x \in I. \ g \ x \neq 1 \} \ \text{finite} \ \{ x \in I. \ h \ x \neq 1 \}\)
  shows \(G \ (\lambda i. \ g \ i \ast h \ i) \ I = G \ g \ I \ast G \ h \ I\)
  proof –
    have \(a \ast a \neq a \Longrightarrow a \neq 1\) \(\text{for} \ a\)
    by \(\text{auto}\)
    then have \(G \ (\lambda i. \ g \ i \ast h \ i) \ I = G \ (\lambda i. \ g \ i \ast h \ i) \ (\{ i \in I. \ g \ i \neq 1 \} \cup \{ i \in I. \ h \ i \neq 1 \})\)
      using \(\text{assms by (force simp: G-def finite-Collect-op intro!: mono-neutral-cong)}\)
    also have \(\ldots = G \ g \ I \ast G \ h \ I\)
  proof –
    have \(F \ g \ (\{ i \in I. \ g \ i \neq 1 \} \cup \{ i \in I. \ h \ i \neq 1 \}) = G \ g \ I\)
      \(F \ h \ (\{ i \in I. \ g \ i \neq 1 \} \cup \{ i \in I. \ h \ i \neq 1 \}) = G \ h \ I\)
    by \(\text{(auto simp: G-def assms intro: mono-neutral-right)}\)
    then show \(?thesis\)
      using \(\text{assms by (simp add: distrib)}\)
    qed
  finally show \(?thesis\).
  qed

lemma \(\text{cong}'\):
  assumes \(A = B\)
  assumes \(g-h: \forall x. \ x \in B \Longrightarrow g \ x = h \ x\)
  shows \(G \ g \ A = G \ h \ B\)
  using \(\text{assms by (auto simp: G-def cong: conj-cong intro: cong)}\)

lemma \(\text{mono-neutral-cong-left}'\):
  assumes \(S \subseteq T\)
    and \(\forall i. \ i \in T - S \Longrightarrow h \ i = 1\)
    and \(\forall x. \ x \in S \Longrightarrow g \ x = h \ x\)
  shows \(G \ g \ S = G \ h \ T\)
proof

have \(*\): \{ \( x \in S. \, g \, x \neq 1 \} = \{ \( x \in T. \, h \, x \neq 1 \) \}
  using assms by\ ((metis DiffI subset-eq)
then have finite \{ \( x \in S. \, g \, x \neq 1 \) \} = finite \{ \( x \in T. \, h \, x \neq 1 \) \}
  by simp
then show \(*\)thesis
  using assms by\ (auto simp add: G-def \( \ast \) intro: cong)
qed

lemma mono-neutral-cong-right':
  \( S \subseteq T \implies \forall i \in T - S. \, g \, i = 1 \implies ( \forall x \in S. \, g \, x = h \, x) \implies G \, g \, T = G \, h \, S \)
  by\ (auto intro!: mono-neutral-cong-left' [symmetric])

lemma mono-neutral-left':
  \( S \subseteq T \implies \forall i \in T - S. \, g \, i = 1 \implies G \, g \, S = G \, g \, T \)
  by\ (blast intro: mono-neutral-cong-left')

lemma mono-neutral-right':
  \( S \subseteq T \implies \forall i \in T - S. \, g \, i = 1 \implies G \, g \, T = G \, g \, S \)
  by\ (blast intro!: mono-neutral-left' [symmetric])

end

48.2 Generalized summation over a set

context comm-monoid-add
begin

sublocale sum: comm-monoid-set plus 0
  defines sum = sum.\( F \) and sum' = sum.\( G \) ..

abbreviation Sum \((\sum)\)
  where \( \sum \equiv \) sum \((\lambda x. \, x)\)

end

Now: lots of fancy syntax. First, \( \sum (\lambda x. \, e) \) \( A \) is written \( \sum x \in A. \, e \).

syntax\ (ASCII)
\(-sum::\pttrn \Rightarrow 'a \Rightarrow 'b::\text{comm-monoid-add} \ ((3SUM \ (-|\)-)/ -) \ [0, 51, 10] 10)\)
syntax\ 
\(-qsum::\pttrn \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a::\text{comm-monoid-add} \ ((2\sum (-|\)-)/ -) \ [0, 51, 10] 10)\)
translations — Beware of argument permutation!
\( \sum i \in A. \, b = \) CONST sum\ (\( \lambda i. \, b \) \( A \))

Instead of \( \sum x \in \{ x. \, P \}. \, e \) we introduce the shorter \( \sum x | P. \, e \).

syntax\ (ASCII)
\(-qsum::\pttrn \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a::\text{comm-monoid-add} \ ((3SUM \ -|\ -)/ -) \ [0, 0, 10] 10)\)
syntax
48.2.1 Properties in more restricted classes of structures

**lemma** sum-Un:

finite \( A \) \( \Rightarrow \) finite \( B \) \( \Rightarrow \) \( \sum f (A \cup B) = \sum f A + \sum f B - \sum f (A \cap B) \)

for \( f :: b \Rightarrow a :: \text{ab-group-add} \)

by (subst \( \text{sum.union-inter [symmetric]} \)) (auto simp add: algebra-simps)

**lemma** sum-Un2:

assumes finite \( (A \cup B) \)

shows \( \sum f (A \cup B) = \sum f (A - B) + \sum f (B - A) + \sum f (A \cap B) \)

proof

have \( A \cup B = A - B \cup (B - A) \cup A \cap B \)

by auto

with assms show \(?thesis\)

by simp (subst \( \text{sum.union-disjoint, auto} \))

qed

**lemma** sum-diff:

fixes \( f :: b \Rightarrow a :: \text{ab-group-add} \)

assumes finite \( A \) \( B \subseteq A \)

shows \( \sum f (A - B) = \sum f A - \sum f B \)

using \( \text{sum.subset-diff [of B A f]} \) assms by simp

**lemma** sum-diff1:

fixes \( f :: b \Rightarrow a :: \text{ab-group-add} \)

assumes finite \( A \)
shows $\sum f (A - \{a\}) = (\text{if } a \in A \text{ then } \sum f A - f a \text{ else } \sum f A)$

using assms by (simp add: sum-diff)

lemma sum-diff1′-aux:
fixes f :: 'a ⇒ 'b::ab-group-add
assumes finite F {i ∈ I. f i ≠ 0} ⊆ F
shows $\sum f (I - \{i\}) = (\text{if } i \in I \text{ then } \sum f I - f i \text{ else } \sum f I)$
using assms proof
induct
  case (insert x F)
  have 1: finite {x ∈ I. f x ≠ 0} =⇒ finite {x ∈ I. x ≠ i ∧ f x ≠ 0}
    by (erule rev-finite-subset) auto
  have 2: finite {x ∈ I. x ≠ i ∧ f x ≠ 0} =⇒ finite {x ∈ I. f x ≠ 0}
    apply (erule rev-finite-subset) auto
  show ?case
    using insert sum-diff1′ [of {i ∈ I. f i ≠ 0} f i] by (auto simp: sum.G-def 1 2 3 set-diff-eq conj-ac)
qed (simp add: sum.G-def)

lemma sum-diff1′:
fixes f :: 'a ⇒ 'b::ab-group-add
assumes finite {i ∈ I. i ≠ 0}
shows $\sum f (I - \{i\}) = (\text{if } i \in I \text{ then } \sum f I - f i \text{ else } \sum f I)$
by (rule sum-diff1′-aux [OF assms order-refl])

lemma (in ordered-comm-monoid-add) sum-mono:
$(\forall i. \in K \Longrightarrow f i \leq g i) \Longrightarrow (\sum i\in K. f i) \leq (\sum i\in K. g i)$
by (induct K rule: infinite-finite-induct) (use add-mono in auto)

lemma (in ordered-cancel-comm-monoid-add) sum-strict-mono-strong:
assumes finite A a ∈ A f a < g a
and $\forall x. x \in A \Longrightarrow f x \leq g x$
shows $\sum f A < \sum g A$
proof
  have $\sum f A = f a + \sum f (A - \{a\})$
    by (simp add: assms sum.remove)
  also have $\ldots \leq f a + \sum g (A - \{a\})$
    using assms by (meson DiffD1 add-left-mono sum-mono)
  also have $\ldots < g a + \sum g (A - \{a\})$
    using assms add-less-le-mono by blast
  also have $\ldots = \sum g A$
    using assms by (intro sum.remove [symmetric])
  finally show ?thesis.
qed

lemma (in strict-ordered-comm-monoid-add) sum-strict-mono:
assumes \( \text{finite } A \neq \{\} \) and \( \forall x \cdot x \in A \implies f x < g x \)
shows \( \sum f A < \sum g A \)
using assms
proof (induct rule: finite-ne-induct)
case singleton
then show \(?case\) by simp
next
case insert
then show \(?case\) by (auto simp: add-strict-mono)
qed

lemma sum-strict-mono-ex1:
fixes \( f, g :: 'i \Rightarrow 'a::ordered-cancel-comm-monoid-add \)
assumes \( \text{finite } A \)
and \( \forall x \cdot x \in A \implies f x \leq g x \)
and \( \exists a \in A. f a < g a \)
shows \( \sum f A < \sum g A \)
proof
  from assms (3) obtain \( a \) where \( a \in A \) \( f a < g a \) by blast
  have \( \sum f A = \sum f ((A - \{a\}) \cup \{a\}) \)
    by (simp add: insert-absorb[OF \( a \in A \)])
  also have \( \ldots = \sum f (A - \{a\}) + \sum f \{a\} \)
    using \( \text{finite } A \) by (subst sum.union_disjoint auto)
  also have \( \sum f (A - \{a\}) \leq \sum g (A - \{a\}) \)
    by (rule sum_mono) (simp add: assms (2))
  also from \( a \) have \( \sum f \{a\} < \sum g \{a\} \) by simp
  also have \( \sum g (A - \{a\}) + \sum g \{a\} = \sum g ((A - \{a\}) \cup \{a\}) \)
    using \( \text{finite } A \) by (subst sum.union_disjoint[symmetric]) auto
  also have \( \ldots = \sum g A \) by (simp add: insert_absorb[OF \( a \in A \)])
  finally show \(?thesis\)
    by (auto simp add: add_right_mono add_strict_left_mono)
qed

lemma sum-mono-inv:
fixes \( f, g :: 'i \Rightarrow 'a::ordered-cancel-comm-monoid-add \)
assumes \( \text{eq} : \sum f I = \sum g I \)
assumes \( \text{le} : \forall i. i \in I \implies f i \leq g i \)
assumes \( \text{i} : i \in I \)
assumes \( \text{I} : \text{finite } I \)
shows \( f i = g i \)
proof (rule ccontr)
  assume \( \neg \text{thesis} \)
  with \( \text{le}[OF I] \) have \( f i < g i \) by simp
  with \( \text{i} \) have \( \exists i \in I. f i < g i \) ..
  from sum-strict-mono-ex1[OF I - this] have \( \sum f I < \sum g I \)
    by blast
  with eq show \( \text{False} \) by simp
qed
**lemma** member-le-sum:

**fixes** \( f : \Rightarrow 'b::{semiring_1, ordered-comm-monoid-add} \)**

**assumes** \( i \in A \) and le: \( \forall x. x \in A - \{i\} \rightarrow 0 \leq f x \) and finite A

**shows** \( f i \leq \text{sum } f A \)

**proof** –

have \( f i \leq \text{sum } f (A \cap \{i\}) \) by (simp add: assms)

also have \( \ldots = (\sum_{x \in A. \text{ if } x \in \{i\} \text{ then } f x \text{ else } 0} \) using assms sum.inter-restrict by blast

also have \( \ldots \leq \text{sum } f A \) apply (rule sum-mono) apply (auto simp: le) done

finally show \( \text{?thesis} \).

**qed**

**lemma** sum-negf: \( (\sum_{x \in A. - f x}) = - (\sum_{x \in A. f x}) \)

**for** \( f : 'b \Rightarrow 'a::{ab-group-add} \)

by (induct A rule: infinite-finite-induct) auto

**lemma** sum-subtractf: \( (\sum_{x \in A. f x - g x}) = (\sum_{x \in A. f x}) - (\sum_{x \in A. g x}) \)

**for** \( f g : 'b \Rightarrow 'a::{ab-group-add} \)

using sum.distrib [of f - g A] by (simp add: sum-negf)

**lemma** sum-subtractf-nat:

\( (\forall x. x \in A \rightarrow g x \leq f x) \rightarrow (\sum_{x \in A. f x - g x}) = (\sum_{x \in A. f x}) - (\sum_{x \in A. g x}) \)

**for** \( f g : 'a \Rightarrow 'a::nat \)

by (induct A rule: infinite-finite-induct) (auto simp: sum-mono)

**context** ordered-comm-monoid-add

**begin**

**lemma** sum-nonneg: \( (\forall x. x \in A \rightarrow 0 \leq f x) \rightarrow 0 \leq \text{sum } f A \)

**proof** (induct A rule: infinite-finite-induct)

case infinite
then show \( \text{?case by simp} \)

next
case empty
then show \( \text{?case by simp} \)

next
case (insert x F)
then have \( 0 + 0 \leq f x + \text{sum } f F \) by (blast intro: add-mono) with insert show \( \text{?case by simp} \)

**qed**
lemma sum-nonpos: \( \forall x. x \in A \implies f x \leq 0 \) \implies \text{sum} f A \leq 0

proof (induct A rule: infinite-finite-induct)
  - case infinite
    then show ?case by simp
  next
    case empty
    then show ?case by simp
  next
    case (insert x F)
    then have \( f x + \text{sum} f F \leq 0 + 0 \) by (blast intro: add-mono)
    with insert show ?case by simp
  qed

lemma sum-nonneg-eq-0-iff:
  \( \text{finite} A \implies (\forall x \in A \implies f x \leq 0) \implies \text{sum} f A = 0 \iff (\forall x \in A. f x = 0) \)
  by (induct set: finite) (simp-all add: add-nonneg-eq-0-iff sum-nonneg)

lemma sum-nonneg-0:
  \( \text{finite} s \implies (\forall i. i \in s \implies f i \geq 0) \implies (\sum i \in s. f i) = 0 \implies i \in s \implies f i = 0 \)
  by (simp add: sum-nonneg-eq-0-iff)

lemma sum-nonneg-bound:
  assumes \( \text{finite} s \land \forall y \in t. 0 \leq g y \land (\forall x \in s. \exists y \in t. i y = x \land f x \leq g y) \)
  shows \( \text{sum} f A \leq \text{sum} f (B - A) \)
  proof -
    have \( \text{sum} f A \leq \text{sum} f A + \text{sum} f (B - A) \)
      by (auto intro: add-increasing2 [OF sum-nonneg] nn)
    also have \( \ldots = \text{sum} f (A \cup (B - A)) \)
      by (simp add: sum.union_disjoint del: Un-Diff-cancel)
    also have \( \text{sub have} A \cup (B - A) = B \) by blast
    finally show ?thesis .
  qed

lemma sum-le-included:
  assumes \( \text{finite} s \land \text{finite} t \)
  and \( \forall y \in t. 0 \leq g y \land (\forall x \in s. \exists y \in t. i y = x \land f x \leq g y) \)
  shows \( \forall x \in s. \exists y \in t. i y = x \land f x \leq g y \)
shows \( \sum f s \leq \sum g t \)

proof –

have \( \sum f s \leq \sum (\lambda y. g \{x. x \in t \land i x = y\}) s \)
proof (rule sum-mono)
fix \( y \)
assume \( y \in s \)
with \( \text{assms} \) obtain \( z \) where \( z \in t \) \( i z = y \)
using \( \text{order-trans[OF} \ ?A (i z) \ \text{sum} g \ \{z\} \ ?B (i z), \text{intro}] \)
by (auto intro!: sum-mono2)
qed
also have \ldots \leq \sum (\lambda y. g \{x. x \in t \land i x = y\}) (i ' t)
using \( \text{assms(2-4)} \) by (auto intro!: sum-mono2 sum-nonneg)
also have \ldots \leq \sum g t
using \( \text{assms} \) by (auto simp: sum_image_gen[symmetric])
finally show \( ?\text{thesis} \).
qed

draw

lemma (in canonically-ordered-monoid-add) sum-eq-0-iff [simp]:
finite \( F \implies (\sum f F = 0) = (\forall a \in F. f a = 0) \)
by (intro ballI sum-nonneg-eq-0-iff zero-le)

code context semiring-0
begin

lemma sum-distrib-left: \( r \times \sum f A = (\sum n \in A. r \times f n) \)
by (induct \( A \) rule: infinite-finite-induct) (simp-all add: algebra-simps)

lemma sum-distrib-right: \( \sum f A \times r = (\sum n \in A. f n \times r) \)
by (induct \( A \) rule: infinite-finite-induct) (simp-all add: algebra-simps)

draw

lemma sum-divide-distrib: \( \sum f A / r = (\sum n \in A. f n / r) \)
for \( r :: 'a::field \)
proof (induct \( A \) rule: infinite-finite-induct)
case infinite
then show \( ?\text{case} \) by simp
next
case empty
then show \( ?\text{case} \) by simp
next
case insert
then show \( ?\text{case} \) by (simp add: add-divide-distrib)
qed

lemma sum-abs[iff]: \( |\sum f A| \leq \sum (\lambda i. |f i|) A \)

for f :: 'a ⇒ 'b::ordered-ab-group-add-abs
proof (induct A rule: infinite-finite-induct)
  case infinite
  then show ?case by simp
next
  case empty
  then show ?case by simp
next
  case insert
  then show ?case by (auto intro: abs_triangle_ineq order_trans)
qed

lemma sum-abs-ge-zero[iff]: 0 ≤ sum (λ i. |f i|) A
for f :: 'a ⇒ 'b::ordered-ab-group-add-abs
by (simp add: sum_nonneg)

lemma abs-sum-abs[simp]: |∑ a∈A. |f a|| = (∑ a∈A. |f a|)
for f :: 'a ⇒ 'b::ordered-ab-group-add-abs
proof (induct A rule: infinite-finite-induct)
  case infinite
  then show ?case by simp
next
  case empty
  then show ?case by simp
next
  case insert
  then have |∑ a∈insert a A. |f a|| = |f a| + (∑ a∈A. |f a||) by simp
  also from insert have . . . = |∑ a∈A. |f a|| by simp
  also have . . . = |∑ a∈A. |f a|| by (simp del: abs_of_nonneg)
  also from insert have . . . = (∑ a∈insert a A. |f a|) by simp
  finally show ?case .
qed

lemma sum-product:
  fixes f :: 'a ⇒ 'b::semiring_0
  shows sum f A * sum g B = (∑ i∈A. ∑ j∈B. f i * g j)
  by (simp add: sum_distrib_left sum_distrib_right) (rule sum_swap)

lemma sum-mult-sum-if-inj:
  fixes f :: 'a ⇒ 'b::semiring_0
  shows inj-on (λ(a, b). f a * g b) (A × B) \implies
  sum f A * sum g B = sum id {f a * g b | a b. a ∈ A ∧ b ∈ B}
  by(auto simp: sum-product sum.cartesian_product_intro: sum.reindex_cong[symmetric])

lemma sum-SucD: sum f A = Suc n \implies ∃ a∈A. 0 < f a
  by (induct A rule: infinite-finite-induct) auto

lemma sum-eq-Suc0-iff:
  finite A \implies sum f A = Suc 0 \iff (∃ a∈A. f a = Suc 0 ∧ (∀ b∈A. a ≠ b → f a < f b)
\begin{verbatim}

THEORY "Groups-Big"

b = 0)
by (induct A rule: finite-induct) (auto simp add: add-is-1)
lemmas sum-eq-1-iff = sum-eq-Suc0-iff[simplified One-nat-def[symmetric]]

lemma sum-Un-nat:
finite A ⇒ finite B ⇒ sum f (A ∪ B) = sum f A + sum f B - sum f (A ∩ B)
for f :: 'a ⇒ nat
— For the natural numbers, we have subtraction.
by (subst sum.union_inter [symmetric]) (auto simp: algebra-simps)

lemma sum-diff1-nat: sum f (A - {a}) = (if a ∈ A then sum f A - f a else sum f A)
for f :: 'a ⇒ nat
proof (induct A rule: infinite-finite-induct)
  case infinite
  then show ?case by simp
next
  case empty
  then show ?case by simp
next
  case (insert x F)
  then show ?case
proof (cases a ∈ F)
    case True
    then have ∃ B. F = insert a B ∧ a /∈ B
    by (auto simp: mk-disjoint-insert)
    then show ?thesis using insert
    by (auto simp: insert-Diff-if)
  qed (auto)
qed

defined sum-diff-nat:
fixes f :: 'a ⇒ nat
assumes finite B and B ⊆ A
shows sum f (A - B) = sum f A - sum f B
using assms
proof induct
  case empty
  then show ?case by simp
next
  case (insert x F)
  note IH = ‹F ⊆ A ‚⇒ sum f (A - F) = sum f A - sum f F›
  from ‹x /∈ F› ‹insert x F ⊆ A› have x ∈ A - F by simp
  then have A: sum f ((A - F) - {x}) = sum f (A - F) - f x
  by (simp add: sum-diff1-nat)
  from ‹insert x F ⊆ A› have F ⊆ A by simp
  with IH have sum f (A - F) = sum f A - sum f F by simp
with A have B: sum f ((A - F) - {x}) = sum f A - sum f F - f x
\end{verbatim}
by simp
from ⟨x ∉ F⟩ have A − insert x F = (A − F) − {x} by auto
with B have C: sum f (A − insert x F) = sum f A − sum f F − f x
  by simp
from ⟨finite F⟩ ⟨x ∉ F⟩ have sum f (insert x F) = sum f F + f x
  by simp
with C have sum f (A − insert x F) = sum f A − sum f (insert x F)
  by simp
then show ?case by simp
qed

lemma sum-comp-morphism:
  h 0 = 0 ⇒ (∀x y. h (x + y) = h x + h y) ⇒ sum (h ∘ g) A = h (sum g A)
  by (induct A rule: infinite-finite-induct) simp-all

lemma (in comm-semiring-1) dvd-sum: (∀a ∈ A ⇒ d dvd f a) ⇒ d dvd sum f A
  by (induct A rule: infinite-finite-induct) simp-all

lemma (in ordered-comm-monoid-add) sum-pos:
  finite I ⇒ I ≠ {} ⇒ (∀i. i ∈ I ⇒ 0 < f i) ⇒ 0 < sum f I
  by (induct I rule: finite-ne-induct) (auto intro: add-pos-pos)

lemma (in ordered-comm-monoid-add) sum-pos2:
  assumes I: finite I i ∈ I 0 < f i \(∀i. i ∈ I \Rightarrow 0 ≤ f i\)
  shows 0 < sum f I
proof –
  have 0 < f i + sum f (I − {i})
    using assms by (intro add-pos-nonneg sum-nonneg) auto
  also have \ldots = sum f I
    using assms by (simp add: sum.remove)
  finally show ?thesis .
qed

lemma sum-strict-mono2:
  fixes f :: 'a ⇒ 'b::ordered-cancel-comm-monoid-add
  assumes finite B A ⊆ B b ∈ B − A f b > 0 and \(∀x. x ∈ B \Rightarrow f x ≥ 0\)
  shows sum f A < sum f B
proof –
  have B − A ≠ {}
    using assms(3) by blast
  have sum f (B − A) > 0
    by (rule sum-pos2) (use assms in auto)
  moreover have sum f B = sum f (B − A) + sum f A
    by (rule sum.subset-diff) (use assms in auto)
  ultimately show ?thesis
    using add-strict-increasing by auto
qed
lemma sum-cong-Suc:
  assumes 0 ⊈ A ∧ x. Suc x ∈ A ⊢ f (Suc x) = g (Suc x)
  shows sum f A = sum g A
proof (rule sum.cong)
  fix x
  assume x ∈ A
  with assms(1) show f x = g x
    by (cases x) (auto intro: assms(2))
qed simp-all

48.2.2 Cardinality as special case of sum

lemma card-eq-sum: card A = sum (λ x. 1) A
proof –
  have plus o (λ -. Suc 0) = (λ -. Suc)
    by (simp add: fun-eq-iff)
  then have Finite-Set.fold (plus o (λ -. Suc 0)) = Finite-Set.fold (λ -. Suc)
    by (rule arg-cong)
  then have Finite-Set.fold (plus o (λ -. Suc 0)) 0 A = Finite-Set.fold (λ -. Suc) 0 A
    by (blast intro: fun-cong)
  then show thesis
    by (simp add: card.eq-fold sum.eq-fold)
qed

context semiring-1
begin

lemma sum-constant [simp]:
  (∑ x ∈ A. y) = of-nat (card A) * y
by (induct A rule: infinite-finite-induct) (simp-all add: algebra-simps)

context
  fixes A
  assumes ‹finite A›
begin

lemma sum-of-bool-eq [simp]:
  (∑ x ∈ A. of-bool (P x)) = of-nat (card (A ∩ {x. P x})), if ‹finite A›
using ‹finite A› by induction simp-all

lemma sum-mult-of-bool-eq [simp]:
  (∑ x ∈ A. f x * of-bool (P x)) = (∑ x ∈ (A ∩ {x. P x}). f x)
by (rule sum.mono-neutral-cong) (use ‹finite A› in auto)

lemma sum-of-bool-mult-eq [simp]:
  (∑ x ∈ A. of-bool (P x) * f x) = (∑ x ∈ (A ∩ {x. P x}). f x)
by (rule sum.mono-neutral-cong) (use ‹finite A› in auto)
lemma sum-Suc: \( \text{sum} (\lambda x. \text{Suc}(f x)) A = \text{sum} f A + \text{card} A \)
using sum.distrib[of \( \lambda x. \text{Suc}(f x) \)] by simp

lemma sum-bounded-above:
fixes \( K :: 'a::{\text{semiring-1,ordered-comm-monoid-add}} \)
assumes le: \( \forall i. i \in A \Rightarrow f i \leq K \)
shows \( \text{sum} f A \leq \text{of-nat} (\text{card} A) \times K \)
proof (cases \text{finite} A)
case True
then show \( \neg thesis \)
using le sum-mono[where \( K=A \) and \( g = \lambda x. K \)] by simp
next
case False
then show \( \neg thesis \) by simp
qed

lemma sum-bounded-above-divide:
fixes \( K :: 'a::{\text{linordered-field}} \)
assumes le: \( \forall i. i \in A \Rightarrow f i \leq K / \text{of-nat} (\text{card} A) \) and fin: \text{finite} A A \( \neq \) \{\}
shows \( \text{sum} f A \leq K \)
using sum-bounded-above [of \( A \) \( f K \)/ \text{of-nat} (\text{card} A) \), \text{OF} le] fin by simp

lemma sum-bounded-above-strict:
fixes \( K :: 'a::{\text{ordered-cancel-comm-monoid-add,semiring-1}} \)
assumes \( \forall i. i \in A \Rightarrow f i < K \) \( \text{card} A > 0 \)
shows \( \text{sum} f A < \text{of-nat} (\text{card} A) \times K \)
using assms sum-strict-mono[where \( A=A \) and \( g = \lambda x. K \)]
by (simp add: \text{card-gt-0-iff})

lemma sum-bounded-below:
fixes \( K :: 'a::{\text{semiring-1,ordered-comm-monoid-add}} \)
assumes le: \( \forall i. i \in A \Rightarrow K \leq f i \)
shows \( \text{of-nat} (\text{card} A) \times K \leq \text{sum} f A \)
proof (cases \text{finite} A)
case True
then show \( \neg thesis \)
using le sum-mono[where \( K=A \) and \( g = \lambda x. K \)] by simp
next
case False
then show \( \neg thesis \) by simp
qed

lemma convex-sum-bound-le:
fixes \( x :: 'a \Rightarrow 'b::{\text{linordered-idom}} \)
assumes 0: \( \forall i. i \in I \Rightarrow 0 \leq x i \) and 1: \text{sum} I = 1
and  \( \delta : \forall i. i \in I \implies |a_i - b| \leq \delta \)
shows \(|(\sum_{i\in I.} a_i \cdot x_i) - b| \leq \delta \)

proof –

have [simp]: \((\sum_{i\in I.} c \cdot x_i) = c \) for \(c\)
by (simp flip: sum-distrib-left 1)
then have \(|(\sum_{i\in I.} a_i \cdot x_i) - b| = |\sum_{i\in I.} (a_i - b) \cdot x_i|\)
by (simp add: sum-subtractf left-diff-distrib)
also have \(\ldots \leq (\sum_{i\in I.} |(a_i - b) \cdot x_i|)\)
using abs-abs abs-of-nonneg by blast
also have \(\ldots \leq (\sum_{i\in I.} |(a_i - b)| \cdot x_i)\)
by (simp add: abs-mult 0)
also have \(\ldots = \delta\)
by simp
finally show \(\)thesis .

qed

lemma card-UN-disjoint:
assumes finite \(I\) and \(\forall i \in I.\) finite \((A_i)\)
and \(\forall i \in I.\) \(\forall j \in I.\) \(i \neq j \implies A_i \cap A_j = \{\}\)
shows \(\operatorname{card}(\bigcup (A ' I)) = (\sum_{i\in I.} \operatorname{card}(A_i))\)

proof –
have \((\sum_{i\in I.} \operatorname{card}(A_i)) = (\sum_{i\in I.} \sum_{x \in A_i} 1)\)
by simp
with assms show \(\)thesis
by (simp add: card-eq-sum sum.UNION-disjoint del: sum-constant)

qed

lemma card-Union-disjoint:
assumes pairwise disjnt \(C\) and \(\forall A.\) \(A \in C \implies \operatorname{finite} A\)
shows \(\operatorname{card}(\bigcup C) = \sum \operatorname{card} C\)

proof (cases finite \(C\))
case True
then show \(\)thesis
using card-UN-disjoint [OF True, of \(\lambda x.\) \(x\)] assms
by (simp add: disjoint-def fin pairwise-def)
next
case False
then show \(\)thesis
using assms card-eq-0-iff finite-UnionD by fastforce

qed

lemma card-Union-le-sum-card-weak:
fixes \(U::'a\) set set
assumes \(\forall u \in U.\) \(\operatorname{finite} u\)
shows \(\operatorname{card}(\bigcup U) \leq \sum \operatorname{card} U\)

proof (cases \(\operatorname{finite} U\))
case False

then show \( \text{card} \left( \bigcup U \right) \leq \text{sum} \text{ card } U \)
using \( \text{card-eq-0-iff} \text{ finite-UnionD} \text{ by } \text{auto} \)
next
\begin{itemize}
\item \text{case } \text{True}
\item then show \( \text{card} \left( \bigcup U \right) \leq \text{sum} \text{ card } U \)
\item proof (\text{induct } U \text{ rule: finite-induct})
\item case \text{empty}
\item then show \( ? \text{case} \) \text{ by } \text{auto} \)
next
\item \text{case } (\text{insert } x F)
\item then have \( \text{card} \left( \bigcup (\text{insert } x F) \right) \leq \text{card}(x) + \text{card} \left( \bigcup F \right) \) \text{ using } \text{card-Un-le} \text{ by } \text{auto} \)
\item also have \( \ldots \leq \text{card}(x) + \text{sum card } F \) \text{ using } \text{insert.hyps} \text{ by } \text{auto} \)
\item also have \( \ldots = \text{sum card} \left( \text{insert } x F \right) \) \text{ using } \text{sum.insert-if} \text{ and } \text{insert.hyps} \text{ by } \text{auto} \)
\item finally show \( ? \text{case} \) .
\item qed
\item qed
\end{itemize}

\textbf{lemma} \text{card-Union-le-sum-card}:
\begin{itemize}
\item \text{fixes } U :: 'a set set
\item \text{shows } \text{card} \left( \bigcup U \right) \leq \text{sum} \text{ card } U \)
\item \text{by } (\text{metis } \text{Union-upper} \text{ card.infinite } \text{card-Union-le-sum-card-weak} \text{ finite-subset zero-le})
\end{itemize}

\textbf{lemma} \text{card-UN-le}:
\begin{itemize}
\item \text{assumes } \text{finite } I
\item \text{shows } \text{card} \left( \bigcup i \in I. A i \right) \leq \left( \sum i \in I. \text{card}(A i) \right) \)
\item \text{using } \text{assms}
\item proof \text{ induction}
\item case (\text{insert } i I)
\item then show \( ? \text{case} \)
\item \text{using } \text{card-Un-le nat-add-left-cancel-le} \text{ by } (\text{force intro: order-trans})
\item qed
\item auto
\end{itemize}

\textbf{lemma} \text{card-quotient-disjoint}:
\begin{itemize}
\item \text{assumes } \text{finite } A \text{ inj-on} (\lambda x. \{x\} \leftrightarrow r) A
\item \text{shows } \text{card} \left( A/\backslash r \right) = \text{card } A \)
\item proof 
\item have \( \forall i \in A. \forall j \in A. i \neq j \rightarrow r " \{j\} \neq r " \{i\} \)
\item \text{using } \text{assms} \text{ by } (\text{fastforce simp add: quotient-def inj-on-def})
\item with \text{assms} \text{ show } ? \text{thesis}
\item \text{by } (\text{simp add: quotient-def card-UN-disjoint})
\item qed
\end{itemize}

\textbf{lemma} \text{sum-multicount-gen}:
\begin{itemize}
\item \text{assumes } \text{finite } s \text{ finite } t \forall j \in t. (\text{card } \{i \in s. R i j\} = k j)
\item \text{shows } \text{sum} (\lambda i. \text{card} \left( \{j \in t. R i j\} \right)) s = \text{sum } k t \)
\item (\text{is } \forall l = \forall r)
\end{itemize}
proof
  
  have ?l = sum (λi. sum (λx.1) {j∈t. R i j}) s
    by auto
  
  also have ... = ?r
    unfolding sum.swap-restrict [OF assms(1-2)]
    using assms(3) by auto
  
  finally show ?thesis .

qed

lemma sum-multicount:
  assumes finite S finite T
  assumes ∀j∈T. (card {i∈S. R i j} = k)
  shows sum (λi. card {j∈T. R i j}) S = k * card T

  proof
    have ?l = sum (λi. k) T
      by (rule sum-multicount-gen) (auto simp: assms)
    
    also have ... = ?r
      by (simp add: mult.commute)
    
    finally show ?thesis by auto
  qed

lemma sum-card-image:
  assumes finite A
  assumes pairwise (λs t. disjnt (f s) (f t)) A
  shows sum card (f ' A) = sum (λa. card (f a)) A
  using assms
  proof (induct A)
    case (insert a A)
    show ?case
      proof cases
        assume f a = {}
        with insert show ?case
          by (subst sum.mono-neutral-right[where S=f ' A]) (auto simp: pairwise-insert)
      next
        assume f a ≠ {}
        then have sum card (insert (f a) (f ' A)) = card (f a) + sum card (f ' A)
          using insert
          by (subst sum.insert) (auto simp: pairwise-insert)
        with insert show ?case by (simp add: pairwise-insert)
      qed
    qed simp

By Jakub Kdzioka:

lemma sum-fun-comp:
  assumes finite S finite R g ' S ⊆ R
  shows (∑x∈S. f (g x)) = (∑y∈R. of-nat (card {x∈S. g x = y})) * f y
  proof
    let ?r = relation-of (λp q. g p = g q) S
    have eqv: equiv S ?r
      unfolding relation-of-def by (auto intro: comp-equivI)
    have finite: C ∈ S/\ ?r ⇒ finite C for C
by (fact finite-equiv-class[OF finite S, equiv-type[OF equiv S ?r]])

have disjoint: A ∈ S // ?r ⇒ B ∈ S // ?r ⇒ A ≠ B ⇒ A ∩ B = {} for A B

using eqv quotient-disj by blast

let ?cls = λ y. { x ∈ S. y = g x }

have quot-as-img: S // ?r = ?cls ' g ' S

by (auto simp add: relation-of-def quotient-def)

have cls-inj: inj-on ?cls (g ' S)

by (auto intro: inj-onI)

have rest-0: (∑ y ∈ R - g ' S. of-nat (card (?cls y)) * f y) = 0

proof –

have of-nat (card (?cls y)) * f y = 0 if asm: y ∈ R - g ' S for y

proof –

from asm have *: ?cls y = {} by auto

show thesis unfolding * by simp

qed

thus thesis by simp

qed

have (∑ x ∈ S. f (g x)) = (∑ C ∈ S // ?r. ∑ x ∈ C. f (g x))

using eqv finite disjoint

by (simp flip: sum.Union-disjoint[simplified] add: Union-quotient)

also have ... = (∑ y ∈ g ' S. ∑ x ∈ ?cls y. f (g x))

unfolding quot-as-img by (simp add: sum.reindex[OF cls-inj])

also have ... = (∑ y ∈ g ' S. ∑ x ∈ ?cls y. f y)

by auto

also have ... = (∑ y ∈ g ' S. of-nat (card (?cls y)) * f y)

by (simp flip: sum-constant)

also have ... = (∑ y ∈ R. of-nat (card (?cls y)) * f y)

using rest-0 by (simp add: sum.subset-diff[OF g ' S ⊆ R] finite R)

finally show thesis

by (simp add: eq-commute)

qed

48.2.3 Cardinality of products

lemma card-SigmaI [simp]:

finite A ⇒ ∀ a ∈ A. finite (B a) ⇒ card (SIGMA x: A. B x) = (∑ a ∈ A. card (B a))

by (simp add: card-eq-sum sum.Sigma del: sum-constant)

lemma card-cartesian-product: card (A × B) = card A * card B

by (cases finite A ∧ finite B)

(auto simp add: card-eq-0-iff dest: finite-cartesian-productD1 finite-cartesian-productD2)

lemma card-cartesian-product-singleton: card ({x} × A) = card A
by (simp add: card-cartesian-product)

48.3 Generalized product over a set

context comm-monoid-mult
begin

sublocale prod: comm-monoid-set times 1
defines prod = prod.F and prod' = prod.G ..

abbreviation Prod (\prod - [100] 999)
where \prod A \equiv prod (\lambda x. x) A

end

syntax (ASCII)
-prod :: pttrn => 'a set => 'b:comm-monoid-mult ((4PROD (-/-)./-) [0, 51, 10] 10)
syntax
-prod :: pttrn => 'a set => 'b:comm-monoid-mult ((2\prod (-/\in)./-) [0, 51, 10] 10)
translations — Beware of argument permutation!
\prod i \in A. b == CONST prod (\lambda i. b) A

Instead of \prod x \in \{x. P\}. e we introduce the shorter \prod x|P. e.

syntax (ASCII)
-gprod :: pttrn => bool => 'a => 'a ((4PROD -)/ -./-) [0, 0, 10] 10)
syntax
-gprod :: pttrn => bool => 'a => 'a ((2\prod -) (\in)./-) [0, 0, 10] 10)
translations
\prod x|P. t => CONST prod (\lambda x. t) \{x. P\}

custom comm-monoid-mult
begin

lemma prod-dvd-prod: (\bigwedge A. a \in A => f a dvd g a) => prod f A dvd prod g A
proof (induct A rule: infinite-finite-induct)
case infinite then show ?case by (auto intro: dvdI)
next
case empty then show ?case by (auto intro: dvdI)
next
case (insert a A)
then have f a dvd g a and prod f A dvd prod g A
by simp-all
then obtain r s where g a = f a \ast r and prod g A = prod f A \ast s
by (auto elim!: dvdE)
then have g a \ast prod g A = f a \ast prod f A \ast (r \ast s)
by (simp add: ac-simps)
with insert.hyps show ?case
  by (auto intro: dvdI)
qed

lemma prod-dvd-prod-subset: finite B ⇒ A ⊆ B ⇒ prod f A dvd prod f B
  by (auto simp add: prod.subset-diff ac-simps intro: dvdI)
end

48.3.1 Properties in more restricted classes of structures

context linordered-nonzero-semiring
begin

lemma prod-ge-1: (∀x. x ∈ A ⇒ 1 ≤ f x) ⇒ 1 ≤ prod f A
proof (induct A rule: infinite-finite-induct)
  case infinite
  then show ?case by simp
next
  case empty
  then show ?case by simp
next
  case (insert x F)
  have 1 * 1 ≤ f x * prod f F
    by (rule mult-mono') (use insert in auto)
  with insert show ?case by simp
qed

lemma prod-le-1:
  fixes f :: 'b ⇒ 'a
  assumes ∀x. x ∈ A ⇒ 0 ≤ f x ∧ f x ≤ 1
  shows prod f A ≤ 1
    using assms
proof (induct A rule: infinite-finite-induct)
  case infinite
  then show ?case by simp
next
  case empty
  then show ?case by simp
next
  case (insert x F)
  then show ?case by (force simp: mult.commute intro: dest: mult-le-one)
qed

end

context comm-semiring-1
begin
lemma dvd-prod-eq [intro]:
assumes finite A and a ∈ A and b = f a
shows b dvd prod f A

proof
from ‹finite A› have prod f (insert a (A - {a})) = f a * prod f (A - {a})
  by (intro prod.insert) auto
also from ‹a ∈ A› have insert a (A - {a}) = A
  by blast
finally have prod f A = f a * prod f (A - {a})
with ‹b = f a› show ?thesis
  by simp
qed

lemma dvd-prodI [intro]: finite A ⇒ a ∈ A ⇒ f a dvd prod f A
  by auto

lemma prod-zero:
assumes finite A and ∃a∈A. f a = 0
shows prod f A = 0
using assms

proof (induct A)
case empty
then show ?case by simp
next
case (insert a A)
then have f a = 0 ∨ (∃a∈A. f a = 0) by simp
then have f a * prod f A = 0 by (rule disjE) (simp-all add: insert)
with insert show ?case by simp
qed

lemma prod-dvd-prod-subset2:
assumes finite B and A ⊆ B and ∏a ∈ A. f a dvd g a
shows prod f A dvd prod g B

proof
from assms have prod f A dvd prod g A
  by (auto intro: prod-dvd-prod)
moreover from assms have prod g A dvd prod g B
  by (auto intro: prod-dvd-prod-subset)
ultimately show ?thesis by (rule dvd-trans)
qed

end

lemma (in semidom) prod-zero-iff [simp]:
fixes f :: 'b ⇒ 'a
assumes finite A
shows prod f A = 0 ⇔ (∃a∈A. f a = 0)
using assms by (induct A) (auto simp: no-zero-divisors)
lemma (in semidom-divide) prod-diff1:
  assumes finite A and f a ≠ 0
  shows prod f (A − {a}) = (if a ∈ A then prod f A div f a else prod f A)
proof (cases a ⊈ A)
  case True
  then show ?thesis by simp
next
  case False
  with assms show ?thesis
proof induct
  case empty
  then show ?case by simp
next
  case (insert b B)
  then show ?case
proof (cases a = b)
  case True
  with insert show ?thesis by simp
next
  case False
  with insert have a ∈ B by simp
  define C where C = B − {a}
  with ‹finite B› ‹a ∈ B› have B = insert a C finite C a /∈ C
    by auto
  with insert show ?thesis
    by (auto simp add: insert-commute ac-simps)
qed
qed

lemma sum-zero-power [simp]: (∑ i∈A. c i * 0^i) = (if finite A ∧ 0 ∈ A then c 0 else 0)
  for c :: nat ⇒ 'a::division-ring
  by (induct A rule: infinite-finite-induct) auto

lemma sum-zero-power' [simp]:
  (∑ i∈A. c i * 0^i / d i) = (if finite A ∧ 0 ∈ A then c 0 / d 0 else 0)
  for c :: nat ⇒ 'a::field
  using sum-zero-power [of λi. c i / d i A] by auto

lemma (in field) prod-inversef: prod (inverse ∘ f) A = inverse (prod f A)
proof (cases finite A)
  case True
  then show ?thesis
    by (induct A rule: finite-induct) simp-all
next
  case False
  then show ?thesis
by auto
qed

lemma (in field) prod-dividef: \( \prod_{x \in A} \frac{f x}{g x} = \prod f A / \prod g A \)
using prod-inversef [of g A] by (simp add: divide-inverse prod.distrib)

lemma prod-Un:
fixes f :: 'b ⇒ 'a :: field
assumes finite A and finite B
and \( \forall x \in A \cap B. f x \neq 0 \)
shows \( \prod f (A \cup B) = \prod f A * \prod f B / \prod f (A \cap B) \)
proof –
from assms have \( \prod f A * \prod f B = \prod f (A \cup B) * \prod f (A \cap B) \)
by (simp add: prod.union-inter [symmetric, of A B])
with assms show ?thesis
by simp
qed

context linordered-semidom
begin

lemma prod-nonneg: \( \forall a \in A \Rightarrow 0 \leq f a \) \Rightarrow 0 \leq \prod f A
by (induct A rule: infinite-finite-induct) simp-all

lemma prod-pos: \( \forall a \in A \Rightarrow 0 < f a \) \Rightarrow 0 < \prod f A
by (induct A rule: infinite-finite-induct) simp-all

lemma prod-mono:
(\( \forall i. i \in A \Rightarrow 0 \leq f i \land f i \leq g i \) \Rightarrow \prod f A \leq \prod g A
by (induct A rule: infinite-finite-induct) (force intro!: prod-nonneg mult-mono)+

Only one needs to be strict

lemma prod-mono-strict:
assumes i ∈ A f i < g i
assumes finite A
assumes \( \forall i. i \in A \Rightarrow 0 \leq f i \land f i \leq g i \)
assumes \( \forall i. i \in A \Rightarrow 0 < g i \)
shows \( \prod f A < \prod g A \)
proof –
have \( \prod f A = f i * \prod f (A - \{i\}) \)
using assms by (intro prod.remove)
also have \( \ldots \leq f i * \prod g (A - \{i\}) \)
using assms by (intro mult-left-mono prod-mono) auto
also have \( \ldots < g i * \prod g (A - \{i\}) \)
using assms by (intro mult-strict-right-mono prod-pos) auto
also have \( \ldots = \prod g A \)
using assms by (intro prod.remove [symmetric])
finally show ?thesis .
qed
lemma prod-le-power:
assumes A: \( \forall i. \ i \in A \implies 0 \leq f i \land f i \leq n \ \operatorname{card} A \leq k \) and \( n \geq 1 \)
shows prod f A \leq n ^ k
using A
proof (induction A arbitrary: k rule: infinite-finite-induct)
case (insert i A)
then obtain k' where k' = Suc k
using Suc-le-D by force
have f i * prod f A \leq n * n ^ k'
using insert \( \{ n \geq 1 \} \) k' by (intro prod-nonneg mult-mono; force)
then show \( \vdots \) by (auto simp: \( k = \operatorname{Suc} k' \) insert.hyps)
qed (use \( \{ n \geq 1 \} \) in auto)
end

lemma prod-mono2:
fixes f :: 'a \Rightarrow 'b :: linordered-idom
assumes fin: finite B
and sub: A \subseteq B
and nn: \( \forall b. \ b \in B - A \implies 1 \leq f b \)
and A: \( \forall a. \ a \in A \implies 0 \leq f a \)
shows prod f A \leq prod f B
proof
  have prod f A \leq prod f A * prod f (B - A)
    by (metis prod-ge-1 A mult-le-cancel-left1 nn not-less prod-nonneg)
  also from fin finite-subset[OF sub fin] have \( \vdots \) = prod f (A \cup (B - A))
    by (simp add: prod.union_disjoint del: Un-Diff-cancel)
  also from sub have A \cup (B - A) = B by blast
  finally show \( \vdots \) thesis .
qed

lemma less-1-prod:
fixes f :: 'a \Rightarrow 'b :: linordered-idom
shows finite I \( \implies I \neq \{ \} \implies (\forall i. \ i \in I \implies 1 < f i) \implies 1 < prod f I \)
by (induct I rule: finite-ne-induct) (auto intro: less-1-mult)

lemma less-1-prod2:
fixes f :: 'a \Rightarrow 'b :: linordered-idom
assumes I: finite I i \in I I < f i \ \forall i. \ i \in I \implies 1 \leq f i
shows I < prod f I
proof
  have I < f i * prod f (I - \{ i \})
    using assms
    by (meson DiffD1 leI less-1-mult less-le-trans mult.le-cancel-left1 prod.ge-1)
  also have \( \vdots \) = prod f I
    using assms by (simp add: prod.remove)
  finally show \( \vdots \) thesis .
THEORY “Groups-Big”

qed

**lemma** (in linordered-field) abs-prod: \( \prod x \in A. |f x| \)
by (induct A rule: infinite-finite-induct) (simp-all add: abs-mult)

**lemma** prod-eq-1-iff [simp]: finite A \( \implies \) prod f A = 1 \( \iff \) (\( \forall a \in A. f a = 1 \))
for f :: 'a \Rightarrow 'nat
by (induct A rule: infinite-finite-induct) simp-all

**lemma** prod-pos-nat-iff [simp]: finite A \( \implies \) prod f A > 0 \( \iff \) (\( \forall a \in A. f a > 0 \))
for f :: 'a \Rightarrow 'nat
using prod-zero-iff by (simp del: neq0_conv add: zero_less_iff_neq_zero)

**lemma** prod-constant [simp]: (\( \prod x \in A. y \)) = y ^ card A
for y :: 'a::comm-monoid-mult
by (induct A rule: infinite-finite-induct) simp-all

**lemma** prod-power-distrib: prod f A ^ n = prod (\( \lambda x. (f x)^n \)) A
for f :: 'a \Rightarrow 'b::comm-semiring-1
by (induct A rule: infinite-finite-induct) (auto simp add: power-mult-distrib)

**lemma** power-sum: c ^ (\( \sum a \in A. f a \)) = (\( \prod a \in A. c ^ f a \))
by (induct A rule: infinite-finite-induct) (simp-all add: power-add)

**lemma** prod-gen-delta:
fixes b :: 'b \Rightarrow 'a::comm-monoid-mult
assumes fin: finite S
shows prod (\( \lambda k. if k = a then b k else c \)) S =
(\( if a \in S then b a * c ^ (card S - 1) \) else c ^ card S)
proof –
let ?f = (\( \lambda k. if k = a then b k else c \))
show ?thesis
proof (cases a \in S)
case False
then have \( \forall k \in S. ?f k = c \) by simp
with False show ?thesis by (simp add: prod-constant)
next
case True
let \( ?A = S - \{a\} \)
let \( ?B = \{a\} \)
from True have eq: S = ?A \cup ?B by blast
have disjoint: ?A \cap ?B = {} by simp
from fin have fin': finite ?A finite ?B by auto
have f-A0: prod \( ?f ?A = prod (\lambda i. c) ?A \)
  by (rule prod.cong) auto
from fin True have card-A: card ?A = card S - 1 by auto
have f-A1: prod \( ?f ?A = c ^ card ?A \)
  unfolding f-A0 by (rule prod-constant)
have prod \( ?f ?A * prod ?f ?B = prod ?f S \)
using prod.union-disjoint[OF fin1 disjoint, of ?f, unfolded eq[symmetric]]
by simp
with True card-A show ?thesis
by (simp add: f-A1 field-simps cong add: prod.cong cong del: if-weak-cong)
qed

lemma sum-image-le:
  fixes g :: 'a ⇒ 'b::ordered-comm-monoid-add
  assumes finite I \&\& i ∈ I ⇒ 0 ≤ g(f i)
  shows sum g (f ' I) ≤ sum (g ◦ f) I
  using assms
  proof induction
    case empty
    then show ?case by auto
  next
    case (insert i I)
    hence *: sum g (f ' I) ≤ g (f i) + sum g (f ' I)
      using add-increasing by blast+
    have sum g (f ' insert i I) = sum g (insert (f i) (f ' I)) by simp
    also have ... ≤ g (f i) + sum g (f ' I) by (simp add: * insert sum.insert-if)
    also from * have ... ≤ g (f i) + sum (g ◦ f) I by (intro add-left-mono)
    also from insert have ... = sum (g ◦ f) (insert i I) by (simp add: sum.insert-if)
    finally show ?case .
  qed

end

49 Chain-complete partial orders and their fix-points

theory Complete-Partial-Order
  imports Product-Type
begin

49.1 Chains

A chain is a totally-ordered set. Chains are parameterized over the order
for maximal flexibility, since type classes are not enough.

definition chain :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool
  where chain ord S ⟷ (∀x∈S. ∀y∈S. ord x y ∨ ord y x)

lemma chainI:
  assumes ∃x y. x ∈ S ⇒ y ∈ S ⇒ ord x y ∨ ord y x
  shows chain ord S
  using assms unfolding chain-def by fast

lemma chainD:
assumes chain ord S and x ∈ S and y ∈ S
shows ord x y ∨ ord y x
using assms unfolding chain-def by fast

lemma chainE:
assumes chain ord S and x ∈ S and y ∈ S
obtains ord x y | ord y x
using assms unfolding chain-def by fast

lemma chain-empty: chain ord {}
by (simp add: chain-def)

lemma chain-equality: chain (=) A ↔ (∀ x∈A. ∀ y∈A. x = y)
by (auto simp add: chain-def)

lemma chain-subset: chain ord A ⇒ B ⊆ A ⇒ chain ord B
by (rule chainI) (blast dest: chainD)

lemma chain-imageI:
assumes chain: chain le-a Y
and mono: ∃ x y. x ∈ Y ⇒ y ∈ Y ⇒ le-a x y ⇒ le-b (f x) (f y)
shows chain le-b (f Y)
by (blast intro: chainI dest: chainD[OF chain mono])

49.2 Chain-complete partial orders

A ccpo has a least upper bound for any chain. In particular, the empty set is a chain, so every ccpo must have a bottom element.

class ccpo = order + Sup +
assumes ccpo-Sup-upper: chain (≤) A ⇒ x ∈ A ⇒ x ≤ Sup A
assumes ccpo-Sup-least: chain (≤) A ⇒ (∀ x ∈ A ⇒ x ≤ z) ⇒ Sup A ≤ z

begin

lemma chain-singleton: Complete-Partial-Order.chain (≤) {x}
by (rule chainI) simp

lemma ccpo-Sup-singleton [simp]: ∩ {x} = x
by (rule order.antisym) (auto intro: ccpo-Sup-least ccpo-Sup-upper simp add: chain-singleton)

49.3 Transfinite iteration of a function

context notes [[inductive-internals]]

begin

inductive-set iterates :: (′a ⇒ ′a) ⇒ ′a set
for f :: ′a ⇒ ′a
where
THEORY "Complete-Partial-Order"

```
step: x ∈ iterates f ⇒ f x ∈ iterates f
| Sup: chain (≤) M ⇒ ∀ x∈M. x ∈ iterates f ⇒ Sup M ∈ iterates f

end

lemma iterates-le-f: x ∈ iterates f ⇒ monotone (≤) (≤) f ⇒ x ≤ f x
  by (induct x rule: iterates.induct)
  (force dest: monotoneD intro: ccpo-Sup-upper ccpo-Sup-least)+

lemma chain-iterates:
  assumes f: monotone (≤) (≤) f
  shows chain (≤) (iterates f) (is chain - ?C)
proof (rule chainI)
  fix x y
  assume x ∈ ?C y ∈ ?C
  then show x ≤ y ∨ y ≤ x
  proof (induct x arbitrary: y rule: iterates.induct)
    fix x y
    assume y: y ∈ ?C
    and IH: ∀ z. z ∈ ?C ⇒ x ≤ z ∨ z ≤ x
    from y show f x ≤ y ∨ y ≤ f x
    proof (induct y rule: iterates.induct)
      case (step y)
      with IH f show ?case by (auto dest: monotoneD)
    next
      case (Sup M)
      then have chM: chain (≤) M
                      and IH': ∀ z. z ∈ M ⇒ f x ≤ z ∨ z ≤ f x by auto
      show f x ≤ Sup M ∨ Sup M ≤ f x
      proof (cases ∃ z∈M. f x ≤ z)
        case True
        then have f x ≤ Sup M
          by (blast intro: ccpo-Sup-upper[OF chM] order-trans)
        then show ?thesis ..
      next
        case False
        with IH' show ?thesis
        by (auto intro: ccpo-Sup-least[OF chM])
      qed
    qed
  next
  case (Sup M y)
  show ?case
  proof (cases ∃ x∈M. y ≤ x)
    case True
    then have y ≤ Sup M
      by (blast intro: ccpo-Sup-upper[OF Sup(1)] order-trans)
    then show ?thesis ..
  next
```
case False with Sup
  show ?thesis by (auto intro: ccpo-Sup-least)
qed
qed
qed

lemma bot-in-iterates: Sup {} ∈ iterates f
  by (auto intro: iterates.Sup simp add: chain-empty)

49.4 Fixpoint combinator

definition fixp :: (′a ⇒ ′a) ⇒ ′a
  where fixp f = Sup (iterates f)

lemma iterates-fixp:
  assumes f: monotone (≤) (≤) f
  shows fixp f ∈ iterates f
  unfolding fixp-def
  by (simp add: iterates.Sup chain-iterates f)

lemma fixp-unfold:
  assumes f: monotone (≤) (≤) f
  shows fixp f = f (fixp f)
  proof (rule order.antisym)
    show fixp f ≤ f (fixp f)
      by (intro iterates-le-f iterates-fixp f)
    have f (fixp f) ≤ Sup (iterates f)
      by (intro ccpo-Sup-upper chain-iterates f iterates.step iterates-fixp)
    then show f (fixp f) ≤ fixp f
      by (simp only: fixp-def)
  qed

lemma fixp-lowerbound:
  assumes f: monotone (≤) (≤) f
    and z: f z ≤ z
  shows fixp f ≤ z
  unfolding fixp-def
  proof (rule ccpo-Sup-least[OF chain-iterates[OF f]])
    fix x
    assume x ∈ iterates f
    then show x ≤ z
      proof (induct x rule: iterates.induct)
        case (step x)
        from f (x ≤ z) have f x ≤ f z
          by (rule monotoneD)
        also note z
        finally show f x ≤ z.
      next
        case (Sup M)
        then show ?case
by (auto intro: ccpo-Sup-least)
qed
qed
end

49.5 Fixpoint induction

setup ⟨Sign.map-naming (Name-Space.mandatory-path ccpo)⟩

definition admissible :: ('a set ⇒ 'a ⇒ bool ⇒ bool) ⇒ ('a ⇒ bool ⇒ bool) ⇒ bool
  where admissible lub ord P ←→ (∀ A. chain ord A −→ A ≠ {} −→ (∀ x∈A. P x) −→ P (lub A))

lemma admissibleI:
  assumes chain ord A =⇒ A ≠ {} =⇒ (∀ x∈A. P x) =⇒ P (lub A)
  shows ccpo.admissible lub ord P
using assms unfolding ccpo.admissible-def by fast

lemma admissibleD:
  assumes ccpo.admissible lub ord P
  assumes chain ord A
  assumes A ≠ {}
  assumes (∀ x. x ∈ A =⇒ P x)
  shows P (lub A)
using assms by (auto simp: ccpo.admissible-def)

setup ⟨Sign.map-naming Name-Space.parent-path⟩

lemma (in ccpo) fixp-induct:
  assumes adm: ccpo.admissible Sup (≤) P
  assumes mono: monotone (≤) (≤) f
  assumes bot: P (Sup {})?
  assumes step: (∀ x. P x =⇒ P (f x))
  shows P (fixp f)
using adm chain-iterates[OF mono]

proof (rule ccpo.admissibleD)
  show iterates f ≠ {} using bot in iterates by auto
next
  fix x
  assume x ∈ iterates f
  then show P x
proof (induct rule: iterates.induct)
  case prems; (step x)
  from this(2) show ?case by (rule step)
next
  case (Sup M)
then show \textbf{?case} by (cases M = {}) (auto intro: step bot ccpo.admissibleD adm)  
\textbf{qed}
\textbf{qed}

lemma \textbf{admissible-True}: ccpo.admissible lub ord (\lambda x. True)  
\textbf{unfolding} ccpo.admissible-def \textbf{by simp}

lemma \textbf{admissible-const}: ccpo.admissible lub ord (\lambda x. t)  
\textbf{by} (auto intro: ccpo.admissibleI)

lemma \textbf{admissible-conj}:
\textbf{assumes} ccpo.admissible lub ord (\lambda x. P x)
\textbf{assumes} ccpo.admissible lub ord (\lambda x. Q x)
\textbf{shows} ccpo.admissible lub ord (\lambda x. P x \land Q x)
\textbf{using} assms \textbf{unfolding} ccpo.admissible-def \textbf{by simp}

lemma \textbf{admissible-all}:
\textbf{assumes} \( \forall y. \) ccpo.admissible lub ord (\lambda x. P x y)
\textbf{shows} ccpo.admissible lub ord (\lambda x. \forall y. P x y)
\textbf{using} assms \textbf{unfolding} ccpo.admissible-def \textbf{by fast}

lemma \textbf{admissible-ball}:
\textbf{assumes} \( \forall y. y \in A \Rightarrow \) ccpo.admissible lub ord (\lambda x. P x y)
\textbf{shows} ccpo.admissible lub ord (\lambda x. \forall y \in A. P x y)
\textbf{using} assms \textbf{unfolding} ccpo.admissible-def \textbf{by fast}

context ccpo  
\textbf{begin}

lemma \textbf{admissible-disj}:
\textbf{fixes} P Q :: 'a \Rightarrow bool
\textbf{assumes} P: ccpo.admissible Sup (\leq) (\lambda x. P x)
\textbf{assumes} Q: ccpo.admissible Sup (\leq) (\lambda x. Q x)
\textbf{shows} ccpo.admissible Sup (\leq) (\lambda x. P x \lor Q x)
\textbf{proof} (rule ccpo.admissibleI)
\textbf{fix} A :: 'a set
\textbf{assume} chain: chain (\leq) A
\textbf{assume} A: A \neq {} \textbf{and} P.Q: \forall x \in A. P x \lor Q x
\textbf{have} (\exists x \in A. P x) \land (\forall x \in A. \exists y \in A. x \leq y \land P y) \lor (\exists x \in A. Q x) \land (\forall x \in A. \exists y \in A. x \leq y \land Q y)
\textbf{is \ ?P \lor \ ?Q is \ ?P1 \ \& \ ?P2 \ \lor -)
\textbf{proof} (rule disjCI)
\textbf{assume} \neg \ ?Q
\textbf{then consider} \forall x \in A. \neg Q x | a \textbf{ where} a \in A \forall y \in A. a \leq y \Rightarrow \neg Q y
by blast 
then show ?P 
proof cases 
case 1 
with P-Q have \( \forall x \in A. \ P x \) by blast 
with A show ?P by blast 
next 
case 2 
note a = \{ a \in A \} 
show ?P 
proof 
from P-Q 2 have *: \( \forall y \in A. \ a \leq y \rightarrow P y \) by blast 
with a have P a by blast 
with a show ?P1 by blast 
show ?P2 
proof 
fix x 
assume x: x \in A 
with chain a show \( \exists y \in A. \ x \leq y \land P y \) 
proof (rule chainE) 
assume le: a \leq x 
with * a x have P x by blast 
with x le show ?thesis by blast 
next 
assume a \geq x 
with a \{ P a \} show ?thesis by blast 
qed 
qed 
qed 
moreover 
have Sup A = Sup \{ x \in A. \ P x \} if \( \land x. \ x \in A \implies \exists y \in A. \ x \leq y \land P y \) for P 
proof (rule order.antisym) 
have chain-P: chain (\( \leq \)) \{ x \in A. \ P x \} 
  by (rule chain-compr [OF chain]) 
show Sup A \leq Sup \{ x \in A. \ P x \} 
proof (rule ccpo-Sup-least [OF chain]) 
  show \( \land x. \ x \in A \implies x \leq \bigsqcup \{ x \in A. \ P x \} \) 
  by (blast intro: ccpo-Sup-upper [OF chain-P] order-trans dest: that) 
  qed 
show Sup \{ x \in A. \ P x \} \leq Sup A 
apply (rule ccpo-Sup-least [OF chain-P]) 
apply (simp add: ccpo-Sup-upper [OF chain]) 
done 
qed 
ultimately 
consider \( \exists x. \ x \in A \land P x \) Sup A = Sup \{ x \in A. \ P x \} 
\mid \exists x. \ x \in A \land Q x \) Sup A = Sup \{ x \in A. \ Q x \}
by blast
then show \( P \supset (\text{Sup } A) \lor Q (\text{Sup } A) \)
proof cases
  case 1
  then show ?thesis
    using ccpo.admissibleD [OF \( P \) chain-compr [OF chain]] by force
next
  case 2
  then show ?thesis
    using ccpo.admissibleD [OF \( Q \) chain-compr [OF chain]] by force
qed
qed
end

instance complete-lattice \( \subseteq \text{ccpo} \)
by standard (fast intro: Sup-upper Sup-least)+

lemma lfp-eq-fixp:
  assumes mono: mono \( f \)
  shows lfp \( f \) = fixp \( f \)
proof (rule order.antisym)
  from mono have \( f' \): monotone \( (\leq) (\leq) \) \( f \)
    unfolding mono-def monotone-def .
  show lfp \( f \) \( \leq \) fixp \( f \)
    by (rule lfp-lowerbound, subst fixp-unfold [OF \( f' \)], rule order-refl)
  show fixp \( f \) \( \leq \) lfp \( f \)
    by (rule fixp-lowerbound [OF \( f' \)]) (simp add: lfp-fixpoint [OF mono])
qed

hide-const (open) iterates fixp
end

50 Datatype option

theory Option
  imports Lifting
begin

datatype 'a option =
  None
| Some (the: 'a)

datatype-compat option

lemma [case-names None Some, cases type: option]:
— for backward compatibility — names of variables differ
  \( y = \text{None} \rightarrow P \) \( \rightarrow (\forall a. y = \text{Some } a \rightarrow P) \rightarrow P \)
by (rule option.exhaust)

**Lemma** [case-names None Some, induct type: option]:
— for backward compatibility – names of variables differ

\[ P \text{ None} \Rightarrow (\forall \text{ option}. \; P (\text{Some option})) \Rightarrow P \text{ option} \]

by (rule option.induct)

Compatibility:

setup `Sign.mandatory-path option`
lemmas inducts = option.induct
lemmas cases = option.case
setup `Sign.parent-path`

**Lemma** not-None-eq [iff]: \(x \neq \text{ None} \iff (\exists \; y. \; x = \text{ Some } y)\)
by (induct \( x \)) auto

**Lemma** not-Some-eq [iff]: \((\forall \; y. \; x \neq \text{ Some } y) \iff x = \text{ None}\)
by (induct \( x \)) auto

**Lemma** comp-the-Some [simp]: the o Some = id
by auto

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform iff attribute.

**Lemma** inj-Some [simp]: inj-on Some A
by (rule inj-onI) simp

**Lemma** case-optionE:
assumes \(c\): \((\text{ case } x \text{ of } \text{ None} \Rightarrow P \mid \text{ Some } y \Rightarrow Q y)\)
obtains
\((\text{ None}) \; x = \text{ None} \; \text{ and } P\)
| \((\text{ Some}) \; y \; \text{where } x = \text{ Some } y \; \text{ and } Q y\)
using \(c\) by (cases \( x \)) simp-all

**Lemma** split-option-all: \((\forall \; x. \; P \; x) \iff P \; \text{ None} \; \land \; (\forall \; x. \; P \; (\text{Some } x))\)
by (auto intro: option.induct)

**Lemma** split-option-ex: \((\exists \; x. \; P \; x) \iff P \; \text{ None} \; \lor \; (\exists \; x. \; P \; (\text{Some } x))\)
using split-option-all[of \( \lambda x. \; \neg \; P \; x \)] by blast

**Lemma** UNIV-option-conv: UNIV = insert \( \text{ None} \) (range Some)
by (auto intro: classical)

**Lemma** rel-option-None1 [simp]: rel-option \( P \; \text{ None} \iff x = \text{ None}\)
by (cases \( x \)) simp-all

**Lemma** rel-option-None2 [simp]: rel-option \( P \; x \; \text{ None} \iff x = \text{ None}\)
by (cases \( x \)) simp-all
lemma option-rel-Some1: rel-option A (Some x) y ⟷ (∃ y'. y = Some y' ∧ A x y')
by (cases y) simp-all

lemma option-rel-Some2: rel-option A x (Some y) ⟷ (∃ x'. x = Some x' ∧ A x' y)
by (cases x) simp-all

lemma rel-option-inf: inf (rel-option A) (rel-option B) = rel-option (inf A B)
(is ?lhs = ?rhs)
proof (rule antisym)
  show ?lhs ≤ ?rhs by (auto elim: option.rel-cases)
  show ?rhs ≤ ?lhs by (auto elim: option.rel-mono-strong)
qed

lemma rel-option-reflI: (⋀ x. x ∈ set-option y =⇒ P x x) =⇒ rel-option P y y
by (cases y) auto

50.0.1 Operations
lemma ospec [dest]: (∀ x ∈ set-option A. P x) =⇒ A = Some x =⇒ P x
by simp

setup {map-theory-claset (fn ctxt => ctxt addSD2 {ospec, @{thm ospec}})}

lemma elem-set [iff]: (x ∈ set-option xo) = (xo = Some x)
by (cases xo) auto

lemma set-empty-eq [simp]: (set-option xo = {}) = (xo = None)
by (cases xo) auto

lemma map-option-case: map-option f y = (case y of None ⇒ None | Some x ⇒ Some (f x))
by (auto split: option.split)

lemma map-option-is-None [iff]: (map-option f opt = None) = (opt = None)
by (simp add: map-option-case split: option.split)

lemma None-eq-map-option-iff [iff]: None = map-option f x =⇒ x = None
by (cases x) simp-all

lemma map-option-eq-Some [iff]: (map-option f xo = Some y) = (∃ z. xo = Some z ∧ f z = y)
by (simp add: map-option-case split: option.split)

lemma map-option-o-case-sum [simp]:
  map-option f ∘ case-sum g h = case-sum (map-option f ∘ g) (map-option f ∘
lemma map-option-cong: \( x = y \implies (\forall a. \ y = \text{Some} \ a \implies f \ a = g \ a) \implies \) 
map-option \( f \ x = \text{map-option} \ g \ y \) 
by (cases \( x \)) auto

lemma map-option-idI: \( (\forall y. y \in \text{set-option} \ x \implies f \ y = y) \implies \text{map-option} \ f \ x = x \) 
by(cases \( x \))(simp-all)

functor map-option: map-option 
by (simp-all add: option.map-comp fun-eq-iff option.map-id)

lemma case-map-option [simp]: \( \text{case-option} \ g \ h \ (\text{map-option} \ f \ x) = \text{case-option} \ g \ (h \circ f) \ x \) 
by (cases \( x \)) simp-all

lemma None-notin-image-Some [simp]: \( \text{None} \notin \text{Some} ' A \) 
by auto

lemma notin-range-Some: \( x \notin \text{range} \text{Some} \iff x = \text{None} \) 
by(auto)

lemma rel-option-iff: \( \text{rel-option} \ R \ x \ y = (\text{case} (x, y) \text{ of} (\text{None, None}) \Rightarrow \text{True} \) 
\| (\text{Some} x, \text{Some} y) \Rightarrow R \ x \ y 
\| - \Rightarrow \text{False}) 
by(auto split: prod.split option.split)

definition combine-options :: ('a ⇒ 'a ⇒ 'a) ⇒ 'a option ⇒ 'a option ⇒ 'a option 
where combine-options \( f \ x \ y = \) 
\( (\text{case} \ x \text{ of} \text{None} \Rightarrow y \ | \text{Some} \ x \Rightarrow (\text{case} \ y \text{ of} \text{None} \Rightarrow \text{Some} \ x \ | \text{Some} \ y \Rightarrow \text{Some} \ (f \ x \ y)))) \)

lemma combine-options-simps [simp]: 
combine-options \( f \ \text{None} \ y = y \)
combine-options \( f \ x \ \text{None} = x \)
combine-options \( f \ (\text{Some} \ a) \ (\text{Some} \ b) = \text{Some} \ (f \ a \ b) \) 
by(simp-all add: combine-options-def split: option.splits)

lemma combine-options-cases [case-names None1 None2 Some]: 
\( (x = \text{None} \Rightarrow P \ x \ y) \Rightarrow (y = \text{None} \Rightarrow P \ x \ y) \Rightarrow \) 
\( (\forall a b. x = \text{Some} \ a \Rightarrow y = \text{Some} \ b \Rightarrow P \ x \ y) \Rightarrow P \ x \ y \) 
by (cases \( x \); cases \( y \)) simp-all

lemma combine-options-commute: 
\( (\forall x. f \ x \ y = f \ y \ x) \Rightarrow \text{combine-options} \ f \ x \ y = \text{combine-options} \ f \ y \ x \)
using combine-options-cases[of x ]
by (induction x y rule: combine-options-cases) simp-all

lemma combine-options-assoc:
(∀x y z. f (f x y) z = f x (f y z)) ⟹
combine-options f (combine-options f x y) z =
combine-options f x (combine-options f y z)
by (auto simp: combine-options-def split: option.splits)

lemma combine-options-left-commute:
(∀x. f x y = f y x) ⟹ (∀x y z. f (f x y) z = f x (f y z)) ⟹
combine-options f y (combine-options f x z) =
combine-options f x (combine-options f y z)
by (auto simp: combine-options-def split: option.splits)

lemmas combine-options-ac =
   combine-options-commute combine-options-assoc combine-options-left-commute

context
begin

qualified definition is-none :: 'a option ⇒ bool
  where [code-post]: is-none x ←→ x = None

lemma is-none-simps [simp]:
  is-none None
  ¬ is-none (Some x)
by (simp-all add: is-none-def)

lemma is-none-code [code]:
  is-none None = True
  is-none (Some x) = False
by simp

lemma rel-option-unfold:
rel-option R x y ←→
(is-none x ←→ is-none y) ∧ (¬ is-none x → ¬ is-none y → R (the x) (the y))
by (simp add: rel-option-iff split: option.split)

lemma rel-option1:
[ is-none x ←→ is-none y; [ ¬ is-none x; ¬ is-none y ] ⟹ P (the x) (the y) ] ⟹
rel-option P x y
by (simp add: rel-option-unfold)

lemma is-none-map-option [simp]: is-none (map-option f x) ←→ is-none x
by (simp add: is-none-def)
lemma the-map-option: \(\neg \text{is-none } x \implies \text{the}(\text{map-option } f x) = f \text{ (the } x)\)
  by (auto simp add: is-none-def)

qualified primrec bind :: 'a option ⇒ ('a ⇒ 'b option) ⇒ 'b option
where
  bind-lzero: bind None f = None
  bind-lunit: bind (Some x) f = f x

lemma is-none-bind: is-none (bind f g) ←→ is-none f ∨ is-none (g (the f))
  by (cases f) simp-all

lemma bind-runit[simp]: bind x Some = x
  by (cases x) auto

lemma bind-assoc[simp]: bind (bind x f) g = bind x (λy. bind (f y) g)
  by (cases x) auto

lemma bind-rzero[simp]: bind x (λx. None) = None
  by (cases x) auto

qualified lemma bind-cong: x = y ⇒ (∀a. y = Some a ⇒ f a = g a) ⇒ bind x f = bind y g
  by (cases x) auto

lemma bind-split: P (bind m f) ←→ (m = None → P None) ∧ (∀v. m = Some v → P (f v))
  by (cases m) auto

lemma bind-split-asn: P (bind m f) ←→ ¬ (m = None ∧ ¬ P None ∨ (∃x. m = Some x ∧ ¬ P (f x)))
  by (cases m) auto

lemmas bind-splits = bind-split bind-split-asn

lemma bind-eq-None-conv: bind a f = Some x ←→ (∃y. f = Some y ∧ g y = Some x)
  by (cases f) simp-all

lemma bind-eq-None-conv: Option.bind a f = None ←→ a = None ∨ f (the a) = None
  by (cases a) simp-all

lemma map-option-bind: map-option f (bind x g) = bind x (map-option f ∘ g)
  by (cases x) simp-all

lemma bind-option-cong:
  \[ x = y; \forall z. z ∈ \text{set-option } y \implies f z = g z \] ⇒ bind x f = bind y g
  by (cases y) simp-all
lemma bind-option-cong-simp:
\[ x = y; \forall z. z \in \text{set-option } y \Rightarrow f z = g z \] \implies bind x f = bind y g
unfolding simp-implies-def by (rule bind-option-cong)

lemma bind-option-cong-code: \( x = y \) \implies bind x f = bind y f
by simp

lemma bind-map-option: bind (map-option f x) g = bind x \((g \circ f)\)
by (cases x) simp-all

lemma set-bind-option [simp]: \( \text{set-option } (\text{bind } x f) = (\bigcup \{\text{set-option } \circ f \} \ \text{of } \text{set-option } x) \)
by (cases x) auto

lemma map-conv-bind-option: map-option f x = Option.bind x (Some \circ f)
by (cases x) simp-all

end

setup (Code-Simp.map-ss (Simplifier.add-cong @{thm bind-option-cong-code}))

context
begin

qualified definition these :: \('a \ \text{option } \text{set} \Rightarrow 'a \ \text{set}\)
where these A = the \'{\{x \in A. x \neq \text{None}\}}

lemma these-empty [simp]: these {} = {} 
by (simp add: these-def)

lemma these-insert-None [simp]: these (insert None A) = these A
by (auto simp add: these-def)

lemma these-insert-Some [simp]: these (insert (Some x) A) = insert x (these A)
proof
  have \{y \in insert (Some x) A. y \neq \text{None}\} = insert (Some x) \{y \in A. y \neq \text{None}\}
  by auto
  then show \?thesis by (simp add: these-def)
qed

lemma in-these-eq: \( x \in \text{these } A \iff \text{Some } x \in A \)
proof
  assume Some x \in A
  then obtain B where A = insert (Some x) B by auto
  then show x \in \text{these } A by (auto simp add: these-def intro: image-eqI)
next
  assume x \in \text{these } A

then show Some $x \in A$ by (auto simp add: these-def)
qed

lemma these-image-Some-eq [simp]: these (Some ' A) = A
by (auto simp add: these-def intro: image-eqI)

lemma Some-image-these-eq: Some ' these A = \{x\in A. x \neq None\}
by (auto simp add: these-def image-image intro: !: image-eqI)

lemma these-empty-eq: these B = {} \iff B = {} \lor B = {None}
by (auto simp add: these-empty-eq)

lemma these-not-empty-eq: these B \neq {} \iff B \neq {} \land B \neq {None}
by (auto simp add: these-empty-eq)

end

lemma finite-range-Some: finite (range (Some :: 'a \Rightarrow 'a option)) = finite (UNIV :: 'a set)
by (auto dest: finite-imageD intro: inj-Some)

50.1 Transfer rules for the Transfer package
context includes lifting-syntax
begin

lemma option-bind-transfer [transfer-rule]:
(rel-option A ===> (A ===> rel-option B) ===> rel-option B)
Option.bind Option.bind
unfolding rel-fun-def split-option-all by simp

lemma pred-option-parametric [transfer-rule]:
((A ===> (=)) ===> rel-option A ===> (=)) pred-option pred-option
by (rule rel-funI)+ (auto simp add: rel-option-unfold Option.is-none-def dest: rel-funD)

end

50.1.1 Interaction with finite sets
lemma finite-option-UNIV [simp]:
finite (UNIV :: 'a option set) = finite (UNIV :: 'a set)
by (auto simp add: UNIV-option-conv elim: finite-imageD intro: inj-Some)

instance option :: (finite) finite
by standard (simp add: UNIV-option-conv)

50.1.2 Code generator setup
lemma equal-None-code-unfold [code-unfold]:

THEORY “Option”
HOL.equal x None \iff Option.is-none x
HOL.equal None = Option.is-none
by (auto simp add: equal Option.is-none-def)

code-printing
type-constructor option \to
(SML) - option
and (OCaml) - option
and (Haskell) Maybe -
and (Scala) Option(\_)
| constant None \to
(SML) NONE
and (OCaml) None
and (Haskell) Nothing
and (Scala) !None
| constant Some \to
(SML) SOME
and (OCaml) Some -
and (Haskell) Just
and (Scala) Some
| class-instance option :: equal \to
(Haskell) -
| constant HOL.equal :: 'a option \Rightarrow 'a option \Rightarrow bool \to
(Haskell) infix 4 ==

code-reserved SML
option NONE SOME
code-reserved OCaml
option None Some
code-reserved Scala
Option None Some

end

51 Partial Function Definitions

theory Partial-Function
  imports Complete-Partial-Order Option
  keywords partial-function :: thy-defn
begin

named-theorems partial-function-mono monotonicity rules for partial function definitions
ML-file ⟨Tools/Function/partial-function.ML⟩

lemma (in ccpo) in-chain-finite:
  assumes Complete-Partial-Order.chain (\leq) A finite A A \neq {}
shows ⨆ A ∈ A
using assms(2,1,3)
proof induction
  case empty thus ?case by simp
next
  case (insert x A)
  note chain = Complete-Partial-Order.chain (≤) (insert x A)
  show ?case
    proof (cases A = {})
      case True thus ?thesis by simp
    next
      case False
      from chain have chain': Complete-Partial-Order.chain (≤) A
      by (rule chain-subset) blast
      hence ⨆ A ∈ A using False by (rule insert.IH)
      show ?thesis
        proof (cases x ≤ ⨆ A)
          case True
          have ⨆ (insert x A) ≤ ⨆ A using chain
          by (rule ccpo-Sup-least)(auto simp add: True intro: ccpo-Sup-upper[OF chain'])
          hence ⨆ (insert x A) = ⨆ A
          by (rule order.antisym)(blast intro: ccpo-Sup-upper[OF chain] ccpo-Sup-least[OF chain'])
        next
          case False
          with ⨆ A ∈ A show ?thesis by simp
        qed
      qed
    qed
lemma (in ccpo) admissible-chfin:
(∀ S. Complete-Partial-Order.chain (≤) S → finite S)
  → ccpo.admissible Sup (≤) P
using in-chain-finite by (blast intro: ccpo.admissibleI)

51.1 Axiomatic setup

This techical locale constains the requirements for function definitions with ccpo fixed points.
definition fun-ord ord f g ←→ (∀ x. ord (f x) (g x))
definition fun-lub L A = (λx. L { g. ∃ f ∈ A. y = f x})
definition img-ord f ord = (λx y. ord (f x) (f y))
definition img-lub f g Lub = (λA. g (Lub (f ∘ A)))
lemma chain-fun:
  assumes A: chain (fun ord ord) A
  shows chain ord { y. ∃ f ∈ A. y = f a } (is chain ord ?C)
proof (rule chainI)
  fix x y assume x ∈ ?C y ∈ ?C
  then obtain f g where fg: f ∈ A g ∈ A
  and [simp]: x = f a y = g a by blast
  from chainD[OF A fg]
  show ord x y ∨ ord y x unfolding fun-ord-def by auto
qed

lemma call-mono[partial-function-mono]: monotone (fun ord ord) ord (λf. f t)
by (rule monotoneI) (auto simp: fun-ord-def)

lemma let-mono[partial-function-mono]:
  (∀ x. monotone orda ordb (λf. b f x))
  ⇒ monotone orda ordb (λf. Let t (b f))
by (simp add: Let-def)

lemma if-mono[partial-function-mono]: monotone orda ordb F
  ⇒ monotone orda ordb G
  ⇒ monotone orda ordb (λf. if c then F f else G f)
unfolding monotone-def by simp

definition mk-less R = (λx y. R x y ∧ ¬ R y x)

locale partial-function-definitions =
  fixes leq :: 'a ⇒ 'a ⇒ bool
  fixes lub :: 'a set ⇒ 'a
  assumes leq-refl: leq x x
  assumes leq-trans: leq x y ⇒ leq y z ⇒ leq x z
  assumes leq-antisym: leq x y ⇒ leq y x ⇒ x = y
  assumes lub-upper: chain leq A ⇒ x ∈ A ⇒ leq x (lub A)
  assumes lub-least: chain leq A ⇒ (∀x. x ∈ A ⇒ leq x z) ⇒ leq (lub A) z

lemma partial-function-lift:
  assumes partial-function-definitions ord lb
  shows partial-function-definitions (fun ord ord) (fun-lub lb) (is partial-function-definitions ?ordf ?lubf)
proof –
  interpret partial-function-definitions ord lb by fact
  show ?thesis
  proof
    fix x show ?ordf x x
    unfolding fun-ord-def by (auto simp: leq-refl)
  next
    fix x y z assume ?ordf x y ?ordf y z
thus \( \text{?ordf } x \ z \) unfolding \text{fun-ord-def}
by (force dest: \text{leq-trans})

next
fix \( x \ y \) assume \( \text{?ordf } x \ y \) \( \text{?ordf } y \ x \)
thus \( x = y \) unfolding \text{fun-ord-def}
by (force intro!: dest: \text{leq-antisym})

next
fix \( A \ f \) assume \( f \in A \) and \( A: \text{chain } \text{?ordf } A \)
thus \( \text{?ordf } f (\text{lubf } A) \)
unfolding \text{fun-lub-def fun-ord-def}
by (blast intro: lub-upper chain-fun[OF A ] f)

next
fix \( A :: (\text{'}b \Rightarrow \text{'}a) \) set and \( g :: (\text{'}b \Rightarrow \text{'}a) \)
assume \( A: \text{chain } \text{?ordf } A \) and \( g: \forall f. f \in A \Rightarrow \text{?ordf } f g \)
show \( \text{?ordf } (\text{lubf } A) g \) unfolding \text{fun-lub-def fun-ord-def}
by (blast intro: lub-least chain-fun[OF A ] dest: g[unfolded fun-ord-def])

qed

qed

lemma \text{ccpo: assumes partial-function-definitions ord lb}
shows class.ccpo lb ord (\text{mk-less ord})
using assms unfolding partial-function-definitions-def mk-less-def
by unfold-locales blast+

lemma partial-function-image:
assumes partial-function-definitions ord \text{Lub}
assumes \text{inj}: \( \forall x \ y. f \ x = f \ y \Rightarrow x = y \)
assumes \text{inv}: \( \forall x. f \ (g \ x) = x \)
shows partial-function-definitions (\text{img-ord } f \ ord) (\text{img-lub } f \ g \ \text{Lub})

proof –
let \( ?iord = \text{img-ord } f \ ord \)
let \( ?ilub = \text{img-lub } f \ g \ \text{Lub} \)

interpret partial-function-definitions ord \text{Lub} by fact
show \( ?thesis \)
proof
fix \( A \ x \) assume \( \text{chain } ?iord A \ x \in A \)
then have \( \text{chain ord } (f \ ' A) f \ x \in f \ ' A \)
by (auto simp: \text{img-ord-def intro: chainI dest: chainD})
thus \( ?iord x (\text{lubf } A) \)
unfolding \text{inv img-lub-def img-ord-def by (rule lub-upper)}

next
fix \( A \ x \) assume \( \text{chain } ?iord A \)
and \( \text{1: } \forall z. z \in A \Rightarrow ?iord z x \)
then have \( \text{chain ord } (f \ ' A) \)
by (auto simp: \text{img-ord-def intro: chainI dest: chainD})
thus \( ?iord (\text{lubf } A) x \)
unfolding \text{inv img-lub-def img-ord-def}
by (rule lub-least) (auto dest: 1[unfolded img-ord-def])

Theory "Partial-Function"

qed (auto simp: img-ord-def intro: leq-refl dest: leq-trans leq-antisym inj)

context partial-function-definitions begin

abbreviation le-fun ≡ fun-ord leq
abbreviation lub-fun ≡ fun-lub lub
abbreviation fixp-fun ≡ ccpo.fixp lub-fun leq
abbreviation mono-body ≡ monotone le-fun leq
abbreviation admissible ≡ ccpo.admissible lub-fun leq

Interpret manually, to avoid flooding everything with facts about orders

lemma ccpo: class.ccpo lub-fun le-fun (mk-less le-fun)
apply (rule ccpo)
apply (rule partial-function-lift)
apply (rule partial-function-definitions-axioms)
done

The crucial fixed-point theorem

lemma mono-body-fixp:
(∀x. mono-body (λf. F f x)) → fixp-fun F = F (fixp-fun F)
by (rule ccpo.fixp-unfold[OF ccpo]) (auto simp: monotone-def fun-ord-def)

Version with curry/uncurry combinators, to be used by package

lemma fixp-rule-uc:
fixes F :: 'c ⇒ 'c and
U :: 'c ⇒ 'b ⇒ 'a and
C :: ('b ⇒ 'a) ⇒ 'c
assumes mono: ∀x. mono-body (λf. U (F (C f)) x)
assumes eq: f ≡ C (fixp-fun (λf. U (F (C f))))
assumes inverse: ∃f. C (U f) = f
shows f = F f
proof
  have f = C (fixp-fun (λf. U (F (C f)))) by (simp add: eq)
also have ... = C (U (F (C (fixp-fun (λf. U (F (C f)))))))
    by (subst mono-body-fixp[of %f. U (F (C f))], OF mono) (rule refl)
also have ... = F (C (fixp-fun (λf. U (F (C f))))) by (rule inverse)
also have ... = F f by (simp add: eq)
finally show f = F f.
qed

Fixpoint induction rule

lemma fixp-induct-uc:
fixes F :: 'c ⇒ 'c
  and U :: 'c ⇒ 'b ⇒ 'a
  and C :: ('b ⇒ 'a) ⇒ 'c
  and P :: ('b ⇒ 'a) ⇒ bool
assumes \( \text{mono} \): \( \bigwedge x. \text{mono-body} (\lambda f. U (F (C f)) x) \)

and \( \text{eq} \): \( f \equiv C (\expfun (\lambda f. U (F (C f)))) \)

and \( \text{inverse} \): \( \bigwedge f. U (C f) = f \)

and \( \text{adm} \): \( \text{ccpo.admissible} \ \text{lub-fun le-fun} \ P \)

and \( \text{bot} \): \( P (\lambda -. \text{lub} \{\}) \)

and \( \text{step} \): \( \bigwedge f. P (U f) \implies P (U (F f)) \)

shows \( P (U f) \)

unfolding \( \text{eq} \) \( \text{inverse} \)

proof (rule ccpo.fixp-induct[OF ccpo adm])

show \( \text{monotone le-fun leq} (\lambda f. U (F (C f))) \)

using \( \text{mono} \) by (auto simp: monotone-def fun-ord-def)

next

show \( P (\text{lub-fun} \{\}) \)

by (auto simp: bot fun-lub-def)

next

fix \( x \)

assume \( P x \)

then show \( P (U (F (C x))) \)

using \( \text{step} [\text{of } C x] \) by (simp add: \( \text{inverse} \))

qed

Rules for \( \text{monotone le-fun leq} \):

lemma \( \text{const-mono[partial-function-mono]}: \text{monotone ord leq} (\lambda f. c) \)

by (rule monotoneI) (rule leq-refl)

end

51.2 Flat interpretation: tailrec and option

definition \( \text{flat-ord} b x y \iff x = b \lor x = y \)

definition \( \text{flat-lub} b A = (\text{if } A \subseteq \{b\} \text{ then } b \text{ else } (\text{T} x. x \in A - \{b\})) \)

lemma \( \text{flat-interpretation:} \)

\( \text{partial-function-definitions} (\text{flat-ord} b) (\text{flat-lub} b) \)

proof

fix \( A x \) assume \( 1: \text{chain} (\text{flat-ord} b) A x \in A \)

show \( \text{flat-ord} b x (\text{flat-lub} b A) \)

proof cases

assume \( x = b \)

thus \( \text{thesis} \) by (simp add: \( \text{flat-ord-def} \))

next

assume \( x \neq b \)

with \( 1 \) have \( A - \{b\} = \{x\} \)

by (auto elim: chainE simp: flat-ord-def)

then have \( \text{flat-lub} b A = x \)

by (auto simp: flat-lub-def)
thus thesis by (auto simp: flat-ord-def)
qed

next

fix A z assume A: chain (flat-ord b) A
and z: \( \forall x, x \in A \Rightarrow flat-ord b x z \)

show \( flat-ord b (flat-lub b A) z \)

proof cases

  assume A \subseteq \{ b \}
  thus thesis by (auto simp: flat-lub-def flat-ord-def)

next

  assume nb: \( \neg A \subseteq \{ b \} \)
  then obtain y where y: y \in A y \neq b by auto
  with A have A - \{ b \} = \{ y \}
    by (auto elim: chainE simp: flat-ord-def)
  with nb have flat-lub b A = y
    by (auto simp: flat-lub-def)
  with z y show thesis by auto

qed

qed (auto simp: flat-ord-def)

lemma flat-ordI: \( x \neq a \Rightarrow x = y \) \Rightarrow flat-ord a x y
by (auto simp add: flat-ord-def)

lemma flat-ord-antisym: \[ flat-ord a x y; flat-ord a y x \] \Rightarrow x = y
by (auto simp add: flat-ord-def)

lemma antisym-flat-ord: antisym (flat-ord a)
by (rule antisymI)(auto dest: flat-ord-antisym)

interpretation tailrec:
  partial-function-definitions flat-ord undefined flat-lub undefined
rewrites flat-lub undefined {} \equiv undefined
by (rule flat-interprettion)(simp add: flat-lub-def)

interpretation option:
  partial-function-definitions flat-ord None flat-lub None
rewrites flat-lub None {} \equiv None
by (rule flat-interprettion)(simp add: flat-lub-def)

abbreviation tailrec-ord \equiv flat-ord undefined
abbreviation mono-tailrec \equiv monotone (fun-ord tailrec-ord) tailrec-ord

lemma tailrec-admissible:
ccpo.admissible (fun-lub (flat-lub c)) (fun-ord (flat-ord c))
(\( \lambda a. \forall x. a x \neq c \rightarrow P x (a x) \))
proof (intro ccpo.admissibleI strip)
fix A x
assume chain: Complete-Partial-Order.chain (fun-ord (flat-ord c)) A
and P [rule-formal]: ∀f ∈ A. ∀x. f x ≠ c → P x (f x)
and defined: fun-lub (flat-lub c) A x ≠ c
from defined obtain f where f: f ∈ A f x ≠ c
by (auto simp add: fun-lub-def flat-lub-def split: if-split-asm)
hence P x (f x) by (rule P)
moreover from chain f have ∀f' ∈ A. f' x = c ∨ f' x = f x
by (auto 4 4 simp add: Complete-Partial-Order.chain-def flat-lub-def fun-ord-def)
hence fun-lub (flat-lub c) A x = f x
using f by (auto simp add: fun-lub-def flat-lub-def)
ultimately show P x (fun-lub (flat-lub c) A x) by simp
qed

lemma fixp-induct-tailrec:
fixes F :: 'c ⇒ 'c and
U :: 'c ⇒ 'b ⇒ 'a and
C :: ('b ⇒ 'a) ⇒ 'c and
P :: 'b ⇒ 'a ⇒ bool and
x :: 'b
assumes mono: ∀x. monotone (fun-ord (flat-ord c)) (fun-ord (flat-ord c)) (λf. U (F (C f)) x)
assumes eq: f ≡ C (ccpo.fixp (fun-lub (flat-lub c)) (fun-ord (flat-ord c))) (λf. U (F (C f))))
assumes inverse2: ∀f. U (C f) = f
assumes step: ∀f x y. (λx y. U f x = y → y ≠ c → P x y) (λx y. U f x = y → y ≠ c → P x y)
assumes result: U f x = y
assumes defined: y ≠ c
shows P x y
proof –
have ∀x y. U f x = y → y ≠ c → P x y
by (rule partial-function-definitions,fixp-induct-ac[OF flat-interpretation, OF U F C, OF mono eq inverse2])
thus ?thesis using result defined by blast
qed

lemma admissible-image:
assumes pfun: partial-function-definitions le lub
assumes adn: ccpo.admissible lub le (P ∘ g)
assumes inj: ∀x y. f x = f y → x = y
assumes inv: ∀x. f (g x) = x
shows ccpo.admissible (img-lub f g lub) (img-ord f le) P
proof (rule ccpo.admissibleI)
fix A assume chain (img-ord f le) A
then have ch': chain le (f' A)
by (auto simp: img-ord-def intro: chainI dest: chainD)
assume A ≠ {}
assume P-A: ∀x ∈ A. P x
have $(P \circ g) (\operatorname{lub} f \cdot A)$ using adm ch'
proof (rule ccpo.admissibleD)
  fix $x$ assume $x \in f \cdot A$
  with $P$-A show $(P \circ g) x$ by (auto simp: inj[OF inv])
qed (simp add: $\cdot A \neq \{\})$
thus $P (\operatorname{img-lub} f g \operatorname{lub} A)$ unfolding img-lub-def by simp
qed

lemma admissible-fun:
assumes pfun: partial-function-definitions le lub
assumes adm: $\forall x. \operatorname{ccpo} \cdot \operatorname{admissible} (\operatorname{lub le}) (Q x)$
shows ccpo.admissible (fun-lub lub) (fun-ord le) ($\lambda f. \forall x. Q x (f x)$)
proof (rule ccpo.admissibleI)
  fix $A :: (\Rightarrow a \Rightarrow b)$ set
  assume Q: $\forall f \in A. \forall x. Q x (f x)$
  assume ch: chain (fun-ord le) $A$
  assume $A \neq \{\}$
  hence non-empty: $\forall a. \{y. \exists f \in A. y = f a\} \neq \{\}$ by auto
  show $\forall x. Q x (\operatorname{fun-lub} lub A x)$ unfolding fun-lub-def
    by (rule allI, rule ccpo.admissibleD[OF adm chain-fun[OF ch] non-empty])
    (auto simp: Q)
qed

abbreviation option-ord ≡ flat-ord None
abbreviation mono-option ≡ monotone (fun-ord option-ord) option-ord

lemma bind-mono[partial-function-mono]:
assumes mf: mono-option B and mg: $\forall y. \operatorname{mono-option} (\lambda f. C y f)$
shows $\operatorname{mono-option} (\lambda f. \operatorname{Option.bind} (B f) (\lambda y. C y f))$
proof (rule monotoneI)
  fix $f g :: \Rightarrow a \Rightarrow b$ option assume fg: fun-ord option-ord f g
  with mf
  have option-ord $(B f) (B g)$ by (rule monotoneD[of - - f g])
  then have option-ord $(\operatorname{Option.bind} (B f) (\lambda y. C y f)) (\operatorname{Option.bind} (B g) (\lambda y. C y f))$
    unfolding flat-ord-def by auto
  also from mg
  have $\forall y'. \operatorname{option-ord} (C y' f) (C y' g)$
    by (rule monotoneD) (rule fg)
  then have option-ord $(\operatorname{Option.bind} (B g) (\lambda y'. C y' f)) (\operatorname{Option.bind} (B g) (\lambda y'. C y' g))$
    unfolding flat-ord-def by (cases B g) auto
  finally (option.leq-trans)
  show option-ord $(\operatorname{Option.bind} (B f) (\lambda y. C y f)) (\operatorname{Option.bind} (B g) (\lambda y'. C y' g))$.
qed
lemma flat-lub-in-chain:
  assumes ch: chain (flat-ord b) A
  assumes lub: flat-lub b A = a
  shows a = b ∨ a ∈ A
proof (cases A ⊆ {b})
  case True
  then have flat-lub b A = b unfolding flat-lub-def by simp
  with lub show ?thesis by simp
next
case False
  then obtain c where c ∈ A and c ≠ b by auto
  { fix z assume z ∈ A
    from chainD[OF ch ‹c ∈ A› this] have z = c ∨ z = b
      unfolding flat-ord-def using ‹c ≠ b› by auto }
  with False have A - {b} = {c} by auto
  with False have flat-lub b A = c by (auto simp: flat-lub-def)
  with ‹c ∈ A› lub show ?thesis by simp
qed

lemma option-admissible: option.admissible (%(f::'a ⇒ 'b option).
  (∀ x y. f x = Some y → P x y))
proof (rule ccpo.admissibleI)
  fix A :: ('a ⇒ 'b option) set
  assume ch: chain option.le-fun A
  and IH: ∀f∈A. ∀ x y. f x = Some y → P x y
from ch have ch': ∀x. chain option-ord {y. ∃f∈A. y = f x} by (rule chain-fun)
show ∀ x y. option.lub-fun A x = Some y → P x y
proof (intro allI impI)
  fix x y assume option.lub-fun A x = Some y
  from flat-lub-in-chain[of ‹x. chain option-ord {y. ∃f∈A. y = f x}› unfolding fun-lub-def]
  have Some y ∈ {y. ∃f∈A. y = f x} by simp
  then have ∃f∈A. f x = Some y by auto
  with IH show P x y by auto
qed

lemma fixp-induct-option:
fixes F :: 'c ⇒ 'c and
  U :: 'c ⇒ 'b ⇒ 'a option and
  C :: ('b ⇒ 'a option) ⇒ 'c and
  P :: 'b ⇒ 'a ⇒ bool
assumes mono: ∀x. mono-option (λf. U (F (C f))) x
assumes eq: f ≡ C (ccpo.fixp (fun-lub (flat-lub None)) (fun-ord-option-ord) (λf. U (F (C f))))
assumes inverse2: ∃f. U (C f) = f
assumes step: ∀f x y. (∀x. U f x = Some y → P x y) → U (F f) x = Some y → P x y
assumes defined: U f x = Some y
shows P x y
using step defined option.fixp-induct-uc[of U F C, OF mono eq inverse2 option-admissible]
unfolding fun-lub-def flat-lub-def by(auto 9 2)
declaration (Partial-Function.init tailrec term (tailrec.fixp-fun)
term (tailrec.mono-body) @(thm tailrec.fixp-rule-uc) @(thm tailrec.fixp-induct-uc)
(SOME @(thm fixp-induct-tailrec[where c = undefined])))
declaration (Partial-Function.init option term (option.fixp-fun)
term (option.mono-body) @(thm option.fixp-rule-uc) @(thm option.fixp-induct-uc)
(SOME @(thm fixp-induct-option))
hide-const (open) chain
end

theory Argo
imports HOL
begin

ML-file (∼/src/Tools/Argo/rego-twist.ML)
ML-file (∼/src/Tools/Argo/rego-expr.ML)
ML-file (∼/src/Tools/Argo/rego-term.ML)
ML-file (∼/src/Tools/Argo/rego-lit.ML)
ML-file (∼/src/Tools/Argo/rego-proof.ML)
ML-file (∼/src/Tools/Argo/rego-rewr.ML)
ML-file (∼/src/Tools/Argo/rego-rewr.ML)
ML-file (∼/src/Tools/Argo/rego-clas.ML)
ML-file (∼/src/Tools/Argo/rego-common.ML)
ML-file (∼/src/Tools/Argo/rego-cc.ML)
ML-file (∼/src/Tools/Argo/rego-simplex.ML)
ML-file (∼/src/Tools/Argo/rego-thy.ML)
ML-file (∼/src/Tools/Argo/rego-heap.ML)
ML-file (∼/src/Tools/Argo/rego-cdcl.ML)
ML-file (∼/src/Tools/Argo/rego-core.ML)
ML-file (∼/src/Tools/Argo/rego-clausify.ML)
ML-file (∼/src/Tools/Argo/rego-solver.ML)

ML-file (Tools/Argo/rego-tactic.ML)
end

52 Reconstructing external resolution proofs for propositional logic

theory SAT
imports Argo
begin
ML-file ⟨Tools/prop-logic.ML⟩
ML-file ⟨Tools/sat-solver.ML⟩
ML-file ⟨Tools/sat.ML⟩
ML-file ⟨Tools/Argo/argo-sat-solver.ML⟩

**method-setup** sat = ⟨Scan.succeed (SIMPLE-METHOD' o SAT.sat-tac)⟩
SAT solver

**method-setup** satx = ⟨Scan.succeed (SIMPLE-METHOD' o SAT.satx-tac)⟩
SAT solver (with definitional CNF)

end

## 53 Function Definitions and Termination Proofs

**theory** Fun-Def
  imports Basic-BNF-LFPs Partial-Function SAT
  keywords
    function termination :: thy-goal-defn and
    fun fun-cases :: thy-defn
begin

### 53.1 Definitions with default value

**definition** THE-default :: 'a ⇒ ('a ⇒ bool) ⇒ 'a
  where THE-default d P = (if (∃!x. P x) then (THE x. P x) else d)

**lemma** THE-defaultI: ∃!x. P x ⇒ P (THE-default d P)
  by (simp add: theI' THE-default-def)

**lemma** THE-default1-equality: ∃!x. P x ⇒ P a ⇒ THE-default d P = a
  by (simp add: the1-equality THE-default-def)

**lemma** THE-default-none: ¬ (∃!x. P x) ⇒ THE-default d P = d
  by (simp add: THE-default-def)

**lemma** fundef-ex1-existence:
  assumes f-def: f ≡ (λx::'a. THE-default (d x) (λy. G x y))
  assumes ex1: ∃!y. G x y
  shows G x (f x)
  apply (simp only: f-def)
  apply (rule THE-defaultI')
  apply (rule ex1)
  done

**lemma** fundef-ex1-uniqueness:
  assumes f-def: f ≡ (λx::'a. THE-default (d x) (λy. G x y))
  assumes ex1: ∃!y. G x y
assumes \( \text{elm} : G x (h x) \)
shows \( h x = f x \)
by (auto simp add: f-def ex1 \( \text{elm} \) THE-default1-equality[symmetric])

lemma fundef-ex1-iff:
assumes \( \text{f-def} : f \equiv (\lambda x::'a. \text{THE-def}\text{ault} (d x) (\lambda y. G x y)) \)
assumes ex1: \( \exists! y. G x y \)
shows \( (G x y) = (f x = y) \)
by (auto simp add: ex1 f-def THE-default1-equality THE-defaultI)

lemma fundef-default-value:
assumes \( \text{f-def} : f \equiv (\lambda x::'a. \text{THE-def}\text{ault} (d x) (\lambda y. G x y)) \)
assumes graph: \( \forall x y. G x y \implies D x \)
assumes \( \neg D x \)
shows \( f x = d x \)
proof
  have \( \neg (\exists y. G x y) \)
  proof
    assume \( \exists y. G x y \)
    then have \( D x \) using graph ..
    with \( \neg D x \) show False ..
  qed
  then have \( \neg (\exists! y. G x y) \) by blast
  then show \( \text{thesis} \)
    unfolding \( \text{f-def} \) by (rule THE-default-none)
  qed

definition in-rel-def[simp]: \( \text{in-rel R x y} \equiv (x, y) \in R \)

lemma wf-in-rel: \( \text{wf R} \implies \text{wfP} (\text{in-rel R}) \)
by (simp add: \( \text{wfP-def} \))

ML-file (Tools/Function/function-core.ML)
ML-file (Tools/Function/mutual.ML)
ML-file (Tools/Function/pattern-split.ML)
ML-file (Tools/Function/relation.ML)
ML-file (Tools/Function/function-elims.ML)

method-setup relation = ';
  Args.term >> (fn t => fn ctxt => SIMPLE-METHOD' (Function-Relation.relation-infer-tac ctxt t))
  > prove termination using a user-specified wellfounded relation

ML-file (Tools/Function/function.ML)
ML-file (Tools/Function/pat-completeness.ML)

method-setup pat-completeness = ';
  Scan.succeed (SIMPLE-METHOD' o Pat-Completeness.pat-completeness-tac)
prove completeness of (co)datatype patterns

ML-file ⟨Tools/Function/fun.ML⟩
ML-file ⟨Tools/Function/induction-schema.ML⟩

method-setup induction-schema = ⟨
  Scan.succeed (CONTEXT-TACTIC oo Induction-Schema.induction-schema-tac)
⟩
prove an induction principle

53.2 Measure functions

inductive is-measure :: ('a ⇒ nat) ⇒ bool
  where is-measure-trivial: is-measure f

named-theorems measure-function rules that guide the heuristic generation of measure functions
ML-file ⟨Tools/Function/measure-functions.ML⟩

lemma measure-size[measure-function]: is-measure size
  by (rule is-measure-trivial)

lemma measure-fst[measure-function]: is-measure f ⇒ is-measure (λp. f (fst p))
  by (rule is-measure-trivial)

lemma measure-snd[measure-function]: is-measure f ⇒ is-measure (λp. f (snd p))
  by (rule is-measure-trivial)

ML-file ⟨Tools/Function/lexicographic-order.ML⟩

method-setup lexicographic-order = ⟨
  Method.sections clasimp-modifiers >>
  (K (SIMPLE-METHOD o Lexicographic-Order.lexicographic-order-tac false))
⟩
termination prover for lexicographic orderings

53.3 Congruence rules

lemma let-cong [fundef-cong]: M = N ⇒ (∀x. x = N ⇒ f x = g x) ⇒ Let M f = Let N g
  unfolding Let-def by blast

lemmas [fundef-cong] =
  if-cong image-cong
  bex-cong ball-cong imp-cong map-option-cong Option.bind-cong

lemma split-cong [fundef-cong]:
  (∀x y. (x, y) = q ⇒ f x y = g x y) ⇒ p = q ⇒ case-prod f p = case-prod g q
  by (auto simp: split-def)

lemma comp-cong [fundef-cong]: f (g x) = f' (g' x') ⇒ (f o g) x = (f' o g') x'
by (simp only: o-apply)

53.4 Simp rules for termination proofs

declare
  trans-less-add1[termination-simp]
  trans-less-add2[termination-simp]
  trans-le-add1[termination-simp]
  trans-le-add2[termination-simp]
  less-imp-le-nat[termination-simp]
  le-imp-less-Suc[termination-simp]

lemma size-prod-simp[termination-simp]:
  size-prod f g p = f (fst p) + g (snd p) + Suc 0
by (induct p) auto

53.5 Decomposition

lemma less-by-empty:
  A = {} \implies A \subseteq B
and union-comp-emptyL:
  A O C = {} \implies B O C = {} \implies (A \cup B) O C = {}
and union-comp-emptyR:
  A O B = {} \implies A O C = {} \implies A O (B \cup C) = {}
and wf-no-loop:
  R O R = {} \implies wf R
by (auto simp add: wf-comp-self[of R])

53.6 Reduction pairs

definition reduction-pair P \longleftrightarrow wf (fst P) \land fst P O snd P \subseteq fst P

lemma reduction-pairI[intro]:
  wf R \implies R O S \subseteq R \implies reduction-pair (R, S)
by (auto simp: reduction-pair-def)

lemma reduction-pair-lemma:
  assumes rp: reduction-pair P
  assumes R \subseteq fst P
  assumes S \subseteq snd P
  assumes wf S
  shows wf (R \cup S)
proof -
  from rp \langle S \subseteq snd P \rangle have (fst P) fst P O S \subseteq fst P
    unfolding reduction-pair-def by auto
  with \langle wf S \rangle have (fst P \cup S)
    by (auto intro: wf-union-compatible)
  moreover from \langle R \subseteq fst P \rangle have R \cup S \subseteq fst P \cup S by auto
  ultimately show ?thesis by (rule wf-subset)
qed

definition rp-inv-image = (\lambda (R, S) f. (inv-image R f, inv-image S f))

lemma rp-inv-image-rp: reduction-pair P \implies reduction-pair (rp-inv-image P f)
unfolding reduction-pair-def rp-inv-image-def split-def by force
53.7 Concrete orders for SCNP termination proofs

**Definition**

- **pair-less** = less-than \(*\text{lex}\) less-than
- **pair-leq** = pair-less
- **max-strict** = max-ext pair-less
- **max-weak** = max-ext pair-leq ∪ \{({}, {})\}
- **min-strict** = min-ext pair-less
- **min-weak** = min-ext pair-leq ∪ \{({}, {})\}

**Lemma**  

- **wf-pair-less**  
  \[\text{simp} : \text{wf pair-less}\]
  by (auto simp: pair-less-def)

- **total-pair-less**  
  \[\text{iff} : \text{total-on } A \text{ pair-less and trans-pair-less} \text{ iff} : \text{trans pair-less}\]
  by (auto simp: total-on-def pair-less-def)

**Introduction rules for pair-less/pair-leq**

- **Lemma** **pair-leqI1**: \(a < b \Rightarrow ((a, s), (b, t)) \in \text{pair-leq}\)
  and **pair-leqI2**: \(a \leq b \Rightarrow s \leq t \Rightarrow ((a, s), (b, t)) \in \text{pair-leq}\)
  and **pair-lessI1**: \(a < b \Rightarrow ((a, s), (b, t)) \in \text{pair-less}\)
  and **pair-lessI2**: \(a \leq b \Rightarrow s < t \Rightarrow ((a, s), (b, t)) \in \text{pair-less}\)
  by (auto simp: pair-leq-def pair-less-def)

- **Lemma** **pair-less-iff1**  
  \[\text{simp} : ((x,y), (x,z)) \in \text{pair-less} \leftrightarrow y < z\]
  by (simp add: pair-less-def)

**Introduction rules for max**

- **Lemma** **smax-emptyI**: finite \(Y \neq \{\} \Rightarrow (\{\}, Y) \in \text{max-strict}\)
  and **smax-insertI**:
    \(y \in Y \Rightarrow (x, y) \in \text{pair-less} \Rightarrow (X, Y) \in \text{max-strict} \Rightarrow (\text{insert } x X, Y) \in \text{max-strict}\)
  and **wmax-emptyI**: finite \(X \Rightarrow (\{\}, X) \in \text{max-weak}\)
  and **wmax-insertI**:
    \(y \in YS \Rightarrow (x, y) \in \text{pair-leq} \Rightarrow (XS, YS) \in \text{max-weak} \Rightarrow (\text{insert } x XS, YS) \in \text{max-weak}\)
  by (auto simp: max-strict-def max-weak-def elim: max-ext_cases)

**Introduction rules for min**

- **Lemma** **smin-emptyI**: \(X \neq \{\} \Rightarrow (X, \{\}) \in \text{min-strict}\)
  and **smin-insertI**:
    \(x \in XS \Rightarrow (x, y) \in \text{pair-less} \Rightarrow (XS, YS) \in \text{min-strict} \Rightarrow (XS, \text{insert } y YS) \in \text{min-strict}\)
  and **wmin-emptyI**: \((X, \{\}) \in \text{min-weak}\)
  and **wmin-insertI**:
    \(x \in XS \Rightarrow (x, y) \in \text{pair-leq} \Rightarrow (XS, YS) \in \text{min-weak} \Rightarrow (XS, \text{insert } y YS) \in \text{min-weak}\)
  by (auto simp: min-strict-def min-weak-def min-ext-def)

**Reduction Pairs.**
lemma max-ext-compat:
  assumes R O S ⊆ R
  shows max-ext R O (max-ext S ∪ {({}, {}))) ⊆ max-ext R
proof –
  have \( \forall X Y Z. (X, Y) \in \text{max-ext } R \implies (Y, Z) \in \text{max-ext } S \implies (X, Z) \in \text{max-ext } R \)
  proof –
    fix X Y Z
    assume \((X, Y)\in \text{max-ext } R \)
    \((Y, Z)\in \text{max-ext } S \)
    then have \(*\): \text{finite } X \text{ finite } Y \text{ finite } Z \neq \emptyset \text{ } Z \neq \emptyset \)
    \((\forall x. x \in X \implies \exists y \in Y. (x, y) \in R) \)
    \((\forall y. y \in Y \implies \exists z \in Z. (y, z) \in S) \)
    by (auto elim: max-ext_cases)
    moreover have \( \forall x. z \in X \implies \exists z \in Z. (x, z) \in R \)
    proof –
      fix x
      assume \(x\in X \)
      then obtain y where \(1\): \(y \in Y. (x, y) \in R \)
      using \(*\) by auto
      then obtain z where \(z \in Z. (y, z) \in S \)
      using \(*\) by auto
      then show \(\exists z \in Z. (x, z) \in R \)
      using assms \(1\) by (auto elim: max-ext_cases)
    qed
    ultimately show \((X, Z)\in \text{max-ext } R \)
    by auto
    qed

lemma min-ext-compat:
  assumes R O S ⊆ R
  shows min-ext R O (min-ext S ∪ {({}, {}))) ⊆ min-ext R
proof –
  have \( \forall X Y Z z. \forall y \in Y. \exists x \in X. (x, y) \in R \implies \forall z \in Z. \exists y \in Y. (y, z) \in S \implies z \in Z \implies \exists x \in X. (x, z) \in R \)
  proof –
    fix X Y Z z
    assume \(*\): \(\forall y \in Y. \exists x \in X. (x, y) \in R \)
    ...
\[ \forall z \in Z. \exists y \in Y. (y, z) \in S \]
then obtain \( y' \) where 1: \( y' \in Y \) \( (y', z) \in S \)
by auto
then obtain \( x' \) where 2: \( x' \in X \) \( (x', y') \in R \)
using \( * \) by auto
show \( \exists x \in X. (x, z) \in R \)
using 1 2 assms by auto
qed
then show \( \forall \)thesis
using assms by (auto simp: min-ext-def)
qed

```ml
lemma min-rpair-set: reduction-pair (min-strict, min-weak)
  unfolding min-strict-def min-weak-def
  apply (intro reduction-pairI min-ext-wf)
  apply simp
  apply (rule min-ext-compat)
  apply (auto simp: pair-less-def pair-leq-def)
done
```

### 53.8 Yet more induction principles on the natural numbers

```ml
lemma nat-descend-induct [case-names base descend]:
  fixes P :: nat \Rightarrow bool
  assumes H1: \( \forall k. k > n \Rightarrow P k \)
  assumes H2: \( \forall k. k \leq n \Rightarrow (\forall i. i > k \Rightarrow P i) \Rightarrow P k \)
  shows P m
  using assms by induction-schema (force intro: wf-measure [of \( \lambda k. Suc n - k \)])+
```

```ml
lemma induct-nat-012 [case-names 0 1 ge2]:
  \( P 0 \Rightarrow P (Suc 0) \Rightarrow (\forall n. P n \Rightarrow P (Suc n)) \Rightarrow P n \)
proof (induction-schema, pat-completeness)
  show wf (Wellfounded.measure id)
    by simp
qed auto
```

### 53.9 Tool setup

ML-file <Tools/Function/termination.ML>
ML-file <Tools/Function/scnp-solve.ML>
ML-file <Tools/Function/scnp-reconstruct.ML>
ML-file <Tools/Function/fun-cases.ML>

ML-val — setup inactive

```
  Context.thy-context (Function-Common.set-termination-prover,
    (K (ScnpReconstruct.decomp-scnp-tac [ScnpSolve.MAX, ScnpSolve.MIN, ScnpSolve.MS])))
```

54  The Integers as Equivalence Classes over Pairs of Natural Numbers

theory Int
  imports Quotient Groups-Big Fun-Def
begin

54.1 Definition of integers as a quotient type

definition intrel :: (nat × nat) ⇒ (nat × nat) ⇒ bool
  where intrel = (λ(x, y) (u, v). x + v = u + y)

lemma intrel-iff [simp]: intrel (x, y) (u, v) ⇐⇒ x + v = u + y
  by (simp add: intrel-def)

quotient-type int = nat × nat / intrel
  morphisms Rep-Integ Abs-Integ
proof (rule equivpI)
  show reflp intrel by (auto simp: reflp-def)
  show symp intrel by (auto simp: symp-def)
  show transp intrel by (auto simp: transp-def)
qed

54.2 Integers form a commutative ring

instantiation int :: comm-ring-1
begin

lift-definition zero-int :: int is (0, 0).

lift-definition one-int :: int is (1, 0).

lift-definition plus-int :: int ⇒ int ⇒ int
  is λ(x, y). (x + u, y + v)
  by clarsimp

lift-definition uminus-int :: int ⇒ int
  is λ(x, y). (y, x)
  by clarsimp

lift-definition minus-int :: int ⇒ int ⇒ int
  is λ(x, y). (x + v, y + u)
  by clarsimp

lift-definition times-int :: int ⇒ int ⇒ int
  is λ(x, y). (x*u + g*v, x*v + g*u)
proof (unfold intrel-def, clarify)
  fix s t u v w x y z :: nat
  assume s + v = u + t and w + z = y + x
  then have (s + v) * w + (u + t) * x + u * (w + z) + v * (y + x) =
    (u + t) * w + (s + v) * x + u * (y + x) + v * (w + z)
    by simp
  then show (s * w + t * x) + (u * z + v * y) = (u * y + v * z) + (s * x + t * w)
    by (simp add: algebra-simps)
qed

instance
  by standard (transfer; clarsimp simp: algebra-simps)+

end

abbreviation int :: nat ⇒ int
  where int ≡ of-nat

lemma int-def: int n = Abs-Integ (n, 0)
  by (induct n) (simp add: zero-int.abs-eq, simp add: one-int.abs-eq plus-int.abs-eq)

lemma int-transfer [transfer-rule]:
  includes lifting-syntax
  shows rel-fun (=) pcr-int (λn. (n, 0)) int
  by (simp fun: rel-fun-def int.pcr-cr-eq cr-int-def int-def)

lemma int-diff-cases: obtains (diff) m n where z = int m − int n
  by transfer clarsimp

54.3 Integers are totally ordered

instantiation int :: linorder
begin

lift-definition less-eq-int :: int ⇒ int ⇒ bool
  is λ(x, y) (u, v). x + v ≤ u + y
  by auto

lift-definition less-int :: int ⇒ int ⇒ bool
  is λ(x, y) (u, v). x + v < u + y
  by auto

instance
  by standard (transfer, force)+

end

instantiation int :: distrib-lattice
begin

definition (inf :: int ⇒ int ⇒ int) = min

definition (sup :: int ⇒ int ⇒ int) = max

instance
  by standard (auto simp add: inf-int-def sup-int-def max-min-distrib2)
end

54.4 Ordering properties of arithmetic operations

instance int :: ordered-cancel-ab-semigroup-add
proof
  fix i j k :: int
  show i ≤ j ⇒ k + i ≤ k + j
    by transfer clarsimp
qed

Strict Monotonicity of Multiplication.

Strict, in 1st argument; proof is by induction on \( k > 0 \).

lemma zmult-zless-mono2-lemma: \( i < j \implies 0 < k \implies \text{int} \ k \ast i < \text{int} \ k \ast j \),
for \( i j :: \text{int} \)
proof (induct k)
  case 0
  then show ?case by simp
next
  case (Suc k)
  then show ?case
    by (cases k = 0) (simp-all add: distrib-right add-strict-mono)
qed

lemma zero-le-imp-eq-int:
  assumes \( k \geq (0 :: \text{int}) \)
  shows \( \exists n. \ k = \text{int} \ n \)
proof
  have \( b \leq a \implies \exists n :: \text{nat}. \ a = n + b \) for \( a b \)
    using exI[of - a - b] by simp
  with assms show ?thesis
    by transfer auto
qed

lemma zero-less-imp-eq-int:
  assumes \( k > (0 :: \text{int}) \)
  shows \( \exists n>0. \ k = \text{int} \ n \)
proof
  have \( b < a \implies \exists n :: \text{nat}. \ n>0 \land a = n + b \) for \( a b \)
    using exI[of - a - b] by simp
  with assms show ?thesis
    by transfer auto
THEORY "Int"

qed

lemma zmult-zless-mono2: i < j 0 < k  k * i < k * j
  for i j k :: int
  by (erule zero-less-imp-eq-int) (auto simp add: zmult-zless-mono2-lemma)

The integers form an ordered integral domain.

instantiation int :: linordered-idom
begin

definition zabs-def: |i::int| = (if i < 0 then −i else i)

definition zsgn-def: sgn (i::int) = (if i = 0 then 0 else if 0 < i then 1 else −1)

instance proof
  fix i j k :: int
  show i < j 0 < k  k * i < k * j
    by (rule zmult-zless-mono2)
  show |i| = (if i < 0 then −i else i)
    by (simp only: zabs-def)
  show sgn (i::int) = (if i=0 then 0 else if 0<i then 1 else −1)
    by (simp only: zsgn-def)
qed

end

instance int :: discrete-linordered-semidom
proof
  fix k l :: int
  show k < l  k + 1 ≤ l (is (?P ↔ ?Q))
  proof
    assume ?Q
    then show ?P
      by simp
  next
    assume ?P
    then have (l − k > 0)
      by simp
    with zero-less-imp-eq-int obtain n where (l − k = int n)
      by blast
    then have (n > 0)
      using (l − k > 0) by simp
    then have (n ≥ 1)
      by simp
    then have (int n ≥ int 1)
      by (rule of-nat-mono)
    with (l − k = int n) show ?Q
      by simp
\begin{document}

\section{Embedding of the Integers into any \textit{ring-1}: of-int}

\begin{code}

\textbf{theory} "Int"

\begin{code}

\begin{lemma}
\begin{proof}
\end{proof}
\end{lemma}

\begin{lemma}
\begin{proof}
\end{proof}
\end{lemma}

\begin{lemma}
\begin{proof}
\end{proof}
\end{lemma}

\end{code}

\end{document}
by transfer simp

lemma of-int-add [simp]: of-int \(w + z\) = of-int \(w\) + of-int \(z\)
by transfer (clarsimp simp add: algebra-simps)

lemma of-int-minus [simp]: of-int \((- z)\) = - (of-int \(z\))
by (transfer fixing: uminus clarsimp)

lemma of-int-diff [simp]: of-int \((w - z)\) = of-int \(w\) - of-int \(z\)
using of-int-add [of \(w - z\)] by simp

lemma of-int-mult [simp]: of-int \((w * z)\) = of-int \(w\) * of-int \(z\)
by (transfer fixing: times) (clarsimp simp add: algebra-simps)

lemma mult-of-int-commute: of-int \(x * y\) = of-int \(y * x\)
by (transfer fixing: times) (auto simp: algebra-simps mult-of-nat-commute)

Collapse nested embeddings.

lemma of-int-of-nat-eq [simp]: of-int \((\text{int }n)\) = of-nat \(n\)
by (induct \(n\)) auto

lemma of-int-numeral [simp, code-post]: of-int \((\text{numeral }k)\) = numeral \(k\)
by (simp add: of-nat-numeral [symmetric] of-int-of-nat-eq [symmetric])

lemma of-int-neg-numeral [code-post]: of-int \((- \text{numeral }k)\) = - numeral \(k\)
by simp

lemma of-int-power [simp]: of-int \((z ^ n)\) = of-int \(z ^ n\)
by (induct \(n\)) simp-all

lemma of-int-of-bool [simp]:
  of-int \((\text{of-bool }P)\) = of-bool \(P\)
by auto

end

context ring-char-0
begin

lemma of-int-eq-iff [simp]: of-int \(w\) = of-int \(z\) \iff \(w = z\)

Special cases where either operand is zero.

lemma of-int-eq-0-iff [simp]: of-int \(z\) = 0 \iff z = 0
using of-int-eq-iff [of \(z\) 0] by simp

lemma of-int-0-eq-iff [simp]: 0 = of-int \(z\) \iff z = 0
using of-int-eq-iff [of 0 \(z\)] by simp
lemma of-int-eq-1-iff [iff]: of-int z = 1 ←→ z = 1 
using of-int-eq-iff [of z 1] by simp

lemma numeral-power-eq-of-int-cancel-iff [simp]:
umeral x ^ n = of-int y ←→ \numeral x ^ n = y
using of-int-eq-iff [of \numeral x ^ n y, unfolded of-int-numeral of-int-power].

lemma of-int-eq-numeral-power-cancel-iff [simp]:
of-int y = \numeral x ^ n ←→ y = \numeral x ^ n
using numeral-power-eq-of-int-cancel-iff [of x n y] by (metis (mono-tags))

lemma neg-numeral-power-eq-of-int-cancel-iff [simp]:
(− \numeral x) ^ n = of-int y ←→ (− \numeral x) ^ n = y
using of-int-eq-iff [of (− \numeral x) ^ n y] by simp

lemma of-int-eq-of-int-power-cancel-iff [simp]:
(of-int b) ^ w = of-int x ←→ b ^ w = x
by (metis of-int-power of-int-eq-iff)

lemma of-int-power-eq-of-int-cancel-iff [simp]:
of-int x = (of-int b) ^ w ←→ x = b ^ w
by (metis of-int-eq-of-int-power-cancel-iff)

end

context linordered-idom
begin
Every linordered-idom has characteristic zero.
subclass ring-char-0 ..

lemma of-int-le-iff [simp]: of-int w ≤ of-int z ←→ w ≤ z
by (transfer fixing: less-eq)

lemma of-int-less-iff [simp]: of-int w < of-int z ←→ w < z
by (simp add: less-le order-less-le)

lemma of-int-0-le-iff [simp]: 0 ≤ of-int z ←→ 0 ≤ z
using of-int-le-iff [of 0 z] by simp

lemma of-int-le-0-iff [simp]: of-int z ≤ 0 ←→ z ≤ 0
using of-int-le-iff [of z 0] by simp
lemma of-int-0-less-iff [simp]: $0 < \text{of-int } z \leftrightarrow 0 < z$
using of-int-less-iff [of 0 z] by simp

lemma of-int-less-0-iff [simp]: $\text{of-int } z < 0 \leftrightarrow z < 0$
using of-int-less-iff [of z 0] by simp

lemma of-int-1-le-iff [simp]: $1 \leq \text{of-int } z \leftrightarrow 1 \leq z$
using of-int-le-iff [of 1 z] by simp

lemma of-int-le-1-iff [simp]: $\text{of-int } z \leq 1 \leftrightarrow z \leq 1$
using of-int-le-iff [of z 1] by simp

lemma of-int-1-less-iff [simp]: $1 < \text{of-int } z \leftrightarrow 1 < z$
using of-int-less-iff [of 1 z] by simp

lemma of-int-less-1-iff [simp]: $\text{of-int } z < 1 \leftrightarrow z < 1$
using of-int-less-iff [of z 1] by simp

lemma of-int-pos: $z > 0 \implies \text{of-int } z > 0$
by simp

lemma of-int-nonneg: $z \geq 0 \implies \text{of-int } z \geq 0$
by simp

lemma of-int-abs [simp]: $\text{of-int } |x| = |\text{of-int } x|$
by (auto simp add: abs-if)

lemma of-int-lessD:
assumes $|\text{of-int } n| < x$
shows $n = 0 \lor x > 1$
proof (cases $n = 0$)
case True
then show ?thesis by simp
next
case False
then have $|n| \neq 0$ by simp
then have $|n| > 0$ by simp
then have $|n| \geq 1$
using zless-imp-add1-zle [of 0 $|n|$] by simp
then have $|\text{of-int } n| \geq 1$
unfolding of-int-1-le-iff [of $|n|$, symmetric] by simp
then have $1 < x$ using assms by (rule le-less-trans)
then show ?thesis ..
qed

lemma of-int-leD:
assumes $|\text{of-int } n| \leq x$
shows $n = 0 \lor 1 \leq x$
proof (cases \( n = 0 \))
  case True
  then show \(?thesis\) by simp
next
  case False
  then have \(|n| \neq 0\) by simp
  then have \(|n| > 0\) by simp
  then have \(|n| \geq 1\)
    using \(\text{zless-imp-add1-zle \[of 0 \mid n\]}\) by simp
  then have \(|n| > 0\)
    by \(\text{simp}\)
  then have \(|n| \geq 1\)
    using \(\text{zless-imp-add1-zle \[of 0 \mid n\]}\) by simp
  then have \(|n| \geq 1\)
    by \(\text{simp}\)
  then show \(?thesis\) ..
qed

lemma numeral-power-le-of-int-cancel-iff [simp]:
  numeral \(x\) \(^\cdot\) \(n\) \(\leq\) of-int \(a\) \(\iff\) numeral \(x\) \(^\cdot\) \(n\) \(\leq\) \(a\)
  by \(\text{metis \{mono-tags\} local.of-int-eq-numeral-power-cancel-iff of-int-le-iff}\)

lemma of-int-le-numeral-power-cancel-iff [simp]:
  of-int \(a\) \(\leq\) numeral \(x\) \(^\cdot\) \(n\) \(\iff\) \(a\) \(\leq\) numeral \(x\) \(^\cdot\) \(n\)
  by \(\text{metis \{mono-tags\} local.of-int-eq-numeral-power-cancel-iff of-int-le-iff}\)

lemma numeral-power-less-of-int-cancel-iff [simp]:
  numeral \(x\) \(^\cdot\) \(n\) \(<\) of-int \(a\) \(\iff\) numeral \(x\) \(^\cdot\) \(n\) \(<\) \(a\)
  by \(\text{metis \{mono-tags\} local.of-int-eq-numeral-power-cancel-iff of-int-less-iff}\)

lemma of-int-less-numeral-power-cancel-iff [simp]:
  of-int \(a\) \(<\) numeral \(x\) \(^\cdot\) \(n\) \(\iff\) \(a\) \(<\) numeral \(x\) \(^\cdot\) \(n\)
  by \(\text{metis \{mono-tags\} local.of-int-eq-numeral-power-cancel-iff of-int-less-iff}\)

lemma neg-numeral-power-le-of-int-cancel-iff [simp]:
  \((-\text{numeral \(x\)})\) \(^\cdot\) \(n\) \(\leq\) of-int \(a\) \(\iff\) \((-\text{numeral \(x\)})\) \(^\cdot\) \(n\) \(\leq\) \(a\)
  by \(\text{metis \{mono-tags\} of-int-le-iff of-int-neg-numeral of-int-power}\)

lemma of-int-le-neg-numeral-power-cancel-iff [simp]:
  of-int \(a\) \(\leq\) \((-\text{numeral \(x\)})\) \(^\cdot\) \(n\) \(\iff\) \(a\) \(\leq\) \((-\text{numeral \(x\)})\) \(^\cdot\) \(n\)
  by \(\text{metis \{mono-tags\} of-int-le-iff of-int-neg-numeral of-int-power}\)

lemma neg-numeral-power-less-of-int-cancel-iff [simp]:
  \((-\text{numeral \(x\)})\) \(^\cdot\) \(n\) \(<\) of-int \(a\) \(\iff\) \((-\text{numeral \(x\)})\) \(^\cdot\) \(n\) \(<\) \(a\)
  using \(\text{of-int-less-iff[of \((-\text{numeral \(x\)})\) \(^\cdot\) \(n\) \(a\)]}\)
  by \(\text{simp}\)

lemma of-int-less-neg-numeral-power-cancel-iff [simp]:
  of-int \(a\) \(<\) \((-\text{numeral \(x\)})\) \(^\cdot\) \(n\) \(\iff\) \(a\) \(<\) \((-\text{numeral \(x::int\)})\) \(^\cdot\) \(n\)
  using \(\text{of-int-less-iff[of \((-\text{numeral \(x\)})\) \(^\cdot\) \(n\)]}\)
  by \(\text{simp}\)
THEORY "Int"

lemma (simp): (of-int b) \(^w\) \(\leq\) of-int x \(\iff\) b \(^w\) \(\leq\) x
    by (metis (mono-tags) of-int-le-iff of-int-power)

lemma (simp): of-int x \(\leq\) (of-int b) \(^w\) \(\iff\) x \(\leq\) b \(^w\)
    by (metis (mono-tags) of-int-le-iff of-int-power)

lemma (simp): (of-int b) \(^w\) < of-int x \(\iff\) b \(^w\) < x
    by (metis (mono-tags) of-int-less-iff of-int-power)

lemma (simp): of-int x < (of-int b) \(^w\) \(\iff\) x < b \(^w\)
    by (metis (mono-tags) of-int-less-iff of-int-power)

lemma (iff): of-int (max x y) = max (of-int x) (of-int y)
    by (auto simp: max-def)

lemma (simp): of-int (min x y) = min (of-int x) (of-int y)
    by (auto simp: min-def)

end

category division-ring
begin

lemmas mult-inverse-of-int-commute =
    mult-commute-imp-mult-inverse-commute[OF mult-of-int-commute]

end

Comparisons involving of-int.

lemma (iff): of-int z = (numeral n :: 'a::ring-char-0) \(\iff\) z
    = numeral n
    using of-int-eq-iff by fastforce

lemma (simp): of-int z \(\leq\) (numeral n :: 'a::linordered-idom) \(\iff\) z \(\leq\) numeral n
    using of-int-le-iff by simp

lemma (simp): (numeral n :: 'a::linordered-idom) \(\leq\) of-int z \(\iff\) numeral n \(\leq\) z
    using of-int-le-iff [of numeral n] by simp

lemma (simp): of-int z < (numeral n :: 'a::linordered-idom) \(\iff\) z < numeral n
    using of-int-less-iff [of numeral n] by simp
THEORY "Int"

lemma of-int-numeral-less-iff [simp]:
  (numeral n :: 'a::linordered-idom) < of-int z ↔ numeral n < z
  using of-int-less-iff [of numeral n z] by simp

lemma of-nat-less-of-int-iff: (of-nat n::'a::linordered-idom) < of-int x ↔ int n < x
  by (metis of-int-of-nat-eq of-int-less-iff)

lemma of-int-eq-id [simp]: of-int = id
proof
  show of-int z = id z for z
    by (cases z rule: int-diff-cases) simp
qed

instance int :: no-top
proof
  fix x::int
  have x < x + 1
    by simp
  then show ∃ y. x < y
    by (rule exI)
qed

instance int :: no-bot
proof
  fix x::int
  have x - 1 < x
    by simp
  then show ∃ y. y < x
    by (rule exI)
qed

54.6 Magnitude of an Integer, as a Natural Number: nat

lift-definition nat :: int ⇒ nat is λ(x, y). x - y
by auto

lemma nat-int [simp]: nat (int n) = n
by transfer simp

lemma int-nat-eq [simp]: int (nat z) = (if 0 ≤ z then z else 0)
by transfer clarsimp

lemma nat-0-le: 0 ≤ z ⟹ int (nat z) = z
by simp

lemma nat-le-0 [simp]: z ≤ 0 ⟹ nat z = 0
by transfer clarsimp
lemma nat-le-eq-zle: \( 0 < w \lor 0 \leq z \implies \text{nat } w \leq \text{nat } z \iff w \leq z \)
  by transfer (clarsimp, arith)

An alternative condition is \((\ast :: a) \leq w\).

lemma nat-mono-iff: \( 0 < z \implies \text{nat } w < \text{nat } z \iff w < z \)
  by (simp add: nat-le-eq-zle linorder-not-le [symmetric])

lemma nat-less-eq-zless: \( 0 \leq w \implies \text{nat } w < \text{nat } z \iff w < z \)
  by (simp add: nat-le-eq-zle linorder-not-le [symmetric])

lemma zless-nat-conj [simp]: \( \text{nat } w < \text{nat } z \iff 0 < z \land w < z \)
  by transfer (clarsimp, arith)

lemma nonneg-int-cases:
  assumes \( 0 \leq k \)
  obtains \( n \) where \( k = \text{int } n \)
proof –
  from assms have \( k = \text{int } (\text{nat } k) \)
    by simp
  then show \( \text{thesis} \)
    by (rule that)
qed

lemma pos-int-cases:
  assumes \( 0 < k \)
  obtains \( n \) where \( k = \text{int } n \) and \( n > 0 \)
proof –
  from assms have \( 0 \leq k \)
    by simp
  then obtain \( n \) where \( k = \text{int } n \)
    by (rule nonneg-int-cases)
  moreover have \( n > 0 \)
    using \( k = \text{int } n \) assms by simp
  ultimately show \( \text{thesis} \)
    by (rule that)
qed

lemma nonpos-int-cases:
  assumes \( k \leq 0 \)
  obtains \( n \) where \( k = -\text{int } n \)
proof –
  from assms have \( -k \geq 0 \)
    by simp
  then obtain \( n \) where \( -k = \text{int } n \)
    by (rule nonneg-int-cases)
  then have \( k = -\text{int } n \)
    by simp
  then show \( \text{thesis} \)
    by (rule that)
qed

lemma neg-int-cases:
  assumes k < 0
  obtains n where k = |int n| and n > 0
proof
  from assms have k > 0
    by simp
  then obtain n where |k| = int n and k > 0
    by (blast elim: pos-int-cases)
  then have k = |int n| and n > 0
    by simp-all
  then show thesis
    by (rule that)
qed

lemma nat-eq-iff: nat w = m <-> (if 0 <= w then w = int m else m = 0)
  by transfer (clarsimp simp add: le-imp-diff-is-add)

lemma nat-eq-iff2: m = nat w <-> (if 0 <= w then w = int m else m = 0)
  using nat-eq-iff[of w m]
  by auto

lemma nat-0 [simp]: nat 0 = 0
  by (simp add: nat-eq-iff)

lemma nat-1 [simp]: nat 1 = Suc 0
  by (simp add: nat-eq-iff)

lemma nat-numeral [simp]: nat (numeral k) = numeral k
  by (simp add: nat-eq-iff)

lemma nat-neg-numeral [simp]: nat (- numeral k) = 0
  by simp

lemma nat-2: nat 2 = Suc (Suc 0)
  by simp

lemma nat-less-iff: 0 <= w ==> nat w < m <-> w < of-nat m
  by transfer (clarsimp, arith)

lemma nat-le-iff: nat x <= n <-> x <= int n
  by transfer (clarsimp simp add: le-diff-conv)

lemma nat-mono: x <= y ==> nat x <= nat y
  by transfer auto

lemma nat-0-iff[simp]: nat i = 0 <-> i <= 0
  for i :: int
  by transfer clarsimp
lemma \textit{int-eq-iff}: of-nat \(m = z \leftrightarrow m = \text{nat } z \land 0 \leq z\) 
by (auto simp add: nat-eq-iff2)

lemma zero-less-nat-eq [simp]: \(0 < \text{nat } z \leftrightarrow 0 < z\) 
using \text{zless-nat-conj [of 0]} by auto

lemma nat-add-distrib: \(0 \leq z \Longrightarrow 0 \leq z' \Longrightarrow \text{nat } (z + z') = \text{nat } z + \text{nat } z'\) 
by transfer clarsimp

lemma nat-diff-distrib': \(0 \leq x \Longrightarrow 0 \leq y \Longrightarrow \text{nat } (x - y) = \text{nat } x - \text{nat } y\) 
by transfer clarsimp

lemma nat-diff-distrib: \(0 \leq z' \Longrightarrow z' \leq z \Longrightarrow \text{nat } (z - z') = \text{nat } z - \text{nat } z'\) 
by (rule nat-diff-distrib') auto

lemma nat-zminus-int [simp]: \(\text{nat } (- \text{int } n) = 0\) 
by transfer simp

lemma le-nat-iff: \(k \geq 0 \Longrightarrow n \leq \text{nat } k \leftrightarrow \text{int } n \leq k\) 
by transfer auto

lemma zless-nat-eq-int-zless: \(m < \text{nat } z \leftrightarrow \text{int } m < z\) 
by transfer (clarsimp simp add: less-diff-conv)

lemma \textit{(in ring-1)} of-nat-nat [simp]: \(0 \leq z \Longrightarrow \text{of-nat } (\text{nat } z) = \text{of-int } z\) 
by transfer (clarsimp simp add: of-nat-diff)

lemma diff-nat-numeral [simp]: \((\text{numeral } v :: \text{nat}) - \text{numeral } v' = \text{nat } (\text{numeral } v - \text{numeral } v')\) 
by (simp only: nat-diff-distrib' zero-le-numeral nat-numeral)

lemma nat-abs-triangle-ineq: \(\text{nat } |k + l| \leq \text{nat } |k| + \text{nat } |l|\) 
by (simp add: nat-add-distrib [symmetric] nat-le-zle abs-triangle-ineq)

lemma nat-of-bool [simp]: \(\text{nat } (\text{of-bool } P) = \text{of-bool } P\) 
by auto

lemma split-nat [linarith-split]: \(P (\text{nat } i) \leftrightarrow (\forall n. i = \text{int } n \longrightarrow P n) \land (i < 0 \longrightarrow P 0)\)
(is \(?P = (?L \land ?R)\))
for \(i :: \text{int}\)
proof (cases \(i < 0\))
  case \(\text{True}\)
  then show \(?\text{thesis}\)
    by auto
next
case False
have ?P = ?L
proof
  assume ?P
  then show ?L using False by auto
next
  assume ?L
  moreover from False have int (nat i) = i
    by (simp add: not-less)
  ultimately show ?P
    by simp
qed
  with False show ?thesis by simp
qed

lemma all-nat: (∀x. P x) ↔ (∀x≥0. P (nat x))
  by (auto split: split-nat)

lemma ex-nat: (∃x. P x) ↔ (∃x. 0 ≤ x ∧ P (nat x))
proof
  assume ∃x. P x
  then obtain x where P x ..
  then have int x ≥ 0 ∧ P (nat (int x)) by simp
  then show ∃x≥0. P (nat x) ..
next
  assume ∃x≥0. P (nat x)
  then show ∃x. P x by auto
qed

For termination proofs:

lemma measure-function-int[measure-function]: is-measure (nat ◦ abs) ..

54.7 Lemmas about the Function of-nat and Orderings

lemma negative-zless-0: ¬ (int (Suc n)) < (0 :: int)
  by (simp add: order-less-le del: of-nat-Suc)

lemma negative-zless [iff]: ¬ (int (Suc n)) < int m
  by (rule negative-zless-0 [THEN order-less-le-trans], simp)

lemma negative-zle-0: ¬ int n ≤ 0
  by (simp add: minus-le-iff)

lemma negative-zle [iff]: ¬ int n ≤ int m
  by (rule order-trans [OF negative-zle-0 of-nat-0-le-iff])

lemma not-zle-0-negative [simp]: ¬ 0 ≤ ¬ int (Suc n)
  by (subst le-minus-iff) (simp del: of-nat-Suc)
lemma int-zle-neg: \( \text{int } n \leq - \text{int } m \iff n = 0 \land m = 0 \)
  by transfer simp

lemma not-int-less-negative [simp]: \( \neg \text{int } n < - \text{int } m \)
  by (simp add: linorder-not-less)

lemma negative-eq-positive [simp]: \( \text{int } n = \text{of-nat } m \iff n = 0 \land m = 0 \)
  by (force simp add: order-eq-iff [of \( - \text{of-nat } n \)] int-zle-neg)

lemma zle-iff-zadd: \( w \leq z \iff (\exists n. z = w + \text{int } n) \)
  (is \( ?\text{lhs} \longleftrightarrow ?\text{rhs} \))
  proof
    assume ?rhs
    then show ?lhs by auto
  next
    assume ?lhs
    then have \( 0 \leq z - w \) by simp
    then obtain \( n \) where \( z - w = \text{int } n \)
      using zero-le-imp-eq-int [of \( z - w \)] by blast
    then have \( z = w + \text{int } n \) by simp
    then show ?rhs ..
  qed

lemma zadd-int-left: \( \text{int } m + (\text{int } n + z) = \text{int } (m + n) + z \)
  by simp

lemma negD:
  assumes \( x < 0 \) shows \( \exists n. x = - (\text{int } (\text{Suc } n)) \)
  proof
    have \( \land \ a b. a < b \implies \exists n. \text{Suc } (a + n) = b \)
    proof
      fix \( a b:: \text{nat} \)
      assume \( a < b \)
      then have \( \text{Suc } (a + (b - \text{Suc } a)) = b \)
        by arith
      then show \( \exists n. \text{Suc } (a + n) = b \)
        by (rule exI)
    qed
    with assms show \( ?\text{thesis} \)
      by transfer auto
  qed

54.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

This version is symmetric in the two subgoals.

lemma int-cases2 [case-names nonneg nonpos, cases type: int]:
(\forall n. z = \text{int } n \rightarrow P) \rightarrow (\forall n. z = - (\text{int } n) \rightarrow P) \rightarrow P

by (cases \( z < 0 \)) (auto simp add: linorder-not-less dest!: negD nat-0-le [THEN sym])

This is the default, with a negative case.

**lemma** int-cases [case-names nonneg neg, cases type: int]:

assumes pos: \( \forall n. z = \text{int } n \rightarrow P \) and neg: \( \forall n. z = - (\text{int } (\text{Suc } n)) \rightarrow P \)

shows P

proof (cases \( z < 0 \))

case True

with neg show \(?thesis

by (blast dest!: negD)

next

case False

with pos show \(?thesis

by (force simp add: linorder-not-less dest: nat-0-le [THEN sym])

qed

**lemma** int-cases3 [case-names zero pos neg]:

fixes \( k :: \text{int} \)

assumes k: \( \text{nat } k > 0 \rightarrow P \) and \( \text{nat } k > 0 \rightarrow P \)

and \( \text{nat } k = \text{int } (\text{nat } k) \rightarrow P \)

shows P

proof (cases \( k \text{ nat } \rightarrow \text{int rule: linorder-cases} \))

case equal

with assms (1) show P by simp

next

case greater

then have \(*: \text{nat } k > 0 \) by simp

moreover from \(* \) have \( \text{nat } (\text{nat } k) \rightarrow P \) by auto

ultimately show P using assms (2) by blast

next

case less

then have \(*: \text{nat } (- k) > 0 \) by simp

moreover from \(* \) have \( \text{nat } (- k) \rightarrow P \) by auto

ultimately show P using assms (3) by blast

qed

**lemma** int-of-nat-induct [case-names nonneg neg, induct type: int]:

\( (\forall n. P \text{ (int n)}) \rightarrow (\forall n. P \text{ (int (Suc n))}) \rightarrow P \)

by (cases \( z \)) auto

**lemma** sgn-mult-dvd-iff [simp]:

\( \text{sgn } r \times l \text{ dvd } k \leftrightarrow l \text{ dvd } k \land (r = 0 \rightarrow k = 0) \) for \( k l r :: \text{int} \)

by (cases r rule: int-cases3) auto

**lemma** mult-sgn-dvd-iff [simp]:

\( l \times \text{sgn } r \text{ dvd } k \leftrightarrow l \text{ dvd } k \land (r = 0 \rightarrow k = 0) \) for \( k l r :: \text{int} \)

using sgn-mult-dvd-iff [of \( r l k \)] by (simp add: ac-simps)
lemma dvd-sgn-mult-iff [simp]:
\[ l \text{ dvd } sgn \, r \ast k \iff l \text{ dvd } k \lor r = 0 \text{ for } k \, l \, r :: \text{int} \]
by (cases \( r \) rule: int-cases3) simp-all

lemma dvd-mult-sgn-iff [simp]:
\[ l \text{ dvd } k \ast \text{sgn } r \iff l \text{ dvd } k \lor r = 0 \text{ for } k \, l \, r :: \text{int} \]
using dvd-sgn-mult-iff [of \( l \) \( r \) \( k \)] by (simp add: ac-simps)

lemma int-sgnE:
\[
\text{fixes } k :: \text{int} \\\n\text{obtains } n \text{ and } l \text{ where } k = \text{sgn } l \ast \text{int } n \]
proof −
\[
\text{have } k = \text{sgn } k \ast \text{int } (\text{nat } |k|) \\\n\text{by (simp add: sgn-mult-abs)} \\\n\text{then show } \text{?thesis} ..
\]
qed

54.8.1 Binary comparisons

Preliminaries

lemma le-imp-0-less:
\[
\text{fixes } z :: \text{int} \\\n\text{assumes } le: 0 \leq z \Rightarrow \text{shows } 0 < 1 + z
\]
proof −
\[
\text{have } 0 \leq z \text{ by fact} \\\n\text{also have } \ldots < z + 1 \text{ by (rule less-add-one)} \\\n\text{also have } \ldots = 1 + z \text{ by (simp add: ac-simps)} \\\n\text{finally show } 0 < 1 + z.
\]
qed

lemma odd-less-0-iff: \( 1 + z + z \not= 0 \leftrightarrow z < 0 \)
\[
\text{for } z :: \text{int} \]
proof (cases \( z \))
case (nonneg \( n \))
\[
\text{then show } \text{?thesis}
\]
by (simp add: linorder-not-less add.assoc add-increasing le-imp-0-less [THEN order-less-imp-le])
next
\[
\text{case (neg } n \text{)}
\]
\[
\text{then show } \text{?thesis}
\]
by (simp del: of-nat-Suc of-nat-add of-nat-1
\[ \text{add: algebra-simps of-nat-1 [where 'a=int, symmetric] of-nat-add [symmetric]} \])
qed

54.8.2 Comparisons, for Ordered Rings

lemma odd-nonzero: \( 1 + z + z \not= 0 \)
for $z :: int$

proof (cases $z$)
  case (nonneg $n$)
  have le: $0 \leq z + z$
    by (simp add: nonneg add-increasing)
  then show ?thesis
    using le-imp-0-less [OF le] by (auto simp: ac-simps)
  next
  case (neg $n$)
  show ?thesis
  proof
    assume eq: $1 + z + z = 0$
    have $0 < 1 + (int n + int n)$
      by (simp add: le-imp-0-less add-increasing)
    also have \ldots\ = $-(1 + z + z)$
      by (simp add: neg add.assoc [symmetric])
    also have \ldots\ = 0 by (simp add: eq)
    finally have $0 < 0$ ..
    then show False by blast
  qed
qed

5.4.9 The Set of Integers

context ring-1
begin

definition Ints :: 'a set (\$Z\$)
where $\$Z\$ = range of-int

lemma Ints-of-int [simp]: of-int $z \in \$Z$
  by (simp add: Ints-def)

lemma Ints-of-nat [simp]: of-nat $n \in \$Z$
  using Ints-of-int [of of-nat $n$] by simp

lemma Ints-0 [simp]: $0 \in \$Z$
  using Ints-of-int [of 0] by simp

lemma Ints-1 [simp]: $1 \in \$Z$
  using Ints-of-int [of 1] by simp

lemma Ints-numeral [simp]: numeral $n \in \$Z$
  by (subst of-nat-numeral [symmetric], rule Ints-of-nat)

lemma Ints-add [simp]: $a \in \$Z \Longrightarrow b \in \$Z \Longrightarrow a + b \in \$Z$
  by (force simp add: Ints-def simp flip: of-int-add intro: range-eql)

lemma Ints-minus [simp]: $a \in \$Z \Longrightarrow \neg a \in \$Z$
by (force simp add: Ints-def simp flip: of-int-minus intro: range-eqI)

lemma minus-in-Ints-iff: \(-x \in \mathbb{Z} \iff x \in \mathbb{Z}\)
using Ints-minus[of x] Ints-minus[of \(-x\)] by auto

lemma Ints-diff [simp]: \(a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a - b \in \mathbb{Z}\)
by (force simp add: Ints-def simp flip: of-int-diff intro: range-eqI)

lemma Ints-mult [simp]: \(a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a \ast b \in \mathbb{Z}\)
by (force simp add: Ints-def simp flip: of-int-mult intro: range-eqI)

lemma Ints-power [simp]: \(a \in \mathbb{Z} \implies a ^ n \in \mathbb{Z}\)
by (induct n) simp-all

lemma Ints-cases [cases set: Ints]:
assumes q \in \mathbb{Z}
obtains (of-int) z where q = of-int z
unfolding Ints-def
proof -
from \(q \in \mathbb{Z}\) have q \in range of-int unfolding Ints-def .
then obtain z where q = of-int z ..
then show thesis ..
qed

lemma Ints-induct [case-names of-int, induct set: Ints]:
q \in \mathbb{Z} \implies (\forall z. P (of-int z)) \implies P q
by (rule Ints-cases) auto

lemma Nats-subset-Ints: \(\mathbb{N} \subseteq \mathbb{Z}\)
unfolding Nats-def Ints-def
by (rule subsetI, elim imageE, hypsubst, subst of-int-of-nat-eq[symmetric], rule imageI) simp-all

lemma Nats-altdef1: \(\mathbb{N} = \{\text{of-int } n \mid n. n \geq 0\}\)
proof (intro subsetI equalityI)
fix x :: 'a
assume x \in \{\text{of-int } n \mid n. n \geq 0\}
then obtain n where x = of-int n n \geq 0
  by (auto elim!: Ints-cases)
then have x = of-nat (nat n)
  by (subst of-nat-nat) simp-all
then show x \in \mathbb{N}
  by simp
next
fix x :: 'a
assume x \in \mathbb{N}
then obtain n where x = of-nat n
  by (auto elim!: Nats-cases)
then have x = of-int (int n) by simp
also have \( \text{int } n \geq 0 \) by simp
then have of-int (\text{int } n) \in \{ \text{of-int } n \mid n \geq 0 \} by blast
finally show \( x \in \{ \text{of-int } n \mid n \geq 0 \} \).
qed
end

lemma Ints-sum [intro]: \( \forall x \in A \Rightarrow f x \in \mathbb{Z} \) \( \Rightarrow \) \( \sum f A \in \mathbb{Z} \)
by (induction A rule: infinite-finite-induct) auto

lemma Ints-prod [intro]: \( \forall x \in A \Rightarrow f x \in \mathbb{Z} \) \( \Rightarrow \) \( \prod f A \in \mathbb{Z} \)
by (induction A rule: infinite-finite-induct) auto

lemma (in linordered-idom) Ints-abs [simp]:
shows \( a \in \mathbb{Z} \) \( \Rightarrow \) \( \text{abs } a \in \mathbb{Z} \)
by (auto simp: abs-if)

lemma (in linordered-idom) Nats-altdef2: \( \mathbb{N} = \{ n \in \mathbb{Z}. n \geq 0 \} \)
proof (intro subsetI equalityI)
fix \( x :: 'a \)
assume \( x \in \{ n \in \mathbb{Z}. n \geq 0 \} \)
then obtain \( n \) where \( x = \text{of-nat} n \) \( n \geq 0 \)
by (auto elim!: Ints-cases)
then show \( x \in \mathbb{N} \)
by simp
qed (auto elim!: Nats-cases)

lemma (in idom-divide) of-int-divide-in-Ints:
\( \text{of-int } a \ \text{div} \ \text{of-int } b \in \mathbb{Z} \) if \( b \ dvd a \)
proof –
from that obtain \( c \) where \( a = b \ast c \) ..
then show \?thesis
by (cases of-int \( b = 0 \)) simp-all
qed

The premise involving \( \mathbb{Z} \) prevents \( a = (1::'a) / (2::'a) \).

lemma Ints-double-eq-0-iff:
fixes \( a :: 'a::ring-char-0 \)
assumes in-Ints: \( a \in \mathbb{Z} \)
shows \( a + a = 0 \) \( \iff \) \( a = 0 \)\hspace{1cm} (is \?lhs \iff \?rhs)
proof –
from in-Ints have \( a \in \text{range of-int} \)
unfolding Ints-def [symmetric] .
then obtain \( z \) where \( a = \text{of-int } z \) ..
show \?thesis
proof
assume ?rhs
then show ?lhs by simp

next
assume ?lhs
with a have of-int (z + z) = (of-int 0 :: 'a) by simp
then have z + z = 0 by (simp only: of-int-eq-iff)
then have z = 0 by (simp only: double-zero)
with a show ?rhs by simp
qed

lemma Ints-odd-nonzero:
  fixes a :: 'a::ring-char-0
  assumes in-Ints: a ∈ ℤ
  shows 1 + a + a ≠ 0
proof (−)
  from in-Ints have a ∈ range of-int
    unfolding Ints-def [symmetric].
  then obtain z where a: a = of-int z ..
  show ?thesis
  proof
    assume 1 + a + a = 0
    with a have of-int (1 + z + z) = (of-int 0 :: 'a) by simp
    then have 1 + z + z = 0 by (simp only: of-int-eq-iff)
    with odd-nonzero show False by blast
  qed
qed

lemma Nats-numeral [simp]: numeral w ∈ ℕ
  using of-nat-in-Nats [of numeral w] by simp

lemma Ints-odd-less-0:
  fixes a :: 'a::linordered-idom
  assumes in-Ints: a ∈ ℤ
  shows 1 + a + a < 0 ←→ a < 0
proof (−)
  from in-Ints have a ∈ range of-int
    unfolding Ints-def [symmetric].
  then obtain z where a: a = of-int z ..
  with a have 1 + a + a < 0 ←→ of-int (1 + z + z) < (of-int 0 :: 'a)
    by simp
  also have ... ←→ z < 0
    by (simp only: of-int-less-iff odd-less-0-iff)
  also have ... ←→ a < 0
    by (simp add: a)
  finally show ?thesis .
qed
THEORY “Int”

54.10 sum and prod
context semiring-1
begin

lemma of-nat-sum [simp]:
  of-nat (sum f A) = (∑ x∈A. of-nat (f x))
by (induction A rule: infinite-finite-induct) auto

end

context ring-1
begin

lemma of-int-sum [simp]:
  of-int (sum f A) = (∑ x∈A. of-int (f x))
by (induction A rule: infinite-finite-induct) auto

end

context comm-semiring-1
begin

lemma of-nat-prod [simp]:
  of-nat (prod f A) = (∏ x∈A. of-nat (f x))
by (induction A rule: infinite-finite-induct) auto

end

context comm-ring-1
begin

lemma of-int-prod [simp]:
  of-int (prod f A) = (∏ x∈A. of-int (f x))
by (induction A rule: infinite-finite-induct) auto

end

54.11 Setting up simplification procedures
ML-file ⟨Tools/int-arith.ML⟩

declaration ⟨K ⟨
  Lin-Arith.add-discrete-type type-name ⟨Int.int⟩
  #> Lin-Arith.add-lessD @{thm zless-imp-add1-zle}
  #> Lin-Arith.add-inj-const (const-name ⟨of-nat⟩, typ ⟨nat ⇒ int⟩)
  #> Lin-Arith.add-simps
    @{thms of-int-0 of-int-1 of-int-add of-int-mult of-int-numeral of-int-neg-numeral

"\text{end}\"
nat-0 nat-1 diff-nat-numeral nat-numeral
neg-less-iff-less
True-implies-equals
distrib-left [where \( a = \text{numeral} \ v \) for \( v \)]
distrib-left [where \( a = - \text{numeral} \ v \) for \( v \)]
div-by-1 div-0
times-divide-eq-right times-divide-eq-left
minus-divide-left [THEN sym] minus-divide-right [THEN sym]
add-divide-distrib diff-divide-distrib
of-int-minus of-int-diff
of-int-of-nat-eq

\[ > \text{Lin-Arith.add-simprocs [Int-Arith.zero-one-idom-simproc]} \]

\texttt{simproc-setup fast-arith}
((\( m::'a::\text{linordered-idom} < n \) |
(\( m::'a::\text{linordered-idom} \leq n \) |
(\( m::'a::\text{linordered-idom} = n \) =
\( 'K \text{Lin-Arith.simproc} \)

54.12 More Inequality Reasoning

\texttt{lemma zless-add1-eq: } w < z + 1 \iff w < z \lor w = z
\texttt{for } w z :: \text{int}
\texttt{by arith}

\texttt{lemma add1-zle-eq: } w + 1 \leq z \iff w < z
\texttt{for } w z :: \text{int}
\texttt{by arith}

\texttt{lemma zle-diff1-eq [simp]: } w \leq z - 1 \iff w < z
\texttt{for } w z :: \text{int}
\texttt{by arith}

\texttt{lemma zle-add1-eq-le [simp]: } w < z + 1 \iff w \leq z
\texttt{for } w z :: \text{int}
\texttt{by arith}

\texttt{lemma int-one-le-iff-zero-less: } 1 \leq z \iff 0 < z
\texttt{for } z :: \text{int}
\texttt{by arith}

\texttt{lemma Ints-nonnzero-abs-ge1:}
\texttt{fixes } x:: 'a :: \text{linordered-idom}
\texttt{assumes } x \in \text{Ints} x \neq 0
\texttt{shows } 1 \leq \text{abs} \ x
\texttt{proof (rule Ints-cases [OF \( \langle x \in \text{Ints} \rangle \)])}
\texttt{fix } z::\text{int}
\texttt{assume } x = \text{of-int} \ z
with \( x \neq 0 \),
show \( 1 \leq |x| \)
apply (auto simp: abs_if)
by (metis diff-0 of-int-1 of-int-le-iff of-int-minus zle-diff1-eq)
qed

**lemma** \( \text{Ints-nonzero-abs-less1} \):
fixes \( x :: \text{'}a :: \text{linordered-idom} \)
shows \([x \in \text{Ints}; \; \text{abs} \; x < 1] \implies x = 0\)
using \( \text{Ints-nonzero-abs-ge1} \; [\text{of} \; x] \) by auto

**lemma** \( \text{Ints-eq-abs-less1} \):
fixes \( x :: \text{'}a :: \text{linordered-idom} \)
shows \([x \in \text{Ints}; \; y \in \text{Ints}] \implies x = y \iff \text{abs} (x-y) < 1\)
using \( \text{eq-iff-diff-eq-0} \) by (fastforce intro: \( \text{Ints-nonzero-abs-less1} \))

### 54.13 The functions nat and int

Simplify the term \( w + - z \).

**lemma** \( \text{one-less-nat-eq} \; [\text{simp}] \): \( \text{Suc} \; 0 < \text{nat} \; z \iff 1 < z \)
using \( \text{zless-nat-conj} \; [\text{of} \; 1 \; z] \) by auto

**lemma** \( \text{int-eq-iff-numeral} \; [\text{simp}] \):
\( \text{int} \; m = \text{numeral} \; v \iff m = \text{numeral} \; v \)
by (simp add: \( \text{int-eq-iff} \))

**lemma** \( \text{nat-abs-int-diff} \):
\( \text{nat} \; | \text{int} \; a - \text{int} \; b| = (\text{if} \; a \leq b \; \text{then} \; b - a \; \text{else} \; a - b)\)
by auto

**lemma** \( \text{nat-int-add} \): \( \text{nat} \; (\text{int} \; a + \text{int} \; b) = a + b \)
by auto

**context** \( \text{ring-1} \)

**begin**

**lemma** \( \text{of-int-of-nat} \; [\text{nitpick-simp}] \):
\( \text{of-int} \; k = (\text{if} \; k < 0 \; \text{then} \; - \; \text{of-nat} \; (\text{nat} \; (- \; k)) \; \text{else} \; \text{of-nat} \; (\text{nat} \; k))\)
proof (cases \( k < 0 \))
  case True
  then have \( 0 \leq - k \) by simp
  then have \( \text{of-nat} \; (\text{nat} \; (- \; k)) = \text{of-int} \; (- \; k) \) by (rule \( \text{of-nat-nat} \))
  with \( \text{True} \) show \(?thesis\) by simp
next
  case False
  then show \(?thesis\) by (simp add: \( \text{not-less} \))
qed

**end**
lemma transfer-rule-of-int:
  includes lifting-syntax
  fixes R :: 'a::ring-1 ⇒ 'b::ring-1 ⇒ bool
  assumes [transfer-rule]: R 0 0 R 1 1
     (R ===> R ===> R) (+) (+)
     (R ===> R) uminus uminus
  shows ((=) ===> R) of-int of-int
proof –
  note assms
  note transfer-rule-of-nat [transfer-rule]
  have [transfer-rule]: ((=) ===> R) of-nat of-nat
     by transfer-prover
  show ?thesis
     by (unfold of-int-of-nat [abs-def]) transfer-prover
qed

lemma nat-mult-distrib:
  fixes z z' :: int
  assumes 0 ≤ z
  shows nat (z * z') = nat z * nat z'
proof (cases 0 ≤ z')
  case False
     with assms have z * z' ≤ 0
     by (simp add: not-le mult-le-0-iff)
  then have nat (z * z') = 0 by simp
  moreover from False have nat z' = 0 by simp
  ultimately show ?thesis by simp
next
  case True
     with assms have ge-0: z * z' ≥ 0 by (simp add: zero-le-mult-iiff)
  show ?thesis
     by (rule injD [of of-nat :: nat ⇒ int, OF inj-of-nat])
     (simp only: of-nat-mult of-nat-nat [OF assms] of-nat-nat [OF ge-0], simp)
qed

lemma nat-mult-distrib-neg:
  assumes z ≤ (0::int) shows nat (z * z') = nat (− z) * nat (− z') (is ?L = ?R)
proof –
  have ?L = nat (− z * − z')
     using assms by auto
  also have ... = ?R
     by (rule nat-mult-distrib) (use assms in auto)
  finally show ?thesis .
qed

lemma nat-abs-mult-distrib: nat |w * z| = nat |w| * nat |z|
  by (cases z = 0 ∨ w = 0)

lemma int-in-range-abs [simp]: int n ∈ range abs
proof (rule range-eqI)
  show int n = |int n| by simp
qed

lemma range-abs-Nats [simp]: range abs = (N :: int set)
proof
  have |k| ∈ N for k :: int
    by (cases k) simp-all
  moreover have k ∈ range abs if k ∈ N for k :: int
    using that by induct simp
  ultimately show ?thesis by blast
qed

lemma Suc-nat-eq-nat-zadd1: 0 ≤ z =⇒ Suc (nat z) = nat (1 + z)
for z :: int
by (rule sym) (simp add: nat-eq-iff)

lemma diff-nat-eq-if:
  nat z − nat z' =
    (if z' < 0 then nat z
    else
      let d = z − z'
      in if d < 0 then 0 else nat d)
  by (simp add: Let-def nat-diff-distrib [symmetric])

lemma nat-numeral-diff-1 [simp]: numeral v − (1 :: nat) = nat (numeral v − 1)
using diff-nat-numeral [of v Num.One] by simp

54.14 Induction principles for int

Well-founded segments of the integers.

definition int-ge-less-than :: int ⇒ (int × int) set
where int-ge-less-than d = {(z', z). d ≤ z' ∧ z' < z}

lemma wf-int-ge-less-than: wf (int-ge-less-than d)
proof
  have int-ge-less-than d ⊆ measure (λz. nat (z − d))
    by (auto simp add: int-ge-less-than-def)
  then show ?thesis
    by (rule wf-subset [OF wf-measure])
qed

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

definition int-ge-less-than2 :: int ⇒ (int × int) set
where \( \text{int-ge-less-than2} \ d = \{(z',z). \ d \leq z \land z' < z\}\)

**lemma** \(\text{wf-int-ge-less-than2}: \text{wf} (\text{int-ge-less-than2} \ d)\)

**proof** –
  have \(\text{int-ge-less-than2} \ d \subseteq \text{measure} (\lambda z. \text{nat} (1 + z - d))\)
  by (auto simp add: int-ge-less-than2-def)
  then show \(?\)thesis
    by (rule wf-subset [OF \(\text{wf-measure}\)])
qed

**theorem** \(\text{int-ge-induct} [\text{case-names base step}, \text{induct set}: \text{int}]:\)
fixes \(i :: \text{int}\)
assumes \(\text{ge}: \ k \leq i\)
and \(\text{base}: \ P \ k\)
and \(\text{step}: \land i. \ k \leq i \Longrightarrow P \ i \Longrightarrow P \ (i + 1)\)
sows \(P \ i\)
proof –
  have \(\land i :: \text{int}. \ n = \text{nat} (i - k) \Longrightarrow k \leq i \Longrightarrow P \ i \ \text{for} \ n\)
  proof (induct \(n\))
    case \(0\)
    then have \(i = k\) by arith
    with \(\text{base}\) show \(P \ i\) by simp
next
  case (Suc \(n\))
  then have \(n = \text{nat} ((i - 1) - k)\) by arith
  moreover have \(k: \ k \leq i - 1\) using Suc.prems by arith
  ultimately have \(P \ (i - 1)\) by (rule Suc.hyps)
  from step [\(\text{OF} \ k\ this\)] show \(?\)case by simp
qed
with \(\text{ge}\) show \(?\)thesis by fast
qed

**theorem** \(\text{int-gr-induct} [\text{case-names base step}, \text{induct set}: \text{int}]:\)
fixes \(i \ k :: \text{int}\)
assumes \(\text{le}: \ i < P \ (k + 1) \land \land i. \ k < i \Longrightarrow P \ i \Longrightarrow P \ (i + 1)\)
sows \(P \ i\)
proof –
  have \(k+1 \leq i\)
    using assms by auto
  then show \(?\)thesis
    by (induction \(i\) rule: int-ge-induct) (auto simp: assms)
qed

**theorem** \(\text{int-le-induct} [\text{consumes} 1, \text{case-names base step}]:\)
fixes \(i \ k :: \text{int}\)
assumes \(\text{le}: \ i \leq k\)
and \(\text{base}: \ P \ k\)
and step: \( \forall i. \ i \leq k \Rightarrow P\ i \Rightarrow P\ (i - 1) \)
shows \( P\ i \)
proof
have \( \forall i::\mathbb{int}. \ n = \mathit{nat}(k-i) \Rightarrow i \leq k \Rightarrow P\ i \) for \( n \)
proof (induct \( n \))
  case 0
  then have \( i = k \) by arith
  with base show \( P\ i \) by simp
next
  case (Suc \( n \))
  then have \( n = \mathit{nat}(k - (i + 1)) \) by arith
  moreover have \( k: i + 1 \leq k \) using Suc.prems by arith
  ultimately have \( P\ (i + 1) \) by (rule Suc.hyps)
  from step[OF \( k \) this] show \( \text{?case} \) by simp
qed
with le show \( \text{?thesis} \) by fast
qed

theorem int-less-induct [consumes 1, case-names base step]:
fixes \( k :: \mathbb{int} \)
assumes \( i < k \) \( P\ (k - 1) \) \( \forall i. \ i < k \Rightarrow P\ i \Rightarrow P\ (i - 1) \)
shows \( P\ i \)
proof
have \( i \leq k - 1 \)
  using assms by auto
  then show \( \text{?thesis} \)
  by (induction \( i \) rule: int-le-induct) (auto simp: assms)
qed

theorem int-induct [case-names base step1 step2]:
fixes \( k :: \mathbb{int} \)
assumes base: \( P\ k \)
and step1: \( \forall i. \ k \leq i \Rightarrow P\ i \Rightarrow P\ (i + 1) \)
and step2: \( \forall i. \ k \geq i \Rightarrow P\ i \Rightarrow P\ (i - 1) \)
shows \( P\ i \)
proof
  have \( i \leq k \lor i \geq k \) by arith
  then show \( \text{?thesis} \)
  proof
    assume \( i \geq k \)
    then show \( \text{?thesis} \)
      using base by (rule int-ge-induct) (fact step1)
  next
    assume \( i \leq k \)
    then show \( \text{?thesis} \)
      using base by (rule int-le-induct) (fact step2)
  qed
qed
54.15 Intermediate value theorems

lemma nat-ivt-aux:
\[ \forall i < n. \vert f \text{ (Suc } i \text{)} - f i \vert \leq 1; f 0 \leq k; k \leq f n \] \[ \Rightarrow \exists i \leq n. f i = k \]
for \( m n :: \text{nat} \) and \( k :: \text{int} \)
proof (induct n)
case (Suc n)
show ?case
proof (cases \( k = f \text{ (Suc } n \text{)} \))
  case False
  with Suc have \( k \leq f n \) by auto
  with Suc show ?thesis
  using \( \text{le-SucI} \)
qed (use Suc in auto)

lemma nat-intermed-int-val:
fixes \( m n :: \text{nat} \) and \( k :: \text{int} \)
assumes \( \forall i. m \leq i \wedge i < n \rightarrow \vert f \text{ (Suc } i \text{)} - f i \vert \leq 1 \)
\( m \leq n f m \leq k k \leq f n \)
shows \( \exists i. m \leq i \wedge i \leq n \wedge f i = k \)
proof –
obtain i where \( i \leq n - m k = f \text{ (m + i)} \)
  using nat-ivt-aux [of \( n - m f \circ \text{plus } m k \)]
  assms by auto
with assms show ?thesis
  using \( \text{exI} \) [of \( m + i \)] by auto
qed

lemma nat0-intermed-int-val:
\[ \exists i \leq n. f i = k \]
if \( \forall i < n. \vert f \text{ (i + 1)} - f i \vert \leq 1 f 0 \leq k k \leq f n \)
for \( n :: \text{nat} \) and \( k :: \text{int} \)
using nat-intermed-int-val [of \( 0 n f k \)]
proof (use assms in auto)

54.16 Products and 1, by T. M. Rasmussen

lemma abs-zmult-eq-1:
fixes \( m n :: \text{int} \)
assumes \( mn \) : \( \vert m \ast n \vert = 1 \)
shows \( \vert m \vert = 1 \)
proof –
from \( mn \) have \( 0; m \neq 0 n \neq 0 \) by auto
have \( \neg 2 \leq \vert m \vert \)
proof
  assume \( \neg 2 \leq \vert m \vert \)
  then have \( 2 \ast \vert n \vert \leq \vert m \ast n \vert \) by (simp add: mult-mono \( 0 \))
  also have \( \ldots = \vert m \ast n \vert \) by (simp add: abs-mult)
  also from \( mn \) have \( \ldots = 1 \) by simp
  finally have \( 2 \ast \vert n \vert \leq 1 \).
  with \( \neg \) show \( \text{False} \) by \( \text{arith} \)
qed
with \( 0 \) show \( \text{thesis} \) by auto
qed

lemma pos-zmult-eq-1-iff-lemma: \( m \ast n = 1 \implies m = 1 \lor m = -1 \)
for \( m \) \( n :: \) \text{int}
using abs-zmult-eq-1 \( \text{[of } m \text{ n]} \) by arith

lemma pos-zmult-eq-1-iff:
  fixes \( m \) \( n :: \) \text{int}
  assumes \( 0 < m \)
  shows \( m \ast n = 1 \iff m = 1 \land n = 1 \)
proof –
  from assms have \( m \ast n = 1 \implies m = 1 \)
  by (auto dest: pos-zmult-eq-1-iff-lemma)
  then show \( \text{thesis} \)
  by (auto dest: pos-zmult-eq-1-iff-lemma)
qed

lemma zmult-eq-1-iff: \( m \ast n = 1 \iff (m = 1 \land n = 1) \lor (m = -1 \land n = -1) \)
for \( m \) \( n :: \) \text{int}
proof
  assume \( L :: \) \text{int}
  show \( R :: \) \text{int}
  using pos-zmult-eq-1-iff-lemma \( \text{[of } a - b\] \) by force
qed
decauto

lemma zmult-eq-neg1-iff: \( a \ast b = (-1 :: \) \text{int} \iff a = 1 \land b = -1 \lor a = -1 \land b = 1 \)
using zmult-eq-1-iff \( \text{[of } a - b\] \) by auto

lemma infinite-UNIV-int \( \text{[simp]} :: \) \neg finite (UNIV::int set)
proof
  assume finite (UNIV::int set)
  moreover have \( \text{inj } (\lambda i :: \text{int. } 2 \ast i) \)
  by (rule injI) simp
  ultimately have \( \text{surj } (\lambda i :: \text{int. } 2 \ast i) \)
  by (rule finite-UNIV-inj-surj)
  then obtain \( i :: \) \text{int where} \( 1 = 2 \ast i \) by (rule surjE)
  then show \( \text{False} \)
  by (simp add: pos-zmult-eq-1-iff)
qed

54.17 The divides relation

lemma zdvd-antisym-nonneg: \( 0 \leq m \implies 0 \leq n \implies m \text{ dvd } n \implies n \text{ dvd } m \implies m = n \)
for \( m \) \( n :: \) \text{int}
by (auto simp add: dvd-def mult.assoc zero-le-mult-iff zmult-eq-1-iff)
lemma zdvd-antisym-abs:
  fixes a b :: int
  assumes a dvd b and b dvd a
  shows \(|a| = |b|\)
proof (cases a = 0)
case True
  with assms show ?thesis by simp
next
case False
  from \(a dvd b\) obtain k where k: b = a * k
    unfolding dvd-def by blast
  from \(b dvd a\) obtain k' where k': a = b * k'
    unfolding dvd-def by blast
  from k k' have a = a * k * k'
    by simp
  with mult-cancel-left \(\text{of } a \neq 0\) \(\text{by simp add: mult_assoc}\)
  then have k = 1 \(\land k' = 1 \lor k = -1 \land k' = -1\)
    \(\text{by (simp add: zmult-eq-1-iff)}\)
  with k k' show ?thesis by auto
qed

lemma zdvd-zdiffD: k dvd m − n \(\implies\) k dvd n \(\implies\) k dvd m
  for k m n :: int
using dvd-add-right-iff \(\text{of } k \cdot n \cdot m\) by simp

lemma zdvd-reduce: k dvd n + k * m \(\iff\) k dvd n
  for k m n :: int
using dvd-add-times-triv-right-iff \(\text{of } k \cdot n \cdot m\) \(\text{by (simp add: ac-simps)}\)

lemma dvd-imp-le-int:
  fixes d i :: int
  assumes i \(\neq\) 0 and d dvd i
  shows \(|d| \leq |i|\)
proof
  from \(d dvd i\) obtain k where i = d * k
    with \(i \neq 0\) \(\text{have } k \neq 0\) by auto
  then have 1 \(\leq |k|\) \(\text{and } 0 \leq |d|\) by auto
  then have \(|d| \times 1 \leq |d| \times |k|\) \(\text{by (rule mult-left-mono)}\)
  with \(i = d \times k\) show ?thesis by (simp add: abs-mult)
qed

lemma zdvd-not-zless:
  fixes m n :: int
  assumes \(0 < m\) and \(m < n\)
  shows \(\neg n \text{ dvd } m\)
proof
  from assms have \(0 < n\) \(\text{by auto}\)
  assume n dvd m then obtain k where k: m = n * k
    with \(\theta < m\) \(\text{have } \theta < n \times k\) \(\text{by auto}\)
with \( \langle 0 < n \rangle \) have \( 0 < k \) by (simp add: zero_less_mult_iff)
with \( k \langle 0 < n \rangle \langle m < n \rangle \) have \( n \cdot k < n \cdot 1 \) by simp
with \( \langle 0 < n \rangle \langle 0 < k \rangle \) show False unfolding mult_less_cancel_left by auto
qed

lemma zdvd_mult_cancel:
  fixes \( k \; m \; n \) :: int
  assumes \( d : k \cdot m \text{ dvd } k \cdot n \) and \( k \neq 0 \)
  shows \( m \text{ dvd } n \)
proof -
  from \( d \) obtain \( h \) where \( h : k \cdot n = k \cdot m \cdot h \)
    unfolding dvd_def by blast
  have \( n = m \cdot h \)
  proof (rule ccontr)
    assume \( \neg \thesis \)
    with \( k \neq 0 \) have \( k \cdot n \neq k \cdot (m \cdot h) \) by simp
    with \( h \) show False
      by (simp add: mult_assoc)
  qed
  then show \( \thesis \) by simp
qed

lemma int_dvd_int_iff [simp]:
  \( \text{int } m \text{ dvd } \text{int } n \longleftrightarrow m \text{ dvd } n \)
proof -
  have \( m \text{ dvd } n \) if \( \text{int } n = \text{int } m \cdot k \) for \( k \)
  proof (cases \( k \))
    case (nonneg \( q \))
    with \( \thesis \) have \( n = m \cdot q \)
      by (simp del: of_nat_mult add: of_nat_mult [symmetric])
    then show \( \thesis \)
  next
    case (neg \( q \))
    with \( \thesis \) have \( n = m \cdot q \)
      by simp
    also have \( \ldots = - (\text{int } m \cdot \text{int } (\text{Suc } q)) \)
      by (simp only: mult_minus_right)
    also have \( \ldots = - \text{int } (m \cdot \text{Suc } q) \)
      by (simp only: of_nat_mult [symmetric])
    finally have \( - \text{int } (m \cdot \text{Suc } q) = \text{int } n \)
      then show \( \thesis \)
        by (simp only: negative_eq_positive) auto
  qed
  then show \( \thesis \) by (auto simp add: dvd_def)
qed

lemma dvd_nat_abs_iff [simp]:
  \( n \text{ dvd } \text{nat } |k| \longleftrightarrow \text{int } n \text{ dvd } k \)
proof
  have \( n \mid \text{nat} \leftrightarrow \text{int} \mid \text{nat} \)
  
  then show \( \text{thesis} \)
  by simp
qed

lemma \( \text{nat-abs-dvd-iff} \) [simp]:
\( \text{nat} \mid k \mid \text{dvd} n \leftrightarrow k \mid \text{dvd} \text{int} n \)

proof
  have \( \text{nat} \mid k \mid \text{dvd} n \leftrightarrow \text{int}(\text{nat} \mid k) \mid \text{dvd} \text{int} n \)
  
  then show \( \text{thesis} \)
  by simp
qed

lemma \( \text{zdvd1-eq} \) [simp]: \( x \mid \text{dvd} 1 \leftrightarrow |x| = 1 \) (is \( \text{lhs} \leftrightarrow \text{rhs} \))
for \( x :: \text{int} \)

proof
  assume \( \text{lhs} \)
  
  then have \( \text{nat} \mid |x| \mid \text{dvd} \text{nat} \mid |1| \)
  
  then have \( \text{nat} \mid |x| = 1 \)
  
  then show \( \text{rhs} \)
  by simp
next
  assume \( \text{rhs} \)
  
  then have \( x = 1 \lor x = -1 \)
  
  by auto
  
  then show \( \text{lhs} \)
  by (auto intro: dvdI)
qed

lemma \( \text{zdvd-mult-cancel1} \):
fixes \( m :: \text{int} \)

assumes \( mp: m \neq 0 \)

shows \( m \mid n \leftrightarrow |n| = 1 \) (is \( \text{lhs} \leftrightarrow \text{rhs} \))

proof
  assume \( \text{rhs} \)
  
  then show \( \text{lhs} \)
  by (cases \( n > 0 \)) (auto simp add: minus-equation-iff)
next
  assume \( \text{lhs} \)
  
  then have \( m \mid n \leftrightarrow m \mid 1 \) by simp
  
  from \( \text{zdvd-mult-cancel1}[OF this mp] \) show \( \text{rhs} \)
  by (simp only: zdvd1-eq)
qed
lemma nat-dvd-iff: nat z dvd m ⟷ (if 0 ≤ z then z dvd int m else m = 0)
  using nat-abs-dvd-iff [of z m] by (cases z ≥ 0) auto

lemma eq-nat-nat-iff: 0 ≤ z ⇒ 0 ≤ z' ⟷ nat z = nat z' ⟷ z = z'
  by (auto elim: nonneg-int-cases)

lemma nat-power-eq: 0 ≤ z =⇒ nat (z ^ n) = nat z ^ n
  by (induct n) (simp-all add: nat-mult-distrib)

lemma numeral-power-eq-nat-cancel-iff [simp]:
  numeral x ^ n = nat y ⟷ numeral x ^ n = y
  using nat-eq-iff2 by auto

lemma numeral-power-le-nat-cancel-iff [simp]:
  numeral x ^ n ≤ nat a ⟷ numeral x ^ n ≤ a
  using nat-le-eq-zle [of numeral x ^ n a]
  by (auto simp: nat-power-eq)

lemma zdvd-imp-le: z ≤ n if z dvd n 0 < n for n z :: int
  proof (cases n)
    case (nonneg n)
    show thesis by (cases z) (use nonneg dvd-imp-le that in auto)
  qed (use that in auto)

lemma zdvd-period:
  fixes a d :: int
  assumes a dvd d
  shows a dvd (x + t) ⟷ a dvd ((x + c * d) + t)
proof –
from assms have a dvd (x + t) ↔ a dvd ((x + t) + c * d)
  by (simp add: dvd-add-left-iff)
then show ?thesis
  by (simp add: ac-simps)
qed

54.18 Powers with integer exponents

The following allows writing powers with an integer exponent. While the
type signature is very generic, most theorems will assume that the underlying
type is a division ring or a field.
The notation ‘powi’ is inspired by the ‘powr’ notation for real/complex ex-
ponentiation.
definition power-int :: 'a :: {inverse, power} ⇒ int ⇒ 'a (infixr powi 80) where
  power-int x n = (if n ≥ 0 then x ^ nat n else inverse x ^ (nat (-n)))

lemma power-int-0-right [simp]: power-int x 0 = 1
  and power-int-1-right [simp]:
    power-int (y :: 'a :: {power, inverse, monoid-mult}) 1 = y
  and power-int-minus1-right [simp]:
    power-int (y :: 'a :: {power, inverse, monoid-mult}) (-1) = inverse y
  by (simp-all add: power-int-def)

lemma power-int-of-nat [simp]: power-int x (int n) = x ^ n
  by (simp add: power-int-def)

lemma power-int-numeral [simp]: power-int x (numeral n) = x ^ numeral n
  by (simp add: power-int-def)

lemma powi-numeral-reduce: x powi numeral n = x * x powi int (pred-numeral n)
  by (simp add: numeral-eq-Suc)

lemma powi-minus-numeral-reduce: x powi − (numeral n) = inverse x * x powi −
  int(pred-numeral n)
  by (simp add: numeral-eq-Suc power-int-def)

lemma int-cases4 [case-names nonneg neg]:
  fixes m :: int
  obtains n where m = int n | n where n > 0 m = −int n
proof (cases m ≥ 0)
  case True
  thus ?thesis using that(1)[of nat m] by auto
next
case False
  thus ?thesis using that(2)[of nat (-m)] by auto
qed
context
  assumes SORT_CONSTRAINT(\'a::division-ring)
begin

lemma powerful_minus: power_int (\'a) (-n) = inverse (power_int n)
  by (auto simp: power_int_def inverse)

lemma powerful_minus_divide: power_int (\'a) (-n) = 1 / (power_int n)
  by (simp add: divide_inverse powerful_minus)

lemma powerful_eq_0_iff: power_int (\'a) n = 0 iff x = 0 ∧ n ≠ 0
  by (auto simp: power_int_def)

lemma powerful_0_left_if: power_int (\'a) m = (if m = 0 then 1 else 0)
  by (auto simp: powerful_0_left)

lemma powerful_0_left: power_int (\'a) m = 0
  by (simp add: powerful_0_left_if)

lemma powerful_1_left: power_int (\'a) 1 = (1 :: \'a :: division_ring)
  by (auto simp: powerful_1_left)

lemma powerful_diff_cone_inverse: x ≠ 0 ⇒ m ≤ n ⇒ (\'a) ^ (n - m) = x ^ n * inverse x ^ m
  by (simp add: field_simps flip: powerful_add)

lemma powerful_mult_inverse_distrib: power_int (\'a) m * inverse (\'a) = inverse (power_int m)
  proof (cases x = 0)
    case True
    therefore ?thesis using Suc by simp
  qed auto

lemma powerful_mult_power_inverse_commute:
  proof (induction n)
    case (Suc n)
    have x ^ m * inverse x ^ Suc n = (x ^ m * inverse x ^ n) * inverse x
      by (simp only: powerful_Suc2 mult_assoc)
    also have ... = inverse x ^ Suc m
      by (simp only: powerful_Suc2 mult_assoc [symmetric])
    finally show ?thesis using Suc by simp
  qed auto
by (rule Suc)
also have \ldots \ast \text{inverse} x = (\text{inverse} x ^ n \ast \text{inverse} x) \ast x ^ m
by (simp add: mult.assoc power-mult-inverse-distrib)
also have \ldots = \text{inverse} x ^ (\text{Suc} n) \ast x ^ m
by (simp only: power-Suc2)
finally show \text{?case} .
qed auto

lemma power-int-add:
assumes \( x \neq 0 \lor m + n \neq 0 \)
shows \( \text{power-int } (x ::'a) (m + n) = \text{power-int } x m \ast \text{power-int } x n \)
proof (cases \( x = 0 \))
  case True
  thus \text{?thesis} using assms by (auto simp: power-int-0-left-If)
next
  case [simp]: False
  show \text{?thesis}
  proof (cases m n rule: int-cases4 case-product int-cases4)
    case (nonneg-nonneg a b)
    thus \text{?thesis}
    by (auto simp: power-int-def nat-add-distrib power-add)
  next
    case (nonneg-neg a b)
    thus \text{?thesis}
    by (auto simp: power-int-def nat-diff-distrib not-le power-diff-conv-inverse
          power-mult-power-inverse-commute)
  next
    case (neg-nonneg a b)
    thus \text{?thesis}
    by (auto simp: power-int-def nat-diff-distrib not-le power-diff-conv-inverse
          power-mult-power-inverse-commute)
  next
    case (neg-neg a b)
    thus \text{?thesis}
    by (auto simp: power-int-def nat-add-distrib add.commute simp flip: power-add)
  qed
qed

lemma power-int-commutes: \( \text{power-int } (x ::'a) n \ast x = x \ast \text{power-int } x n \)

lemma power-int-add-1:
assumes \( x \neq 0 \lor m \neq -1 \)
shows \( \text{power-int } (x ::'a) (m + 1) = \text{power-int } x m \ast x \)
using assms by (subst power-int-add) auto

lemma power-int-add-1':
assumes \( x \neq 0 \lor m \neq -1 \)
shows \( \text{power-int } (x ::'a) (m + 1) = x \ast \text{power-int } x m \)
using assms by (subst add.commute, subst power-int-add) auto

lemma power-int-commutes: \( \text{power-int } (x ::'a) n \ast x = x \ast \text{power-int } x n \)
by (cases x = 0) (auto simp flip: power-int-add-1 power-int-add-1')

lemma power-int-inverse [field-simps, field-split-simps, divide-simps]:
  power-int (inverse (x :: 'a)) n = inverse (power-int x n)
by (auto simp: power-int-def power-inverse)

lemma power-int-mult: power-int (x :: 'a) (m * n) = power-int (power-int x m) n
  by (auto simp: power-int-def zero-le-mult-iff simp flip: power-mult power-inverse)

end

context
  assumes SORT-CONSTRAINT ('a::field)
begin

lemma power-int-diff:
  assumes x ≠ 0 ∨ m ≠ n
  shows  power-int (x :: 'a) (m - n) = power-int x m / power-int x n
  using power-int-add[of x m - n] assms by (auto simp: field-simps power-int-minus)

lemma power-int-minus-mult: x ≠ 0 ∨ n ≠ 0 ⇒ power-int (x :: 'a) (n - 1) * x = power-int x n
  by (auto simp flip: power-int-add-1)

lemma power-int-mult-distrib: power-int (x * y :: 'a) m = power-int x m * power-int y m
  by (auto simp: power-int-def power-mult-distrib)

lemmas power-int-mult-distrib-numeral1 = power-int-mult-distrib [where x = numeral w for w, simp]
lemmas power-int-mult-distrib-numeral2 = power-int-mult-distrib [where y = numeral w for w, simp]

lemma power-int-divide-distrib: power-int (x / y :: 'a) m = power-int x m / power-int y m
  using power-int-mult-distrib[of x inverse y m] unfolding power-int-inverse
  by (simp add: field-simps)

end

lemma power-int-add-numeral [simp]:
  power-int x (numeral m) * power-int x (numeral n) = power-int x (numeral (m + n))
  for x :: 'a :: division-ring
  by (simp add: power-int-add [symmetric])

lemma power-int-add-numeral2 [simp]:
power-int \(x\) (numeral \(m\)) * (power-int \(x\) (numeral \(n\)) * \(b\)) = power-int \(x\) (numeral \((m + n)\)) * \(b\)
for \(x\) :: 'a :: division-ring
by (simp add: mult.assoc [symmetric])

lemma power-int-mult-numeral [simp]:
  power-int (power-int \(x\) (numeral \(m\))) (numeral \(n\)) = power-int \(x\) (numeral \((m * n)\))
for \(x\) :: 'a :: division-ring
by (simp only: numeral-mult power-int-mult)

lemma power-int-not-zero: \((x :: 'a :: division-ring) \neq 0 \vee n = 0 \implies power-int x\)
\(n \neq 0\)
by (subst power-int-eq-0-iff) auto

lemma power-int-one-over [field-simps, field-split-simps, divide-simps]:
  power-int \((1 / x) :: 'a :: division-ring\) \(n = 1 / power-int x\)
using power-int-inverse[of \(x\)] by (simp add: divide-inverse)

context
  assumes SORT-CONSTRAINT('a :: linordered-field)
begin

lemma power-int-numeral-neg-numeral [simp]:
  power-int (numeral \(m\)) (-numeral \(n\)) = (inverse (numeral (Num.pow \(m\) \(n\))) :: 'a)
by (simp add: power-int-minus)

lemma zero-less-power-int [simp]: \(0 < (x :: 'a) \implies 0 < power-int x\)
\(n\)
by (auto simp: power-int-def)

lemma zero-le-power-int [simp]: \(0 \leq (x :: 'a) \implies 0 \leq power-int x\)
\(n\)
by (auto simp: power-int-def)

lemma power-int-mono: \((x :: 'a) \leq y \implies n \geq 0 \implies 0 \leq x \implies power-int x\)
\(n\) \(n\)
by (cases n rule: int-cases4) (auto intro: power-mono)

lemma one-le-power-int [simp]: \(1 \leq (x :: 'a) \implies n \geq 0 \implies 1 \leq power-int x\)
\(n\)
\(n\)
using power-int-mono[of \(1 x\) \(n\)] by simp

lemma power-int-le-one: \(0 \leq (x :: 'a) \implies n \geq 0 \implies x \leq 1 \implies power-int x\)
\(n\) \(n\)
\(1\)
using power-int-mono[of \(x 1\) \(n\)] by simp

lemma power-int-le-imp-le-exp:
  assumes gt1: \(1 < (x :: 'a :: linordered-field)\)
  assumes power-int \(x\) \(m\) \(\leq power-int x\)
\(n\) \(n\) \(\geq 0\)
shows $m \leq n$

proof (cases $m < 0$)
  case True
  with ($n \geq 0$) show ?thesis by simp
next
  case False
  with assms have $x ^ \text{nat } m \leq x ^ \text{nat } n$
    by (simp add: power-int-def)
  from gt1 and this show ?thesis using False ($n \geq 0$) by auto
qed

lemma power-int-le-imp-less-exp:
  assumes gt1: $1 < (x :: 'a :: linordered-field)$
  assumes power-int $x m < \text{power-int } x n n \geq 0$
  shows $m < n$
proof (cases $m < 0$)
  case True
  with assms show ?thesis by simp
next
  case False
  with assms have $x ^ \text{nat } m < x ^ \text{nat } n$
    by (simp add: power-int-def)
  from gt1 and this show ?thesis using False ($n \geq 0$) by auto
qed

lemma power-int-strict-mono:
  $(a :: 'a :: linordered-field) < b \Rightarrow 0 \leq a \Rightarrow 0 < n \Rightarrow \text{power-int } a n < \text{power-int } b n$
  by (auto simp: power-int-def intro!: power-strict-mono)

lemma power-int-mono-iff [simp]:
  fixes $a b :: 'a :: linordered-field$
  shows $[a \geq 0; b \geq 0; n > 0] \Rightarrow \text{power-int } a n \leq \text{power-int } b n \iff a \leq b$
  by (auto simp: power-int-def intro!: power-strict-mono)

lemma power-int-strict-increasing:
  fixes $a :: 'a :: linordered-field$
  assumes $n < N 1 < a$
  shows $\text{power-int } a N > \text{power-int } a n$
proof
  have $*: a ^ \text{nat } (N - n) > a ^ \theta$
    using assms by (intro power-strict-increasing) auto
  have $\text{power-int } a N = \text{power-int } a n * \text{power-int } a (N - n)$
    using assms by (simp flip: power-int-add)
  also have $\ldots > \text{power-int } a n * 1$
    using assms *
    by (intro mult-strict-left-mono zero-less-power-int) (auto simp: power-int-def)
finally show \( \text{thesis by simp} \)
qed

lemma power-int-increasing:

fixes \( a \) :: linordered-field
assumes \( n \leq N \) \( a \geq 1 \)
shows \( \text{power-int } a \ N \geq \text{power-int } a \ n \)
proof
have \( \ast \): \( a \ ^{\text{nat}} \ (N - n) \geq a ^{\ 0} \)
using assms by (intro power-increasing) auto
have power-int \( a \ N = \text{power-int } a \ n * \text{power-int } a \ (N - n) \)
using assms by (simp flip: power-int-add)
also have \( \ldots \geq \text{power-int } a \ n * 1 \)
using assms \( \ast \) by (intro mult-left-mono) (auto simp: power-int-def)
finally show \( \text{thesis by simp} \)
qed

lemma power-int-strict-decreasing:

fixes \( a \) :: linordered-field
assumes \( n < N \) \( 0 < a \ < 1 \)
shows \( \text{power-int } a \ N < \text{power-int } a \ n \)
proof
have \( \ast \): \( a ^{\text{nat}} \ (N - n) < a ^{\ 0} \)
using assms by (intro power-strict-decreasing) auto
have power-int \( a \ N = \text{power-int } a \ n * \text{power-int } a \ (N - n) \)
using assms by (simp flip: power-int-add)
also have \( \ldots < \text{power-int } a \ n * 1 \)
using assms \( \ast \)
by (intro mult-strict-left-mono zero-less-power-int) (auto simp: power-int-def)
finally show \( \text{thesis by simp} \)
qed

lemma power-int-decreasing:

fixes \( a \) :: linordered-field
assumes \( n \leq N \) \( 0 \leq a \ < 1 \)
shows \( \text{power-int } a \ N \leq \text{power-int } a \ n \)
proof (cases \( a = 0 \))
case False
have \( \ast \): \( a ^{\text{nat}} \ (N - n) \leq a ^{\ 0} \)
using assms by (intro power-decreasing) auto
have power-int \( a \ N = \text{power-int } a \ n * \text{power-int } a \ (N - n) \)
using assms False by (simp flip: power-int-add)
also have \( \ldots \leq \text{power-int } a \ n * 1 \)
using assms \( \ast \)
by (intro mult-left-mono) (auto simp: power-int-def)
finally show \( \text{thesis by simp} \)
qed (use assms in (auto simp: power-int-0-left-If))

lemma one-less-power-int: \( 1 < (a :: 'a) \Rightarrow \ 0 < n \Rightarrow 1 < \text{power-int } a \ n \)
using power-int-strict-increasing[of \( 0 \ n \ a \)] by simp
lemma \textit{power-int-abs}: \(\mid\text{power-int } a \ n :: 'a\mid = \text{power-int } \mid a\mid n\)
by (auto simp: power-int-def power-abs)

lemma \textit{power-int-sgn} [simp]: \(\text{sgn (power-int } a \ n :: 'a\} = \text{power-int } (\text{sgn } a) \ n\)
by (auto simp: power-int-def)

lemma \textit{abs-power-int-minus} [simp]: \(\mid \text{power-int } (-a) \ n :: 'a\mid = \mid \text{power-int } a \ n\mid\)
by (simp add: power-int-abs)

lemma \textit{power-int-strict-antimono}:
assumes \(a :: 'a :: \text{linordered-field}\) \(< b\ 0 < a \ n < 0\)
shows \(\text{power-int } a \ n > \text{power-int } b \ n\)
proof –
  have \(\text{inverse } (\text{power-int } a \ (-n)) \mid > \text{inverse } (\text{power-int } b \ (-n))\)
  using assms by (intro less-imp-inverse-less power-int-strict-mono zero-less-power-int)
  auto
  thus ?thesis by (simp add: power-int-minus)
qed

lemma \textit{power-int-antimono}:
assumes \(a :: 'a :: \text{linordered-field}\) \(\leq b\ 0 < a \ n < 0\)
shows \(\text{power-int } a \ n \geq \text{power-int } b \ n\)
using \textit{power-int-strict-antimono}[of \(a\) \(b\) \(n\)] assms by (cases \(a = b\)) auto

end

54.19 Finiteness of intervals

lemma \textit{finite-interval-int1} [iff]: \(\text{finite } \{ i :: \text{int. } a \leq i \land i \leq b\}\)
proof (cases \(a \leq b\))
  case True
  then show ?thesis
  proof (induct \(b\) rule: int-ge-induct)
    case base
    have \(\{i. a \leq i \land i \leq a\} = \{a\}\) by auto
    then show ?case by simp
  next
    case (step \(b\))
    then have \(\{i. a \leq i \land i \leq b + 1\} = \{i. a \leq i \land i \leq b\} \cup \{b + 1\}\) by auto
    with step show ?case by simp
  qed
next
  case False
  then show ?thesis
  by (metis (lifting, no-types) Collect-empty-eq finite.emptyI order-trans)
qed

lemma \textit{finite-interval-int2} [iff]: \(\text{finite } \{ i :: \text{int. } a \leq i \land i < b\}\)
by (rule rev-finite-subset[of finite-interval-int1[of a b]]) auto

lemma finite-interval-int3 [iff]: finite \{ i :: int. a < i ∧ i ≤ b \}
  by (rule rev-finite-subset[of finite-interval-int1[of a b]]) auto

lemma finite-interval-int4 [iff]: finite \{ i :: int. a < i ∧ i < b \}
  by (rule rev-finite-subset[of finite-interval-int1[of a b]]) auto

54.20 Configuration of the code generator

Constructors

definition Pos :: num ⇒ int
  where [simp, code-abbrev]: Pos = numeral

definition Neg :: num ⇒ int
  where [simp, code-abbrev]: Neg n = − (Pos n)

code-datatype 0::int Pos Neg

Auxiliary operations.

definition dup :: int ⇒ int
  where [simp]: dup k = k + k

lemma dup-code [code]:
  dup 0 = 0
  dup (Pos n) = Pos (Num.Bit0 n)
  dup (Neg n) = Neg (Num.Bit0 n)
  by (simp-all add: numeral-Bit0)

definition sub :: num ⇒ num ⇒ int
  where [simp]: sub m n = numeral m − numeral n

lemma sub-code [code]:
  sub Num.One Num.One = 0
  sub (Num.Bit0 m) Num.One = Pos (Num.BitM m)
  sub (Num.Bit1 m) Num.One = Pos (Num.Bit0 m)
  sub Num.One (Num.Bit0 n) = Neg (Num.BitM n)
  sub Num.One (Num.Bit1 n) = Neg (Num.Bit0 n)
  sub (Num.Bit0 m) (Num.Bit0 n) = dup (sub m n)
  sub (Num.Bit1 m) (Num.Bit0 n) = dup (sub m n) + 1
  sub (Num.Bit0 m) (Num.Bit1 n) = dup (sub m n) − 1
  by (simp-all only: sub-def dup-def numeral.simps Pos-def Neg-def numeral-BitM)

lemma sub-BitM-One-eq:
  ⟨(Num.sub (Num.BitM n) num.One) = 2 * (Num.sub n Num.One :: int)⟩
  by (cases n) simp-all

Implementations.
lemma one-int-code [code]: \(1 = \text{Pos Num.One}\)
\hfill by simp

lemma plus-int-code [code]:
\[k + 0 = k\]
\[0 + l = l\]
\[\text{Pos } m + \text{Pos } n = \text{Pos } (m + n)\]
\[\text{Pos } m + \text{Neg } n = \text{sub } m n\]
\[\text{Neg } m + \text{Pos } n = \text{sub } n m\]
\[\text{Neg } m + \text{Neg } n = \text{Neg } (m + n)\]
\hfill for \(k \cdot l :: \text{int}\)
\hfill by simp-all

lemma uminus-int-code [code]:
\[\text{uminus } 0 = (0 :: \text{int})\]
\[\text{uminus } (\text{Pos } m) = \text{Neg } m\]
\[\text{uminus } (\text{Neg } m) = \text{Pos } m\]
\hfill by simp-all

lemma minus-int-code [code]:
\[k - 0 = k\]
\[0 - l = \text{uminus } l\]
\[\text{Pos } m - \text{Pos } n = \text{sub } m n\]
\[\text{Pos } m - \text{Neg } n = \text{Pos } (m + n)\]
\[\text{Neg } m - \text{Pos } n = \text{Neg } (m + n)\]
\[\text{Neg } m - \text{Neg } n = \text{sub } n m\]
\hfill for \(k \cdot l :: \text{int}\)
\hfill by simp-all

lemma times-int-code [code]:
\[k * 0 = 0\]
\[0 * l = 0\]
\[\text{Pos } m * \text{Pos } n = \text{Pos } (m * n)\]
\[\text{Pos } m * \text{Neg } n = \text{Neg } (m * n)\]
\[\text{Neg } m * \text{Pos } n = \text{Neg } (m * n)\]
\[\text{Neg } m * \text{Neg } n = \text{Pos } (m * n)\]
\hfill for \(k \cdot l :: \text{int}\)
\hfill by simp-all

instantiation \(\text{int :: equal}\)

begin

definition \(\text{HOL.equal } k \ l \longleftrightarrow \text{ (}t::\text{int})\)

instance
\hfill by standard (rule equal-int-def)

end
lemma equal-int-code [code]:
HOL.equal 0 (0::int) ↔ True
HOL.equal 0 (Pos l) ↔ False
HOL.equal 0 (Neg l) ↔ False
HOL.equal (Pos k) 0 ↔ False
HOL.equal (Pos k) (Pos l) ↔ HOL.equal k l
HOL.equal (Pos k) (Neg l) ↔ False
HOL.equal (Neg k) 0 ↔ False
HOL.equal (Neg k) (Pos l) ↔ False
HOL.equal (Neg k) (Neg l) ↔ HOL.equal k l
by (auto simp add: equal)

lemma equal-int-refl [code nbe]: HOL.equal k k ↔ True
for k :: int
by (fact equal-refl)

lemma less-eq-int-code [code]:
0 ≤ (0::int) ↔ True
0 ≤ Pos l ↔ True
0 ≤ Neg l ↔ False
Pos k ≤ 0 ↔ False
Pos k ≤ Pos l ↔ k ≤ l
Pos k ≤ Neg l ↔ False
Neg k ≤ 0 ↔ True
Neg k ≤ Pos l ↔ True
Neg k ≤ Neg l ↔ l ≤ k
by simp-all

lemma less-int-code [code]:
0 < (0::int) ↔ False
0 < Pos l ↔ True
0 < Neg l ↔ False
Pos k < 0 ↔ False
Pos k < Pos l ↔ k < l
Pos k < Neg l ↔ False
Neg k < 0 ↔ True
Neg k < Pos l ↔ True
Neg k < Neg l ↔ l < k
by simp-all

lemma nat-code [code]:
nat (Int.Neg k) = 0
nat 0 = 0
nat (Int.Pos k) = nat-of-num k
by (simp-all add: nat-of-num-numeral)

lemma (in ring-1) of-int-code [code]:
of-int (Int.Neg k) = - numeral k
of-int 0 = 0
of-int (Int.Pos k) = numeral k 
by simp-all

Serializer setup.
code-identifier
code-module Int → (SML) Arith and (OCaml) Arith and (Haskell) Arith

quickcheck-params [default-type = int]

hide-const (open) Pos Neg sub dup

De-register int as a quotient type:
lifting-update int.lifting
lifting-forget int.lifting

54.21 Duplicates

lemmas int-sum = of-nat-sum [where 'a=int]
lemmas int-prod = of-nat-prod [where 'a=int]
lemmas zle-int = of-nat-le-iff [where 'a=int]
lemmas int-int-eq = of-nat-eq-iff [where 'a=int]
lemmas nonneg-eq-int = nonneg-int-cases
lemmas double-eq-0-iff = double-zero

lemmas int-distrib =
  distrib-right [of z1 z2 w]
  distrib-left [of w z1 z2]
  left-diff-distrib [of z1 z2 w]
  right-diff-distrib [of w z1 z2]
  for z1 z2 w :: int

end

55 Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets

theory Lattices-Big
  imports Groups-Big Option
begin

55.1 Generic lattice operations over a set

55.1.1 Without neutral element

locale semilattice-set = semilattice
begin

interpretation comp-fun-idem f
by standard (simp-all add: fun-eq-iff left-commute)

definition $F :: 	ext{'a set} \Rightarrow \text{'a}$

where

$eq\text{-fold} : F A = \text{the (Finite-Set.fold} (\lambda x y. \text{Some (case y of None } \Rightarrow x \mid \text{Some z } \Rightarrow f x z)) \text{ None A}$

lemma $eq\text{-fold}$:

assumes finite $A$

shows $F (\text{insert } x A) = \text{Finite-Set.fold} f x A$

proof (rule sym)

let $?f = \lambda x y. \text{Some (case y of None } \Rightarrow x \mid \text{Some z } \Rightarrow f x z)$

interpret comp-fun-idem $?f$

by standard (simp-all add: fun-eq-iff commute left-commute split: option.split)

from assms show $\text{Finite-Set.fold} f x A = F (\text{insert } x A)$

proof

case empty then show $?case$ by (simp add: $eq\text{-fold}'$)

next

case $(\text{insert } y B)$ then show $?case$ by (simp add: insert-commute [of $x$] $eq\text{-fold}'$)

qed

lemma $singleton$ [simp]:

$F \{x\} = x$

by (simp add: $eq\text{-fold}$)

lemma $insert\text{-not-elem}$:

assumes finite $A$ and $x \notin A$ and $A \neq \{\}$

shows $F (\text{insert } x A) = x \ast F A$

proof

from $\langle A \neq \{\} \rangle$ obtain $b$ where $b \in A$ by blast

then obtain $B$ where $\ast : A = \text{insert } b B b \notin B$ by (blast dest: mk-disjoint-insert)

with $\langle \text{finite } A \rangle$ and $\langle x \notin A \rangle$

have finite $(\text{insert } x B)$ and $b \notin \text{insert } x B$ by auto

then have $F (\text{insert } b (\text{insert } x B)) = x \ast F (\text{insert } b B)$

by (simp add: $eq\text{-fold}$)

then show $?thesis$ by (simp add: $\ast \text{ insert-commute}$)

qed

lemma $in\text{-idem}$:

assumes finite $A$ and $x \in A$

shows $x \ast F A = F A$

proof

from assms have $A \neq \{\}$ by auto

with $\langle \text{finite } A \rangle$ show $?thesis$ using $\langle x \in A \rangle$

by (induct $A$ rule: finite-ne-induct) (auto simp add: ac-simps insert-not-elem)

qed

lemma $insert$ [simp]:
assumes finite $A$ and $A \neq \{\}$
shows $F (\text{insert } x \ A) = x \ast F A$
using assms by (cases $x \in A$) (simp-all add: insert-absorb in-idem insert-not-elem)

lemma union:
assumes finite $A$ $A \neq \{}$ and finite $B$ $B \neq \{}$
shows $F (A \cup B) = F A \ast F B$
using assms by (induct $A$ rule: finite-ne-induct) (simp-all add: ac-simps)

lemma remove:
assumes finite $A$ and $x \in A$
shows $F A = (\text{if } A - \{x\} = \{\} \text{ then } x \ast \ \text{F } (A - \{x\}) \text{ else } x)$
proof –
  from assms obtain $B$ where $A = \text{insert } x \ B$ and $x \notin B$ by (blast dest: mk-disjoint-insert)
  with assms show ?thesis by simp
qed

lemma insert-remove:
assumes finite $A$
shows $F (\text{insert } x \ A) = (\text{if } A - \{x\} = \{\} \text{ then } x \ast \ \text{F } (A - \{x\}) \text{ else } x)$
using assms by (cases $x \in A$) (simp-all add: insert-absorb remove)

lemma subset:
assumes finite $A$ $B \neq \{}$ and $B \subseteq A$
shows $F B \ast F A = F A$
proof –
  from assms have $A \neq \{}$ and finite $B$ by (auto dest: finite-subset)
  with assms show ?thesis by (simp add: union [symmetric] Un-absorb1)
qed

lemma closed:
assumes finite $A$ $A \neq \{}$ and elem: $\forall x \ y. \ x \ast y \in \{x, y\}$
shows $F A \in A$
using assms proof (induct rule: finite-ne-induct)
  case singleton then show ?case by simp
next
  case (insert $n \ N$)
  then have $F (\text{insert } n \ N) = h (n \ast F N)$ by simp
qed

lemma hom-commute:
assumes hom: $\forall x \ y. \ h (x \ast y) = h x \ast h y$
and $N$: finite $N$ $N \neq \{}$
shows $h (F N) = F \ (h \ast N)$
using $N$ proof (induct rule: finite-ne-induct)
  case singleton thus ?case by simp
next
  case (insert $n \ N$)
  then have $h (F (\text{insert } n \ N)) = h (n \ast F N)$ by simp

also have \ldots = h \cdot n \cdot (F \cdot N) by (rule hom)
also have \( h \cdot (F \cdot N) = F \cdot (h \cdot N) \) by (rule insert)
also have \( h \cdot n \cdot \ldots = F \cdot (\text{insert} \cdot (h \cdot n) \cdot (h \cdot N)) \)
using \( \text{insert} \) by simp
also have \( \text{insert} \cdot (h \cdot n) \cdot (h \cdot N) = h \cdot \text{insert} \cdot n \cdot N \) by simp
finally show \( ?\text{case} \).

\text{qed}

\textbf{lemma infinite:} ~ finite \( A \) \( \Rightarrow \) \( F \cdot A \) = \text{the None}
\textbf{unfolding} eq-fold' by (cases finite (UNIV::'a set)) (auto intro: finite-subset fold-infinite)

\textbf{end}

\textbf{locale} semilattice-order-set = binary?: semilattice-order + semilattice-set
\textbf{begin}

\textbf{lemma} bounded-iff:
\quad \textbf{assumes} finite \( A \) \textbf{and} \( A \neq \{\} \)
\quad \textbf{shows} \( x \leq F \cdot A \iff (\forall a \in A. \ x \leq a) \)
\quad \textbf{using} \textbf{assms} by (induct rule: finite-ne-induct) simp-all

\textbf{lemma} boundedI:
\quad \textbf{assumes} finite \( A \)
\quad \textbf{assumes} \( A \neq \{\} \)
\quad \textbf{assumes} \( \forall a. \ a \in A \Rightarrow x \leq a \)
\quad \textbf{shows} \( x \leq F \cdot A \)
\quad \textbf{using} \textbf{assms} by (simp add: bounded-iff)

\textbf{lemma} boundedE:
\quad \textbf{assumes} finite \( A \) \textbf{and} \( A \neq \{\} \) \textbf{and} \( x \leq F \cdot A \)
\quad \textbf{obtains} \( \forall a. \ a \in A \Rightarrow x \leq a \)
\quad \textbf{using} \textbf{assms} by (simp add: bounded-iff)

\textbf{lemma} coboundedI:
\quad \textbf{assumes} finite \( A \)
\quad \textbf{and} \( a \in A \)
\quad \textbf{shows} \( F \cdot A \leq a \)
\quad \textbf{proof} ~
\quad \quad \textbf{from} \textbf{assms} \textbf{have} \( A \neq \{\} \) \textbf{by} auto
\quad \quad \textbf{from} \( \text{finite} \cdot A \cdot A \neq \{\} \cdot a \in A \cdot \textbf{show} \ ?\text{thesis} \)
\quad \quad \textbf{proof} (induct rule: finite-ne-induct)
\quad \quad \quad \textbf{case} singleton \quad \textbf{thus} \ ?\text{case} \textbf{by} (simp add: refl)
\quad \quad \quad \textbf{next}
\quad \quad \quad \quad \textbf{case} \ (\text{insert} \ x \ B)
\quad \quad \quad \quad \quad \textbf{from} \ (\text{insert}) \ \textbf{have} \ a = x \lor a \in B \ \textbf{by} \ \textbf{simp}
\quad \quad \quad \quad \quad \textbf{then} \ \textbf{show} \ ?\text{case} \ \textbf{using} \ \textbf{insert} \ \textbf{by} (auto \ intro: \ coboundedI2)
\textbf{qed}
\textbf{qed}
lemma subset-imp:
  assumes A ⊆ B and A ≠ {} and finite B
  shows F B ≤ F A
proof (cases A = B)
  case True then show ?thesis by (simp add: refl)
next
  case False
  have B: B = A ∪ (B - A) using A ⊆ B by blast
  then have F B = F (A ∪ (B - A)) by simp
  also have . . . = F A * F (B - A) using False assms by (subst union) (auto intro: finite-subset)
  also have . . . ≤ F A by simp
  finally show ?thesis.
qed

55.1.2 With neutral element

locale semilattice-neutr-set = semilattice-neutr begin
interpretation comp-fun-idem f by standard (simp-all add: fun-eq-iff left-commute)
definition F :: 'a set ⇒ 'a
where
eq-fold: F A = Finite-Set.fold f 1 A

lemma infinite [simp]:
  ¬ finite A ⇒ F A = 1
by (simp add: eq-fold)

lemma empty [simp]:
  F {} = 1
by (simp add: eq-fold)

lemma insert [simp]:
  assumes finite A
  shows F (insert x A) = x * F A
  using assms by (simp add: eq-fold)

lemma in-idem:
  assumes finite A and x ∈ A
  shows x * F A = F A
proof -
  from assms have A ≠ {} by auto
  with finite A show ?thesis using x ∈ A
  qed
by (induct A rule: finite-ne-induct) (auto simp add: ac-simps)

qed

lemma union:
  assumes finite A and finite B
  shows $F(A \cup B) = F A \ast F B$
  using assms by (induct A) (simp-all add: ac-simps)

lemma remove:
  assumes finite A and $x \in A$
  shows $F A = x \ast F (A - \{x\})$
proof --
  from assms obtain B where $A = \text{insert } x B$ and $x \notin B$
  by (blast dest: mk-disjoint-insert)
  with assms show ?thesis by simp
qed

lemma insert-remove:
  assumes finite A
  shows $F (\text{insert } x A) = x \ast F (A - \{x\})$
  using assms by (cases $x \in A$) (simp-all add: insert-absorb remove)

lemma subset:
  assumes finite A and $B \subseteq A$
  shows $F B \ast F A = F A$
proof --
  from assms have finite B by (auto dest: finite-subset)
  with assms show ?thesis by (simp add: union [symmetric] Un-absorb1)
qed

lemma closed:
  assumes finite A $A \neq \{\}$ and elem: $\forall x \ y. x \ast y \in \{x, y\}$
  shows $F A \in A$
using {finite A} $A \neq \{\}$
proof (induct rule: finite-ne-induct)
  case singleton then show ?case by simp
next
  case insert with elem show ?case by force
qed

end

locale semilattice-order-neutr-set = binary?: semilattice-neutr-order + semilattice-neutr-set
begin

lemma bounded-iff:
  assumes finite A
  shows $x \leq F A \iff (\forall a \in A. x \leq a)$
  using assms by (induct A) simp-all
lemma boundedI:
assumes finite A
assumes \( \forall a. a \in A \implies x \leq a \)
shows \( x \leq F A \)
using assms by (simp add: bounded-iff)

lemma boundedE:
assumes finite A and \( x \leq F A \)
obtains \( \forall a. a \in A \implies x \leq a \)
using assms by (simp add: bounded-iff)

lemma coboundedI:
assumes finite A and \( a \in A \)
shows \( F A \leq a \)
proof –
from assms have \( A \neq \{\} \) by auto
from \( \text{finite } A \) \( \{\} \text{ } \{a \in A\} \) show ?thesis
proof (induct rule: finite-ne-induct)
  case singleton thus ?case by (simp add: refl)
next
  case (insert x B)
  from insert have \( a = x \lor a \in B \) by simp
  then show ?case using insert by (auto intro: coboundedI2)
qed

lemma subset-imp:
assumes \( A \subseteq B \) and finite B
shows \( F B \leq F A \)
proof (cases \( A = B \))
  case True then show ?thesis by (simp add: refl)
next
  case False
  have \( B : B = A \cup (B - A) \) using \( \{A \subseteq B\} \) by blast
  then have \( F B = F (A \cup (B - A)) \) by simp
  also have \( \ldots = F A \ast F (B - A) \) using False assms by (subst union) (auto intro: finite-subset)
  also have \( \ldots \leq F A \) by simp
  finally show ?thesis .
qed

end

55.2 Lattice operations on finite sets

context semilattice-inf
begin
sublocale Inf-fin: semilattice-order-set inf less-eq less
  defines
  Inf-fin \( (\bigsqcap_{f \in n} - [900] \ 900) = \text{Inf-fin}.F \).
end

context semilattice-sup
begin
sublocale Sup-fin: semilattice-order-set sup greater-eq greater
  defines
  Sup-fin \( (\bigvee_{f \in n} - [900] \ 900) = \text{Sup-fin}.F \).
end

55.3 Infimum and Supremum over non-empty sets

context lattice
begin
lemma Inf-fin-le-Sup-fin [simp]:
  assumes finite A and A \( \neq \{\} \)
  shows \( \bigsqcap_{f \in n} A \leq \bigvee_{f \in n} A \)
proof -
  from \( A \neq \{\} \) obtain a where a \( \in A \) by blast
  with \( \text{finite } A \) have \( \bigsqcap_{f \in n} A \leq a \) by (rule Inf-fin.coboundedI)
  moreover from \( \text{finite } A \), \( a \in A \), have \( a \leq \bigvee_{f \in n} A \) by (rule Sup-fin.coboundedI)
  ultimately show \( \text{thesis} \) by (rule order-trans)
qed

lemma sup-Inf-absorb [simp]:
  finite A \( \Rightarrow \) a \( \in A \) \( \Rightarrow \) \( \bigsqcap_{f \in n} A \sqcup a = a \)
  by (rule sup-absorb2) (rule Inf-fin.coboundedI)

lemma inf-Sup-absorb [simp]:
  finite A \( \Rightarrow \) a \( \in A \) \( \Rightarrow \) a \( \sqcap \bigvee_{f \in n} A = a \)
  by (rule inf-absorb1) (rule Sup-fin.coboundedI)
end

context distrib-lattice
begin
lemma sup-Inf1-distrib:
  assumes finite A and A \( \neq \{\} \)
  shows sup x \( (\bigsqcap_{f \in n} A) \) = \( \bigsqcap_{f \in n} \{\text{sup } x \ a | a \in A\}\)
  using assms by (simp add: image-def Inf-fin.hom-commute [where h=sup x, OF)
sup-inf-distrib1)
  (rule arg-cong [where \(f=\text{Inf-fin}\)], blast)

lemma sup-Inf2-distrib:
  assumes \(A: \text{finite } A \neq \{\}\) and \(B: \text{finite } B \neq \{\}\)
  shows \(\sup (\bigcap_{f \in \text{fin}A} f) = \bigcap_{f \in \text{fin}B} \{\sup a \mid a \in A \land b \in B\}\) using \(A\)
  proof (induct rule: finite-ne-induct)
    case singleton then show \(?case\)
      by (simp add: sup-inf-distrib \([OF B]\))
  next
    case (insert \(x\) \(A\))
    have \(\text{finB: finite } \{\sup a \mid a \in A \land b \in B\}\) by blast
    have \(\text{finAB: finite } \{\sup a \mid a \in A \land b \in B\}\)
      proof -
        have \(\{\sup a \mid a \in A \land b \in B\} = (\bigcup a \in A. \bigcup b \in B. \{\sup a b\})\)
          by blast
        thus \(?thesis\) by (simp add: insert(1) \(B(1)\))
      qed
    have \(\text{ne: } \{\sup a \mid a \in A \land b \in B\} \neq \{\}\) using insert \(B\) by blast
    have \(\sup (\bigcap_{f \in \text{fin}(\text{insert } x A)} f) = \sup (\inf (\bigcap_{f \in \text{fin}A}) f)\)
      using insert by simp
    also have \(\ldots = \inf (\text{sup } (\bigcap_{f \in \text{fin}B}) f) (\sup (\bigcap_{f \in \text{fin}A}) f)\) by (rule sup-inf-distrib2)
    also have \(\ldots = \inf (\bigcap_{f \in \text{fin}(\text{sup } x b)} \{\sup a \mid a \in A \land b \in B\})\)
      using insert by simp add: sup-inf-distrib[OF \(B\)]
    also have \(\ldots = \bigcap_{f \in \text{fin}(\{\sup a \mid a \in \text{insert } x A \land b \in B\})}\)
      (is - = \(\bigcap_{f \in \text{fin}(\text{insert } x A)} f\))
      using \(B\) insert
      by (simp add: Inf-fin.union \([OF \text{finB - finAB ne}]\))
    also have \(?M = \{\sup a \mid a \in \text{insert } x A \land b \in B\}\)
      by blast
    finally show \(?case\)
  qed

lemma inf-Sup1-distrib:
  assumes \(\text{finite } A\) and \(A \neq \{\}\)
  shows \(\inf (\bigcup_{f \in \text{fin}A} f) = \bigcup_{f \in \text{fin}B} \{\inf a \mid a \in A\}\) using \(\text{assms}\)
  by (simp add: image-def Sup-fin.hom-commute [where \(h=\text{inf } x\), OF inf-sup-distrib1])
  (rule arg-cong [where \(f=\text{Sup-fin}\)], blast)

lemma inf-Sup2-distrib:
  assumes \(A: \text{finite } A \neq \{\}\) and \(B: \text{finite } B \neq \{\}\)
  shows \(\inf (\bigcup_{f \in \text{fin}A} f) (\bigcup_{f \in \text{fin}B} f) = \bigcup_{f \in \text{fin}} \{\inf a \mid a \in A \land b \in B\}\)
  using \(A\)
  proof (induct rule: finite-ne-induct)
    case singleton thus \(?case\)
      by (simp add: inf-Sup1-distrib \([OF B]\))
  next
case (insert x A)

have finB: finite { inf x b | b ∈ B }
  by (rule finite-surj[where f = %b. inf x b, OF B(1)], auto)

have finAB: finite { inf a b | a b. a ∈ A ∧ b ∈ B }
  proof
    have { inf a b | a b. a ∈ A ∧ b ∈ B } = (∪ a∈A. ∪ b∈B. { inf a b })
      by blast
    thus ?thesis by (simp add: insert(1) B(1))
  qed

have nc: { inf a b | a b. a ∈ A ∧ b ∈ B } ≠ {} using insert B by blast

have inf (∪ f in (insert x A)) (∪ f in B) = inf (sup x (∪ f in A)) (∪ f in B)
  using insert by simp

also have ... = sup (inf x (∪ f in B)) (inf (∪ f in A)) (∪ f in B) by (rule inf-sup-distrib2)

also have ... = sup (∪ f in (inf x b | b ∈ B)) (∪ f in (inf a b | a b. a ∈ A ∧ b ∈ B))
  using insert by simp

also have ... = ∪ f in (inf x b | b ∈ B) ∪ { inf a b | a b. a ∈ A ∧ b ∈ B }
  (is - = ∪ f in ?M)

using B insert
  by (simp add: Sup-fin.union [OF finB - finAB nc])

also have ?M = { inf a b | a b. a ∈ insert x A ∧ b ∈ B }
  by blast

finally show ?case .

ded

context complete-lattice
begin

lemma Inf-fin-Inf:
  assumes finite A and A ≠ {}
  shows ∩ f in A = ∩ A
proof
  from assms obtain b B where A = insert b B and finite B by auto
  then show ?thesis
    by (simp add: Inf-fin.eq-fold inf-inf-fold-inf inf.commute [of b])

lemma Sup-fin-Sup:
  assumes finite A and A ≠ {}
  shows ∪ f in A = ∪ A
proof
  from assms obtain b B where A = insert b B and finite B by auto
  then show ?thesis
    by (simp add: Sup-fin.eq-fold sup-sup-fold-sup sup.commute [of b])

ded
55.4 Minimum and Maximum over non-empty sets

context linorder
begin

sublocale Min: semilattice-order-set min less-eq less + Max: semilattice-order-set max greater-eq greater defines Min = Min.F and Max = Max.F ..

end

syntax
-MIN1 :: pttrns ⇒ 'b ⇒ 'b ((3MIN -./ .) [0, 10] 10)
-MIN :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3MIN -∈ .) [0, 0, 10] 10)
-MAX1 :: pttrns ⇒ 'b ⇒ 'b ((3MAX -./ .) [0, 10] 10)
-MAX :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3MAX -∈ .) [0, 0, 10] 10)

translations
MIN x y. f ≜ MIN x. MIN y. f
MIN x. f ≜ CONST Min (CONST range (λx. f))
MIN x∈A. f ≜ CONST Min ((λx. f) ' A)
MAX x y. f ≜ MAX x. MAX y. f
MAX x. f ≜ CONST Max (CONST range (λx. f))
MAX x∈A. f ≜ CONST Max ((λx. f) ' A)

An aside: Min/Max on linear orders as special case of Inf-fin/Sup-fin

lemma Inf-fin-Min:
Inf-fin = (Min : 'a::{semilattice-inf, linorder} set ⇒ 'a)
by (simp add: Inf-fin-def Min-def inf-min)

lemma Sup-fin-Max:
Sup-fin = (Max : 'a::{semilattice-sup, linorder} set ⇒ 'a)
by (simp add: Sup-fin-def Max-def sup-max)

context linorder
begin

lemma dual-min:
ord.min greater-eq = max
by (auto simp add: ord.min-def max-def fun-eq-iff)

lemma dual-max:
ord.max greater-eq = min
by (auto simp add: ord.max-def min-def fun-eq-iff)

lemma dual-Min:
linorder.Min greater-eq = Max
proof –
interpret dual: linorder greater-eq greater by (fact dual-linorder)
THEORY “Lattices-Big”

show ?thesis by (simp add: dual.Min-def dual-min Max-def)
qed

lemma dual-Max:
  linorder.Max greater-eq = Min
proof –
  interpret dual: linorder greater-eq greater by (fact dual-linorder)
  show ?thesis by (simp add: dual.Max-def dual-max Min-def)
qed

lemmas Min-singleton = Min.singleton
lemmas Max-singleton = Max.singleton
lemmas Min-insert = Min.insert
lemmas Max-insert = Max.insert
lemmas Min-Un = Min.union
lemmas Max-Un = Max.union
lemmas hom-Min-commute = Min.hom-commute
lemmas hom-Max-commute = Max.hom-commute

lemma Min-in [simp];
  assumes finite A and A ≠ {}
  shows Min A ∈ A
  using assms by (auto simp add: min-def Min.closed)

lemma Max-in [simp];
  assumes finite A and A ≠ {}
  shows Max A ∈ A
  using assms by (auto simp add: max-def Max.closed)

lemma Min-insert2:
  assumes finite A and min: ∀ b. b ∈ A ⇒ a ≤ b
  shows Min (insert a A) = a
proof (cases A = { })
  case True
  then show ?thesis by simp
next
case False
with ⟨finite A; have Min (insert a A) = min a (Min A) ⟩
  by simp
moreover from ⟨finite A; ⟨A ≠ {}; min have a ≤ Min A by simp
ultimately show ?thesis by (simp add: min.absorb1)
qed

lemma Max-insert2:
  assumes finite A and max: ∀ b. b ∈ A ⇒ b ≤ a
  shows Max (insert a A) = a
proof (cases A = { })
  case True
  then show ?thesis by simp
next
  case False
  with \{finite A\} have \text{Max} (\text{insert} a A) = \text{max} a (\text{Max} A)
    by simp
  moreover from \{finite A\} \{A \neq \{\}\} \text{max} \text{have} \text{Max} A \leq a \text{ by simp}
  ultimately show \{thesis by (simp add: max.absorb1)\}
  qed

lemma \text{Max-const}[simp]: [\{finite A; A \neq \{\}\} \implies \text{Max} ((\lambda_. c) ' A) = c
using \text{Max-in image-is-empty} by blast

lemma \text{Min-const}[simp]: [\{finite A; A \neq \{\}\} \implies \text{Min} ((\lambda_. c) ' A) = c
using \text{Min-in image-is-empty} by blast

lemma \text{Min-le}[simp]:
  assumes \text{finite} A \text{ and} x \in A
  shows \text{Min} A \leq x
  using assms by (fact Min.coboundedI)

lemma \text{Max-ge}[simp]:
  assumes \text{finite} A \text{ and} x \in A
  shows x \leq \text{Max} A
  using assms by (fact Max.coboundedI)

lemma \text{Min-eqI}:
  assumes \text{finite} A
  assumes \\(\forall y. y \in A \implies y \geq x\)
    and \text{x \in A}
  shows \text{Min} A = x
proof (rule order.antisym)
  from \{x \in A\} \text{have} A \neq \{\} \text{ by auto}
  with assms show \text{Min} A \geq x \text{ by simp}
next
  from assms show x \geq \text{Min} A \text{ by simp}
  qed

lemma \text{Max-eqI}:
  assumes \text{finite} A
  assumes \\(\forall y. y \in A \implies y \leq x\)
    and \text{x \in A}
  shows \text{Max} A = x
proof (rule order.antisym)
  from \{x \in A\} \text{have} A \neq \{\} \text{ by auto}
  with assms show \text{Max} A \leq x \text{ by simp}
next
  from assms show x \leq \text{Max} A \text{ by simp}
  qed

lemma \text{eq-Min-iff}:
THEORY “Lattices-Big”

[ finite A; A ≠ {} ] ⇒ m = Min A ←→ m ∈ A ∧ (∀ a ∈ A. m ≤ a)
by (meson Min-in Min-le order.antisym)

lemma Min-eq-iff:
[ finite A; A ≠ {} ] ⇒ Min A = m ←→ m ∈ A ∧ (∀ a ∈ A. m ≤ a)
by (meson Min-in Min-le order.antisym)

lemma eq-Max-iff:
[ finite A; A ≠ {} ] ⇒ m = Max A ←→ m ∈ A ∧ (∀ a ∈ A. a ≤ m)
by (meson Max-in Max-ge order.antisym)

lemma Max-eq-iff:
[ finite A; A ≠ {} ] ⇒ Max A = m ←→ m ∈ A ∧ (∀ a ∈ A. a ≤ m)
by (meson Max-in Max-ge order.antisym)

context
  fixes A :: 'a set
  assumes fin-nonempty: finite A A ≠ {}
begin

lemma Min-le-iff [simp]:
x ≤ Min A ←→ (∀ a ∈ A. x ≤ a)
using fin-nonempty by (fact Min.bounded-iff)

lemma Max-le-iff [simp]:
Max A ≤ x ←→ (∀ a ∈ A. a ≤ x)
using fin-nonempty by (fact Max.bounded-iff)

lemma Min-gr-iff [simp]:
x < Min A ←→ (∀ a ∈ A. x < a)
using fin-nonempty by (induct rule: finite-ne-induct) simp-all

lemma Max-less-iff [simp]:
Max A < x ←→ (∀ a ∈ A. a < x)
using fin-nonempty by (induct rule: finite-ne-induct) simp-all

lemma Min-le-iff:
Min A ≤ x ←→ (∃ a ∈ A. a ≤ x)
using fin-nonempty by (induct rule: finite-ne-induct) (simp-all add: min-le-iff-disj)

lemma Max-ge-iff:
x ≤ Max A ←→ (∃ a ∈ A. x ≤ a)
using fin-nonempty by (induct rule: finite-ne-induct) (simp-all add: le-max-iff-disj)

lemma Min-less-iff:
Min A < x ←→ (∃ a ∈ A. a < x)
using fin-nonempty by (induct rule: finite-ne-induct) (simp-all add: min-less-iff-disj)

lemma Max-gr-iff:
x < Max A ←→ (∃a∈A. x < a)
using fin-nonempty by (induct rule: finite-ne-induct) (simp-all add: less-max-iff-disj)

end

lemma Max-eq-if:
assumes finite A finite B ∀ a∈A. ∃ b∈B. a ≤ b ∀ b∈B. ∃ a∈A. b ≤ a
shows Max A = Max B
proof cases
  assume A = {} thus ?thesis using assms by simp
next
  assume A ≠ {} thus ?thesis using assms
  by (blast intro: order.antisym Max-in Max-ge-iff [THEN iffD2])
qed

lemma Min-antimono:
assumes M ⊆ N and M ≠ {} and finite N
shows Min N ≤ Min M
using assms by (fact Min.subset-imp)

lemma Max-mono:
assumes M ⊆ N and M ≠ {} and finite N
shows Max M ≤ Max N
using assms by (fact Max.subset-imp)

lemma mono-Min-commute:
assumes mono f
assumes finite A and a ≠ {} shows f (Min A) = Min (f ' A)
proof (rule linorder-class.Min-eqI [symmetric])
  from finite A show finite (f ' A) by simp
  from assms show f (Min A) ∈ f ' A by simp
  fix x
  assume x ∈ f ' A
  then obtain y where y ∈ A and x = f y ..
  with assms have Min A ≤ y by auto
  with mono f have f (Min A) ≤ f y by (rule monoE)
  with x = f y show f (Min A) ≤ x by simp
qed

lemma mono-Max-commute:
assumes mono f
assumes finite A and A ≠ {} shows f (Max A) = Max (f ' A)
proof (rule linorder-class.Max-eqI [symmetric])
  from finite A show finite (f ' A) by simp
  from assms show f (Max A) ∈ f ' A by simp
  fix x
  assume x ∈ f ' A
then obtain \( y \) where \( y \in A \) and \( x = f y \).

with assms have \( y \leq \text{Max} A \) by auto

with \( \langle \text{mono} f \rangle \) have \( f y \leq f (\text{Max} A) \) by (rule monoE)

with \( \langle x = f y \rangle \) show \( x \leq f (\text{Max} A) \) by simp

qed

lemma finite-linorder-max-induct [consumes 1, case-names empty insert]:

assumes fin: finite A

and empty: \( P \{\} \)

and insert: \( \forall b A. \text{finite} A \implies \forall a \in A. a < b \implies P A \implies P (\text{insert} b A) \)

shows \( P A \)

using \( \text{fin} \) empty insert

proof (induct rule: finite-psubset-induct)

case (psubset A)

have IH: \( \forall A. [B < A; P \{\}; (\forall A b. [\text{finite} A; \forall a \in A. a b; P A] \implies P (\text{insert} b A))] \implies P B \) by fact

have fin: finite A by fact

have empty: \( P \{\} \) by fact

have step: \( \forall A. [\text{finite} A; \forall a \in A. a < b; P A] \implies P (\text{insert} b A) \) by fact

show \( P A \)

proof (cases \( A = \{\} \))

assume \( A = \{\} \)

then show \( P A \) using \( P \{\} \) by simp

next

let \( ?B = A - \{\text{Max} A\} \)

let \( ?A = \text{insert} (\text{Max} A) ?B \)

have finite ?B using \( \text{finite} ?A \) by simp

assume \( A \neq \{\} \)

with \( \text{finite} ?A \) have \( \text{Max} A \in A \) by auto

then have \( A: ?A = A \) using insert-Diff-single insert-absorb by auto

then have \( P ?B \) using \( P \{\} \) step IH [of ?B] by blast

moreover

have \( \forall a \in ?B. a < \text{Max} A \) using Max-ge [OF \( \text{finite} ?A \) ] by fastforce

ultimately show \( P A \) using \( A \) insert-Diff-single step [OF \( \text{finite} ?B \) ] by fastforce

qed

qed

lemma finite-linorder-min-induct [consumes 1, case-names empty insert]:

\[ \text{finite} A; P \{\}; \forall b A. [\text{finite} A; \forall a \in A. b < a; P A] \implies P (\text{insert} b A) \implies P A \]

by (rule linorder.finite-linorder-max-induct [OF dual-linorder])

lemma finite-ranking-induct[consumes 1, case-names empty insert]:

fixes \( f :: b \Rightarrow 'a \)

assumes finite S

assumes \( P \{\} \)

assumes \( \forall x S. \text{finite} S \implies (\forall y. y \in S \implies f y \leq f x) \implies P S \implies P (\text{insert} x S) \)
shows \( P S \)

**proof** (induction rule: finite-psubset-induct)

**case** (psubset \( A \))

\{
  
  assume \( A \neq \{\} \)

  hence \( f \cdot A \neq \{\} \) and \( \text{finite} (f \cdot A) \)

  using psubset finite-image-iff by simp+

  then obtain \( a \) where \( f a = \text{Max} (f \cdot A) \) and \( a \in A \)

  by (metis Max-in[of \( f \cdot A \)] imageE)

  then have \( P (A - \{a\}) \)

  using psubset member-remove by blast

  moreover

  have \( \forall y. y \in A \Longrightarrow f y \leq f a \)

  using \( f a = \text{Max} (f \cdot A) \cdot \langle \text{finite} (f \cdot A) \rangle \) by simp

  ultimately

  have \( \text{?case} \)

  by (metis \( a \in A \cdot \text{Diff} \text{D1 insert-Diff} \text{assms(3)} \text{finite-Diff} \text{psubset.hyps})

  thus \( \text{?case} \)

  using assms(2) by blast

**qed**

**lemma** Least-Min:

assumes finite \( \{a. P a\} \) and \( \exists a. P a \)

shows \( (\text{LEAST} a. P a) = \text{Min} \{a. P a\} \)

**proof** −

\{
  
  fix \( A :: 'a \ set \)

  assume \( A: \text{finite} A A \neq \{\} \)

  have \( (\text{LEAST} a. a \in A) = \text{Min} A \)

  using \( A \) proof (induct \( A \) rule: finite-ne-induct)

  case singleton show \( \text{?case} \) by (rule Least-equality) simp-all

  next

  case (insert \( a \) \( A \))

  have \( (\text{LEAST} b. b = a \lor b \in A) = \text{min} a (\text{LEAST} a. a \in A) \)

  by (auto intro!: Least-equality simp add: min_def not-le Min-le-iff insert.hyps dest!: less-imp-le)

  with insert show \( \text{?case} \) by simp

**qed**

**lemma** infinite-growing:

assumes \( X \neq \{\} \)

assumes \( *: \forall x. x \in X \Longrightarrow \exists y \in X. y > x \)

shows \( \neg \text{finite} X \)

**proof**

assume \( \text{finite} X \)

with \( \langle X \neq \{\} \rangle \) have \( \text{Max} X \in X \forall x \in X. x \leq \text{Max} X \)
by auto

with *[of Max X] show False
by auto

qed

end

lemma sum-le-card-Max: finite A \Rightarrow \sum f A \leq \card A \cdot \Max (f \cdot A)
using sum-bounded-above[of A f Max (f \cdot A)] by simp

lemma card-Min-le-sum: finite A \Rightarrow \card A \cdot \Min (f \cdot A) \leq \sum f A
using sum-bounded-below[of A Min (f \cdot A) f] by simp

context linordered-ab-semigroup-add
begin

lemma Min-add-commute:
fixes k
assumes finite S and S \neq {} 
shows Min ((\lambda x. f x + k) \cdot S) = Min(f \cdot S) + k
proof –
have m: \forall x y. \min x y + k = \min (x+k) (y+k)
  by (simp add: min-def order.antisym add-right-mono)
have (\lambda x. f x + k) \cdot S = (\lambda y. y + k) \cdot (f \cdot S) by auto
also have Min \ldots = Min (f \cdot S) + k
  using assms hom-Min-commute [of \lambda y. y+k f \cdot S, OF m, symmetric] by simp
finally show ?thesis by simp

qed

lemma Max-add-commute:
fixes k
assumes finite S and S \neq {} 
shows Max ((\lambda x. f x + k) \cdot S) = Max(f \cdot S) + k
proof –
have m: \forall x y. \max x y + k = \max (x+k) (y+k)
  by (simp add: max-def order.antisym add-right-mono)
have (\lambda x. f x + k) \cdot S = (\lambda y. y + k) \cdot (f \cdot S) by auto
also have Max \ldots = Max (f \cdot S) + k
  using assms hom-Max-commute [of \lambda y. y+k f \cdot S, OF m, symmetric] by simp
finally show ?thesis by simp

qed

end

context linordered-ab-group-add
begin

lemma minus-Max-eq-Min [simp]:
finite S \Rightarrow S \neq {} \Rightarrow \neg \Max S = Min (uminus \cdot S)
by (induct S rule: finite-ne-induct) (simp-all add: minus-max-eq-min)

lemma minus-Min-eq-Max [simp]:
  finite S =⇒ S ≠ {} =⇒ − Min S = Max (uminus ' S)
by (induct S rule: finite-ne-induct) (simp-all add: minus-min-eq-max)

end

context complete-linorder
begin

lemma Min-Inf:
  assumes finite A and A ≠ {} shows Min A = Inf A
proof –
from assms obtain b B where A = insert b B and finite B by auto
then show ?thesis
  by (simp add: Min.eq-fold complete-linorder-inf-min[symmetric] inf-Inf-fold-inf
       inf.commute[of b])
qed

lemma Max-Sup:
  assumes finite A and A ≠ {} shows Max A = Sup A
proof –
from assms obtain b B where A = insert b B and finite B by auto
then show ?thesis
  by (simp add: Max.eq-fold complete-linorder-sup-max[symmetric] sup-Sup-fold-sup
       sup.commute[of b])
qed

end

lemma disjnt-ge-max:
  ⟨disjnt X Y⟩ if ⟨finite Y⟩ ⟨∀x. x ∈ X =⇒ x > Max Y⟩
using that by (auto simp add: disjnt-def) (use Max-less-iff in blast)

55.5 Arg Min

context ord
begin

definition is-arg-min :: ('b ⇒ 'a) ⇒ ('b ⇒ bool) ⇒ 'b ⇒ bool where
  is-arg-min f P x = (P x ∧ ¬ (∃ y. P y ∧ f y < f x))

definition arg-min :: ('b ⇒ 'a) ⇒ ('b ⇒ bool) ⇒ 'b where
  arg-min f P = (SOME x. is-arg-min f P x)

definition arg-min-on :: ('b ⇒ 'a) ⇒ 'b set ⇒ 'b where
arg-min-on \( f \subseteq S \) = arg-min \( f \ (\lambda x. \ x \in S) \) 

end

syntax

\(-\text{arg-min} :: \ ('b \Rightarrow \ 'a) \Rightarrow \ \text{pttrn} \Rightarrow \ \text{bool} \Rightarrow \ 'b\)

((\text{3ARG}^{-1}\text{-MIN} \ (-/-) \ [1000, \ 0, \ 10] \ 10)\))

translations

\( \text{ARG-MIN} \ f \ x \ . \ P \\rightleftharpoons \ \text{CONST} \ \text{arg-min} \ f \ (\lambda x. \ P) \)

lemma is-arg-min-linorder: fixes \( f :: \ 'a \Rightarrow \ 'b :: \ \text{linorder} \)

shows is-arg-min \( f \ P \ x = (P \ x \ \land \ (\forall y. \ P \ y \ \rightarrow \ f \ x \ \leq \ f \ y)) \)

by (auto simp add: is-arg-min-def)

lemma is-arg-min-antimono: fixes \( f :: \ 'a \Rightarrow \ 'b :: \ \text{order} \)

shows \( [ \ [ \ \text{is-arg-min} \ f \ P \ x ; \ f \ y \ \leq \ f \ x ; \ P \ y ] ] \ \Rightarrow \ \text{is-arg-min} \ f \ P \ y \)

by (simp add: order.order-iff-strict is-arg-min-def)

lemma arg-minI:

\[ [ P x; \ \bigwedge x. \ P x = \bigwedge x. \ [ P x; \ \forall y. \ P y \ \rightarrow \ f y \ < \ f x ] \ \Rightarrow \ Q x ] \]

\( \Rightarrow \ Q (\text{arg-min} \ f \ P) \)

apply (simp add: arg-min-def is-arg-min-def)

apply (rule someI2-ex)

apply blast

apply blast

done

lemma arg-min-equality:

\[ [ P k; \ \bigwedge x. \ P x = \bigwedge x. \ [ P x; \ \forall y. \ P y \ \rightarrow \ f k \ \leq \ f x ] \ \Rightarrow \ f (\text{arg-min} \ f \ P) = f k \]

for \( f :: -. \Rightarrow \ 'a::\text{order} \)

apply (rule arg-minI)

apply assumption

apply (simp add: less-le-not-le)

by (metis le-less)

lemma wf-linord-ex-has-least:

\[ [ \ \text{wf} \ r ; \ \forall x y. \ (x, \ y) \in r^+ \ \leftrightarrow \ (y, \ x) \notin r^* ; \ P k ] \]

\( \Rightarrow \ \exists x. \ P x \ \land \ (\forall y. \ P y \ \rightarrow \ (m x, \ m y) \in r^*) \)

apply (drule wf-trancl [THEN wf-eq-minimal [THEN iffD1]])

apply (drule-tac \( x = m \) Collect \( P \) in spec)

by force

lemma ex-has-least-nat: \( P k \Rightarrow \ \exists x. \ P x \ \land \ (\forall y. \ P y \ \rightarrow \ m x \ \leq \ m y) \)

for \( m :: \ 'a \Rightarrow \ \text{nat} \)

apply (simp only: pred-nat-trancl-eq-le [symmetric])

apply (rule wf-pred-nat [THEN wf-linord-ex-has-least])

apply (simp add: less-eq linorder-not-le pred-nat-trancl-eq-le)
by assumption

lemma arg-min-nat-lemma:
  \( P k \implies P(\arg\text{-}\min m \ P) \land (\forall y. \ P y \implies m (\arg\text{-}\min m \ P) \leq m y) \)
for \( m :: 'a \Rightarrow nat \)
apply (simp add: arg-min-def is-arg-min-linorder)
apply (rule someI-ex)
apply (erule ex-has-least-nat)
done

lemmas arg-min-natI = arg-min-nat-lemma [THEN conjunct1]

lemma is-arg-min-arg-min-nat: fixes \( m :: 'a \Rightarrow nat \)
  shows \( P x \implies is\text{-}arg\text{-}min m \ P (\arg\text{-}\min m \ P) \)
by (metis arg-min-nat-lemma is-arg-min-linorder)

lemma arg-min-nat-le: \( P x \implies m (\arg\text{-}\min m \ P) \leq m x \)
for \( m :: 'a \Rightarrow nat \)
by (rule arg-min-nat-lemma [THEN conjunct2, THEN spec, THEN mp])

lemma ex-min-if-finite: \( \text{finite } S; S \neq \{\} \implies \exists m \in S. \neg(\exists x \in S. x < (m::'a::order)) \)
by (induction rule: finite.induct) (auto intro: order.strict-trans)

lemma ex-is-arg-min-if-finite: fixes \( f :: 'a \Rightarrow 'b :: order \)
  shows \( \text{finite } S; S \neq \{\} \implies \exists x. \text{is-arg-min } f (\lambda x. x \in S) x \)
unfolding is-arg-min-def
using ex-min-if-finite[of f ' S]
by auto

lemma arg-min-SOME-Min:
  \( \text{finite } S \implies \text{arg-min-on } f \ S = (\text{SOME } y. y \in S \land f \ y = \text{Min}(f \ S)) \)
unfolding arg-min-on-def arg-min-def is-arg-min-linorder
apply (rule arg-cong[where \( f = \text{Eps} \)])
apply (auto simp: fun-eq-iff intro: Min-eqI[ symmetric])
done

lemma arg-min-if-finite: fixes \( f :: 'a \Rightarrow 'b :: order \)
assumes \( \text{finite } S \ S \neq \{\} \)
shows \( \text{arg-min-on } f \ S \in S \text{ and } \neg(\exists x \in S. f x < f (\text{arg-min-on } f \ S)) \)
using ex-is-arg-min-if-finite[of assms, of f]
unfolding arg-min-on-def arg-min-def is-arg-min-def
by (auto dest!: some1-ex)

lemma arg-min-least: fixes \( f :: 'a \Rightarrow 'b :: \text{linorder} \)
  shows \( \text{finite } S; S \neq \{\}; y \in S \implies f(\text{arg-min-on } f \ S) \leq f y \)
by (simp add: arg-min-SOME-Min inv-into-def2[ symmetric] f-inv-into-f)

lemma arg-min-inj-eq: fixes \( f :: 'a \Rightarrow 'b :: order \)
shows \[ \text{inj-on } f \{ x. \ P x \}; \ P a; \ \forall \ y. \ P y \rightarrow f a \leq f y \] \implies \text{arg-min } f P = a

apply (simp add: arg-min-def is-arg-min-def)
apply (rule someI2[of \ a])
apply (simp add: less-le-not-le)
by (metis inj-on-eq-iff less-le mem-Collect-eq)

55.6 Arg Max

context ord
begin

definition is-arg-max :: \('b \Rightarrow 'a\) \Rightarrow \('b \Rightarrow \text{bool}\) \Rightarrow \('b\)

where

\[ \text{is-arg-max } f P x = (P x \land \neg (\exists y. \ P y \land f y > f x)) \]

definition arg-max :: \('b \Rightarrow 'a\) \Rightarrow \('b \Rightarrow \text{bool}\) \Rightarrow \('b\)

where

\[ \text{arg-max } f P = (\text{SOME } x. \ \text{is-arg-max } f P x) \]

definition arg-max-on :: \('b \Rightarrow 'a\) \Rightarrow \('b\) set \Rightarrow \('b\)

where

\[ \text{arg-max-on } f S = \text{arg-max } f (\lambda x. \ x \in S) \]

end

syntax

-arg-max :: \('b \Rightarrow 'a\) \Rightarrow \text{pttrn} \Rightarrow \text{bool} \Rightarrow \('a\)

((3ARG'\-MAX - -) [1000, 0, 10] 10)

translations

ARG-MAX f x. P \equiv \text{CONST arg-max } f (\lambda x. P)

lemma is-arg-max-linorder: fixes f :: 'a \Rightarrow 'b :: linorder
shows is-arg-max f P x = (P x \land (\forall y. \ P y \rightarrow f x \geq f y))
by (auto simp add: is-arg-max-def)

lemma arg-maxI:

\[ P x \Rightarrow \]

\[ (\forall y. \ P y \rightarrow \neg f y > f x) \Rightarrow \]

\[ (\forall x. \ P x \rightarrow \forall y. \ P y \rightarrow \neg f y > f x \Rightarrow Q x) \Rightarrow \]

\[ Q \ (\text{arg-max } f P) \]

apply (simp add: arg-max-def is-arg-max-def)
apply (rule someI2-ex)
apply blast
apply blast

done

lemma arg-max-equality:

\[ [ P k; \ \forall x. \ P x \Rightarrow f x \leq f k ] \Rightarrow f \ (\text{arg-max } f P) = f k \]

for f :: \- \Rightarrow 'a::order

apply (rule arg-maxI [where f = f])
apply assumption
apply (simp add: less-le-not-le)
by (metis le-less)

lemma ex-has-greatest-nat-lemma:
  \( P \, k \implies \forall \, x. \, P \, x \implies (\exists \, y. \, P \, y \land \neg \, f \, y \leq \, f \, x) \implies \exists \, y. \, P \, y \land \neg \, f \, y < f \, k + n \)
  for \( f :: 'a \Rightarrow \mbox{nat} \)
  by (induct \( n \)) (force simp: le-Suc-eq)+

lemma ex-has-greatest-nat:
  assumes \( P \, k \)
    and \( \forall \, y. \, P \, y \implies (f :: \mbox{nat}) \, y < b \)
  shows \( \exists \, x. \, P \, x \land (\forall \, y. \, P \, y \implies f \, y \leq f \, x) \)
  proof (rule ccontr)
    assume \( \neg \exists \, x. \, P \, x \land (\forall \, y. \, P \, y \implies f \, y \leq f \, x) \)
    then have \( (\exists \, x. \, P \, x) \land (\forall \, y. \, P \, y \land \neg \, f \, y \leq f \, x) \)
      by auto
    then have \( (\exists \, y. \, P \, y \land \neg \, f \, y < f \, k + (b - f \, k)) \)
      using assms ex-has-greatest-nat-lemma[of \( P \, k \) \( f \) \( b \)]
      by blast
    then show \( \mbox{False} \)
      using assms by auto
  qed

lemma arg-max-nat-lemma:
  \[
  [ \, P \, k ; \, \forall \, y. \, P \, y \implies f \, y < b \,] \]
  \( \implies P \, (\arg\,-\max P) \land (\forall \, y. \, P \, y \implies f \, y \leq f \, (\arg\,-\max f \, P)) \)
  for \( f :: 'a \Rightarrow \mbox{nat} \)
  apply (simp add: arg-max-def is-arg-max-linorder)
  apply (rule someI-ex)
  apply (erule (1) ex-has-greatest-nat)
  done

lemmas arg-max-natI = arg-max-nat-lemma [THEN conjunct1]

lemma arg-max-nat-le: \( P \, x \implies \forall \, y. \, P \, y \implies f \, y < b \implies f \, x \leq f \, (\arg\,-\max f \, P) \)
  for \( f :: 'a \Rightarrow \mbox{nat} \)
  by (blast dest: arg-max-nat-lemma [THEN conjunct2, THEN spec, of \( P \)])

end

56 Division in euclidean (semi)rings

theory Euclidean-Rings
  imports Int Lattices-Big
begin

56.1 Euclidean (semi)rings with explicit division and remainder

class euclidean-semiring = semidom-modulo +
fixes euclidean-size :: 'a ⇒ nat
assumes size-0 [simp]: euclidean-size 0 = 0
assumes mod-size-less:
  \( b \neq 0 \implies \text{euclidean-size} (a \mod b) < \text{euclidean-size} b \)
assumes size-mult-mono:
  \( b \neq 0 \implies \text{euclidean-size} a \leq \text{euclidean-size} (a \times b) \)

lemma euclidean-size-eq-0-iff [simp]:
  \( \text{euclidean-size} b = 0 \iff b = 0 \)
proof
  assume b = 0
  then show euclidean-size b = 0
    by simp
next
  assume euclidean-size b = 0
  show b = 0
    proof (rule ccontr)
      assume \( \neg b = 0 \)
      with mod-size-less have euclidean-size (b mod b) < euclidean-size b .
      with \( \text{euclidean-size} b = 0 \) show False
        by simp
    qed
  qed

lemma euclidean-size-greater-0-iff [simp]:
  \( \text{euclidean-size} b > 0 \iff b \neq 0 \)
using euclidean-size-eq-0-iff [symmetric, of b] by safe simp

lemma size-mult-mono': \( b \neq 0 \implies \text{euclidean-size} a \leq \text{euclidean-size} (b \times a) \)
by (subst mult.commute) (rule size-mult-mono)

lemma dvd-euclidean-size-eq-imp-dvd:
  assumes \( a \neq 0 \) and \( \text{euclidean-size} a = \text{euclidean-size} b \)
  and \( b \text{ dvd } a \)
  shows \( a \text{ dvd } b \)
proof (rule ccontr)
  assume \( \neg a \text{ dvd } b \)
  hence \( b \mod a \neq 0 \) using mod-0-imp-dvd [of b a] by blast
  then have \( b \mod a \neq 0 \) by (simp add: mod-0-iff-dvd)
  from \( \langle b \text{ dvd } a \rangle \) have \( b \text{ dvd } b \mod a \) by (simp add: dvd-mod-iff)
  then obtain \( c \) where \( b \mod a = b \times c \) unfolding dvd-def by blast
    with \( \langle b \mod a \neq 0 \rangle \) have \( c \neq 0 \) by auto
    with \( \langle b \mod a = b \times c \rangle \) have \( \text{euclidean-size} (b \mod a) \geq \text{euclidean-size} b \)
      using size-mult-mono by force
  moreover from \( \langle \neg a \text{ dvd } b \rangle \) and \( \langle a \neq 0 \rangle \)
  have \( \text{euclidean-size} (b \mod a) < \text{euclidean-size} a \)
    using mod-size-less by blast
  ultimately show False using \( \langle \text{euclidean-size} a = \text{euclidean-size} b \rangle \)
lemma euclidean-size-times-unit:
assumes is-unit a
shows euclidean-size (a * b) = euclidean-size b
proof (rule antisym)
  from assms have [simp]: a ≠ 0 by auto
  hence 1 div a ≠ 0 by (intro notI) simp-all
  hence euclidean-size (a * b) ≥ euclidean-size ((1 div a) * (a * b))
    by (rule size-mult-mono’)
  also from assms have (1 div a) * (a * b) = b
    by (simp add: algebra-simps unit-div-mult-swap)
  finally show euclidean-size (a * b) ≤ euclidean-size b .
qed

lemma euclidean-size-unit:
is-unit a ⇒ euclidean-size a = euclidean-size 1
using euclidean-size-times-unit [of a 1] by simp

lemma unit-iff-euclidean-size:
is-unit a ←→ euclidean-size a = euclidean-size 1 ∧ a ≠ 0
proof safe
  assume A: a ≠ 0 and B: euclidean-size a = euclidean-size 1
  show is-unit a
    by (rule dvd-euclidean-size-eq-imp-dvd [OF A B]) simp-all
qed (auto intro: euclidean-size-unit)

lemma euclidean-size-times-nonunit:
assumes a ≠ 0 b ≠ 0 ¬ is-unit a
shows euclidean-size b < euclidean-size (a * b)
proof (rule contr)
  assume ¬euclidean-size b < euclidean-size (a * b)
  with size-mult-mono’[OF assms(1), of b]
    have eq: euclidean-size (a * b) = euclidean-size b by simp
  have a * b dvd b
    by (rule dvd-euclidean-size-eq-imp-dvd [OF - eq])
    (use assms in simp-all)
  hence a * b dvd 1 * b by simp
  with (b ≠ 0) have is-unit a by (subst (asm) dvd-times-right-cancel-iff)
  with assms(3) show False by contradiction
qed

lemma dvd-imp-size-le:
assumes a dvd b b ≠ 0
shows euclidean-size a ≤ euclidean-size b
using assms by (auto simp: size-mult-mono)
lemma dvd-proper-imp-size-less:
assumes \( a \text{ dvd } b \Rightarrow b \text{ dvd } a \neq 0 \)
shows \( \text{euclidean-size } a < \text{euclidean-size } b \)
proof
from assms(1) obtain \( c \) where \( b = a \cdot c \) by (erule dvdE)
hence \( z : b = c \cdot a \) by (simp add: mult.commute)
from \( z \) assms have \( \neg \text{is-unit } c \) by (auto simp: mult.commute mult-unit-dvd-iff)
with \( z \) assms show \( \text{?thesis} \)
  by (auto intro!: euclidean-size-times-nonunit)
qed

lemma unit-imp-mod-eq-0:
\( a \mod b = 0 \) if \( \text{is-unit } b \)
using that by (simp add: mod-eq-0-iff-dvd unit-imp-dvd)

lemma mod-eq-self-iff-div-eq-0:
\( a \mod b = a \iff a \div b = 0 \) (is \( ?P \iff ?Q \)
proof
assume \( ?P \)
with \( \text{div-mult-mod-eq [of } a b \) show \( ?Q \)
  by auto
next
assume \( ?Q \)
with \( \text{div-mult-mod-eq [of } a b \) show \( ?P \)
  by simp
qed

lemma coprime-mod-left-iff [simp]:
coprime \( (a \mod b) \) \( b \iff \text{coprime } a \text{ if } b \neq 0 \)
by (rule iffI; rule coprimeI)
  (use that in \( \text{auto dest!: dvd-mod-imp-dvd coprime-common-divisor simp add: dvd-mod-iff} \))

lemma coprime-mod-right-iff [simp]:
coprime \( a \mod (b \mod a) \iff \text{coprime } a \text{ if } a \neq 0 \)
using that coprime-mod-left-iff [of \( a b \) by (simp add: ac-simps)

end

class euclidean-ring = idom-modulo + euclidean-semiring

begin

lemma dvd-diff-commute [ac-simps]:
a \text{ dvd } c - b \iff a \text{ dvd } b - c
proof
  have a \text{ dvd } c - b \iff a \text{ dvd } (c - b) \ast - 1
    by (subst dvd-mult-unit-iff) simp-all
  then show \( ?\text{thesis} \)
end
56.2 Euclidean (semi)rings with cancel rules

class euclidean-semiring-cancel = euclidean-semiring +
  assumes div-mult-self1 [simp]: \( b \neq 0 \implies (a + c \cdot b) \div b = c + a \div b \)
  and div-mult-mult1 [simp]: \( c \neq 0 \implies (c \cdot a) \div (c \cdot b) = a \div b \)

begin

lemma div-mult-self2 [simp]:
  assumes \( b \neq 0 \)
  shows \( (a + b \cdot c) \div b = c + a \div b \)
  using assms div-mult-self1 [of b a c] by (simp add: mult.commute)

lemma div-mult-self3 [simp]:
  assumes \( b \neq 0 \)
  shows \( (c \cdot b + a) \div b = c + a \div b \)
  using assms by (simp add: add.commute)

lemma div-mult-self4 [simp]:
  assumes \( b \neq 0 \)
  shows \( (b \cdot c + a) \div b = c + a \div b \)
  using assms by (simp add: add.commute)

lemma mod-mult-self1 [simp]: \( (a + c \cdot b) \mod b = a \mod b \)
proof (cases b = 0)
  case True then show \(?thesis\) by simp
next
  case False
  have \( a + c \cdot b = (a + c \cdot b) \div b \cdot b + (a + c \cdot b) \mod b \)
    by (simp add: div-mult-mod-eq)
  also from False div-mult-self1 [of b a c] have
    \( \ldots = (c + a \div b) \cdot b + (a + c \cdot b) \mod b \)
    by (simp add: algebra-simps)
  finally have \( a = a \div b \cdot b + (a + c \cdot b) \mod b \)
    by (simp add: add.commute [of a] add.assoc distrib-right)
  then have \( a \div b \cdot b + (a + c \cdot b) \mod b = a \div b \cdot b + a \mod b \)
    by (simp add: div-mult-mod-eq)
  then show \(?thesis\) by simp
qed

lemma mod-mult-self2 [simp]:
  \( (a + b \cdot c) \mod b = a \mod b \)
  by (simp add: mult.commute [of b])

lemma mod-mult-self3 [simp]:
(c * b + a) mod b = a mod b
by (simp add: add.commute)

lemma mod-mul-self4 [simp]:
(b * c + a) mod b = a mod b
by (simp add: add.commute)

lemma mod-mul-self1-is-0 [simp]:
b * a mod b = 0
using mod-mul-self2 [of 0 b a] by simp

lemma mod-mul-self2-is-0 [simp]:
a * b mod b = 0
using mod-mul-self1 [of 0 a b] by simp

lemma div-add-self1:
assumes b ≠ 0
shows (b + a) div b = a div b + 1
using assms div-mult-self1 [of b a 1] by (simp add: add.commute)

lemma div-add-self2:
assumes b ≠ 0
shows (a + b) div b = a div b + 1
using assms div-add-self1 [of b a] by (simp add: add.commute)

lemma mod-add-self1 [simp]:
(b + a) mod b = a mod b
using mod-mul-self1 [of a 1 b] by (simp add: add.commute)

lemma mod-add-self2 [simp]:
(a + b) mod b = a mod b
using mod-mul-self1 [of a 1 b] by simp

lemma mod-div-trivial [simp]:
a mod b div b = 0
proof (cases b = 0)
assume b = 0
thus ?thesis by simp
next
assume b ≠ 0
hence a div b + a mod b div b = (a mod b + a div b * b) div b
by (rule div-mult-self1 [symmetric])
also have . . . = a div b
by (simp only: mod-div-mult-eq)
also have . . . = a div b + 0
by simp
finally show ?thesis
by (rule add-left-imp-eq)
qed
lemma mod-mod-trivial [simp]:
\[ a \mod b \mod b = a \mod b \]
proof -
  have \[ a \mod b \mod b = (a \mod b + a \div b \times b) \mod b \]
    by (simp only: mod-mult-self1)
  also have \[ \ldots = a \mod b \]
    by (simp only: mod-div-mult-eq)
  finally show \[ \text{thesis} \].
qed

lemma mod-mod-cancel:
  assumes \[ c \dvd b \]
  shows \[ a \mod b \mod c = a \mod c \]
proof -
  from \[ \langle c \dvd b \rangle \]
  obtain \[ k \] where \[ b = c \times k \]
    by (rule dvdE)
  have \[ a \mod b \mod c = a \mod (c \times k) \mod c \]
    by (simp only: \[ \langle b = c \times k \rangle \])
  also have \[ \ldots = (a \mod (c \times k) + a \div (c \times k) \times k \times c) \mod c \]
    by (simp only: mod-mult-self1)
  also have \[ \ldots = (a \div (c \times k) \times (c \times k) + a \mod (c \times k)) \mod c \]
    by (simp only: ac-simps)
  also have \[ \ldots = a \mod c \]
    by (simp only: div-mult-mod-eq)
  finally show \[ \text{thesis} \].
qed

lemma div-mult-mult2 [simp]:
  \[ c \neq 0 \implies (a \times c) \div (b \times c) = a \div b \]
by (drule div-mult-mult1) (simp add: mult.commute)

lemma div-mult-mult1-if [simp]:
  \[ (c \times a) \div (c \times b) = (if c = 0 \text{ then } 0 \text{ else } a \div b) \]
by simp-all

lemma mod-mult-mult1:
  \[ (c \times a) \mod (c \times b) = c \times (a \mod b) \]
proof (cases \[ c = 0 \])
  case True then show \[ \text{thesis} \] by simp
next
  case False
  from div-mult-mod-eq
  have \[ ((c \times a) \div (c \times b)) \times (c \times b) \times (c \times a) \mod (c \times b) = c \times a \]
    by simp
  also have False \[ (a \div b) \times b + a \mod b + (c \times a) \mod (c \times b) = c \times a + (a \mod b) \]
    by (simp add: algebra-simps)
  with \[ \text{div-mult-mod-eq} \]
  show \[ \text{thesis} \] by simp
qed
lemma mod-mult-mult2:  
\((a * c) \mod (b * c) = (a \mod b) * c\)  
using mod-mult-mult1 [of c a b] by (simp add: mult.commute)

lemma mult-mod-left:  
\((a \mod b) * c = (a * c) \mod (b * c)\)  
by (fact mod-mult-mult2 [symmetric])

lemma mult-mod-right:  
\(c * (a \mod b) = (c * a) \mod (c * b)\)  
by (fact mod-mult-mult1 [symmetric])

lemma dvd-mod:  
k dvd m \implies k dvd n \implies k dvd (m \mod n)  
unfolding dvd-def by (auto simp add: mod-mult-mult1)

lemma div-plus-div-distrib-dvd-left:  
\(c dvd a \implies (a + b) \div c = a \div c + b \div c\)  
by (cases c = 0) auto

lemma div-plus-div-distrib-dvd-right:  
\(c dvd b \implies (a + b) \div c = a \div c + b \div c\)  
using div-plus-div-distrib-dvd-left [of c b a]  
by (simp add: ac-simps)

lemma sum-div-partition:  
\((\sum a\in A. f a) \div b = (\sum a\in A \cap \{a \div b dvd f a\}. f a \div b) + (\sum a\in A \cap \{a \div b dvd f a\}. f a)\)  
if \(\finite A\)  

proof  
  have \(A = A \cap \{a \div b dvd f a\} \cup A \cap \{a \div b dvd f a\}\)  
    by auto  
  then have \((\sum a\in A. f a) = (\sum a\in A \cap \{a \div b dvd f a\} \cup A \cap \{a \div b dvd f a\}. f a)\)  
    by simp  
  also have \(\ldots = (\sum a\in A \cap \{a \div b dvd f a\}. f a) + (\sum a\in A \cap \{a \div b dvd f a\}. f a)\)  
    using \(\finite A\) by (auto intro: sum.union_inter_neutral)  
  finally have \(*: (\sum f A = \text{sum} f (A \cap \{a \div b dvd f a\}) + \text{sum} f (A \cap \{a \div b dvd f a\}\})\).  
  define B where \(B = A \cap \{a \div b dvd f a\}\)  
  with \(\finite A\) have \(\finite B\) and \(a \in B \implies b \div d f\\) for \(a\)  
    by simp-all  
  then have \((\sum a\in B. f a) \div b = (\sum a\in B. f a \div b)\) and \(b \div d \sum a\in B. f a\)  
    by induction (simp-all add: div-plus-div-distrib-dvd-left)  
  then show \(?thesis using \(*\)  
    by (simp add: B div-plus-div-distrib-dvd-left)

qed

named-theorems mod-simps

Addition respects modular equivalence.
lemma mod-add-left-eq [mod-simps]:
(a mod c + b) mod c = (a + b) mod c
proof
have (a + b) mod c = (a div c * c + a mod c + b) mod c
  by (simp only: div-mult-mod-eq)
also have . . . = (a mod c + b + a div c * c) mod c
  by (simp only: ac-simps)
also have . . . = (a mod c + b) mod c
  by (rule mod-mult-self1)
finally show ?thesis
  by (rule sym)
qed

lemma mod-add-right-eq [mod-simps]:
(a + b mod c) mod c = (a + b) mod c
using mod-add-left-eq [of b c a] by (simp add: ac-simps)

lemma mod-add-eq:
(a mod c + b mod c) mod c = (a + b) mod c
by (simp add: mod-add-left-eq mod-add-right-eq)

lemma mod-sum-eq [mod-simps]:
(\sum_{i \in A}. f i mod a) mod a = sum f A mod a
proof (induct A rule: infinite-finite-induct)
case (insert i A)
  then have (\sum_{i \in insert i A}. f i mod a) mod a
    = (f i mod a + (\sum_{i \in A}. f i mod a)) mod a
    by simp
  also have . . . = (f i + (\sum_{i \in A}. f i mod a) mod a) mod a
    by (simp add: mod-simps)
  also have . . . = (f i + (sum_{i \in A}. f i) mod a) mod a
    by (simp add: insert.hyps)
finally show ?case
  by (simp add: insert.hyps mod-simps)
qed simp-all

lemma mod-add-cong:
assumes a mod c = a' mod c
assumes b mod c = b' mod c
shows (a + b) mod c = (a' + b') mod c
proof
have (a mod c + b mod c) mod c = (a' mod c + b' mod c) mod c
  unfolding assms ..
then show ?thesis
  by (simp add: mod-add-eq)
qed

Multiplication respects modular equivalence.

lemma mod-mult-left-eq [mod-simps]:


THEORY "Euclidean-Rings"

((a mod c) * b) mod c = (a * b) mod c

proof -
  have (a * b) mod c = ((a div c * c + a mod c) * b) mod c
    by (simp only: div-mult-mod-eq)
  also have ... = (a mod c * b + a div c * b * c) mod c
    by (simp only: algebra-simps)
  also have ... = (a mod c * b) mod c
    by (rule mod-mult-self1)
  finally show ?thesis
    by (rule sym)
qed

lemma mod-mult-right-eq [mod-simps]:
  (a * (b mod c)) mod c = (a * b) mod c
  using mod-mult-left-eq [of b c a] by (simp add: ac-simps)

lemma mod-mult-eq:
  ((a mod c) * (b mod c)) mod c = (a * b) mod c
  by (simp add: mod-mult-left-eq mod-mult-right-eq)

lemma mod-prod-eq [mod-simps]:
  (∏ i∈A. f i mod a) mod a = prod f A mod a
proof (induct A rule: infinite-finite-induct)
  case (insert i A)
  then have (∏ i∈insert i A. f i mod a) mod a
    = (f i mod a * (∏ i∈A. f i mod a)) mod a
    by simp
  also have ... = (f i * (∏ i∈A. f i mod a)) mod a
    by (simp add: mod-simps)
  also have ... = (f i * (∏ i∈A. f i) mod a)) mod a
    by (simp add: insert.hyps)
  finally show ?case
    by (simp add: insert.hyps mod-simps)
qed simp-all

lemma mod-mult-cong:
  assumes a mod c = a' mod c
  assumes b mod c = b' mod c
  shows (a * b) mod c = (a' * b') mod c
proof -
  have (a mod c * (b mod c)) mod c = (a' mod c * (b' mod c)) mod c
    unfolding assms ..
  then show ?thesis
    by (simp add: mod-mult-eq)
qed

Exponentiation respects modular equivalence.

lemma power-mod [mod-simps]:
  ((a mod b) ^ n) mod b = (a ^ n) mod b
proof (induct n)
case 0
  then show ?case by simp
next
  case (Suc n)
  have \((a \mod b) ^ \text{Suc} n \mod b = (a \mod b) \ast ((a \mod b) ^ \text{n} \mod b) \mod b\)
    by (simp add: mod-mult-right-eq)
  with Suc show ?case
    by (simp add: mod-mult-left-eq mod-mult-right-eq)
qed

lemma power-diff-power-eq:
\(<a ^ m \div a ^ n = (if n \leq m then a ^ (m - n) else 1 \div a ^ (n - m))>\)
proof (cases \(<n \leq m>\))
  case True
  with that power-diff [symmetric, of a n m] show ?thesis by simp
next
  case False
  then obtain q where n:\(<n = m + Suc q>\)
    by (auto simp add: not-le dest: less-imp-Suc-add)
  then have \(<a ^ m \div a ^ n = (a ^ m \ast Suc q)\)
    by (simp add: power-add ac-simps)
  moreover from that have \(<a ^ m \neq 0>\)
    by simp
  ultimately have \(<a ^ m \div a ^ n = 1 \div a ^ Suc q>\)
    by (subst (asm) div-mult-mult1) simp
  with False n show ?thesis
    by simp
qed

end

class euclidean-ring-cancel = euclidean-ring + euclidean-semiring-cancel
begin
subclass idom-divide ..

lemma div-minus-minus [simp]: \((- a) \div (- b) = a \div b\)
  using div-mult-mult1 [of - 1 a b] by simp

lemma mod-minus-minus [simp]: \((- a) \mod (- b) = -(a \mod b)\)
  using mod-mult-mult1 [of - 1 a b] by simp

lemma div-minus-right: \(a \div (- b) = -(a) \div b\)
  using div-minus-minus [of - a b] by simp

lemma mod-minus-right: \(a \mod (- b) = - ((- a) \mod b)\)
using mod-minus-minus [of \(- a \ b\)] by simp

lemma div-minus1-right [simp]: \(a \div (- 1) = - a\)
using div-minus-right [of \(a \ b\)] by simp

lemma mod-minus1-right [simp]: \(a \mod (- 1) = 0\)
using mod-minus-right [of \(a \ b\)] by simp

Negation respects modular equivalence.

lemma mod-minus-eq [mod-simps]:
\((- (a \mod b)) \mod b = (- a) \mod b\)
proof -
  have \((- a) \mod b = (- (a \div b * b + a \mod b)) \mod b\)
    by (simp only: div-mult-mod-eq)
  also have \(\ldots = (- (a \mod b) + - (a \div b) * b) \mod b\)
    by (simp add: ac-simps)
  also have \(\ldots = (- (a \mod b)) \mod b\)
    by (rule mod-mult-self1)
  finally show ?thesis
    by (rule sym)
qed

lemma mod-minus-cong:
  assumes \(a \mod b = a' \mod b\)
  shows \((- a) \mod b = (- a') \mod b\)
proof -
  have \((- (a \mod b)) \mod b = (- (a' \mod b)) \mod b\)
    unfolding assms ..
  then show ?thesis
    by (simp add: mod-minus-eq)
qed

Subtraction respects modular equivalence.

lemma mod-diff-left-eq [mod-simps]:
\((a \mod c - b) \mod c = (a - b) \mod c\)
using mod-add-cong [of \(a \ c \ a \mod c - b - b\)]
by simp

lemma mod-diff-right-eq [mod-simps]:
\((a - b \mod c) \mod c = (a - b) \mod c\)
using mod-add-cong [of \(a \ c \ a - b - (b \mod c)\)] mod-minus-cong [of \(b \mod c \ c \ b\)]
by simp

lemma mod-diff-eq:
\((a \mod c - b \mod c) \mod c = (a - b) \mod c\)
using mod-add-cong [of \(a \ c \ a \mod c - b - (b \mod c)\)] mod-minus-cong [of \(b \mod c \ c \ b\)]
by simp
lemma mod-diff-cong:
\hspace{1em} \text{assumes } a \mod c = a' \mod c
\hspace{1em} \text{assumes } b \mod c = b' \mod c
\hspace{1em} \text{shows } (a - b) \mod c = (a' - b') \mod c
\hspace{1em} \text{using } \text{assms } \text{mod-add-cong } [\text{of } a \text{ c} \text{ a'} - b - b'] \text{ mod-minus-cong } [\text{of } b \text{ c} \text{ b'}]
\hspace{1em} \text{by } \text{simp}

lemma minus-mod-self2 [simp]:
\hspace{1em} (a - b) \mod b = a \mod b
\hspace{1em} \text{using } \text{mod-diff-right-eq } [\text{of } a \text{ b} \text{ b'}]
\hspace{1em} \text{by } (\text{simp add: mod-diff-right-eq})

lemma minus-mod-self1 [simp]:
\hspace{1em} (b - a) \mod b = -a \mod b
\hspace{1em} \text{using } \text{mod-add-self2 } [\text{of } -a \text{ b} \text{ b'}]
\hspace{1em} \text{by } \text{simp}

lemma mod-eq-dvd-iff:
\hspace{1em} a \mod c = b \mod c \iff c \dvd a - b
\hspace{1em} \text{proof}
\hspace{2em} \text{assume } ?P
\hspace{2em} \text{then have } (a \mod c - b \mod c) \mod c = 0
\hspace{3em} \text{by } \text{simp}
\hspace{2em} \text{then show } ?Q
\hspace{3em} \text{by } (\text{simp add: dvd-eq-mod-eq-0 mod-simps})
\hspace{1em} \text{next}
\hspace{2em} \text{assume } ?Q
\hspace{2em} \text{then obtain } d \text{ where } d: a - b = c \ast d
\hspace{3em} \text{then have } a = c \ast d + b
\hspace{4em} \text{by } (\text{simp add: algebra-simps})
\hspace{2em} \text{then show } ?P \text{ by } \text{simp}
\hspace{1em} \text{qed}

lemma mod-eqE:
\hspace{1em} \text{assumes } a \mod c = b \mod c
\hspace{1em} \text{obtains } d \text{ where } b = a + c \ast d
\hspace{1em} \text{proof}
\hspace{2em} \text{from } \text{assms } \text{have } c \dvd a - b
\hspace{3em} \text{by } (\text{simp add: mod-eq-dvd-iff})
\hspace{2em} \text{then obtain } d \text{ where } a - b = c \ast d
\hspace{3em} \text{then have } b = a + c \ast d
\hspace{4em} \text{by } (\text{simp add: algebra-simps})
\hspace{2em} \text{with that } \text{show } \text{thesis .}
\hspace{1em} \text{qed}

lemma invertible-coprime:
\hspace{1em} \text{coprime } a \text{ c if } a \ast b \mod c = 1
\hspace{1em} \text{by } (\text{rule coprimeI}) \text{ (use that } \text{dvd-mod-iff } [\text{of } -c a \text{ b}] \text{ in auto})

end
56.3 Uniquely determined division

class unique-euclidean-semiring = euclidean-semiring +
  assumes euclidean-size-mult: ‹euclidean-size (a * b) = euclidean-size a *
  euclidean-size b›
  fixes division-segment :: ‹a ⇒ 'a›
  assumes is-unit-division-segment [simp]: ‹is-unit (division-segment a)›
  and division-segment-mult:
    ‹a ≠ 0 ⇒ b ≠ 0 ⇒ division-segment (a * b) = division-segment a * divi-
    sion-segment b›
  and division-segment-mod:
    ‹b ≠ 0 ⇒ ¬ b dvd a ⇒ division-segment (a mod b) = division-segment b›
  assumes div-bounded:
    ‹b ≠ 0 ⇒ division-segment r = division-segment b
    ⇒ euclidean-size r < euclidean-size b
    ⇒ (q * b + r) div b = q›

begin

lemma division-segment-not-0 [simp]:
  ‹division-segment a ≠ 0›
using is-unit-division-segment [of a] is-unitE [of ‹division-segment a›] by blast

lemma euclidean-relationI [case-names by0 divides euclidean-relation]:
  ‹a div b, a mod b = (q, r)›
if by0: ‹b = 0 ⇒ q = 0 ∧ r = a›
  and divides: ‹b ≠ 0 ⇒ b dvd a ⇒ r = 0 ∧ a = q * b›
  and euclidean-relation: ‹b ≠ 0 ⇒ ¬ b dvd a ⇒ division-segment r = divi-
    sion-segment b
    ∧ euclidean-size r < euclidean-size b ∧ a = q * b + r›
proof (cases ‹b = 0›)
  case True
  with by0 show ‹thesis›
  by simp
next
  case False
  show ‹thesis›
proof (cases ‹b dvd a›)
  case True
  with ‹b ≠ 0› divides
  show ‹thesis›
  by simp
next
  case False
  with ‹b ≠ 0› euclidean-relation
  have ‹division-segment r = division-segment b›
    ‹euclidean-size r < euclidean-size b› ‹a = q * b + r›
    by simp-all
  from ‹b ≠ 0› ‹division-segment r = division-segment b›
    ‹euclidean-size r < euclidean-size b›
  have ‹(q * b + r) div b = q›
by (rule div-bounded)
with \((a = q * b + r)\)
have \((q = a \div b)\)
  by simp
from \((a = q * b + r)\)
have \((a \div b * b + a \mod b = q * b + r)\)
  by (simp add: div-mult-mod-eq)
with \((q = a \div b)\)
have \((q * b + a \mod b = q * b + r)\)
  by simp
then have \((r = a \mod b)\)
  by simp
with \((q = a \div b)\)
show \(?thesis\)
  by simp
qed

qed

subclass euclidean-semiring-cancel
proof
  fix \(a\) \(b\) \(c\)
  assume \(\langle b \neq 0 \rangle\)
  have \(\langle((a + c * b) \div b, (a + c * b) \mod b) = (c + a \div b, a \mod b)\rangle\)
  proof (induction rule: euclidean-relationI)
  case by0
  with \(\langle b \neq 0 \rangle\)
  show \(?case\)
    by simp
  next
  case divides
  then show \(?case\)
    by (simp add: algebra-simps dvd-add-left-iff)
  next
  case euclidean-relation
  then have \(\langle\neg b \ dvd a\rangle\)
    by (simp add: dvd-add-left-iff)
  have \(\langle a \mod b + (b * c + b * (a \div b)) = b * c + ((a \div b) * b + a \mod b)\rangle\)
    by (simp add: ac-simps)
  with \(\langle b \neq 0 \rangle\) have \*: \(\langle a \mod b + (b * c + b * (a \div b)) = b * c + a\rangle\)
    by (simp add: div-mult-mod-eq)
  from \(\langle\neg b \ dvd a\rangle\) euclidean-relation show \(?case\)
    by (simp-all add: algebra-simps division-segment-mod mod-size-less \*)
  qed
  then show \(\langle(a + c * b) \div b = c + a \div b\rangle\)
    by simp
next
  fix \(a\) \(b\) \(c\)
  assume \(\langle c \neq 0 \rangle\)
  have \(\langle((c + a) \div (c * b), (c + a) \mod (c * b)) = (a \div b, c + (a \mod b))\rangle\)
proof (induction rule: euclidean-relationI)

  case by0
  with \( c \neq 0 \) show ?case
    by simp

next
  case divides
  then show ?case
    by (auto simp add: algebra-simps)

next
  case euclidean-relation
  then have \( b \neq 0 \)
    using mod-size-less [of b a] by simp
  with \( b \neq 0 \) \( c \neq 0 \) show ?case
    by (simp add: algebra-simps division-segment-mult division-segment-mod euclidean-size-mult *)

qed

then show \( (c * a) \ div (c * b) = a \ div b \)
  by simp

qed

lemma div-eq-0-iff:
  \( \langle a \ div b = 0 \longleftrightarrow \text{euclidean-size } a < \text{euclidean-size } b \vee b = 0 \rangle \) (is - \( \longleftrightarrow \) ?P)

if \( \langle \text{division-segment } a = \text{division-segment } b \rangle \)

proof (cases \( a = 0 \vee b = 0 \))
  case True
  then show ?thesis by auto

next
  case False
  then have \( \langle a \neq 0 \rangle \langle b \neq 0 \rangle \)
    by simp-all
  have \( \langle a \ div b = 0 \longleftrightarrow \text{euclidean-size } a < \text{euclidean-size } b \rangle \)
  proof
    assume \( \langle a \ div b = 0 \rangle \)
    then have \( \langle a \ mod b = a \rangle \)
      using div-mult-mod-eq [of a b] by simp
    with \( b \neq 0 \) mod-size-less [of b a]
    show \( \langle \text{euclidean-size } a < \text{euclidean-size } b \rangle \)
      by simp
  next
    assume \( \langle \text{euclidean-size } a < \text{euclidean-size } b \rangle \)
    have \( \langle a \ div b, a \ mod b \rangle = (0, a) \)
    proof (induction rule: euclidean-relationI)
case by0
show ?case
by simp
next
case divides
with ∨ euclidean-size a < euclidean-size b show ?case
using dvd-imp-size-le [of b a] ∨ a ≠ 0 by simp
next
case euclidean-relation
with ∨ euclidean-size a < euclidean-size b that
show ?case
by simp
qed
then show ∨ a div b = 0
by simp
qed
with ∨ b ≠ 0, show ?thesis
by simp
qed

lemma div-mult1-eq:
(a * b) div c = a * (b div c) + a * (b mod c) div c

proof –
have *: ∨ (a * b) mod c + (a * (c * (b div c)) + c * (a * (b mod c) div c)) = a
* b: (is ∨ A + (∨ B + ∨ C) = -1)
proof –
have ∨ A = a * (b mod c) mod c
by (simp add: mod-mult-right-eq)
then have ∨ C + ∨ A = a * (b mod c)
by (simp add: multi-div-mod-eq)
then have ∨ B + (∨ C + ∨ A) = a * (c * (b div c) + (b mod c))
by (simp add: algebra-simps)
also have ∨ . . . = a * b
by (simp add: multi-div-mod-eq)
finally show ?thesis
by (simp add: algebra-simps)
qed

have ∨ ((a * b) div c, (a * b) mod c) = (a * (b div c) + a * (b mod c) div c, (a
* b) mod c)
proof (induction rule: euclidean-relationI)

case by0
then show ?case by simp
next
case divides
with * show ?case
by (simp add: algebra-simps)
next
case euclidean-relation
with * show ?case
by (simp add: division-segment-mod mod-size-less algebra-simps)
qed
then show ?thesis
  by simp
qed

lemma div-add1-eq:
  \langle (a + b) \mod c = a \div c + b \div c + (a \mod c + b \mod c) \div c \rangle

proof
have *: \langle (a + b) \mod c + (c * (a \div c) + (c * (b \div c) + c * ((a \mod c + b 
mod c) \div c))) = a + b \rangle
(is \langle ?A + (?B + (?C + ?D)) = \cdot \rangle)
proof
  by (simp add: ac-simps)
also have \langle ?A + ?D = (a \mod c + b \mod c) \mod c + ?D \rangle
  by (simp add: mod-add-eq)
also have \ldots = a \mod c + b \mod c
  by (simp add: mod-mult-div-eq)
finally have \langle ?A + (?B + (?C + ?D)) = (a \mod c + ?B) + (b \mod c + ?C) \rangle.
  by (simp add: ac-simps)
then show ?thesis
  by (simp add: mod-mult-div-eq)
qed

have \langle ((a + b) \div c, (a + b) \mod c) = (a \div c + b \div c + (a \mod c + b \mod 
c) \div c, (a + b) \mod c) \rangle
proof (induction rule: euclidean-relationI)
  case by0
  then show ?case
  by simp
next
  case divides
  with * show ?case
  by (simp add: algebra-simps)
next
  case euclidean-relation
  with * show ?case
  by (simp add: division-segment-mod mod-size-less algebra-simps)
qed
then show ?thesis
  by simp
qed

end

class unique-euclidean-ring = euclidean-ring + unique-euclidean-semiring
begin
subclass euclidean-ring-cancel ..
end

56.4 Division on nat

instantiation nat :: normalization-semidom
begin

definition normalize-nat :: (nat ⇒ nat)
where [simp]: normalize = (id :: nat ⇒ nat)

definition unit-factor-nat :: (nat ⇒ nat)
where unit-factor n = of-bool (n > 0)
for n :: nat

lemma unit-factor-simps [simp]:
〈unit-factor 0 = (0::nat)〉
〈unit-factor (Suc n) = 1〉
by (simp-all add: unit-factor-nat-def)

definition divide-nat :: (nat ⇒ nat ⇒ nat)
where m div n = (if n = 0 then 0 else Max {k. k * n ≤ m})
for m n :: nat

instance
by standard (auto simp add: divide-nat-def ac-simps unit-factor-nat-def intro: Max-eqI)

end

lemma coprime-Suc-0-left [simp]:
coprime (Suc 0) n
using coprime-1-left [of n] by simp

lemma coprime-Suc-0-right [simp]:
coprime n (Suc 0)
using coprime-1-right [of n] by simp

lemma coprime-common-divisor-nat: coprime a b ⇒ x dvd a ⇒ x dvd b ⇒ x = 1
for a b :: nat
by (drule coprime-common-divisor [of - - x]) simp-all

instantiation nat :: unique-euclidean-semiring
begin

definition euclidean-size-nat :: (nat ⇒ nat)
where [simp]: euclidean-size-nat = id

definition division-segment-nat :: (nat ⇒ nat)
where [simp]: division-segment n = 1
for n :: nat
definition modulo-nat :: (nat ⇒ nat ⇒ nat)
where ⟨m mod n = m − (m div n * n)⟩ for m n :: nat

instance proof
  fix m n :: nat
  have ex: ∃k. k * n ≤ l for l :: nat
    by (rule exI [of - 0]) simp
  have fin: finite {k. k * n ≤ l} if n > 0 for l
    proof (cases n)
      case True
      moreover have {l. l = 0 ∧ l ≤ m} = {0::nat}
      by auto
      then show ?thesis
      by simp
    next
      case False
      with ex [of m] fin have n * (m div n) = Max {l. l ≤ m ∧ n dvd l}
      proof (cases n = 0)
        case True
        moreover have {l. l = 0 ∧ l ≤ m} = {0::nat}
        by auto
        ultimately show ?thesis
        by simp
      next
        case False
        with ex have n * Max {k. k * n ≤ m} = Max (times n ' {k. k * n ≤ m})
        by (auto simp add: nat-mult-max-right intro: hom-Max-commute)
        also have times n ' {k. k * n ≤ m} = {l. l ≤ m ∧ n dvd l}
        by (auto simp add: ac-simps elim: dvdE)
        finally show ?thesis
        using False by (simp add: divide-nat-def ac-simps)
      qed
    qed
  proof
    have less-eq: m div n * n ≤ m
    by (auto simp add: mult-div-unfold ac-simps intro: Max.boundedI)
    then show m div n * n + m mod n = m
    by (simp add: modulo-nat-def)
    assume n ≠ 0
    show euclidean-size (m mod n) < euclidean-size n
    proof
      have m < Suc (m div n) * n
      proof (rule ccontr)
        assume ¬ m < Suc (m div n) * n
        then have Suc (m div n) * n ≤ m
        by (simp add: not-less)
        moreover from n ≠ 0 have Max {k. k * n ≤ m} < Suc (m div n)
        by (simp add: divide-nat-def)
        with n ≠ 0 ex fin have ∃k. k * n ≤ m ⇒ k < Suc (m div n)
        by auto
        ultimately have Suc (m div n) < Suc (m div n)
by blast
then show False
by simp
qed
with \( n \neq 0 \) show ?thesis
by (simp add: modulo-nat-def)
qed
show euclidean-size \( m \) \leq euclidean-size \( m \ast n \)
using \( n \neq 0 \) by (cases n) simp-all
fix \( q \) \( r \) :: nat
show \((q \ast n + r) \div n = q\) if euclidean-size \( r \) < euclidean-size \( n \)
proof 
  from that have \( r < n \)
  by simp
  have \( k \leq q \) if \( k \ast n \leq q \ast n + r \) for \( k \)
  proof (rule ccontr)
  assume \( \neg k \leq q \)
  then have \( q < k \)
  by simp
  then obtain \( l \) where \( k = \text{Suc} (q + l) \)
  by (auto simp add: less-iff-Suc-add)
  with \( r < n \) that show False
  by (simp add: algebra-simps)
qed
with \( n \neq 0 \) ex fin show ?thesis
  by (auto simp add: divide-nat-def Max-eq-iff)
qed
qed simp-all
end

lemma euclidean-relation-natI [case-names by0 divides euclidean-relation]:
\((m \div n, m \mod n) = (q, r)\)
if by0: \( n = 0 \implies q = 0 \land r = m \)
and divides: \( n > 0 \implies n \vdots m \implies r = 0 \land m = q \ast n \)
and euclidean-relation: \( n > 0 \implies \neg n \vdots m \implies r < n \land m = q \ast n + r \)
for \( m \) \( n \) \( q \) \( r \) :: nat
by (rule euclidean-relationI) (use that in simp-all)

lemma div-nat-eqI:
\( m \div n = q \) if \( \langle n \ast q \leq m \rangle \) and \( \langle m < n \ast \text{Suc} q \rangle \) for \( m \) \( n \) \( q \) :: nat
proof 
  have \( \langle m \div n, m \mod n \rangle = (q, m - n \ast q) \)
  proof (induction rule: euclidean-relation-natI)
  case by0
  with that show ?case
  by simp
next 
case divides
from \( n \, \text{dvd} \, m \) obtain \( s \) where \( m = n \ast s \) ..
with \( n > 0 \) that have \( s < \text{Suc} \, q \)
with \( m = n \ast s \) \( n > 0 \) that have \( q = s \)
by \( \text{simp only} : \text{mult-less-cancel1} \)
with \( m = n \ast s \) show \(?\text{case}\)
by \( \text{simp add} : \text{ac-simps} \)
next
\( \text{case} \, \text{euclidean-relation} \)
with that show \(?\text{case}\)
by \( \text{simp add} : \text{ac-simps} \)
qed
then show \(?\text{thesis}\)
by \( \text{simp} \)
qed

\textbf{lemma} \, \	extit{mod-nat-eqI}:
\( m \, \text{mod} \, n = r \) if \( r < n \) and \( r \leq m \) and \( n \, \text{dvd} \, m - r \) for \( m \, n \, r :: \text{nat} \)
\textbf{proof} –
have \( (m \, \text{div} \, n, m \, \text{mod} \, n) = ((m - r) \, \text{div} \, n, r) \)
\textbf{proof} (\text{induction \, rule: \, euclidean-relation-natI})
\begin{itemize}
  \item \( \text{case} \, \text{by0} \)
  \item with that show \(?\text{case}\)
  by \( \text{simp} \)
\end{itemize}
next
\begin{itemize}
  \item \( \text{case} \, \text{divides} \)
  \item from that dvd-minus-add [of \( r \) \( m \) \( I \, n \)]
  have \( n \, \text{dvd} \, m + (n - r) \)
  by \( \text{simp} \)
  \item with divides have \( n \, \text{dvd} \, n - r \)
  by \( \text{simp add} : \text{dvd-add-right-iff} \)
  \item then have \( n \leq n - r \)
  by \( \text{rule dvd-imp-le} \) (use \( r < n \) in \( \text{simp} \))
  \item with \( n > 0 \) have \( r = 0 \)
  by \( \text{simp} \)
  \item with \( n > 0 \) that show \(?\text{case}\)
  by \( \text{simp} \)
\end{itemize}
next
\begin{itemize}
  \item \( \text{case} \, \text{euclidean-relation} \)
  \item with that show \(?\text{case}\)
  by \( \text{simp add} : \text{ac-simps} \)
\end{itemize}
qed
then show \(?\text{thesis}\)
by \( \text{simp} \)
qed

Tool support

\textbf{ML}:
\textbf{structure} \, \textit{Cancel-Div-Mod-Nat} = \textit{Cancel-Div-Mod}
THEORY "Euclidean-Rings"

{ val div-name = const-name (divide); 
val mod-name = const-name (modulo); 
val mk-binop = HOLogic.mk-binop; 
val dest-plus = HOLogic.dest-bin const-name (Groups.plus) HOLogic.natT; 
val mk-sum = Arith-Data.mk-sum; 
fun dest-sum tm = 
  if HOLogic.is-zero tm then [] 
  else 
    (case try HOLogic.dest-Suc tm of 
      SOME t => HOLogic.Suc-zero :: dest-sum t 
    | NONE => (case try dest-plus tm of 
      SOME (t, u) => dest-sum t @ dest-sum u 
    | NONE => [tm]));
val div-mod-eqs = map mk-meta-eq @{thms cancel-div-mod-rules};
val prove-eq-sums = Arith-Data.prove-conv2 all-tac 
  (Arith-Data.simp-all-tac @{thms add-0-left add-0-right ac-simps})
}

simpproc-setup cancel-div-mod-nat ((m::nat) + n) = 
  (K Cancel-Div-Mod-Nat.proc)

lemma div-mult-self-is-m [simp]: 
m * n div n = m if n > 0 for m n :: nat 
using that by simp

lemma div-mult-self1-is-m [simp]: 
n * m div n = m if n > 0 for m n :: nat 
using that by simp

lemma mod-less-divisor [simp]: 
m mod n < n if n > 0 for m n :: nat 
using mod-size-less [of n m] that by simp

lemma mod-le-divisor [simp]: 
m mod n ≤ n if n > 0 for m n :: nat 
using that by (auto simp add: le-less)

lemma div-times-less-eq-dividend [simp]: 
m div n * n ≤ m for m n :: nat 
by (simp add: minus-mod-eq-div-mult [symmetric])

lemma times-div-less-eq-dividend [simp]: 
n * (m div n) ≤ m for m n :: nat 
using div-times-less-eq-dividend [of m n]
by (simp add: ac-simps)

lemma dividend-less-div-times:
  \( m < n + (m \div n) \times n \) if \( 0 < n \) for \( m, n :: \text{nat} \)
proof
  from that have \( m \mod n < n \)
  by simp
  then show \( \text{thesis} \)
  by (simp add: minus-mod-eq-div-mult [symmetric])
qed

lemma dividend-less-times-div:
  \( m < n + n \times (m \div n) \) if \( 0 < n \) for \( m, n :: \text{nat} \)
using dividend-less-div-times [of \( n, m \)] that
by (simp add: ac-simps)

lemma mod-Suc-le-divisor [simp]:
  \( m \mod \text{Suc} n \leq n \)
using mod-less-divisor [of Suc \( n, m \)] by arith

lemma mod-less-eq-dividend [simp]:
  \( m \mod n \leq m \) for \( m, n :: \text{nat} \)
proof (rule add-leD2)
  from \( \text{div-mult-mod-eq} \) have \( m \div n \times n + m \mod n = m \).
  then show \( m \div n \times n + m \mod n \leq m \) by auto
qed

lemma div-less [simp]: \( m \div n = 0 \)
and mod-less [simp]: \( m \mod n = m \)
if \( m < n \) for \( m, n :: \text{nat} \)
using that by (auto intro: div-nat-eqI mod-nat-eqI)

lemma split-div:
\( \langle P (m \div n) \rangle \longleftrightarrow \langle n = 0 \rightarrow P 0 \rangle \land \langle n \neq 0 \rightarrow (\forall i. j < n \land m = n \times i + j \rightarrow P i) \rangle \) (\( \text{is div} \))
and split-mod:
\( \langle Q (m \mod n) \rangle \longleftrightarrow \langle n = 0 \rightarrow Q m \rangle \land \langle n \neq 0 \rightarrow (\forall i. j < n \land m = n \times i + j \rightarrow Q j) \rangle \) (\( \text{is mod} \))
for \( m, n :: \text{nat} \)
proof
  have \( *: \langle R (m \div n) (m \mod n) \rangle \longleftrightarrow \langle n = 0 \rightarrow R 0 m \rangle \land \langle n \neq 0 \rightarrow (\forall i. j < n \land m = n \times i + j \rightarrow R i j) \rangle \) for \( R \)
  by (cases \( \langle n = 0 \rangle \)) auto
  from \( * \) [of \( \lambda q. P q \)] show \( \text{div} \).
  from \( * \) [of \( \lambda r. Q r \)] show \( \text{mod} \).
declare split-div [of - - ‹numeral n›, linarith-split] for n
declare split-mod [of - - ‹numeral n›, linarith-split] for n

lemma split-div':
P (m div n) ←→ n = 0 ∧ P 0 ∨ (∃q. (n * q ≤ m ∧ m < n * Suc q) ∧ P q)
proof (cases n = 0)
  case True
  then show ?thesis
  by simp
next
case False
  then have n * q ≤ m ∧ m < n * Suc q ←→ m div n = q for q
  by (auto intro: div-nat-eqI dividend-less-times-div)
  then show ?thesis
  by auto
qed

lemma le-div-geq:
m div n = Suc ((m - n) div n) if 0 < n and n ≤ m for m n :: nat
proof –
from ‹n ≤ m› obtain q where m = n + q
  by (auto simp add: le_iff_add)
with ‹0 < n› show ?thesis
  by (simp add: div_add_self1)
qed

lemma le-mod-geq:
m mod n = (m - n) mod n if n ≤ m for m n :: nat
proof –
from ‹n ≤ m› obtain q where m = n + q
  by (auto simp add: le_iff_add)
then show ?thesis
  by simp
qed

lemma div-if:
m div n = (if m < n ∨ n = 0 then 0 else Suc ((m - n) div n))
by (simp add: le-div-geq)

lemma mod-if:
m mod n = (if m < n then m else (m - n) mod n) for m n :: nat
by (simp add: le-mod-geq)

lemma div-eq-0-iff:
m div n = 0 ←→ m < n ∨ n = 0 for m n :: nat
by (simp add: div-eq-0-iff)
lemma div-greater-zero-iff:
  \( m \div n > 0 \iff n \leq m \land n > 0 \) for \( m, n : \text{nat} \)
using div-eq-0-iff [of \( m, n \)] by auto

lemma mod-greater-zero-iff-not-dvd:
  \( m \mod n > 0 \iff \neg n \dvd m \) for \( m, n : \text{nat} \)
by (simp add: dvd-eq-mod-eq-0)

lemma div-by-Suc-0 [simp]:
  \( m \div \text{Suc} 0 = m \)
using div-by-1 [of \( m \)] by simp

lemma mod-by-Suc-0 [simp]:
  \( m \mod \text{Suc} 0 = 0 \)
using mod-by-1 [of \( m \)] by simp

lemma div2-Suc-Suc [simp]:
  \( \text{Suc} (\text{Suc} m) \div 2 = \text{Suc} (m \div 2) \)
by (simp add: numeral-2-eq-2 le-div-geq)

lemma Suc-n-div-2-gt-zero [simp]:
  \( 0 < \text{Suc} n \div 2 \) if \( n > 0 \) for \( n : \text{nat} \)
using that by (cases \( n \)) simp-all

lemma div-2-gt-zero [simp]:
  \( 0 < n \div 2 \) if \( \text{Suc} 0 < n \) for \( n : \text{nat} \)
using that Suc-n-div-2-gt-zero [of \( n - 1 \)] by simp

lemma mod2-Suc-Suc [simp]:
  \( \text{Suc} (\text{Suc} m) \mod 2 = m \mod 2 \)
by (simp add: numeral-2-eq-2 le-mod-geq)

lemma add-self-div-2 [simp]:
  \( (m + m) \div 2 = m \) for \( m : \text{nat} \)
by (simp add: mult-2 [symmetric])

lemma add-self-mod-2 [simp]:
  \( (m + m) \mod 2 = 0 \) for \( m : \text{nat} \)
by (simp add: mult-2 [symmetric])

lemma mod2-gr-0 [simp]:
  \( 0 < m \mod 2 \iff m \mod 2 = 1 \) for \( m : \text{nat} \)
proof -
  have \( m \mod 2 < 2 \)
    by (rule mod-less-divisor) simp
  then have \( m \mod 2 = 0 \lor m \mod 2 = 1 \)
    by arith
  then show \( \text{thesis} \)
    by auto
lemma mod-Suc-eq [mod-simps]:
\[ \text{Suc} (m \mod n) \mod n = \text{Suc} m \mod n \]
\begin{proof}
\begin{itemize}
\item have \((m \mod n + 1) \mod n = (m + 1) \mod n\)
\item then show \(?\text{thesis}\)
\end{itemize}
\end{proof}
\end{lemma}

lemma mod-Suc-Suc-eq [mod-simps]:
\[ \text{Suc} (\text{Suc} (m \mod n)) \mod n = \text{Suc} (\text{Suc} m) \mod n \]
\begin{proof}
\begin{itemize}
\item have \((m \mod n + 2) \mod n = (m + 2) \mod n\)
\item then show \(?\text{thesis}\)
\end{itemize}
\end{proof}
\end{lemma}

lemma Suc-mod-mult-self1 [simp]:
\[ \text{Suc} (m + k \cdot n) \mod n = \text{Suc} m \mod n \]
and Suc-mod-mult-self2 [simp]:
\[ \text{Suc} (m + n \cdot k) \mod n = \text{Suc} m \mod n \]
and Suc-mod-mult-self3 [simp]:
\[ \text{Suc} (k \cdot n + m) \mod n = \text{Suc} m \mod n \]
and Suc-mod-mult-self4 [simp]:
\[ \text{Suc} (k \cdot m + n) \mod n = \text{Suc} m \mod n \]
\begin{proof}
\begin{itemize}
\item subst mod-Suc-eq [symmetric], simp add: mod-simps
\end{itemize}
\end{proof}
\end{lemma}

lemma Suc-0-mod-eq [simp]:
\[ \text{Suc} 0 \mod n = \text{of-bool} (n \neq \text{Suc} 0) \]
\begin{proof}
\begin{itemize}
\item cases \(n\) simp-all
\end{itemize}
\end{proof}
\end{lemma}

lemma div-mult2-eq:
\[ m \div (n \cdot q) = (m \div n) \div q \qquad (\text{is } ?Q) \]
and mod-mult2-eq:
\[ m \mod (n \cdot q) = n \cdot (m \mod n \mod q) + m \mod n \qquad (\text{is } ?R) \]
\begin{proof}
\begin{itemize}
\item for \(m n q :: \text{nat}\)
\item case by0
\item then show \(?\text{case}\)
\item by auto
\end{itemize}
\end{proof}
\end{lemma
case euclidean-relation
then have \( \langle n > 0 \rangle \langle q > 0 \rangle 
by simp-all
from \( \langle n > 0 \rangle \) have \( \langle m \mod n < n \rangle 
by (rule mod-less-divisor)
from \( \langle q > 0 \rangle \) have \( \langle m \div n \mod q < q \rangle 
by (rule mod-less-divisor)
then obtain \( s \) where \( \langle q = Suc \ (m \div n \mod q + s) \rangle 
by (blast dest: less-imp-Suc-add)
moreover have \( \langle m \mod n + n \ast (m \div n \mod q) < n \ast Suc \ (m \div n \mod q + s) \rangle 
using \( \langle m \mod n < n \rangle \) by (simp add: add-mult-distrib2)
ultimately have \( \langle m \mod n + n \ast (m \div n \mod q) < n \ast q \rangle 
by simp
then show ?case
by (simp add: algebra-simps flip: add-mult-distrib2)
qed
then show ?Q and ?R
by simp-all
qed

lemma div-le-mono:
\( m \div k \leq n \div k \) if \( m \leq n \) for \( m \ n \ k :: \text{nat} \)
proof –
from that obtain \( q \) where \( n = m + q \)
by (auto simp add: le-iff-add)
then show ?thesis
by (simp add: div-add1-eq [of \( m \ q \ k \)])
qed

Antimonotonicity of \( \text{(div)} \) in second argument

lemma div-le-mono2:
\( k \div n \leq k \div m \) if \( 0 < m \) and \( m \leq n \) for \( m \ n \ k :: \text{nat} \)
using that proof (induct \( k \) arbitrary: \( m \) rule: less-induct)
case \( \langle \text{less} \ k \rangle \)
show ?case
proof (cases \( n \leq k \))
case False
then show ?thesis
by simp
next
case True
have \( \langle k - n \rangle \div n \leq (k - m) \div n \)
using less-prems
by (blast intro: div-le-mono diff-le-mono2)
also have \( \ldots \leq (k - m) \div m \)
using \( \langle n \leq k \rangle \) less-prems less.hyps [of \( k - m \ m \)]
by simp
finally show ?thesis
using \( n \leq k \) less.prems
by (simp add: le-div-geq)
qed

lemma div-le-dividend [simp]:
\[ m \div n \leq m \] for \( m n :: \text{nat} \)
using div-le-mono2 [of \( 1 n m \)] by (cases \( n = 0 \)) simp-all

lemma div-less-dividend [simp]:
\[ m \div n < m \] if \( 1 < n \) and \( \theta < m \) for \( m n :: \text{nat} \)
using that proof (induct \( m \) rule: less-induct)
case \( \text{less } m \)
show \( ?\text{case} \)
proof (cases \( n < m \))
  case False
  with \( \text{less} \) show \( ?\text{thesis} \)
  by (cases \( n = m \)) simp-all
next
  case True
  then show \( ?\text{thesis} \)
  using \( \text{less.hyps} \) [of \( m - n \)] less.prems
  by (simp add: le-div-geq)
qed

lemma div-eq-dividend-iff:
\[ m \div n = m \iff n = 1 \] if \( m > \theta \) for \( m n :: \text{nat} \)
proof
  assume \( n = 1 \)
  then show \( m \div n = m \)
  by simp
next
  assume \( P : m \div n = m \)
  show \( n = 1 \)
  proof (rule ccontr)
    have \( n \neq 0 \)
    by (rule ccontr) (use \( P \) in auto)
    moreover assume \( n \neq 1 \)
    ultimately have \( n > 1 \)
    by simp
    with \( \text{that} \) have \( m \div n < m \)
    by simp
    with \( P \) show \( \text{False} \)
    by simp
  qed
qed

lemma less-mult-imp-div-less:
$m \div n < i$ if $m < i \times n$ for $m, n, i :: \text{nat}$

proof -
from that have $i \times n > 0$
  by \texttt{(cases $i \times n = 0$) simp-all}
then have $i > 0$ and $n > 0$
  by simp-all
have $m \div n \times n \leq m$
  by simp
then have $m \div n \div n < i \times n$
  using that by \texttt{(rule le-less-trans)}
with $n > 0$ show ?thesis
  by simp
qed

lemma \texttt{div-less-iff-less-mult}:
  \texttt{\langle m \div q < n \iff m < n \times q \rangle \ (is \ \langle P \iff Q \rangle)}
if $q > 0$ for $m, n, q :: \text{nat}$
proof
assume ?Q then show ?P
  by \texttt{(rule less-mult-imp-div-less)}
next
assume ?P
then obtain $h$ where \texttt{\langle n = Suc (m \div q + h) \rangle}
  using \texttt{less-natE by blast}
moreover have \texttt{\langle m < m + (Suc h \times q - m \mod q) \rangle}
  using that by \texttt{(simp add: trans-less-add1)}
ultimately show ?Q
  by \texttt{(simp add: algebra-simps flip: minus-mod-eq-mult-div)}
qed

lemma \texttt{less-eq-div-iff-mult-less-eq}:
  \texttt{\langle m \leq n \div q \iff m \times q \leq n \rangle \ if \ q > 0 \ for \ m, n, q :: \text{nat}}
using \texttt{div-less-iff-mult [of q n m] that by auto}

lemma \texttt{div-Suc}:
  \texttt{\langle Suc m \div n = (if Suc m \mod n = 0 then Suc (m \div n) else m \div n) \rangle}
proof \texttt{(cases \langle n = 0 \lor n = 1 \rangle)}
  case True
  then show ?thesis by auto
next
  case False
  then have \texttt{\langle n > 1 \rangle}
    by simp
  then have \texttt{\langle Suc m \div n = m \div n + Suc (m \mod n) \div n \rangle}
    using \texttt{div-add1-eq [of m 1 n] by simp}
  also have \texttt{\langle Suc (m \mod n) \div n = of-bool (n dvd Suc m) \rangle}
    proof \texttt{(cases \langle n dvd Suc m \rangle)}
      case False
      moreover have \texttt{\langle Suc (m \mod n) \neq n \rangle}
    qed
proof (rule ccontr)
  assume \( \neg \text{Suc} (m \mod n) \neq n \)
  then have \( m \mod n = n - \text{Suc} 0 \)
    by simp
  with \( n > 1 \) have \( (m + 1) \mod n = 0 \)
    by (subst mod-add-left-eq [symmetric]) simp
  then have \( n \dv Suc m \)
    by auto
  with False show False ..
qed

moreover have \( \text{Suc} (m \mod n) \leq n \)
  using \( n > 1 \) by (simp add: Suc-le-eq)
ultimately show \( \text{thesis} \)
  by (simp add: div-nat-eqI)
next
  case True
  then obtain q where q: \( \text{Suc} m = n \times q \).
  moreover have \( q > 0 \)
    by (rule ccontr)
  ultimately have \( m \mod n = n - \text{Suc} 0 \)
    using \( n > 1 \) mult-le-cancel1 [of n \text{Suc} 0 q]
    by (auto intro: mod-nat-eqI intro: neq-le-trans simp add: Suc-le-eq)
qed

lemma mod-Suc:
  \( \text{Suc} m \mod n = (\text{if} \text{Suc} (m \mod n) = n \text{ then 0 else} \text{Suc} (m \mod n)) \)
proof (cases \( n = 0 \))
  case True
  then show \( \text{thesis} \)
    by simp
next
  case False
  moreover have \( \text{Suc} m \mod n = \text{Suc} (m \mod n) \mod n \)
    by (simp add: mod-simps)
ultimately show \( \text{thesis} \)
  by (auto intro!: mod-nat-eqI intro: neq-le-trans simp add: Suc-le-eq)
qed

lemma Suc-times-mod-eq:
  \( \text{Suc} (m * n) \mod m = 1 \) if \( \text{Suc} 0 < m \)
using that by (simp add: mod-Suc)

lemma Suc-times-numeral-mod-eq [simp]:
  \( \text{Suc} (\text{numeral} k * n) \mod \text{numeral} k = 1 \) if \( \text{numeral} k \neq (1::\text{nat}) \)
by (rule Suc-times-mod-eq) (use that in simp)

lemma Suc-div-le-mono [simp]:
\[
m \div n \leq Suc \ m \div n
\]
by (simp add: div-le-mono)

These lemmas collapse some needless occurrences of Suc: at least three Sucs, since two and fewer are rewritten back to Suc again! We already have some rules to simplify operands smaller than 3.

lemma div-Suc-eq-div-add3 [simp]:
\[
m \div Suc (Suc (Suc n)) = m \div (3 + n)
\]
by (simp add: div-le-mono)

lemma mod-Suc-eq-mod-add3 [simp]:
\[
m \mod Suc (Suc (Suc n)) = m \mod (3 + n)
\]
by (simp add: Suc3-eq-add-3)

lemma Suc-div-eq-add3-div:
\[
Suc (Suc (Suc m)) \div n = (3 + m) \div n
\]
by (simp add: Suc3-eq-add-3)

lemma Suc-mod-eq-add3-mod:
\[
Suc (Suc (Suc m)) \mod n = (3 + m) \mod n
\]
by (simp add: Suc3-eq-add-3)


lemma (in field-char-0) of-nat-div:
\[
of-nat (m \div n) = ((of-nat m - of-nat (m \mod n)) / of-nat n)
\]
proof -
  have of-nat (m \div n) = ((of-nat (m \div n \times n + m \mod n) - of-nat (m \mod n)) / of-nat n :: 'a)
    unfolding of-nat-add by (cases n = 0) simp-all
  then show ?thesis
  by simp
qed

An “induction” law for modulus arithmetic.

lemma mod-induct [consumes 3, case-names step]:
\[
P m \text{ if } P n \text{ and } n < p \text{ and } m < p
\]
and step: \(n, n < p \Rightarrow P n \Rightarrow P (Suc n \mod p)\)

using \(m < p\) proof (induct m)
  case 0
  show ?case
proof (rule ccontr)
assume $\neg P \ 0$

from $\langle n < p \rangle$ have $\theta < p$
by simp

from $\langle n < p \rangle$ obtain $m$ where $\theta < m$ and $p = n + m$
by (blast dest: less-imp-add-positive)

with $\langle P \ n \rangle$ have $P (p - m)$
by simp

moreover have $\neg P (p - m)$
using $\langle \theta < m \rangle$ proof (induct m)

case 0
then show $?case$
by simp

next


case $(\operatorname{Suc} \ m)$

show $?case$

proof

assume $P: P (p - \operatorname{Suc} \ m)$

with $\langle \neg P \ 0 \rangle$ have $\operatorname{Suc} \ m < p$
by (auto intro: ccontr)

then have $\operatorname{Suc} (p - \operatorname{Suc} \ m) = p - m$
by arith

moreover from $\langle \theta < p \rangle$ have $p - \operatorname{Suc} \ m < p$ by arith

with $P$ step have $P ((\operatorname{Suc} (p - \operatorname{Suc} \ m)) \mod p)$
by blast

ultimately show False
using $\langle \neg P \ 0 \rangle$, $\operatorname{Suc}$, hyps by (cases $m = 0$) simp-all

qed

next

case $(\operatorname{Suc} \ m)$
then have $m < p$ and mod: $\operatorname{Suc} \ m \mod p = \operatorname{Suc} \ m$
by simp-all

from $\langle m < p \rangle$ have $P m$
by (rule Suc.hyps)

with $\langle m < p \rangle$ have $P (\operatorname{Suc} \ m \mod p)$
by (rule step)

with mod show $?case$
by simp

qed

lemma funpow-mod-eq:
\langle(f \ ^\langle m \mod n \rangle) \ x = (f \ ^\langle m \rangle) \ x\rangle \ if \ \langle(f \ ^\langle n \rangle) \ x = x\rangle

proof

have $\langle(f \ ^\langle m \rangle) \ x = (f \ ^\langle m \mod n + m \div n \ast n \rangle) \ x\rangle$
by simp
also have \((\ldots = (f \mod (m \mod n)) (((f \mod n) \mod (m \div n)) x))\)
by \((\text{simp only: funpow-add funpow-mult ac-simps})\) \text{simp}
also have \(((f \mod n) \mod q) x = x\) \text{ for } q
by \((\text{induction } q) (\text{use } (f \mod n) x = x) \text{ in simp-all})
finally show \(\text{thesis}\)
by \text{simp}
\qed

\begin{lem}
\text{mod-eq-dvd-iff-nat}:
\langle m \mod q = n \mod q \longleftrightarrow q \text{ dvd } m - n \rangle
\langle \text{is } ?P \longleftrightarrow ?Q \rangle
if \langle m \geq n \rangle \text{ for } m, n, q :: \text{nat}
\end{lem}
\begin{proof}
assume \(\text {?Q}\)
then obtain \(s\) where \(\langle m - n = q \ast s \rangle\) ..
with that have \(\langle m = q \ast s + n \rangle\)
by \text{simp}
then show \(\text {?P}\)
by \text{simp}
next
assume \(\text {?P}\)
have \(\langle m - n = m \div q \ast q + m \mod q - (n \div q \ast q + n \mod q) \rangle\)
by \text{simp}
also have \(\langle \ldots = q \ast (m \div q - n \div q) \rangle\)
by \((\text{simp only: algebra-simps } ?P)\)
finally show \(\text {?Q} \) ..
\end{proof}
\end{lem}
\begin{lem}
\text{mod-eq-dvd-symdiff-nat}:
\langle m \mod q = n \mod q \longleftrightarrow q \text{ dvd } \text{nat} \mid \text{int } m - \text{int } n \rangle
by \((\text{auto simp add: abs-if mod-eq-dvd-iff-nat nat-diff-distrib dest: sym intro: sym})\)
\end{lem}
\begin{lem}
\text{mod-eq-nat1E}:
\begin{fixes}
m n q :: \text{nat}
\end{fixes}
\begin{assumes}
m \mod q = n \mod q \text{ and } m \geq n
\end{assumes}
\begin{obtains}
s \text{ where } m = n + q \ast s
\end{obtains}
\begin{proof}
from assms have \(q \text{ dvd } m - n\)
by \((\text{simp add: mod-eq-dvd-iff-nat})\)
then obtain \(s\) where \(m - n = q \ast s \) ..
with \(\langle m \geq n \rangle\) have \(m = n + q \ast s\)
by \text{simp}
with that show \text{thesis} .
\end{proof}
\end{lem}
\begin{lem}
\text{mod-eq-nat2E}:
\begin{fixes}
m n q :: \text{nat}
\end{fixes}
\begin{assumes}
m \mod q = n \mod q \text{ and } n \geq m
\end{assumes}
\begin{obtains}
s \text{ where } n = m + q \ast s
\end{obtains}
\begin{using}
assms \text{ mod-eq-nat1E } [\text{of } n \ q \ m] \text{ by } \text{(auto simp add: ac-simps)}
\end{using}
lemma nat-mod-eq-iff:
\[(x::nat) \mod n = y \mod n \iff (\exists q1 \ q2. x + n \ast q1 = y + n \ast q2)\] (is \?lhs = \?rhs)
proof
  assume H: x mod n = y mod n
  \{ assume xy: x \leq y \\
  from H have th: y mod n = x mod n by simp \\
  from mod-eq-nat1E [OF th xy] obtain q where y = x + n \ast q .
  then have x + n \ast q = y + n \ast 0
    by simp
  then have \exists q1 q2. x + n \ast q1 = y + n \ast q2
    by blast \}
moreover
  \{ assume xy: y \leq x \\
  from mod-eq-nat1E [OF H xy] obtain q where x = y + n \ast q .
  then have x + n \ast 0 = y + n \ast q
    by simp
  then have \exists q1 q2. x + n \ast q1 = y + n \ast q2
    by blast \}
ultimately show \?rhs using linear[of x y] by blast
next
  assume \?rhs then obtain q1 q2 where q12: x + n \ast q1 = y + n \ast q2 by blast
  hence (x + n \ast q1) mod n = (y + n \ast q2) mod n by simp
  thus \?lhs by simp
qed

56.5 Division on \textit{int}

The following specification of integer division rounds towards minus infinity
and is advocated by Donald Knuth. See [5] for an overview and terminology
of different possibilities to specify integer division; there division rounding
towards minus infinity is named “F-division”.

56.5.1 Basic instantiation

instantiation \textit{int} :: {normalization-semidom, idom-modulo}
begin

definition normalize-int :: \langle \textit{int} \Rightarrow \textit{int} \rangle
  where [simp]: \langle normalize = (abs :: \textit{int} \Rightarrow \textit{int}) \rangle

definition unit-factor-int :: \langle \textit{int} \Rightarrow \textit{int} \rangle
  where [simp]: \langle unit-factor = (sgn :: \textit{int} \Rightarrow \textit{int}) \rangle

definition divide-int :: \langle \textit{int} \Rightarrow \textit{int} \Rightarrow \textit{int} \rangle
  where \langle k \div l = (sgn k \ast sgn l \ast int (nat |k| \div nat |l|)) \rangle
THEORY “Euclidean-Rings”

lemma divide-int-unfold:
\((\text{sgn } k \ast \text{int } m) \div (\text{sgn } l \ast \text{int } n) = (\text{sgn } k \ast \text{sgn } l \ast \text{int } (m \div n))\)
\((\text{of-bool } ((k = 0 \longleftrightarrow m = 0) \land l \neq 0 \land n \neq 0 \land \text{sgn } k \neq \text{sgn } l \land \neg n \text{ dvd } m))\)
by (simp add: divide-int-def sgn-mult nat-mult-distrib abs-mult sgn-eq-0-iff ac-simps)

definition modulo-int :: \(\text{int} \Rightarrow \text{int} \Rightarrow \text{int}\)
where
\(k \mod l = \text{sgn } k \ast \text{int } (\text{nat } |k| \mod \text{nat } |l|) + l \ast \text{of-bool } (\text{sgn } k \neq \text{sgn } l \land \neg l \text{ dvd } k)\)

lemma modulo-int-unfold:
\((\text{sgn } k \ast \text{int } m) \mod (\text{sgn } l \ast \text{int } n) = (\text{sgn } k \ast \text{int } (\text{mod } (\text{of-bool } (l \neq 0) \ast n)) + (\text{sgn } l \ast \text{int } n) \ast \text{of-bool } ((k = 0 \longleftrightarrow m = 0) \land \text{sgn } k \neq \text{sgn } l \land \neg n \text{ dvd } m))\)
by (auto simp add: modulo-int-def sgn-mult abs-mult)

instance proof
fix \(k :: \text{int}\) show \(k \div 0 = 0\)
by (simp add: divide-int-def)

next
fix \(k l :: \text{int}\)
assume \(l \neq 0\)
obtain \(n m \text{ and } s t\) where \(k: k = \text{sgn } s \ast \text{int } n \text{ and } l: l = \text{sgn } t \ast \text{int } m\)
by (blast intro: int-sgnE elim: that)
then have \(k \ast l = \text{sgn } (s \ast t) \ast \text{int } (n \ast m)\)
by (simp add: ac-simps sgn-mult)
with \(k l \text{ \(l \neq 0\)}\) show \(k \ast l \div l = k\)
by (simp only: divide-int-unfold)
(auto simp add: algebra-simps sgn-mult sgn-1-pos sgn-0-0)

next
fix \(k l :: \text{int}\)
obtain \(n m \text{ and } s t\) where \(k: k = \text{sgn } s \ast \text{int } n \text{ and } l: l = \text{sgn } t \ast \text{int } m\)
by (blast intro: int-sgnE elim: that)
then show \(k \div l \ast l + k \mod l = k\)
by (simp add: divide-int-unfold modulo-int-unfold algebra-simps modulo-nat-def of-nat-diff)
qed (auto simp add: sgn-mult mult-sgn-abs abs-eq-iff)

end

56.5.2 Algebraic foundations

lemma coprime-int-iff [simp]:
coprime (\text{int } m) (\text{int } n) \longleftrightarrow coprime m n (is \ ?P \longleftrightarrow \ ?Q)

proof
assume \(?P\)
show \(?Q\)
proof (rule coprimeI)
  fix q
  assume q dvd m q dvd n
  then have int q dvd int m int q dvd int n
    by simp-all
  with (?P, have is-unit (int q))
    by (rule coprime-common-divisor)
  then show is-unit q
    by simp
qed

next
  assume ?Q
  show ?P
  proof (rule coprimeI)
    fix k
    assume k dvd int m k dvd int n
    then have nat |k| dvd m nat |k| dvd n
      by simp-all
    with (?Q, have is-unit (nat |k|))
      by (rule coprime-common-divisor)
    then show is-unit k
      by simp
  qed
qed

lemma coprime-abs-left-iff [simp]:
coprime |k| l ↔ coprime k l  for k l :: int
using coprime-normalize-left-iff [of k l] by simp

lemma coprime-abs-right-iff [simp]:
coprime k |l| ↔ coprime k l  for k l :: int
using coprime-abs-left-iff [of l k] by (simp add: ac-simps)

lemma coprime-nat-abs-left-iff [simp]:
coprime (nat |k|) n ↔ coprime k (int n)
proof –
  define m where m = nat |k|
  then have |k| = int m
    by simp
  moreover have coprime k (int n) ↔ coprime |k| (int n)
    by simp
  ultimately show ?thesis
    by simp
qed

lemma coprime-nat-abs-right-iff [simp]:
coprime n (nat |k|) ↔ coprime (int n) k
using coprime-nat-abs-left-iff [of k n] by (simp add: ac-simps)
lemma coprime-common-divisor-int: coprime a b ⟹ x dvd a ⟹ x dvd b ⟹ |x| = 1 
  for a b :: int
  by (drule coprime-common-divisor [of - - x]) simp-all

56.5.3 Basic conversions

lemma div-abs-eq-div-nat:
  |k| div |l| = int (nat |k| div nat |l|)
  by (auto simp add: divide-int-def)

lemma div-eq-div-abs:
  k div l = sgn k * sgn l * (|k| div |l|)
    - of-bool (l ≠ 0 ∧ sgn k ≠ sgn l ∧ ¬ l dvd k)
  for k l :: int
  by (simp add: divide-int-def [of k l] div-abs-eq-div-nat)

lemma mod-abs-eq-div-nat:
  |k| mod |l| = int (nat |k| mod nat |l|)
  by (simp add: modulo-int-def)

lemma mod-eq-mod-abs:
  k mod l = sgn k * (|k| mod |l|) + l * of-bool (sgn k ≠ sgn l ∧ ¬ l dvd k)
  for k l :: int
  by (auto simp add: modulo-int-def [of k l] mod-abs-eq-div-nat)

lemma div-sgn-abs-cancel:
  fixes k l v :: int
  assumes v ≠ 0
  shows (sgn v * |k|) div (sgn v * |l|) = |k| div |l|
  using assms by (simp add: sgn-mult abs-mult sgn-0-0 divide-int-def [of sgn v * |k| sgn v * |l|] flip: div- abs-eq-div-nat)

lemma div-eq-sgn- abs:
  fixes k l v :: int
  assumes sgn k = sgn l
  shows k div l = |k| div |l|
  using assms by (auto simp add: divide-int-def)
lemma div-dvd-sgn-abs:
fixes \( k, l :\) \( \mathbb{int} \)
assumes \( l \mid k \)
shows \( k \div l = (sgn k \times sgn l) \times (|k| \div |l|) \)
using assms by (auto simp add: div-abs-eq ac-simps)

lemma div-noneq-sgn-abs:
fixes \( k, l :\) \( \mathbb{int} \)
assumes \( l \neq 0 \)
assumes \( sgn k \neq sgn l \)
shows \( k \div l = -(|k| \div |l|) - of\text{-}bool (-\ l \mid k) \)
using assms by (auto simp add: div-abs-eq ac-simps sgn-0-0 dest: sgn-not-eq-imp)

56.5.4 Euclidean division

instantiation \( \mathbb{int} :\) \( \text{unique-euclidean-ring} \)
begin

definition euclidean-size-int :: \( \mathbb{int} \Rightarrow \mathbb{nat} \)
where [simp]: euclidean-size-int = (nat o abs :: \( \mathbb{int} \Rightarrow \mathbb{nat} \))

definition division-segment-int :: \( \mathbb{int} \Rightarrow \mathbb{int} \)
where division-segment-int \( k \) = (if \( k \geq 0 \) then \( 1 \) else \( -1 \))

lemma division-segment-eq-sgn:
Division-segment \( k \) = \( sgn k \) if \( k \neq 0 \)
for \( k :: \mathbb{int} \)
using that by (simp add: division-segment-int-def)

lemma abs-division-segment [simp]:
| division-segment \( k | \) = \( 1 \) for \( k :: \mathbb{int} \)
by (simp add: division-segment-int-def)

lemma abs-mod-less:
| \( k \mod l | < | l | \) if \( l \neq 0 \)
for \( k, l :: \mathbb{int} \)
proof -
obtain \( n, m \) and \( s, t \) where \( k = sgn s \times int n \) and \( l = sgn t \times int m \)
by (blast intro: int-sgnE elim: that)
with that show \( \text{thesis} \)
by (auto simp add: modulo-int-unfold abs-mult mod-greater-zero-iff-not-dvd
simp flip: right-diff-distrib dest!: sgn-not-eq-imp)
(simp add: sgn-0-0)

qed

lemma sgn-mod:
sgn \( (k \mod l) | = sgn l \) if \( l \neq 0 \) \( \neg l \mid k \)
for \( k, l :: \mathbb{int} \)
proof -
obtain \( n, m \) and \( s, t \) where \( k = sgn s \times int n \) and \( l = sgn t \times int m \)
by (blast intro: int-sgnE elim: that)
with that show \( \text{thesis} \)

by (auto simp add: modulo-int-unfold sgn-mult \mod greater-zero-iff-not-dvd
    simp flip: right-diff-distrib dest: sgn-not-eq-imp)

qed

instance proof

  fix \(k \cdot l :: \text{int}\)

  show \(\text{division-segment} \ (k \mod l) = \text{division-segment} \ l\ \text{if}\)
  \(l \neq 0\ \text{and} \ \neg l \ \text{dvd} \ k\)
  using that by (simp add: division-segment-eq-sgn dvd-eq-mod-eq-0 sgn-mod)

next

  fix \(l \cdot q \cdot r :: \text{int}\)

  obtain \(n \cdot m \cdot s \cdot t\)
    where \(l \cdot l = sgn \cdot s \cdot int \ n\ \text{and} \ q = sgn \cdot t \cdot int \ m\)
      by (blast intro: int-sgnE elim: that)

  assume \(\lnot l \neq 0\)

  with \(l \cdot l \neq 0\ \text{and} \ n > 0\)
    by (simp-all add: sgn-0-0)

  assume \(\text{division-segment} \ r = \text{division-segment} \ l\)

  moreover have \(r = sgn \cdot r \cdot |r|\)
    by (simp add: sgn-mult-abs)

  moreover define \(u\)
    where \(u = \text{nat} \cdot |r|\)

  ultimately have \(r = sgn \cdot l \cdot int \ u\)

    using \(\text{division-segment-eq-sgn} \ (l \neq 0)\ \text{by} \ (cases \ r = 0)\ \text{simp-all}\)

  with \(l \cdot n > 0\)
    have \(r = sgn \cdot s \cdot int \ u\)
      by (simp add: sgn-mul)

  assume \(\text{euclidean-size} \ r < \text{euclidean-size} \ l\)

  with \(l \cdot r \neq 0\)
    have \(u < n\)
      by (simp add: abs-mult)

  show \(q \cdot l + r \div l = q\)

  proof (cases \ q = 0 \lor r = 0\)

    case True

    then show \ ?thesis

    proof

      assume \(q = 0\)

      then show \ ?thesis

        using \(l \cdot r \cdot (u < n)\) by (simp add: divide-int-unfold)

    next

      assume \(r = 0\)

      from \(r = 0\) have *: \(q \cdot l + r = sgn \cdot (t \cdot s) \ast int \ (n \ast m)\)

      using \(q \cdot l\) by (simp add: ac-simps sgn-mult)

      from \(s \neq 0\) \(n > 0\) show \ ?thesis

        by (simp only: *, simp only: * q l divide-int-unfold)

    (auto simp add: sgn-mult ac-simps)

    qed

  next

    case False

    with \(q \cdot r\)
      have \(t \neq 0\ \text{and} \ m > 0\ \text{and} \ s \neq 0\ \text{and} \ u > 0\)

    by (simp-all add: sgn-0-0)

    moreover from \(0 < m\) \(0 < u\) have \(u \leq m \ast n\)
THEORY "Euclidean-Rings"

using mult-le-less-imp-less [of 1 m u n] by simp

ultimately have \( \ast \cdot q + l + r = \operatorname{sgn} (s \ast l) \)
* int (if \( t < 0 \) then \( m \ast n - u \) else \( m \ast n + u \))

using \( I q r \)

by (simp add: sgn-mult algebra-simps of-nat-diff)

have \( (m \ast n - u) \) div \( n = m - 1 \) if \( u > 0 \)

using \( \langle 0 < m \rangle \langle u < n \rangle \) that

by (auto intro: div-nat-eqI simp add: algebra-simps)

have \( (m \ast n - u) \) div \( n = m - 1 \) if \( u > 0 \)

using \( \langle u \leq m \ast n \rangle \) dvd-diffD1 [of \( n m \ast n u \)] by auto

ultimately show \( ?\text{thesis} \)

using \( \langle s \neq 0 \rangle \langle m > 0 \rangle \langle u > 0 \rangle \langle u < n \rangle \langle u \leq m \ast n \rangle \)

by simp: sgn-mult sgn-0-0 sgn-1-pos algebra-simps dest: dvd-imp-le

qed

qed (use mult-le-mono2 [of 1] in \( \langle u \leq m \ast n u \rangle \) by auto simp add: division-segment-int-def not-le

zero-less-mult-iff mult-less-0-iff abs-mult sgn-mult abs-mod-less sgn-mod nat-mult-distrib)

end

lemma euclidean-relation-intI [case-names by0 divides euclidean-relation]:
\( (k \div l, k \mod l) = (q, r) \)

if by0': \( l = 0 \Longrightarrow q = 0 \wedge r = k \)

and divides': \( l \neq 0 \Longrightarrow l \vdash d k \Longrightarrow r = 0 \wedge k = q \ast l \)

and euclidean-relation': \( l \neq 0 \Longrightarrow \neg l \vdash d k \Longrightarrow \operatorname{sgn} r = \operatorname{sgn} l \\
\wedge |r| < |l| \wedge k = q \ast l + r \) for \( k l :: \text{int} \)

proof (induction rule: euclidean-relationI)

case by0
then show \( ?\text{case} \)
by (rule by0')

next

case divides
then show \( ?\text{case} \)
by (rule divides')

next

case euclidean-relation

with euclidean-relation' have \( \langle \operatorname{sgn} r = \operatorname{sgn} l \rangle \langle |r| < |l| \rangle \langle k = q \ast l + r \rangle \)

by simp: all

from \( \langle \operatorname{sgn} r = \operatorname{sgn} l \rangle \langle l \neq 0 \rangle \) have \( \langle \text{division-segment} r = \text{division-segment} l \rangle \\
by (simp add: division-segment-int-def sgn-if split: if-splits)

with \( \langle |r| < |l| \rangle \langle k = q \ast l + r \rangle \)
show \( ?\text{case} \)
by simp

qed

56.5.5 Trivial reduction steps

lemma div-pos-pos-trivial [simp]:
THEORY "Euclidean-Rings"

\[
k \div l = 0 \text{ if } k \geq 0 \text{ and } k < l \text{ for } k, l :: \text{int}
\]
using that by \((\text{simp add: unique-euclidean-semiring-class.div-eq-0-iff division-segment-int-def})\)

**lemma** mod-pos-pos-trivial [simp]:
\[
k \mod l = k \text{ if } k \geq 0 \text{ and } k < l \text{ for } k, l :: \text{int}
\]
using that by \((\text{simp add: mod-eq-self-iff-div-eq-0})\)

**lemma** div-neg-neg-trivial [simp]:
\[
k \div l = 0 \text{ if } k \leq 0 \text{ and } l < k \text{ for } k, l :: \text{int}
\]
using that by \((\text{cases } k = 0)\) \((\text{simp, simp add: unique-euclidean-semiring-class.div-eq-0-iff division-segment-int-def})\)

**lemma** mod-neg-neg-trivial [simp]:
\[
k \mod l = k \text{ if } k \leq 0 \text{ and } l < k \text{ for } k, l :: \text{int}
\]
using that by \((\text{simp add: mod-eq-self-iff-div-eq-0})\)

**lemma** div-pos-neg-trivial:
\[
\langle k \div l = -1 \rangle \quad (\text{is } ?Q)
\]
and **mod-pos-neg-trivial**: \langle k \mod l = k + l \rangle \quad (\text{is } ?R)
if \langle 0 < k \rangle \text{ and } \langle k + l \leq 0 \rangle \text{ for } k, l :: \text{int}

**proof**
from that have \langle l < 0 \rangle
by simp
have \langle k \div l, k \mod l \rangle = (-1, k + l)
**proof** \((\text{induction rule: euclidean-relation-intI})\)
  case by0
    with \langle l < 0 \rangle show ?case
    by simp
next
case divides
  from \langle l \div k \rangle obtain j where \langle k = l \times j \rangle .
with \langle l < 0 \rangle \langle 0 < k \rangle have \langle j < 0 \rangle
by \((\text{simp add: zero-less-mult-iff})\)
  moreover from \langle k + l \leq 0 \rangle \langle k = l \times j \rangle have \langle l \times (j + 1) \leq 0 \rangle
by \((\text{simp add: algebra-simps})\)
  with \langle l < 0 \rangle have \langle j + 1 \geq 0 \rangle
by \((\text{simp add: mult-le-0-iff})\)
  with \langle j < 0 \rangle have \langle j = -1 \rangle
by simp
  with \langle k = l \times j \rangle show ?case
  by simp
next
case euclidean-relation
  with \langle k + l \leq 0 \rangle have \langle k + l < 0 \rangle
by \((\text{auto simp add: less-le-add-eq-0-iff})\)
  with \langle 0 < k \rangle show ?case
by simp
qed
then show ?Q and ?R
by simp-all
qed

There is neither \textit{div-neg-pos-trivial} nor \textit{mod-neg-pos-trivial} because $(0::'a) \div l = (0::'a)$ would supersede it.

### 56.5.6 More uniqueness rules

**Lemma**

\begin{align*}
\text{fixes } & a \ b \ q \ r :: \text{int} \\
\text{assumes } & \langle a = b \ast q + r \mid 0 \leq r \rangle \langle r < b \rangle \\
\text{shows } & \langle a \ div b = q \rangle \ (\text{is } ?Q) \\
\text{and } & \langle a \ mod b = r \rangle \ (\text{is } ?R) \\
\text{proof } & \text{ –}
\end{align*}

\text{have } \langle a \ div b, a \ mod b \rangle = (q, r)

\text{by (induction rule: euclidean-relation-intI)}

\text{(use assms in (auto simp add: ac-simps dvd-add-left-iff sgn-1-pos le-less dest: zdvd-imp-le))}

\text{then show } ?Q \text{ and } ?R

\text{by simp-all}

**Lemma** \textit{int-div-neg-eq}:

\begin{align*}
\langle a \ div b = q \rangle \text{ if } & \langle a = b \ast q + r \mid r \leq 0 \rangle \langle b < r \rangle \text{ for } a \ b \ q \ r :: \text{int} \\
\text{using that } & \text{int-div-pos-eq } [\text{of } a \langle \neg b \rangle \langle \neg q \rangle \langle \neg r \rangle] \text{ by simp-all}
\end{align*}

**Lemma** \textit{int-mod-neg-eq}:

\begin{align*}
\langle a \ mod b = r \rangle \text{ if } & \langle a = b \ast q + r \mid r \leq 0 \rangle \langle b < r \rangle \text{ for } a \ b \ q \ r :: \text{int} \\
\text{using that } & \text{int-div-neg-eq } [\text{of } a \ b \ q \ r] \text{ by simp}
\end{align*}

### 56.5.7 Laws for unary minus

**Lemma** \textit{zmod-zminus1-not-zero}:

\begin{align*}
\text{fixes } & k \ l :: \text{int} \\
\text{shows } & \neg k \ mod l \neq 0 \implies k \ mod l \neq 0 \\
\text{by (simp add: mod-eq-0-iff-dvd)}
\end{align*}

**Lemma** \textit{zmod-zminus2-not-zero}:

\begin{align*}
\text{fixes } & k \ l :: \text{int} \\
\text{shows } & k \ mod l \neq 0 \implies k \ mod l \neq 0 \\
\text{by (simp add: mod-eq-0-iff-dvd)}
\end{align*}

**Lemma** \textit{zdiv-zminus1-eq-if}:

\begin{align*}
\langle (\neg a) \ div b = (\text{if } a \ mod b = 0 \text{ then } (a \ div b) \text{ else } (a \ div b) - 1) \rangle \\
\text{if } & \langle b \neq 0 \rangle \text{ for } a \ b :: \text{int} \\
\text{using that } & \text{sgn-not-eq-imp } [\text{of } b \langle \neg a \rangle] \\
\text{by (cases } & a \langle = 0 \rangle) \text{ (auto simp add: div-eq-dvd-div-abs [of } \langle \neg a \rangle \b] \text{ div-eq-div-abs [of } a \ b] \text{ sgn-eq-0-iff})}
\end{align*}
lemma zdiv-zminus2-eq-if:
\[ a \div (-b) = (\text{if } a \mod b = 0 \text{ then } - (a \div b) \text{ else } -(a \div b) - 1) \]
if \( b \neq 0 \) for \( a, b :: \text{int} \)
using that by (auto simp add: zdiv-zminus1-eq-if div-minus-right)

lemma zmod-zminus1-eq-if:
\[ (-a) \mod b = (\text{if } a \mod b = 0 \text{ then } 0 \text{ else } b - (a \mod b)) \]
for \( a, b :: \text{int} \)
by (cases \( b = 0 \))
(auto simp flip: minus-div-mult-eq-mod simp add: zdiv-zminus1-eq-if algebra-simps)

lemma zmod-zminus2-eq-if:
\[ a \mod (-b) = (\text{if } a \mod b = 0 \text{ then } 0 \text{ else } (a \mod b) - b) \]
for \( a, b :: \text{int} \)
by (auto simp add: zmod-zminus1-eq-if mod-minus-right)

56.5.8 Borders

lemma pos-mod-bound [simp]:
\[ k \mod l < l \text{ if } l > 0 \text{ for } k, l :: \text{int} \]
proof –
obtain \( m \) and \( s \) where \( k = \text{sgn } s \ast \text{int } m \)
by (rule int-sgnE)
moreover from that obtain \( n \) where \( l = \text{sgn } 1 \ast \text{int } n \)
by (cases \( l \)) simp-all
moreover from this that have \( n > 0 \)
by simp
ultimately show \(?\thesis\)
by (simp only: modulo-int-unfold)
(auto simp add: mod-greater-zero-iff-not-dvd sgn-1-pos)
qed

lemma neg-mod-bound [simp]:
\[ l < k \mod l \text{ if } l < 0 \text{ for } k, l :: \text{int} \]
proof –
obtain \( m \) and \( s \) where \( k = \text{sgn } s \ast \text{int } m \)
by (rule int-sgnE)
moreover from that obtain \( q \) where \( l = \text{sgn } (-1) \ast \text{int } (Suc q) \)
by (cases \( l \)) simp-all
moreover define \( n \) where \( n = \text{Suc } q \)
them have \( \text{Suc } q = n \)
by simp
ultimately show \(?\thesis\)
by (simp only: modulo-int-unfold)
(auto simp add: mod-greater-zero-iff-not-dvd sgn-1-neg)
qed

lemma pos-mod-sign [simp]:
$0 \leq k \mod l$ if $l > 0$ for $k,l :: \text{int}$

**proof** –

obtain $m$ and $s$ where $k = sgn\ s \ast \text{int}\ m$

by (rule $\text{int-sgnE}$)

moreover from that obtain $n$ where $l = sgn\ (-1) \ast \text{int}\ (Suc\ q)$

by (cases $l$) simp-all

moreover define $n$ where $n = Suc\ q$

then have $Suc\ q = n$

by simp

moreover have $\langle\ \text{int}\ (m\ \text{mod}\ n) \leq\ \text{int}\ n\rangle$

using $\langle\ Suc\ q = n\rangle$ by simp

then have $\langle\ sgn\ s \ast\ \text{int}\ (m\ \text{mod}\ n) \leq\ \text{int}\ n\rangle$

by (cases $s\ 0 :: \text{int}$, rule: linorder-cases) simp-all

ultimately show $\langle\ ?thesis\ \rangle$

by (simp only: modulo-int-unfold) auto

qed

**lemma** $\text{neg-mod-sign [simp]}$:

$k \mod l \leq 0$ if $l < 0$ for $k,l :: \text{int}$

**proof** –

obtain $m$ and $s$ where $k = sgn\ s \ast \text{int}\ m$

by (rule $\text{int-sgnE}$)

moreover from that obtain $q$ where $l = sgn\ (-1) \ast \text{int}\ (Suc\ q)$

by (cases $l$) simp-all

ultimately show $\langle?thesis\ \rangle$

by (simp only: modulo-int-unfold) (auto simp add: $\text{sgn-1-pos}$)

qed

56.5.9 Splitting Rules for $\text{div}$ and $\text{mod}$

**lemma** $\text{split-zdiv}$:

$\langle P\ (n\ \text{div}\ k)\ \langle k = 0 \rightarrow P\ 0\rangle \wedge$

$(0 < k \rightarrow (\forall\ i\ j.\ 0 \leq j < k \wedge n = k \ast i + j \rightarrow P\ i)) \wedge$

$(k < 0 \rightarrow (\forall\ i\ j.\ k < j \wedge 0 \leq n = k \ast i + j \rightarrow P\ i))\rangle$ (is $\langle?\text{div}\rangle$

and $\text{split-zmod}$:

$\langle Q\ (n\ \text{mod}\ k)\ \rightarrow$

$(k = 0 \rightarrow Q\ n) \wedge$

$(0 < k \rightarrow (\forall\ i\ j.\ 0 \leq j < k \wedge n = k \ast i + j \rightarrow Q\ j)) \wedge$

$(k < 0 \rightarrow (\forall\ i\ j.\ k < j \wedge 0 \leq n = k \ast i + j \rightarrow Q\ j))\rangle$ (is $\langle?\text{mod}\rangle$

for $n,k :: \text{int}$

**proof** –

have $\ast: \langle R\ (n\ \text{div}\ k)\ (n\ \text{mod}\ k)\ \rightarrow$

$(k = 0 \rightarrow R\ 0\ n) \wedge$

$(0 < k \rightarrow (\forall\ i\ j.\ 0 \leq j < k \wedge n = k \ast i + j \rightarrow R\ i\ j)) \wedge$

$(k < 0 \rightarrow (\forall\ i\ j.\ k < j \wedge 0 \leq n = k \ast i + j \rightarrow R\ i\ j))\rangle$ for $R$

by (cases $k = 0$)
(auto simp add: linorder_class.neq_iff)
from * [of (λq · P q)] show ?div .
from * [of (λ· q · Q r)] show ?mod .
qed

Enable (lin)arith to deal with (div) and (mod) when these are applied to some constant that is of the form numeral k:

declare split-zdiv [of - - (numeral n), linarith-split] for n
declare split-zdiv [of - (numeral n), linarith-split] for n
declare split-zmod [of - - (numeral n), linarith-split] for n
declare split-zmod [of - - (numeral n), linarith-split] for n

lemma zdiv-eq-0-iff:
  i div k = 0 ←→ k = 0 ∨ 0 ≤ i ∧ i < k ∧ i ≤ 0 ∧ k < i (is ?L = ?R)
  for i k :: int
proof
  assume ?L
  moreover have ?L → ?R
    by (rule split-zdiv [THEN iffD2]) simp
  ultimately show ?R
    by blast
next
  assume ?R then show ?L
    by auto
qed

lemma zmod-trivial-iff:
  fixes i k :: int
  shows i mod k = i ←→ k = 0 ∨ 0 ≤ i ∧ i < k ∧ i ≤ 0 ∧ k < i
proof
  have i mod k = i ←→ i div k = 0
    using div-mult-mod-eq [of i k] by safe auto
  with zdiv-eq-0-iff
  show ?thesis
    by simp
qed

56.5.10 Algebraic rewrites

lemma zdiv-zmult2-eq: ⟨a div (b · c) = (a div b) div c⟩ (is ?Q)
  and zmod-zmult2-eq: ⟨a mod (b · c) = b · (a div b mod c) + a mod b⟩ (is ?P)
  if c ≥ 0, for a b c :: int
proof
  have *: ⟨(a div (b · c), a mod (b · c)) = ((a div b) div c, b · (a div b mod c) + a mod b)⟩
    if b > 0 for a b
  proof (induction rule: euclidean-relationI)
    case by0
    then show ?case by auto
next
  case divides
  then obtain d where \( a = b \cdot c \cdot d \)
    by blast
  with divides that show \(?\)case
    by (simp add: ac-simps)
next
case euclidean-relation
with \( \langle b > 0 \rangle \cdot \langle c \geq 0 \rangle \) have \( \langle 0 < c \rangle \cdot \langle b > 0 \rangle \)
  by simp-all
then have \( \langle a \bmod b < b \rangle \)
  by simp
moreover have \( \langle 1 \leq c - a \bmod b \rangle \)
  using \( \langle c > 0 \rangle \) by (simp add: int-one-le-iff-zero-less)
ultimately have \( \langle a \bmod b * 1 < b * (c - a \bmod b) \rangle \)
  by (rule mult-less-le-imp-less) (use \( \langle b > 0 \rangle \) in simp-all)
with \( \langle 0 < b \rangle \cdot \langle 0 < c \rangle \) show \(?\)case
  by (simp add: division-segment-int-def algebra-simps flip: minus-mod-eq-mult-div)
qed
show \(?\)Q
proof (cases \( \langle b \geq 0 \rangle \))
  case True
    with * [of b a] show \(?\)thesis
      by (cases \( \langle b = 0 \rangle \)) simp-all
next
  case False
    with * [of \(- b\) \(- a\)] show \(?\)thesis
      by simp
qed
show \(?\)P
proof (cases \( \langle b \geq 0 \rangle \))
  case True
    with * [of b a] show \(?\)thesis
      by (cases \( \langle b = 0 \rangle \)) simp-all
next
  case False
    with * [of \(- b\) \(- a\)] show \(?\)thesis
      by simp
qed
dlemma zdiv-zmult2-eq':
\( \langle k \div (l * j) = \langle (\sgn j * k) \div l \rangle \div |j| \rangle \) for \( k \cdot l \cdot j :: \text{int} \)
proof
  have \( \langle k \div (l * j) = (\sgn j * k) \div (\sgn j * (l \cdot j)) \rangle \)
    by (simp add: sgn-0-0)
  also have \( \langle (\sgn j * (l \cdot j)) = l \cdot |j| \rangle \)
    by (simp add: mult.left-commute of - l abs-sgn) (simp add: ac-simps)
  also have \( \langle (\sgn j * k) \div (l \cdot |j|) = ((\sgn j * k) \div l) \div |j| \rangle \)
by (simp add: zdiv-zmult2-eq)

finally show ?thesis .

qed

lemma half-nonnegative-int-iff [simp]:
  \( \langle k \div 2 \geq 0 \longleftrightarrow k \geq 0 \rangle \) for \( k :: \text{int} \)
  by auto

lemma half-negative-int-iff [simp]:
  \( \langle k \div 2 < 0 \longleftrightarrow k < 0 \rangle \) for \( k :: \text{int} \)
  by auto

56.5.11 Distributive laws for conversions.

lemma zdiv-int:
  \( \langle \text{int} (m \div n) = \text{int} m \div \text{int} n \rangle \)
  by (cases \( \langle m = 0 \rangle \)) (auto simp add: divide-int-def)

lemma zmod-int:
  \( \langle \text{int} (m \mod n) = \text{int} m \mod \text{int} n \rangle \)
  by (cases \( \langle m = 0 \rangle \)) (auto simp add: modulo-int-def)

lemma nat-div-distrib:
  \( \langle \text{nat} (x \div y) = \text{nat} x \div \text{nat} y \rangle \) if \( \langle 0 \leq x \rangle \)
  using that by (simp add: divide-int-def sgn-if)

lemma nat-div-distrib':
  \( \langle \text{nat} (x \div y) = \text{nat} x \div \text{nat} y \rangle \) if \( \langle 0 \leq y \rangle \)
  using that by (simp add: divide-int-def sgn-if)

lemma nat-mod-distrib:
  \( \langle \text{nat} (x \mod y) = \text{nat} x \mod \text{nat} y \rangle \) if \( \langle 0 \leq x \rangle \langle 0 \leq y \rangle \)
  using that by (simp add: modulo-int-def sgn-if)

56.5.12 Monotonicity in the First Argument (Dividend)

lemma zdiv-mono1:
  \( \langle a \div b \leq a' \div b \rangle \)
  if \( \langle a \leq a' \rangle \langle 0 < b \rangle \)
  for \( a \ b \ b' :: \text{int} \)

proof
  from \( \langle a \leq a' \rangle \) have \( \langle b * (a \div b) + a \mod b \leq b * (a' \div b) + a' \mod b \rangle \)
  by simp
  then have \( \langle b * (a \div b) \leq (a' \mod b - a \mod b) + b * (a' \div b) \rangle \)
  by (simp add: algebra-simps)
  moreover have \( \langle a' \mod b < b + a \mod b \rangle \)
  by (rule less-le-trans [of \( \langle b \rangle \)]) (use \( \langle 0 < b \rangle \) in simp-all)
  ultimately have \( \langle b * (a \div b) < b * (1 + a' \div b) \rangle \)
  by (simp add: distrib-left)
with \(0 < b\) have \(a \div b < 1 + a' \div b\)
by (simp add: mult-less-cancel-left)
then show \(?thesis\)
by simp
qed

lemma \(\text{zdiv-mono1-neg}\):
\[a' \div b \leq a \div b\]
if \(a \leq a'\) \(b < 0\)
for \(a\) \(a'\) \(b\) :: int
using that \(\text{zdiv-mono1}\) [of \((-a') \ (-a) \ (-b)\) by simp]

56.5.13 Monotonicity in the Second Argument (Divisor)

lemma \(\text{zdiv-mono2}\):
\(a \div b \leq a \div b'\) if \(0 \leq a\) \(0 < b\) \(b' \leq b\) for \(a\) \(b\) \(b'\) :: int
proof –
define \(q\) \(q'\) \(r\) \(r'\) where **: \(q = a \div b\) \(q' = a \div b'\) \(r = a \mod b\) \(r' = a \mod b'\)
then have **: \(b * q + r = b' * q' + r'\) \(0 \leq b' * q' + r'\)
\(r' < b'\) \(0 \leq r\) \(0 < b'\) \(b' \leq b\)
using that by simp-all
have \(0 < b' * (q' + 1)\)
using * by (simp add: distrib-left)
with * have \(0 \leq q'\)
by (simp add: zero-less-mult-iff)
moreover have \(b * q = r' - r + b' * q'\)
using * by linarith
ultimately have \(b * q < b * (q' + 1)\)
using mult-right-mono * unfolding distrib-left by fastforce
with * have \(q \leq q'\)
by (simp add: mult-less-cancel-left-pos)
with ** show \(?thesis\)
by simp
qed

lemma \(\text{zdiv-mono2-neg}\):
\(a \div b' \leq a \div b\) if \(a < 0\) \(0 < b\) \(b' \leq b\) for \(a\) \(b\) \(b'\) :: int
proof –
define \(q\) \(q'\) \(r\) \(r'\) where **: \(q = a \div b\) \(q' = a \div b'\) \(r = a \mod b\) \(r' = a \mod b'\)
then have **: \(b * q + r = b' * q' + r'\) \(b' * q' + r' < 0\)
\(r < b\) \(0 \leq r'\) \(0 < b'\) \(b' \leq b\)
using that by simp-all
have \(b' * q' < 0\)
using * by linarith
with * have \(q' \leq 0\)
by (simp add: mult-less-0-iff)
have \(b * q' \leq b' * q'\)
by (simp add: \(q' \leq 0\) * mult-right-mono-neg)
then have \(b * q' < b * (q + 1)\)
  using * by (simp add: distrib-left)
then have \(q' \leq q\)
  using * by (simp add: mult-less-cancel-left)
then show \(?thesis\)
  by (simp add: **) 
qed

56.5.14 Quotients of Signs

lemma div-eq-minus1:
\(<0 < b \implies -1 \ div \ b = -1\> \ for \ b :: \ int\)
by (simp add: divide-int-def)

lemma zmod-minus1:
\(<0 < b \implies -1 \ mod \ b = b - 1\> \ for \ b :: \ int\)
by (auto simp add: modulo-int-def)

lemma minus-mod-int-eq:
\(<- k \ mod \ l = l - 1 - (k - 1) \ mod \ l \> \ if \ <l \geq 0\> \ for \ k \ l :: \ int\)
proof (cases (\(l = 0\)))
  case True
  then show \(?thesis\)
  by simp
next
  case False
  with that have \(<l > 0\) 
  by simp
then show \(?thesis\)
proof (cases (\(l \ dvd \ k\))
  case True
  then obtain \(j\) where \(<k = l * j\) ..
  moreover have \(<l * j \ mod \ l - 1) \ mod \ l = l - 1\>
    using \(<l > 0\) by (simp add: zmod-minus1)
  then have \(<l * j - 1) \ mod \ l = l - 1\>
    by (simp only: mod-simps)
  ultimately show \(?thesis\)
    by simp
next
  case False
  moreover have \(<1 < k \ mod \ l\>
    using \(<0 < l\) False le-less by fastforce
  moreover have \(<2 < l \ mod \ l < 1 + l\>
    using \(<0 < l\) pos-mod-bound[of \(l k\)] by linarith
  from \(1\ \ 2\ \ (<l > 0)\) have \(<(k \ mod \ l - 1) \ mod \ l = k \ mod \ l - 1\>
    by (simp add: zmod-trivial-iff)
  ultimately show \(?thesis\)
    by (simp only: zmod-minus1-eq-if)
THEORY “Euclidean-Rings”

(simp add: mod-eq-0-iff-dvd algebra-simps mod-simps)

qed

lemma div-pos-nonneg-le0:

\[ a \div b \leq 0 \] if \( a \leq 0 \) \( b < 0 \) for \( a, b :: \text{int} \).

proof

have \( a \div b \leq -1 \div b \)
  using zdiv-mono1 that by auto

also have \( \ldots \leq -1 \)
  by (simp add: that(2) div-eq-minus1)

finally show \( \text{thesis} \)
  by force

qed

lemma pos-imp-zdiv-nonneg-iff:

\[ 0 \leq a \div b \iff 0 \leq a \] if \( 0 < b \) for \( a, b :: \text{int} \).

proof

assume \( 0 \leq a \div b \)

show \( 0 \leq a \)

proof (rule ccontr)

assume \( \neg 0 \leq a \)

then have \( a < 0 \)
  by simp

then have \( a \div b < 0 \)
  using that by (rule div-pos-nonneg-le0)

with \( 0 \leq a \div b \) show False
  by simp

qed

next

assume \( 0 \leq a \)

then have \( 0 \div b \leq a \div b \)
  using zdiv-mono1 that by blast

then show \( 0 \leq a \div b \)
  by auto

qed
lemma neg-imp-zdiv-nonneg-iff:
\(<0 \leq a \div b \iff a \leq 0>\) if \(<b \leq 0>\) for \(a, b :: \text{int}\)
using that pos-imp-zdiv-nonneg-iff [of \(\neg b \iff \neg a\)] by simp

lemma pos-imp-zdiv-pos-iff:
\(<0 < (i :: \text{int}) \div k \iff k \leq i>\) if \(<0 < k>\) for \(i, k :: \text{int}\)
using that pos-imp-zdiv-nonneg-iff [of \(\neg i \iff \neg k\)] by arith

lemma pos-imp-zdiv-neg-iff:
— But not \((a \div b \leq 0) = (a \leq 0)\); consider \(a = 1, b = 2\) when \(a \div b = 0\).
using that by (simp add: pos-imp-zdiv-nonneg-iff flip: linorder-not-le)

lemma neg-imp-zdiv-neg-iff:
— But not \((a \div b < 0) = (0 \leq a)\); consider \(a = -1, b = -2\) when \(a \div b = 0\).
using that \(\neg (0 < b)\) for \(a :: \text{int}\)

lemma nonneg1-imp-zdiv-pos-iff:
\(a \div b > 0 \iff a \geq b \land b > 0\) if \(\neg (0 \leq a)\) for \(a, b :: \text{int}\)
proof –
have \(0 < a \div b \implies b \leq a\)
using div-pos-pos-trivial [of \(a, b\)] that by arith
moreover have \(0 < a \div b \implies b > 0\)
using that div-nonneg-neg-le0 [of \(a, b\)] by (cases \(b\) = 0; force)
moreover have \(b \leq a \land 0 < b \implies 0 < b \div a\)
using int-one-le-iff-zero-less [of \(a, b\)] zdiv-mono1 [of \(b, a, b\)] by simp
ultimately show \(?thesis\)
by blast
qed

lemma zmod-le-nonneg-dividend:
\((m \mod k \leq m)\) if \((m :: \text{int}) \geq 0\) for \(m, k :: \text{int}\)
proof –
from that have \((m > 0 \lor m = 0)\)
by auto
then show \(?thesis\) proof
assume \((m = 0)\) then show \(?thesis\)
by simp
next
assume \((m > 0)\) then show \(?thesis\)
proof (cases \(k :: \text{int}\) rule: linorder-cases)
case less
moreover define \(l\) where \((l = -k)\)
ultimately have \((l > 0)\)
by simp
with \((m > 0)\) have \((\text{nat } m \mod \text{nat } l \leq m)\)
by (simp flip: le-nat-iff)
then have \( \langle \text{int } (\text{nat } m \mod \text{nat } l) - l \leq m \rangle \)
using \( \langle l > 0 \rangle \) by simp
with \( \langle m > 0 \rangle \) \( \langle l > 0 \rangle \) show \( \text{thesis} \)
by \( \langle \text{simp add: modulo-int-def l-def flip: le-nat-iff} \rangle \)
qed \( \langle \text{simp-all add: modulo-int-def flip: le-nat-iff} \rangle \)

qed

lemma \texttt{sgn-div-eq-sgn-mult}:
\( \langle \text{sgn } (k \div l) = \text{of-bool } (k \div l \neq 0) \ast \text{sgn } (k \ast l) \rangle \)
for \( k l :: \text{int} \)
proof \( \langle \text{cases } (k \div l = 0) \rangle \)
  case True
  then show \( \text{thesis} \)
  by simp
next
case False
have \( \langle 0 \leq |k| \div |l| \rangle \)
  by \( \langle \text{cases } (l = 0) \rangle \) \( \langle \text{simp-all add: pos-imp-zdiv-nonneg-iff} \rangle \)
then have \( \langle |k| \div |l| \neq 0 \iff 0 < |k| \div |l| \rangle \)
  by \( \langle \text{simp add: less-le} \rangle \)
also have \( \langle \ldots \iff |k| \geq |l| \rangle \)
  using \( \langle \text{False nonneg1-imp-zdiv-pos-iff} \rangle \) by auto
finally have \( \ast: \langle |k| \div |l| \neq 0 \iff |l| \leq |k| \rangle . \)
  show \( \text{thesis} \)
  using \( \ast \) by \( \langle \text{auto simp add: div-eq-div-abs [of } k \text{ ] div-eq-sgn-abs [of } k \text{ ] sgn-mult sgn-1-pos sgn-1-neg sgn-eq-0-iff nonneg1-imp-zdiv-pos-iff * dest: sgn-not-eq-imp} \rangle \)
qed

56.5.15 Further properties

lemma \texttt{div-int-pos-iff}:
\( k \div l \geq 0 \iff k = 0 \lor l = 0 \lor k \geq 0 \land l \geq 0 \)
\( \lor k < 0 \land l < 0 \)
for \( k l :: \text{int} \)
proof \( \langle \text{cases } k = 0 \lor l = 0 \rangle \)
  case False
  then have \( \ast: k \neq 0 \land l \neq 0 \)
  by auto
  then have \( 0 \leq k \div l \Longrightarrow \neg k < 0 \Longrightarrow 0 \leq l \)
  by \( \langle \text{meson neg-imp-zdiv-neg-iff not-le not-less-iff-gr-or-eq} \rangle \)
  then show \( \text{thesis} \)
  using \( \ast \) by \( \langle \text{auto simp add: pos-imp-zdiv-nonneg-iff neg-imp-zdiv-nonneg-iff} \rangle \)
qed auto

lemma \texttt{mod-int-pos-iff}:
\( \langle k \mod l \geq 0 \iff l \mod k \mod l = 0 \land k \geq 0 \lor l > 0 \rangle \)
for \( k l :: \text{int} \)
proof (cases \( l > 0 \))
  case False
  then show ?thesis
    by (simp add: dvd-eq-mod-eq-0) (use neg-mod-sign [of \( l \) \( k \)] in (auto simp add: le-less not-less))

qed auto

lemma abs-div:
  \(| x \) div \( y \)\| = |\( x \) div \( y \)\| if \( y \) dvd \( x \) for \( x \) \( y \) :: int
  using that by (cases \( y = 0 \)) (auto simp add: abs-mult)

lemma int-power-div-base:
  \( k ^ m \) div \( k \) = \( k ^ (m - Suc 0) \) if \( 0 < m \) \( 0 < n \) for \( k :: int \)
  using that by (cases \( m \)) simp-all

lemma int-div-less-self:
  \( x \) div \( k \) < \( x \) if \( 0 < x \) \( 1 < k \) for \( x \) \( k :: int \)
proof -
  from that have \( \text{nat} (x \) div \( k) = \text{nat} x \) div \( \text{nat} k \)
    by (simp add: nat-div-distrib)
  also from that have \( \text{nat} x \) div \( k < \text{nat} x \)
    by simp
  finally show ?thesis
    by simp
qed

56.5.16 Computing div and mod by shifting

lemma div-pos-geq:
  \( k \) div \( l \) = \((k - l) \) div \( l + 1 \) if \( 0 < l \) \( l \leq k \) for \( k \) \( l :: int \)
proof -
  have \( k = (k - l) + l \) by simp
  then obtain \( j \) where \( k = j + l \) ..
  with that show ?thesis by (simp add: div-add-self2)

qed

lemma mod-pos-geq:
  \( k \) mod \( l \) = \((k - l) \) mod \( l \) if \( 0 < l \) \( l \leq k \) for \( k \) \( l :: int \)
proof -
  have \( k = (k - l) + l \) by simp
  then obtain \( j \) where \( k = j + l \) ..
  with that show ?thesis by simp

qed

lemma pos-zdiv-mult-2: \( (1 + 2 \cdot b) \) div \( (2 \cdot a) = b \) div \( a \) \( (\text{is} \ ?Q) \)
  and pos-zmod-mult-2: \( (1 + 2 \cdot b) \) mod \( (2 \cdot a) = 1 + 2 \cdot (b \) mod \( a) \) \( (\text{is} \ ?R) \)
if \( 0 \leq a \) for \( a \) \( b :: \text{int} \)
proof -
  have \( ((1 + 2 \cdot b) \) div \( (2 \cdot a), (1 + 2 \cdot b) \) mod \( (2 \cdot a) = (b \) div \( a, 1 + 2 \cdot b \) mod \( a) \)
  by (simp add: div-and-mod-first2) (use neg-mod-sign [of \( b \)] in (auto simp add: div-and-mod-first2))

qed
((b mod a))

proof (induction rule: euclidean-relation-intI)
  case by0
  then show ?case
    by simp
next
  case divides
  have ‹2 dvd (2 * a)›
    by simp
  then have ‹2 dvd (1 + 2 * b)›
    using ‹2 * a dvd 1 + 2 * b› by (rule dvd-trans)
  then have ‹2 dvd (1 + b * 2)›
    by (simp add: ac-simps)
  then have ‹is-unit (2 :: int)›
    by simp
  then show ?case
    by simp
next
  case euclidean-relation
  with that have ‹a > 0›
    by simp
  moreover have ‹b mod a < a›
    using ‹a > 0› by simp
  then have ‹1 + 2 * (b mod a) < 2 * a›
    by simp
  moreover have ‹2 * (b mod a) + a * (2 * (b div a)) = 2 * (b div a * a + b mod a)›
    by (simp only: algebra-simps)
  moreover have ‹0 ≤ 2 * (b mod a)›
    using ‹a > 0› by simp
  ultimately show ?case
    by (simp add: algebra-simps)
qed
then show ?Q and ?R
  by simp-all
qed

lemma neg-zdiv-mult-2: ‹(1 + 2 * b) div (2 * a) = (b + 1) div a› (is ?Q)
  and neg-zmod-mult-2: ‹(1 + 2 * b) mod (2 * a) = 2 * ((b + 1) mod a) - 1›
(is ?R)
  if ‹a ≤ 0› for a b :: int
proof -
  have ‹((1 + 2 * b) div (2 * a), (1 + 2 * b) mod (2 * a)) = ((b + 1) div a, 2 * ((b + 1) mod a) - 1)›
    proof (induction rule: euclidean-relation-intI)
      case by0
      then show ?case
        by simp
    next
THEORY "Euclidean-Rings"

case divides
  have \(2 \text{ dvd } (2 \ast a)\)
    by simp
  then have \(2 \text{ dvd } (1 + 2 \ast b)\)
    using \(2 \ast a \text{ dvd } 1 + 2 \ast b\) by (rule dvd-trans)
  then have \(2 \text{ dvd } (1 + b + 2)\)
    by (simp add: ac-simps)
  then have \((\text{is-unit } (2 :: \text{int}))\)
    by simp
  then show \(?\) by simp
next
  case euclidean-relation
  with that have \(< a < 0\)\)
    by simp
  moreover have \((b + 1) \text{ mod } a > a\)
    using \(< a < 0\) by simp
  then have \(2 \ast ((b + 1) \text{ mod } a) > 1 + 2 \ast a\)
    by simp
  moreover have \(((1 + b) \text{ mod } a) \leq 0\)
    using \(< a < 0\) by simp
  then have \(2 \ast ((1 + b) \text{ mod } a) \leq 0\)
    by simp
  moreover have \(2 \ast ((1 + b) \text{ mod } a) + a \ast (2 \ast ((1 + b) \text{ div } a)) =
    2 \ast ((1 + b) \text{ div } a \ast a + (1 + b) \text{ mod } a)\)
    by (simp only: algebra-simps)
  ultimately show \(?\) by (simp add: algebra-simps sgn-mult abs-mult)
  qed
  then show \(?Q\) and \(?R\)
    by simp-all
qed

lemma zdiv-numeral-1 [simp]:
\(\text{numeral (Num.Bit0 v) div numeral (Num.Bit0 w)} =
\text{numeral v div (numeral w :: int)}\)
unfolding numeral.simps unfolding mult-2 [symmetric]
by (rule div-mul-mult1) simp

lemma zdiv-numeral-1 [simp]:
\(\text{numeral (Num.Bit1 v) div numeral (Num.Bit0 w)} =
\text{numeral v div (numeral w :: int)}\)
unfolding numeral.simps
unfolding mult-2 [symmetric] add.commute [of \(-1\)]
by (rule pos-zdiv-mult-2) simp

lemma zmod-numeral-1 [simp]:
\(\text{numeral (Num.Bit0 v) mod numeral (Num.Bit0 w)} =
\text{(2::int) * (numeral v mod numeral w)}\)
THEORY “Parity”

unfolding numeral-Bit0 [of v] numeral-Bit0 [of w]
unfolding mult-2 [symmetric] by (rule mod-mult-mult1)

lemma zmod-numeral-Bit1 [simp]:
numeral (Num.Bit1 v) mod numeral (Num.Bit0 w) =
2 * (numeral v mod numeral w) + (1::int)
unfolding numeral-Bit1 [of v] numeral-Bit0 [of w]
unfolding mult-2 [symmetric] add.commute [of - 1]
by (rule pos-zmod-mult-2) simp

56.6 Code generation
context
begin

qualified definition divmod-nat :: nat ⇒ nat ⇒ nat × nat
where divmod-nat m n = (m div n, m mod n)

qualified lemma divmod-nat-if [code]:
divmod-nat m n = (if n = 0 ∨ m < n then (0, m) else
let (q, r) = divmod-nat (m - n) n in (Suc q, r))
by (simp add: divmod-nat-def prod-eq-iff case-prod-beta not-less le-div-geq le-mod-geq)

qualified lemma [code]:
m div n = fst (divmod-nat m n)
m mod n = snd (divmod-nat m n)
by (simp-all add: divmod-nat-def)

end

code-identifier
code-module Euclidean-Rings → (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

57 Parity in rings and semirings

theory Parity
imports Euclidean-Rings
begin

57.1 Ring structures with parity and even/odd predicates

class semiring-parity = comm-semiring-1 + semiring-modulo +
assumes mod-2-eq-odd: (a mod 2 = of-bool (∼ 2 dvd a))
and odd-one [simp]: (∼ 2 dvd 1)
and even-half-succ-eq [simp]: (2 dvd a ⇒ (1 + a) div 2 = a div 2)
begin
THEORY “Parity”

abbreviation even :: 'a ⇒ bool
  where (even a ≡ 2 dvd a)

abbreviation odd :: 'a ⇒ bool
  where (odd a ≡ ¬ 2 dvd a)

end

class ring-parity = ring + semiring-parity
begin

subclass comm-ring-1 ..

end

instance nat :: semiring-parity
  by standard (auto simp add: dvd-eq-mod-eq-0)

instance int :: ring-parity
  by standard (auto simp add: dvd-eq-mod-eq-0)

context semiring-parity
begin

lemma evenE [elim?]:
  assumes (even a)
  obtains b where (a = 2 * b)
  using assms by (rule dvdE)

lemma oddE [elim?]:
  assumes (odd a)
  obtains b where (a = 2 * b + 1)
  proof –
    have (a = 2 * (a div 2) + a mod 2)
      by (simp add: mult-div-mod-eq)
    with assms have (a = 2 * (a div 2) + 1)
      by (simp add: mod-2-eq-odd)
    then show thesis ..
  qed

lemma of-bool-odd-eq-mod-2:
  (of-bool (odd a) = a mod 2)
  by (simp add: mod-2-eq-odd)

lemma odd-of-bool-self [simp]:
  (odd (of-bool p) ←→ p)
  by (cases p) simp-all
lemma not-mod-2-eq-0-eq-1 [simp]:
\[ a \mod 2 \neq 0 \iff a \mod 2 = 1 \]
by (simp add: mod-2-eq-odd)

lemma not-mod-2-eq-1-eq-0 [simp]:
\[ a \mod 2 \neq 1 \iff a \mod 2 = 0 \]
by (simp add: mod-2-eq-odd)

lemma even-iff-mod-2-eq-zero:
\[ 2 \mid a \iff a \mod 2 = 0 \]
by (simp add: mod-2-eq-odd)

lemma odd-iff-mod-2-eq-one:
\[ \neg 2 \mid a \iff a \mod 2 = 1 \]
by (simp add: mod-2-eq-odd)

lemma even-mod-2-iff:
\[ \text{even} \ (a \mod 2) \iff \text{even} \ a \]
by (simp add: mod-2-eq-odd)

lemma mod2-eq-if:
\[ a \mod 2 = \text{if even} \ a \text{ then } 0 \text{ else } 1 \]
by (simp add: mod-2-eq-odd)

lemma zero-mod-two-eq-zero [simp]:
\[ 0 \mod 2 = 0 \]
by (simp add: mod-2-eq-odd)

lemma one-mod-two-eq-one [simp]:
\[ 1 \mod 2 = 1 \]
by (simp add: mod-2-eq-odd)

lemma parity-cases [case_names even odd]:
assumes \[ \text{even} \ a \implies a \mod 2 = 0 \implies P \]
assumes \[ \text{odd} \ a \implies a \mod 2 = 1 \implies P \]
shows \( P \)
using assms by (auto simp add: mod-2-eq-odd)

lemma even-zero [simp]:
\[ \text{even} \ 0 \]
by (fact dvd-0-right)

lemma odd-even-add:
\[ \text{even} \ (a + b) \text{ if odd} \ a \text{ and odd} \ b \]
proof -
from that obtain \( c \ d \) where \( a = 2 \cdot c + 1 \) and \( b = 2 \cdot d + 1 \)
by (blast elim: oddE)
then have \( a + b = 2 \cdot c + 2 \cdot d + (1 + 1) \)
by (simp only: ac-simps)
also have \( \ldots = 2 \ast (c + d + 1) \)
by (simp add: algebra-simps)
finally show \(?thesis \).
qed

lemma even-add [simp]:
even \((a + b) \leftarrow\rightarrow (even\ a \leftarrow\rightarrow even\ b)\)
by (auto simp add: dvd-add-right-iff dvd-add-left-iff odd-even-add)

lemma odd-add [simp]:
odd \((a + b) \leftarrow\rightarrow \neg (odd\ a \leftarrow\rightarrow odd\ b)\)
by simp

lemma even-plus-one-iff [simp]:
even \((a + 1) \leftarrow\rightarrow odd\ a\)
by (auto simp add: dvd-add-right-iff intro: odd-even-add)

lemma even-mult-iff [simp]:
even \((a \ast b) \leftarrow\rightarrow even\ a \lor even\ b\ (is \ ?P \leftarrow\rightarrow ?Q)\)
proof
  assume \(?Q\)
  then show \(?P\)
    by auto
next
  assume \(?P\)
  show \(?Q\)
  proof (rule ccontr)
    assume \(\neg (even\ a \lor even\ b)\)
    then have \(odd\ a\ and\ odd\ b\)
      by auto
    then obtain \(r\ s\ where\ a = 2 \ast r + 1\ and\ b = 2 \ast s + 1\)
      by (blast elim: oddE)
    then have \(a \ast b = (2 \ast r + 1) \ast (2 \ast s + 1)\)
      by simp
    also have \(\ldots = 2 \ast (2 \ast r \ast s + r + s + 1)\)
      by (simp add: algebra-simps)
    finally have \(odd\ (a \ast b)\)
      by simp
    with \(?P\), show False
      by auto
  qed
qed

lemma even-numeral [simp]: even \((numeral\ (Num.Bit0\ n))\)
proof
  have even \((2 \ast numeral\ n)\)
    unfolding even-mult-iff by simp
  then have even \((numeral\ n + numeral\ n)\)
    unfolding mult-2 .
THEORY “Parity”

then show ?thesis
  unfolding numeral.simps .
qed

lemma odd-numeral [simp]: odd (numeral (Num.Bit1 n))
proof
  assume even (numeral (Num.Bit1 n))
  then have even (numeral n + numeral n + 1)
    unfolding numeral.simps .
  then have even (2 * numeral n + 1)
    unfolding mult-2 .
  then have 2 dvd numeral n * 2 + 1
    by (simp add: ac-simps)
  then have 2 dvd 1
    using dvd-add-times-triv-left-iff [of 2 numeral n 1] by simp
  then show False by simp
qed

lemma odd-numeral-BitM [simp]:
  (odd (numeral (Num.BitM w))
by (cases w) simp-all

lemma even-power [simp]: even (a ^ n) <-> even a ∧ n > 0
by (induct n) auto

lemma even-prod-iff:
  even (prod f A) <-> (∃a∈A. even (f a)) if finite A
using that by (induction A) simp-all

lemma even-half-maybe-succ-eq [simp]:
  even a --> (of-bool b + a) div 2 = a div 2;
by simp

lemma even-half-maybe-succ'-eq [simp]:
  even a --> (b mod 2 + a) div 2 = a div 2;
by (simp add: mod2-eq-if)

lemma mask-eq-sum-exp:
  2 ^ n - 1 = (∑ m∈{q. q < n}. 2 ^ m)
proof
  have *: {q. q < Suc m} = insert m {q. q < m}; for m
  by auto
  have 2 ^ n = (∑ m∈{q. q < n}. 2 ^ m) + 1;
    by (induction n) (simp-all add: ac-simps mult-2 *)
  then have 2 ^ n - 1 = (∑ m∈{q. q < n}. 2 ^ m) + 1 - 1;
    by simp
  then show ?thesis
    by simp
qed
lemma (in −) mask-eq-sum-exp-nat:
\[2^n - \text{Suc } 0 = \left(\sum_{q.<n} 2^q\right)\]
using mask-eq-sum-exp [where ?a = nat] by simp

end

context ring-parity
begin

lemma even-minus:
even \((- a) \leftrightarrow\) even a
by (fact dvd-minus-iff)

lemma even-diff [simp]:
even \((a - b) \leftrightarrow\) even \((a + b)\)
using even-add [of \(a - b\)] by simp

end

57.2 Instance for nat

lemma even-Suc-Suc-iff [simp]:
even \((\text{Suc } (\text{Suc } n)) \leftrightarrow\) even \(n\)
using dvd-add-triv-right-iff [of \(2 n\)] by simp

lemma even-Suc [simp]: even \((\text{Suc } n) \leftrightarrow\) odd \(n\)
using even-plus-one-iff [of \(n\)] by simp

lemma even-diff-nat [simp]:
even \((m - n) \leftrightarrow\) \(m < n \lor\) even \((m + n)\) for \(m n::\text{nat}\)
proof (cases \(n \leq m\))
  case True
  then have \((m - n + n * 2) = m + n\) by (simp add: mult-2-right)
  moreover have \(((m - n) \leftrightarrow\) even \((m - n + n * 2)\) by simp
  ultimately have \((m - n) \leftrightarrow\) even \((m + n)\) by (simp only:)
  then show \(\text{thesis by auto}\)
next
  case False
  then show \(\text{thesis by simp}\)
qed

lemma odd-pos:
odd \(n \Longrightarrow 0 < n\) for \(n::\text{nat}\)
by (auto elim: oddE)

lemma Suc-double-not-eq-double:
Suc \((2 * m) \neq 2 * n\)
proof
THEORY "Parity"

assume Suc (2 * m) = 2 * n
moreover have odd (Suc (2 * m)) and even (2 * n)
   by simp-all
ultimately show False by simp
qed

lemma double-not-eq-Suc-double:
  2 * m \neq Suc (2 * n)
using Suc-double-not-eq-double [of n m] by simp

lemma odd-Suc-minus-one [simp]: odd n \implies Suc (n - Suc 0) = n
by (auto elim: oddE)

lemma even-Suc-div-two [simp]:
even n \implies Suc n div 2 = n div 2
by auto

lemma odd-Suc-div-two [simp]:
odd n \implies Suc n div 2 = Suc (n div 2)
by (auto elim: oddE)

lemma odd-two-times-div-two-nat [simp]:
assumes odd n
shows 2 * (n div 2) = n - (1 :: nat)
proof -
from assms have 2 * (n div 2) + 1 = n
   by (auto elim: oddE)
then have Suc (2 * (n div 2)) - 1 = n - 1
   by simp
then show ?thesis
   by simp
qed

lemma not-mod2-eq-Suc-0-eq-0 [simp]:
n mod 2 \neq Suc 0 \iff n mod 2 = 0
using not-mod-2-eq-1-eq-0 [of n] by simp

lemma odd-card-imp-not-empty:
\langle A \neq \{\} \rangle if \langle odd (card A)\rangle
using that by auto

lemma nat-induct2 [case-names 0 1 step]:
assumes P 0 P 1 and step: \\( \forall n::nat. \ P n \implies P (n + 2) \)
shows P n
proof (induct n rule: less-induct)
case (less n)
show ?case
proof (cases n < Suc (Suc 0))
  case True
then show \( ?\text{thesis} \)
  using assms by (auto simp: less-Suc-eq)

next
  case False
  then obtain \( k \) where \( k: n = \text{Suc} (\text{Suc} k) \)
    by (force simp: not-less nat-le-iff-add)
  then have \( k < n \)
    by simp
  with less assms have \( P (k+2) \)
    by blast
  then show \( ?\text{thesis} \)
    by (simp add: k)
qed

lemma mod-double-nat:
  \(< n \mod (2 * m) = n \mod m \lor n \mod (2 * m) = n \mod m + m \> 
for \( m \) \( n \) :: nat
  by (cases \( \text{even} (n \div m) \))
    (simp-all add: mod-mult2-eq ac-simps even-iff-mod-2-eq-zero)

context semiring-parity
begin

lemma even-sum-iff:
  \(< \text{even} (\sum f A) \leftrightarrow \text{even} (\text{card} \{ a\in A. \text{odd} (f a)\}) \> 
if \( \text{finite} A \)
using that proof (induction A)
  case empty
    then show \( ?\text{case} \)
      by simp
next
  case (insert a A)
  moreover have \( \{ b \in \text{insert} a A. \text{odd} (f b) \} = (\text{if odd} (f a) \text{ then } \{a\} \text{ else } \{\}) \cup \{b \in A. \text{odd} (f b)\} \)
    by auto
  ultimately show \( ?\text{case} \)
    by simp
qed

lemma even-mask-iff [simp]:
  \(< \text{even} (2 ^ n - 1) \leftrightarrow n = 0 \> 
proof (cases \( \text{int} n = 0 \))
  case True
    then show \( ?\text{thesis} \)
      by simp
next
  case False
  then have \( \{a. a = 0 \land a < n\} = \{0\} \)
    by auto
then show \( \text{thesis} \)
  by \((\text{auto simp add: mask-eq-sum-exp even-sum-iff})\)

qed

lemma even-of-nat-iff [simp]:
  even \((\text{of-nat } n)\) \(\iff\) even \(n\)
by \((\text{induction } n)\) simp-all

end

57.3 Parity and powers

context ring_1 begin

lemma power-minus-even [simp]: even \(n\) \(\Rightarrow\) \((-a) \^ n = a \^ n\)
by \((\text{auto elim: evenE})\)

lemma power-minus-odd [simp]: odd \(n\) \(\Rightarrow\) \((-a) \^ n = -(a \^ n)\)
by \((\text{auto elim: oddE})\)

lemma uminus-power-if:
  \((-a) \^ n = (\text{if even } n \text{ then } a \^ n \text{ else } -(a \^ n))\)
by auto

lemma neg-one-even-power [simp]: even \(n\) \(\Rightarrow\) \((-1) \^ n = 1\)
by simp

lemma neg-one-odd-power [simp]: odd \(n\) \(\Rightarrow\) \((-1) \^ n = -1\)
by simp

lemma neg-one-power-add-eq-neg-one-power-diff: \(k \leq n \Rightarrow (-1) \^ (n + k) = (-1) \^ (n - k)\)
by \((\text{cases even } (n + k))\) auto

lemma minus-one-power-iff: \((-1) \^ n = (\text{if even } n \text{ then } 1 \text{ else } -1)\)
by \((\text{induct } n)\) auto

end

context linordered_idom begin

lemma zero-le-even-power: even \(n\) \(\Rightarrow\) \(0 \leq a \^ n\)
by \((\text{auto elim: evenE})\)

lemma zero-le-odd-power: odd \(n\) \(\Rightarrow\) \(0 \leq a \^ n \iff 0 \leq a\)
by \((\text{auto simp add: power-even-eq zero-le-mult-iff elim: oddE})\)
lemma zero-le-power-eq: $0 \leq a \Rightarrow even \, n \lor odd \, n \land 0 \leq a$
by (auto simp add: zero-le-even-power zero-le-odd-power)

lemma zero-less-power-eq: $0 < a \Rightarrow n = 0 \lor even \, n \land a \neq 0 \lor odd \, n \land 0 < a$
proof
  have [simp]: $0 = a \Rightarrow a = 0 \land n > 0$
  unfolding power-equ-0-iff [of a n, symmetric] by blast
  show ?thesis
  unfolding less-le zero-le-power-eq by auto
qed

lemma power-less-zero-eq [simp]: $a \n < 0 \Rightarrow odd \, n \land a < 0$
unfolding not-le [symmetric] zero-le-power-eq by auto

lemma power-le-zero-eq: $a \n \leq 0 \Rightarrow n > 0 \land (odd \, n \land a \leq 0 \lor even \, n \land a = 0)$
unfolding not-less [symmetric] zero-less-power-eq by auto

lemma power-even-abs [simp]: $even \, n \Rightarrow |a| \n = a \n$
using power-abs [of a n] by (simp add: zero-le-even-power)

lemma power-mono-even:
  assumes $even \, n$ and $|a| \leq |b|$
  shows $a \n \leq b \n$
proof
  have $0 \leq |a|$ by auto
  with $|a| \leq |b|$ have $|a| \n \leq |b| \n$
  by (rule power-mono)
  with $even \, n$ show ?thesis
  by (simp add: power-even-abs)
qed

lemma power-mono-odd:
  assumes $odd \, n$ and $a \leq b$
  shows $a \n \leq b \n$
proof (cases $b < 0$)
  case True
  with $a \leq b$ have $-b \leq -a$ and $0 \leq -b$ by auto
  then have $(-b) \n \leq (-a) \n$ by (rule power-mono)
  with $odd \, n$ show ?thesis by simp
next
  case False
  then have $0 \leq b$ by auto
  show ?thesis
  proof (cases $a < 0$)
    case True
    then have $n \neq 0$ and $a \leq 0$ using $odd \, n$ [THEN odd-pos] by auto
    then have $a \n \leq 0$ unfolding power-le-zero-eq using $odd \, n$ by auto
  qed

qed
moreover from \(0 \leq b\) have \(0 \leq b^n\) by auto
ultimately show \(\text{thesis}\) by auto

next
  case False
  then have \(0 \leq a\) by auto
  with \(a \leq b\) show \(\text{thesis}\)
    using power-mono by auto

qed

qed

Simplify, when the exponent is a numeral

lemma zero-le-power-eq-numeral [simp]:
  \(0 \leq a^{\text{numeral } w}\) \iff \((\text{even } \text{numeral } w :: \text{nat}) \lor \text{odd } (\text{numeral } w :: \text{nat}) \land 0 \leq a\)
by (fact zero-le-power-eq)

lemma zero-less-power-eq-numeral [simp]:
  \(0 < a^{\text{numeral } w}\) \iff
    \((\text{numeral } w = (0 :: \text{nat}) \lor \text{even } (\text{numeral } w :: \text{nat}) \land a \neq 0) \lor\)
    \(\text{odd } (\text{numeral } w :: \text{nat}) \land 0 < a\)
by (fact zero-less-power-eq)

lemma power-le-zero-eq-numeral [simp]:
  \(a^{\text{numeral } w} \leq 0\) \iff
    \((0 :: \text{nat}) < \text{numeral } w \land\)
    \((\text{odd } (\text{numeral } w :: \text{nat}) \land a \leq 0 \lor \text{even } (\text{numeral } w :: \text{nat}) \land a = 0)\)
by (fact power-le-zero-eq)

lemma power-less-zero-eq-numeral [simp]:
  \(a^{\text{numeral } w} < 0\) \iff \(\text{odd } (\text{numeral } w :: \text{nat}) \land a < 0\)
by (fact power-less-zero-eq)

lemma power-even-abs-numeral [simp]:
  \(\text{even } (\text{numeral } w :: \text{nat}) \Longrightarrow |a|^{\text{numeral } w} = a^{\text{numeral } w}\)
by (fact power-even-abs)

end

57.4 Instance for \(\text{int}\)

lemma even-diff-iff:
  \(\text{even } (k - l) \iff \text{even } (k + l)\) for \(k l :: \text{int}\)
by (fact even-diff)

lemma even-abs-add-iff:
  \(\text{even } (|k| + l) \iff \text{even } (k + l)\) for \(k l :: \text{int}\)
by simp
lemma even-add-abs-iff:
  even (k + |l|) ↔ even (k + l) for k l :: int
by simp

lemma even-nat-iff:
  0 ≤ k ⇒ even (nat k) ↔ even k
by (simp add: even-of-nat-iff [of nat k, where ?a = int, symmetric])

context
  assumes SORT-CONSTRAINT('a::division-ring)
begin

lemma power-int-minus-left:
  power-int (-a :: 'a) n = (if even n then power-int a n else -power-int a n)
by (auto simp: power-int-def minus-one-power-iff even-nat-iff)

lemma power-int-minus-left-even [simp]: even n ⇒ power-int (-a :: 'a) n = power-int a n
by (simp add: power-int-minus-left)

lemma power-int-minus-left-odd [simp]: odd n ⇒ power-int (-a :: 'a) n = -power-int a n
by (simp add: power-int-minus-left)

lemma power-int-minus-left-distrib:
  NO-MATCH (-1) x ⇒ power-int (-a :: 'a) n = power-int (-1) n * power-int a n
by (simp add: power-int-minus-left)

lemma power-int-minus-one-minus: power-int (-1 :: 'a) (-n) = power-int (-1) n
by (simp add: power-int-minus-left)

lemma power-int-minus-one-minus-diff-commute: power-int (-1 :: 'a) (a - b) = power-int (-1) (b - a)
by (subst power-int-minus-one-minus [symmetric]) auto

lemma power-int-minus-one-mult-self [simp]:
  power-int (-1 :: 'a) m * power-int (-1) m = 1
by (simp add: power-int-minus-left)

lemma power-int-minus-one-mult-self' [simp]:
  power-int (-1 :: 'a) m * (power-int (-1) m * b) = b
by (simp add: power-int-minus-left)

end
57.5 Special case: euclidean rings structurally containing the natural numbers

```plaintext
class linordered-euclidean-semiring = discrete-linordered-semidom + unique-euclidean-semiring +
  assumes of-nat-div: of-nat (m div n) = of-nat m div of-nat n
  and division-segment-of-nat [simp]: division-segment (of-nat n) = 1
  and division-segment-euclidean-size [simp]: division-segment a * of-nat (euclidean-size a) = a
begin

lemma division-segment-eq-iff:
  a = b if division-segment a = division-segment b
  and euclidean-size a = euclidean-size b
using that division-segment-euclidean-size [of a] by simp

lemma euclidean-size-of-nat [simp]:
  euclidean-size (of-nat n) = n
proof –
  have division-segment (of-nat n) * of-nat (euclidean-size (of-nat n)) = of-nat n
    by (fact division-segment-euclidean-size)
  then show ?thesis by simp
qed

lemma of-nat-euclidean-size:
  of-nat (euclidean-size a) = a div division-segment a
proof –
  have of-nat (euclidean-size a) = division-segment a * of-nat (euclidean-size a)
    div division-segment a
    by (subst nonzero-mult-div-cancel-left) simp-all
  also have . . . = a div division-segment a
    by simp
  finally show ?thesis.
qed

lemma division-segment-1 [simp]:
  division-segment 1 = 1
using division-segment-of-nat [of 1] by simp

lemma division-segment-numeral [simp]:
  division-segment (numeral k) = 1
using division-segment-of-nat [of numeral k] by simp

lemma euclidean-size-1 [simp]:
  euclidean-size 1 = 1
using euclidean-size-of-nat [of 1] by simp

lemma euclidean-size-numeral [simp]:
  euclidean-size (numeral k) = numeral k
using euclidean-size-of-nat [of numeral k] by simp
```

THEORY “Parity”
lemma of-nat-dvd-iff:
    of-nat m dvd of-nat n ⟷ m dvd n
  (is ?P ⟷ ?Q)
proof (cases m = 0)
  case True
  then show ?thesis
    by simp
next
  case False
  show ?thesis
    proof
      assume ?Q
      then show ?P
        by auto
    next
      assume ?P
      with False have of-nat n = of-nat n div of-nat m * of-nat m
        by simp
      then have of-nat n = of-nat (n div m * m)
        by (simp add: of-nat-div)
      then have n = n div m * m
        by (simp only: of-nat-eq-iff)
      then have n = m * (n div m)
        by (simp add: ac-simps)
      then show ?Q ..
    qed
qed

lemma of-nat-mod:
    of-nat (m mod n) = of-nat m mod of-nat n
proof
  have of-nat m div of-nat n * of-nat n + of-nat m mod of-nat n = of-nat m
    by (simp add: div-mult-mod-eq)
  also have of-nat m = of-nat (m div n * n + m mod n)
    by simp
  finally show ?thesis
    by (simp only: of-nat-div of-nat-mult of-nat-add) simp
qed

lemma one-div-two-eq-zero [simp]:
    1 div 2 = 0
proof
  from of-nat-div [symmetric] have of-nat 1 div of-nat 2 = of-nat 0
    by (simp only:) simp
  then show ?thesis
    by simp
qed

lemma one-mod-2-pow-eq [simp]:
\[
1 \mod (2 \sim n) = \text{of-bool}\ (n > 0)
\]

**proof**
- have \(1 \mod (2 \sim n) = \text{of-nat}\ (1 \mod (2 \sim n))\)
  - using \text{of-nat-mod} [of 1 2 \sim n] by \text{simp}
- also have \(\ldots = \text{of-bool}\ (n > 0)\)
  - by \text{simp}
- finally show \(\text{thesis}\).

**qed**

**lemma** one-div-2-pow-eq [simp]:
\[
1 \div (2 \sim n) = \text{of-bool}\ (n = 0)
\]

**using** \text{div-mult-mod-eq} [of 1 2 \sim n] by auto

**lemma** div-mult2-eq':
\[
\langle a \div (\text{of-nat } m \times \text{of-nat } n) = a \div \text{of-nat } m \div \text{of-nat } n \rangle
\]

**proof** (cases \((m = 0 \lor n = 0)\))
- case True
  then show \(\text{thesis}\)
    - by auto
- case False
  then have \(\langle m > 0 \land n > 0 \rangle\)
    - by \text{simp-all}
  show \(\text{thesis}\)
    **proof** (cases \(\langle \text{of-nat } m \ast \text{of-nat } n \text{ dvd } a \rangle\))
      - case True
        then obtain \(b\) where \(\langle a = (\text{of-nat } m \ast \text{of-nat } n) \ast b \rangle\)
        ..
        then have \(\langle a = \text{of-nat } m \ast (\text{of-nat } n \ast b) \rangle\)
          - by (simp add: \text{ac-simps})
        then show \(\text{thesis}\)
          - by \text{simp}
- next
  define \(q\) where \(\langle q = a \div (\text{of-nat } m \ast \text{of-nat } n) \rangle\)
  define \(r\) where \(\langle r = a \mod (\text{of-nat } m \ast \text{of-nat } n) \rangle\)
  from \(\langle m > 0 \land n > 0 \rangle\) \(\langle \neg \text{of-nat } m \ast \text{of-nat } n \text{ dvd } a \rangle\) \(r\)-def** have** division-segment

\[
r = 1
\]

**using** division-segment-of-nat [of \(m \ast n\)] by (simp add: division-segment-mod)

**with** division-segment-euclidean-size [of \(r\)]

**have** \(\text{of-nat}\ (\text{euclidean-size } r) = r\)
  - by \text{simp}

**have** \(a \mod (\text{of-nat } m \ast \text{of-nat } n) \div (\text{of-nat } m \ast \text{of-nat } n) = 0\)
  - by \text{simp}

**with** \(\langle m > 0 \land n > 0 \rangle\) \(r\)-def** have** \(r \div (\text{of-nat } m \ast \text{of-nat } n) = 0\)
  - by \text{simp}

**with** \(\langle \text{of-nat}\ (\text{euclidean-size } r) = r \rangle\)
**have** \(\text{of-nat}\ (\text{euclidean-size } r) \div (\text{of-nat } m \ast \text{of-nat } n) = 0\)
  - by \text{simp}

**then have** \(\text{of-nat}\ (\text{euclidean-size } r \div (m \ast n)) = 0\)
by (simp add: of-nat-div)
then have of-nat (euclidean-size r div m div n) = 0
  by (simp add: div-mult2-eq)
with of-nat (euclidean-size r) = r
  have r div of-nat m div of-nat n = 0
  by (simp add: of-nat-div)
with m > 0, n > 0
  have q = (r div of-nat m + q * of-nat n * of-nat m div of-nat m) div of-nat n
  by simp
moreover have \(<a = q * (of-nat m * of-nat n) + r\>
  by (simp add: q-def r-def div-mult-mod-eq)
ultimately show \(<a div (of-nat m * of-nat n) = a div of-nat m div of-nat n\>
  using q-def \[\text{symmetric}\] div-plus-div-distrib-dvd-right [of \(<of-nat m \cdot q * (of-nat m * of-nat n)\) \(r\)]
  by (simp add: ac-simps)
qed

lemma mod-mult2-eq':
a mod (of-nat m * of-nat n) = of-nat m * (a div of-nat m mod of-nat n) + a mod of-nat m
proof
  have a div (of-nat m * of-nat n) * (of-nat m * of-nat n) + a mod (of-nat m * of-nat n) = a div of-nat m div of-nat n * of-nat n * of-nat m + (a div of-nat m mod of-nat n * of-nat m + a mod of-nat m)
    by (simp add: combine-common-factor div-mult-mod-eq)
  moreover have a div of-nat m div of-nat n * of-nat n * of-nat m = of-nat n * of-nat m * (a div of-nat m div of-nat n)
    by (simp add: ac-simps)
  ultimately show \(?thesis\)
    by (simp add: div-mult2-eq' mult-commute)
qed

lemma div-mult2-numeral-eq:
a div numeral k div numeral l = a div numeral (k * l) \(\text{is } ?A = ?B\)
proof
  have \(<A = a \text{ div of-nat } (\text{numeral } k) \text{ div of-nat } (\text{numeral } l)\>
    by simp
  also have \(\ldots = a \text{ div of-nat } (\text{numeral } k) \text{ div of-nat } (\text{numeral } l)\)
    by (fact div-mult2-eq' [symmetric])
  also have \(\ldots = ?B\)
    by simp
  finally show \(?thesis\).
qed

lemma numeral-Bit0-div-2:
numeral (num.Bit0 n) div 2 = numeral n
proof
  have numeral (num.Bit0 n) = numeral n + numeral n
    by (simp only: numeral.simps)
also have \( \ldots = \text{numeral } n \ast 2 \)
  by (simp add: mult-2-right)
finally have \( \text{numeral } (\text{num.Bit0 } n) \div 2 = \text{numeral } n \ast 2 \div 2 \)
  by simp
also have \( \ldots = \text{numeral } n \)
  by (rule nonzero-mult-div-cancel-right) simp
finally show \(?\text{thesis}\.\)
qed

lemma numeral-Bit1-div-2:
\( \text{numeral } (\text{num.Bit1 } n) \div 2 = \text{numeral } n \)
proof –
  have \( \text{numeral } (\text{num.Bit1 } n) = \text{numeral } n + \text{numeral } n + 1 \)
    by (simp only: numeral.simps)
  also have \( \ldots = \text{numeral } n \ast 2 + 1 \)
    by (simp add: mult-2-right)
finally have \( \text{numeral } (\text{num.Bit1 } n) \div 2 = (\text{numeral } n \ast 2 + 1) \div 2 \)
  by simp
also have \( \ldots = \text{numeral } n \ast 2 \div 2 + 1 \div 2 \)
  using dvd-triv-right by (rule div-plus-div-distrib-dvd-left)
also have \( \ldots = \text{numeral } n \ast 2 \div 2 \)
  by simp
also have \( \ldots = \text{numeral } n \)
  by (rule nonzero-mult-div-cancel-right) simp
finally show ?thesis .
qed

lemma exp-mod-exp:
\( 2 ^ m \mod 2 ^ n = \text{ofoob } (m < n) \ast 2 ^ m \)
proof –
  have \( 2 ^ n \mod 2 ^ m = 2 ^ m \mod 2 ^ m = \text{ofoob } (m < n) \ast 2 ^ m \) (is ?lhs = ?rhs)
    by (auto simp add: linorder-class.not-less monoid-mult-class.power-add dest!: le-Suc-ex)
  then have \( \text{ofoob } ?\text{lhs} = \text{ofoob } ?\text{rhs} \)
    by simp
  then show ?thesis
    by (simp add: ofoob-mod)
qed

lemma mask-mod-exp:
\( (2 ^ n - 1) \mod 2 ^ m = 2 ^ \min m n - 1 \)
proof –
  have \( (2 ^ n - 1) \mod 2 ^ m = 2 ^ \min m n - (1::nat) \) (is ?lhs = ?rhs)
    proof (cases \( n \leq m \))
      case True
      then show ?thesis
        by (simp add: Suc-le-lessD)
    next
      case False
then have \langle m < n \rangle
  by simp

then obtain \( q \) where \( n = \text{Suc} \( q \) + m \)
  by (auto dest: less-imp-Suc-add)
then have \langle \text{min} m n = m \rangle
  by simp
moreover have \langle (2::nat) \sim m \leq 2 \sim 2 \sim q \sim 2 \sim m \rangle
  using mult-le-monot1 [of 1 \langle 2 \sim 2 \sim q \rangle \langle 2 \sim m \rangle] by simp
with \( n \) have \langle 2 \sim n - 1 = (2 \sim \text{Suc} q - 1) \sim 2 \sim m + (2 \sim m - (1::nat)) \rangle
  by (simp add: monoid-mult-class.power-add algebra-simps)
ultimately show \?thesis
  by (simp only: euclidean-semiring-cancel-class.mod-mult-self3) simp
qed
then have \langle \text{of-nat} ?lhs = \text{of-nat} ?rhs \rangle
  by simp
then show \?thesis
  by (simp add: of-nat-mod of-nat-diff)
qed

lemma of-bool-half-eq-0 [simp]:
\langle \text{of-bool} b \div 2 = 0 \rangle
  by simp

lemma of-nat-mod-double:
\langle \text{of-nat} n \mod (2 \ast \text{of-nat} m) = \text{of-nat} n \mod \text{of-nat} m \lor \text{of-nat} n \mod (2 \ast \text{of-nat} m) = \text{of-nat} n \mod \text{of-nat} m + \text{of-nat} m \rangle
end

instance nat :: linordered-euclidean-semiring
  by standard simp-all

instance int :: linordered-euclidean-semiring
  by standard (auto simp add: divide-int-def division-segment-int-def elim: contra-pos-np)

context linordered-euclidean-semiring
begin
subclass semiring-parity
proof
  show \langle a \mod 2 = \text{of-bool} (\sim 2 \text{ dvd} a) \rangle for a
  proof (cases \langle 2 \text{ dvd} a \rangle)
    case True
    then show \?thesis
      by (simp add: dvd-eq-mod-eq-0)
  next
case False


have eucl: euclidean-size (a mod 2) = 1
proof (rule Orderings.order-antisym)
  show euclidean-size (a mod 2) ≤ 1
    using mod-size-less [of 2 a] by simp
  show 1 ≤ euclidean-size (a mod 2)
    using (~ 2 dvd a) by (simp add: Suc-le-eq dvd-eq-mod-eq-0)
qed

from (∼ 2 dvd a) have ∼ of-nat 2 dvd division-segment a * of-nat (euclidean-size a)
  by simp
then have ∼ of-nat 2 dvd of-nat (euclidean-size a)
  by (auto simp only: dvd-mul-unit-iff is-unit-division-segment)
then have ∼ 2 dvd euclidean-size a
  using of-nat-dvd-iff [of 2] by simp
then have euclidean-size a mod 2 = 1
  by (simp add: semidom-modulo-class.dvd-eq-mod-eq-0)
then have of-nat (euclidean-size a mod 2) = of-nat 1
  by simp
then have of-nat (euclidean-size a) mod 2 = 1
  by (simp add: of-nat-mod)
from (∼ 2 dvd a) eucl
have a mod 2 = 1
  by (auto intro: division-segment-eq-iff simp add: division-segment-mod)
with (∼ 2 dvd a) show ?thesis
  by simp
qed

show (∼ is-unit 2)
proof
  assume (∼ is-unit 2)
  then have (∼ of-nat 2 dvd of-nat 1)
    by simp
  then have (∼ is-unit (2::nat))
    by (simp only: of-nat-dvd-iff)
  then show False
    by simp
qed

show (1 + a) div 2 = a div 2 if (2 dvd a) for a
  using that by auto
qed

lemma even-succ-div-two [simp]:
  even a → (a + 1) div 2 = a div 2
by (cases a = 0) (auto elim!: evenE dest: mult-not-zero)

lemma odd-succ-div-two [simp]:
  odd a → (a + 1) div 2 = a div 2 + 1
by (auto elim!: oddE simp add: add.assoc)

lemma even-two-times-div-two:
even $a \Rightarrow 2 \ast (a \div 2) = a$
by (fact dvd-mult-div-cancel)

lemma odd-two-times-div-two-succ [simp]:
$odd a \Rightarrow 2 \ast (a \div 2) + 1 = a$
using mult-div-mod-eq [of 2 a]
by (simp add: even-iff-mod-2-eq-zero)

lemma coprime-left-2-iff-odd [simp]:
coprime 2 $a \iff add a$
proof
assume odd $a$
show coprime 2 $a$
proof (rule coprimeI)
  fix $b$
  assume $b \dvd 2$ $b \dvd a$
  then have $b \dvd a \mod 2$
  by (auto intro: dvd-mod)
  with $\langle odd a \rangle$ show is-unit $b$
  by (simp add: mod-2-eq-odd)
qed
next
assume coprime 2 $a$
show odd $a$
proof (rule notI)
  assume even $a$
  then obtain $b$ where $a = 2 \ast b$ ..
  with $\langle coprime 2 a \rangle$ have coprime 2 $(2 \ast b)$
  by simp
  moreover have $\neg$ coprime 2 $(2 \ast b)$
  by (rule not-coprimeI [of 2]) simp-all
  ultimately show False
  by blast
qed
qed

lemma coprime-right-2-iff-odd [simp]:
coprime $a 2 \iff odd a$
using coprime-left-2-iff-odd [of $a$] by (simp add: ac-simps)
end

lemma minus-1-mod-2-eq [simp]:
$(-1 \mod 2 = (1::int))$
by (simp add: mod-2-eq-odd)

lemma minus-1-div-2-eq [simp]:
$(-1 \div 2 = - (1::int))$
proof –
from \texttt{div-mult-mod-eq \{of \ -1 :: int \}}
have \(-1 \div 2 * 2 = -1 * (2 :: int)\)
using \texttt{add-implies-diff} by \texttt{fastforce}
then show \(?thesis\)
using \texttt{mult-right-cancel \{of 2 - 1 \ div 2 - (1 :: int)\}} by \texttt{simp}
qed

context \texttt{linordered-euclidean-semiring}
begin

lemma \texttt{even-decr-exp-div-exp-iff}:
\langle even \((2 ^ m - 1) \div 2 ^ n) \longleftrightarrow m \leq n \rangle

proof

have \langle even \((2 ^ m - 1) \div 2 ^ n) \longleftrightarrow even \((\text{of-nat})((2 ^ m - Suc 0) \div 2 ^ n)\rangle

by \texttt{(simp only: of-nat-div)} (\texttt{simp add: of-nat-diff})

also have \langle \ldots \longleftrightarrow even \((2 ^ m - Suc 0) \div 2 ^ n)\rangle

by \texttt{simp}

also have \langle \ldots \longleftrightarrow m \leq n \rangle

proof (cases \langle m \leq n \rangle)

case True
then show \(?thesis\)
by \texttt{(simp add: Suc-le-lessD)}

case False
then obtain \(r\) where \(r\): \langle m = n + Suc r)\rangle

using \texttt{less-imp-Suc-add} by \texttt{fastforce}

from \(r\) have \langle \{q. q < m) \cap \{q. 2 ^ n dvd (2 :: nat) ^ q) = \{q. n \leq q \land q < m)\rangle

by \texttt{(auto simp add: dvd-power-iff-le)}

moreover from \(r\) have \langle \{q. q < m) \cap \{q. \neg 2 ^ n dvd (2 :: nat) ^ q) = \{q. q < n)\rangle

by \texttt{(auto simp add: dvd-power-iff-le)}

moreover from False have \langle \{q. n \leq q \land q < m \land q \leq n\) = \{n)\rangle

by \texttt{auto}

then have \langle odd \((\sum a \in \{q. n \leq q \land q < m\}. 2 ^ n dvd (2 :: nat) ^ n) + sum \rangle

\langle 2 \rangle \{q. q < n) \ div 2 ^ n)\rangle

by \texttt{(simp-all add: euclidean-semiring-cancel-class.power-diff-power-eq semiring-parity-class.even-sum-iff)}

\texttt{linorder-class.not-less mask-eq-sum-exp-nat [symmetric]}

ultimately have \langle odd \((\sum (2 :: nat)) \{q. q < m) \ div 2 ^ n)\rangle

by \texttt{(subst euclidean-semiring-cancel-class.sum-div-partition) simp-all}

with \texttt{False show \(?thesis\)}

by \texttt{(simp add: mask-eq-sum-exp-nat)}

qed

finally show \(?thesis\).

qed

end
57.6  Generic symbolic computations

The following type class contains everything necessary to formulate a division algorithm in ring structures with numerals, restricted to its positive segments.

```
class linordered-euclidean-semiring-division = linordered-euclidean-semiring +
  fixes divmod :: 'a ⇒ 'a ⇒ 'a × 'a — These are conceptually definitions
  and divmod-step :: 'a ⇒ 'a × 'a ⇒ 'a × 'a — These are conceptually definitions
  but force generated code to be monomorphic wrt. particular instances of this class
  which yields a significant speedup.

assumes divmod-def: divmod m n = (numeral m div numeral n, numeral m mod numeral n)
  and divmod-step-def [simp]: divmod-step l (q, r) =
    (if euclidean-size l ≤ euclidean-size r then (2 * q + 1, r - l)
     else (2 * q, r)) — This is a formulation of one step (referring to one
digit position) in school-method division: compare the dividend at the current
digit position with the remainder from previous division steps and evaluate accordingly.
```

```
begin

lemma fst-divmod:
  fst (divmod m n) = numeral m div numeral n
by (simp add: divmod-def)

lemma snd-divmod:
  snd (divmod m n) = numeral m mod numeral n
by (simp add: divmod-def)

Following a formulation of school-method division. If the divisor is smaller
than the dividend, terminate. If not, shift the dividend to the right until
termination occurs and then reiterate single division steps in the opposite
direction.

lemma divmod-divmod-step:
  divmod m n = (if m < n then (0, numeral m)
   else divmod-step (numeral n) (divmod m (Num.Bit0 n)))
proof (cases (m < n))
  case True
  then show ?thesis
    by (simp add: prod-eq-iff fst-divmod snd-divmod flip: of-nat-numeral of-nat-div
         of-nat-mod)
next
  case False
  define r s t where r = (numeral m :: nat) s = (numeral n :: nat) t = 2 * s
  then have *: numeral m = of-nat r numeral n = of-nat s numeral (Num.Bit0 n) = of-nat t
    and ¬ s ≤ r mod s
    by (simp-all add: linorder-class.not-le)
  have t: (2 * (r div t) = r div s - r div s mod 2)
```

```
\( r \ mod \ t = s \ast (r \ div \ s \ mod \ 2) + r \ mod \ s \)

by (simp add: Rings.minus-mod-eq-mul-div Groups.mult.commute [of 2] Euclidean-Rings.div-mult2-eq \( t = 2 * s \))

(use mod-mult2-eq [of \( r \ s \)] in (simp add: ac-simps \( t = 2 * s \))

have \( rs \); \( r \ div \ s \ mod \ 2 = 0 \lor r \ div \ s \ mod \ 2 = \text{Suc 0} \)

by auto

from \( \neg s \leq r \ mod \ s \) have \( s \leq r \ mod \ t \implies r \ div \ s = \text{Suc} (2 * (r \ div \ t)) \land r \ mod \ s = r \ mod \ t - s \)

using \( rs \)

by (auto simp add: \( t \))

moreover have \( r \ mod \ t < s \implies r \ div \ s = 2 \ast (r \ div \ t) \land r \ mod \ s = r \ mod \ t \)

using \( rs \)

by (auto simp add: \( t \))

ultimately show \(?\)thesis

by (simp add: divmod-def prod-eq-iff split-def Let-def
not-less mod-eq-0-iff-dvd Rings.mod-eq-0-iff-dvd False not-le *

qed

The division rewrite proper – first, trivial results involving \( 1 \)

lemma \( \text{divmod-trivial} \) [simp]:
\( \text{divmod} \ m \ \text{Num.One} = (\text{numeral} \ m, 0) \)
\( \text{divmod} \ \text{num.One} \ (\text{num.Bit0} \ n) = (0, \text{Numeral1}) \)
\( \text{divmod} \ \text{num.One} \ (\text{num.Bit1} \ n) = (0, \text{Numeral1}) \)

using \( \text{divmod-divmod-step} \) [of \( \text{Num.One} \)] by (simp-all add: divmod-def)

Division by an even number is a right-shift

lemma \( \text{divmod-cancel} \) [simp]:
\( \langle \text{divmod} \ (\text{Num.Bit0} \ m) \ (\text{Num.Bit0} \ n) = (\text{case divmod} \ m \ n \ of \ (q, r) \Rightarrow (q, 2 \ast r)) \rangle \) (is \(?P\))
\( \langle \text{divmod} \ (\text{Num.Bit1} \ m) \ (\text{Num.Bit0} \ n) = (\text{case divmod} \ m \ n \ of \ (q, r) \Rightarrow (q, 2 \ast r + 1)) \rangle \) (is \(?Q\))

proof –

define \( r \ s \) where \( \langle r = (\text{numeral} \ m :: \text{nat}) \rangle \ \langle s = (\text{numeral} \ n :: \text{nat}) \rangle \)

then have \(* \): \( \langle \text{numeral} \ m = \text{of-nat} \ r \rangle \ \langle \text{numeral} \ n = \text{of-nat} \ s \rangle \)
\( \langle \text{numeral} \ (\text{num.Bit0} \ m) = \text{of-nat} \ (2 \ast r) \rangle \ \langle \text{numeral} \ (\text{num.Bit0} \ n) = \text{of-nat} \ (2 \ast s) \rangle \)

by simp-all

have \( ** \): \( \langle \text{Suc} \ (2 \ast r) \ \text{div} \ 2 = r \rangle \)

by simp

show \( ?P \) and \( ?Q \)

by (simp-all add: divmod-def *)

(simp-all flip: of-nat-numeral of-nat-div of-nat-mod of-nat-mult add.commute [of \( 1 \] of-nat-Suc
THEORY “Parity”


divmod-steps [simp]:
divmod (num.Bit0 m) (num.Bit1 n) =
  (if m ≤ n then (0, numeral (num.Bit0 m))
   else divmod-step (numeral (num.Bit1 n))
    (divmod (num.Bit0 m)
     (num.Bit0 (num.Bit1 n))))
divmod (num.Bit1 m) (num.Bit1 n) =
  (if m < n then (0, numeral (num.Bit1 m))
   else divmod-step (numeral (num.Bit1 n))
    (divmod (num.Bit1 m)
     (num.Bit0 (num.Bit1 n))))
by (simp-all add: divmod-divmod-step)

lemmas divmod-algorithm-code = divmod-trivial divmod-cancel divmod-steps

Special case: divisibility

definition divides-aux :: 'a × 'a ⇒ bool
where
  divides-aux qr =→ snd qr = 0

lemma divides-aux-eq [simp]:
divides-aux (q, r) =→ r = 0
by (simp add: divides-aux-def)

lemma dvd-numeral-simp [simp]:
numeral m dvd numeral n =→ divides-aux (divmod n m)
by (simp add: divmod-def mod-iff-dvd)

Generic computation of quotient and remainder

lemma numeral-div-numeral [simp]:
umeral k div numeral l = fst (divmod k l)
by (simp add: fst-divmod)

lemma numeral-mod-numeral [simp]:
umeral k mod numeral l = snd (divmod k l)
by (simp add: snd-divmod)

lemma one-div-numeral [simp]:
1 div numeral n = fst (divmod num.One n)
by (simp add: fst-divmod)

lemma one-mod-numeral [simp]:
1 mod numeral n = snd (divmod num.One n)
by (simp add: snd-divmod)
THEORY "Parity"

end

instantiation nat :: linordered-euclidean-semiring-division
begin

definition divmod-nat :: num ⇒ num ⇒ nat × nat
where
  divmod'-nat-def: divmod-nat m n = (numeral m div numeral n, numeral m mod numeral n)

definition divmod-step-nat :: nat ⇒ nat × nat ⇒ nat × nat
where
  divmod-step-nat l qr = (let (q, r) = qr
      in if r ≥ l then (2 * q + 1, r - l)
      else (2 * q, r))

instance
  by standard (simp-all add: divmod'-nat-def divmod-step-nat-def)

end

declare divmod-algorithm-code [where $a = nat, code]

lemma Suc-0-div-numeral [simp]:
  Suc 0 div numeral Num.One = 1
  Suc 0 div numeral (Num.Bit0 n) = 0
  Suc 0 div numeral (Num.Bit1 n) = 0
by simp

lemma Suc-0-mod-numeral [simp]:
  Suc 0 mod numeral Num.One = 0
  Suc 0 mod numeral (Num.Bit0 n) = 1
  Suc 0 mod numeral (Num.Bit1 n) = 1
by simp

instantiation int :: linordered-euclidean-semiring-division
begin

definition divmod-int :: num ⇒ num ⇒ int × int
where
  divmod-int m n = (numeral m div numeral n, numeral m mod numeral n)

definition divmod-step-int :: int ⇒ int × int ⇒ int × int
where
  divmod-step-int l qr = (let (q, r) = qr
      in if |l| ≤ |r| then (2 * q + 1, r - l)
      else (2 * q, r))
instance
  by standard (auto simp add: divmod-int-def divmod-step-int-def)

end

declare divmod-algorithm-code [where 'a = int, code]

context
begin

qualified definition adjust-div :: int × int ⇒ int
where
adjust-div qr = (let (q, r) = qr in q + of-bool (r ≠ 0))

qualified lemma adjust-div-eq [simp, code]:
  adjust-div (q, r) = q + of-bool (r ≠ 0)
  by (simp add: adjust-div-def)

qualified definition adjust-mod :: num ⇒ int ⇒ int
where
[simp]: adjust-mod l r = (if r = 0 then 0 else numeral l − r)

lemma minus-numeral-div-numeral [simp]:
  − numeral m div numeral n = − (adjust-div (divmod m n) :: int)

proof −
  have int (fst (divmod m n)) = fst (divmod m n)
    by (simp only: fst-divmod divide-int-def) auto
  then show ?thesis
    by (auto simp add: split-def Let-def adjust-div-def divides-aux-def divide-int-def)
  qed

lemma minus-numeral-mod-numeral [simp]:
  − numeral m mod numeral n = adjust-mod n (snd (divmod m n) :: int)

proof (cases snd (divmod m n) = (0::int))
  case True
  then show ?thesis
    by (simp add: mod-eq-0-iff-dvd divides-aux-def)
next
  case False
  then have int (snd (divmod m n)) = snd (divmod m n) if snd (divmod m n) ≠ (0::int)
    by (simp only: snd-divmod modulo-int-def) auto
  then show ?thesis
    by (simp add: divides-aux-def adjust-div-def divides-aux-def modulo-int-def)
  qed

lemma numeral-div-minus-numeral [simp]:
  numeral m div − numeral n = − (adjust-div (divmod m n) :: int)
proof –
have int (fst (divmod m n)) = fst (divmod m n)
  by (simp only: fst-divmod divide-int-def) auto
then show ?thesis
  by (auto simp add: split-def Let-def adjust-div-def divides-aux-def divide-int-def)
qed

lemma numeral-mod-minus-numeral [simp]:
umeral m mod − numeral n = − adjust-mod n (snd (divmod m n) :: int)
proof (cases snd (divmod m n)) = (0::int))
case True
  then show ?thesis
    by (simp add: mod-eq-0-iff-dvd divides-aux-def)
next
case False
  then have int (snd (divmod m n)) = snd (divmod m n)
    if snd (divmod m n) ≠ (0::int)
    by (simp only: snd-divmod modulo-int-def) auto
  then show ?thesis
    by (simp add: divides-aux-def adjust-div-def)
      (simp add: divides-aux-def modulo-int-def)
qed

lemma minus-one-div-numeral [simp]:
− 1 div numeral n = − (adjust-div (divmod Num.One n) :: int)
using minus-numeral-div-numeral [of Num.One n] by simp

lemma minus-one-mod-numeral [simp]:
− 1 mod numeral n = adjust-mod n (snd (divmod Num.One n) :: int)
using minus-numeral-mod-numeral [of Num.One n] by simp

lemma one-div-minus-numeral [simp]:
1 div − numeral n = − (adjust-div (divmod Num.One n) :: int)
using numeral-div-minus-numeral [of Num.One n] by simp

lemma one-mod-minus-numeral [simp]:
1 mod − numeral n = − (adjust-mod n (snd (divmod Num.One n) :: int)
using numeral-mod-minus-numeral [of Num.One n] by simp

lemma [code]:
fixes k :: int
shows k div 0 = 0
  k mod 0 = k
  0 div k = 0
  0 mod k = 0
  k div Int.Pos Num.One = k
  k mod Int.Pos Num.One = 0
  k div Int.Neg Num.One = − k
THEORY “Parity”

\[ k \mod \text{Int. Neg Num.One} = 0 \]
\[ \text{Int. Pos m div Int. Pos n} = (\text{fst (divmod m n :: int)} \] \[ \text{Int. Pos m mod Int. Pos n} = (\text{snd (divmod m n :: int)} \] \[ \text{Int. Neg m div Int. Pos n} = \text{adjust-div (divmod m n :: int)} \]
\[ \text{Int. Pos m mod Int. Neg n} = \text{adjust-mod n (snd (divmod m n :: int)} \]
\[ \text{Int. Neg m div Int. Neg n} = (\text{fst (divmod m n :: int)} \] \[ \text{Int. Neg m mod Int. Neg n} = (\text{snd (divmod m n :: int)} \]

by simp-all

end

lemma divmod-BitM-2-eq [simp]:
\[ \text{divmod (Num.BitM m) (Num.Bit0 Num.One)} = (\text{numeral m - 1, (1 :: int)})) \]
by (cases m) simp-all

57.6.1 Computation by simplification

lemma euclidean-size-nat-less-eq-iff:
\[ \text{euclidean-size m} \leq \text{euclidean-size n} \leftrightarrow m \leq n \]
for m n :: nat
by simp

lemma euclidean-size-int-less-eq-iff:
\[ \text{euclidean-size k} \leq \text{euclidean-size l} \leftrightarrow |k| \leq |l| \]
for k l :: int
by auto

simproc-setup numeral-divmod
\[ (0 \div 0 :: 'a :: linordered-euclidean-semiring-division | 0 \mod 0 :: 'a :: linordered-euclidean-semiring-division | \]
\[ 0 \div 1 :: 'a :: linordered-euclidean-semiring-division | 0 \mod 1 :: 'a :: linordered-euclidean-semiring-division | \]
\[ 0 \div -1 :: \text{int} | 0 \mod -1 :: \text{int} | \]
\[ 0 \div \text{numeral b :: 'a :: linordered-euclidean-semiring-division} | 0 \mod \text{numeral b :: 'a :: linordered-euclidean-semiring-division} | \]
\[ 0 \div -\text{numeral b :: int} | 0 \mod -\text{numeral b :: int} | \]
\[ 1 \div 0 :: 'a :: linordered-euclidean-semiring-division | 1 \mod 0 :: 'a :: linordered-euclidean-semiring-division | \]
\[ 1 \div 1 :: 'a :: linordered-euclidean-semiring-division | 1 \mod 1 :: 'a :: linordered-euclidean-semiring-division | \]
\[ 1 \div -1 :: \text{int} | 1 \mod -1 :: \text{int} | \]
\[ 1 \div \text{numeral b :: 'a :: linordered-euclidean-semiring-division} | 1 \mod \text{numeral b :: 'a :: linordered-euclidean-semiring-division} | \]
\[ 1 \div -\text{numeral b :: int} | 1 \mod -\text{numeral b :: int} | \]
\[ \text{numeral a \div 0 :: 'a :: linordered-euclidean-semiring-division} | \text{numeral a mod} \]
0 : 'a :: linordered-euclidean-semiring-division | numeral a div 1 :: 'a :: linordered-euclidean-semiring-division | numeral a mod
1 :: 'a :: linordered-euclidean-semiring-division | numeral a div -1 :: int | numeral a mod -1 :: int |
numeral a div numeral b :: 'a :: linordered-euclidean-semiring-division | numeral a mod numeral b :: 'a :: linordered-euclidean-semiring-division |
numeral a div - numeral b :: int | numeral a mod - numeral b :: int |
- numeral a div 0 :: int | - numeral a mod 0 :: int |
- numeral a div 1 :: int | - numeral a mod 1 :: int |
- numeral a div -1 :: int | - numeral a mod -1 :: int |
- numeral a div numeral b :: int | - numeral a mod numeral b :: int |
- numeral a div - numeral b :: int | - numeral a mod - numeral b :: int |
- numeral a div numeral b :: int | - numeral a mod - numeral b :: int |
- numeral a div - numeral b :: int | - numeral a mod - numeral b :: int | = |

let
val if-cong = the (Code.get-case-cong theory const-name (If));
fun successful-rewrite ctxt ct =
let
val thm = Simplifier.rewrite ctxt ct
in if Thm.is-reflexive thm then NONE else SOME thm end;
val simps = @
{ thms div-0 mod-0 div-by-0 mod-by-0 div-by-1 mod-by-1
one-div-numeral one-mod-numeral minus-one-div-numeral minus-one-mod-numeral
one-div-minus-numeral one-mod-minus-numeral
numeral-div-numeral numeral-mod-numeral minus-one-div-numeral minus-one-mod-numeral
minus-numeral-div-numeral minus-numeral-mod-numeral
numeral-div-minus-numeral numeral-mod-minus-numeral
div-minus-minus mod-minus-minus Parity.adjust-div-eq of-bool-eq one-neq-zero
numeral-neq-zero neg-equal-0-iff-equal arith-simps arith-special divmod-trivial
divmod-cancel divmod-steps divmod-step-def fst-conv snd-conv numeral-One
case-prod-beta rel-simps Parity.adjust-mod-def div-minus1-right mod-minus1-right
minus-minus numeral-times-numeral mult-zero-right mult-1-right
euclidean-size-nat-less-eq-iff euclidean-size-int-less-eq-iff diff-nat-numeral nat-numeral
} @
{ lemma 0 = 0 ⇔ True by simp ]];
val simpset =
HOL-ss |> Simplifier.simpset-map context
(Simplifier.add-cong if-cong #> fold Simplifier.add-simp simps);
in K (fn ctxt => successful-rewrite (Simplifier.put-simpset simpset ctxt)) end
do — There is space for improvement here: the calculation itself could be carried out
outside the logic, and a generic simproc (simplifier setup) for generic calculation
would be helpful.

57.7 Computing congruences modulo $2^q$

context linordered-euclidean-semiring-division
begin

lemma cong-exp-iff-simps:
numeral n mod numeral Num.One = 0
⇔ True
numeral (Num.Bit0 n) mod numeral (Num.Bit0 q) = 0
⇔ numeral n mod numeral q = 0
numeral (Num.Bit1 n) mod numeral (Num.Bit0 q) = 0
  ←→ False
numeral m mod numeral Num.One = (numeral n mod numeral Num.One)
  ←→ True
numeral Num.One mod numeral (Num.Bit0 q) = (numeral Num.One mod numeral (Num.Bit0 q))
  ←→ True
numeral Num.One mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n) mod numeral (Num.Bit0 q))
  ←→ False
numeral Num.One mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
  ←→ False
numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n) mod numeral (Num.Bit0 q))
  ←→ numeral m mod numeral q = (numeral n mod numeral q)
numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
  ←→ False
numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral Num.One mod numeral (Num.Bit0 q))
  ←→ (numeral m mod numeral q) = 0
numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n) mod numeral (Num.Bit0 q))
  ←→ False
numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
  ←→ numeral m mod numeral q = (numeral n mod numeral q)
by (auto simp add: case-prod-beta dest: arg-cong |of - even)

end

code-identifier code-module Parity ↦ (SML) Arith and (OCaml) Arith and (Haskell) Arith

lemmas even-of-nat = even-of-nat-iff

end

58 Combination and Cancellation Simprocs for Numeral Expressions

theory Numeral-Simprocs
imports Parity
begin

ML-file (~~/src/Provers/Arith/assoc-fold.ML)
ML-file (~~/src/Provers/Arith/cancel-numerals.ML)
ML-file (~~/src/Provers/Arith/combine-numerals.ML)
ML-file (~~/src/Provers/Arith/cancel-numeral-factor.ML)
ML-file (~~/src/Provers/Arith/extract-common-term.ML)

lemmas semiring-norm =
   Let-def arith-simps diff-nat-numeral rel-simps
   if-False if-True
   add-Suc add-numeral-left add-neg-numeral-left mult-numeral-left
   numeral-One [symmetric] uminus-numeral-One [symmetric] Suc-eq-plus1
   eq-numeral-iff-iszero not-iszero-Numeral1

For combine-numerals

lemma left-add-mult-distrib: i*u + (j*u + k) = (i+j)*u + (k::nat)
   by (simp add: add-mult-distrib)

For cancel-numerals

lemma nat-diff-add-eq1:
   j <= (i::nat) ===> ((i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)
   by (simp split: nat-diff-split add: add-mult-distrib)

lemma nat-diff-add-eq2:
   i <= (j::nat) ===> ((i*u + m) - (j*u + n)) = (m - ((j-i)*u + n))
   by (simp split: nat-diff-split add: add-mult-distrib)

lemma nat-eq-add-iff1:
   j <= (i::nat) ===> (i*u + m = j*u + n) = ((i-j)*u + m = n)
   by (auto split: nat-diff-split simp add: add-mult-distrib)

lemma nat-eq-add-iff2:
   i <= (j::nat) ===> (i*u + m = j*u + n) = (m = (j-i)*u + n)
   by (auto split: nat-diff-split simp add: add-mult-distrib)

lemma nat-less-add-iff1:
   j <= (i::nat) ===> (i*u + m < j*u + n) = ((i-j)*u + m < n)
   by (auto split: nat-diff-split simp add: add-mult-distrib)

lemma nat-less-add-iff2:
   i <= (j::nat) ===> (i*u + m < j*u + n) = (m < (j-i)*u + n)
   by (auto split: nat-diff-split simp add: add-mult-distrib)

lemma nat-le-add-iff1:
   j <= (i::nat) ===> (i*u + m <= j*u + n) = ((i-j)*u + m <= n)
   by (auto split: nat-diff-split simp add: add-mult-distrib)
lemma nat-le-add-iff2:
\[ i \leq (j::nat) \iff (i \cdot u + m \leq j \cdot u + n) = (m \leq (j - i) \cdot u + n) \]
by (auto split: nat-diff-split simp add: add-mult-distrib)

For cancel-numeral-factors
lemma nat-mult-le-cancel1: (0::nat) < k ==> (k \cdot m \leq k \cdot n) = (m \leq n)
by auto

lemma nat-mult-less-cancel1: (0::nat) < k ==> (k \cdot m < k \cdot n) = (m < n)
by auto

lemma nat-mult-eq-cancel1: (0::nat) < k ==> (k \cdot m = k \cdot n) = (m = n)
by auto

lemma nat-mult-div-cancel1: (0::nat) < k ==> (k \cdot m) div (k \cdot n) = (m div n)
by auto

lemma nat-mult-dvd-cancel-disj[simp]:
(\(k \cdot m\)) dvd \((k \cdot n)\) = (\(k = 0 \lor m\) dvd \((n::nat)\))
by (auto simp: dvd-eq-mod-eq-0 mod-mult-mult1)

lemma nat-mult-dvd-cancel1: 0 < k ==> (k \cdot m) dvd (k \cdot n::nat) = (m dvd n)
by (auto)

For cancel-factor
lemmas nat-mult-le-cancel-disj = mult-le-cancel1

lemmas nat-mult-less-cancel-disj = mult-less-cancel1

lemma nat-mult-eq-cancel-disj:
fixes k m n :: nat
shows \(k \cdot m = k \cdot n \iff k = 0 \lor m = n\)
by (fact mult-cancel-left)

lemma nat-mult-div-cancel-disj:
fixes k m n :: nat
shows \((k \cdot m)\) div \((k \cdot n)\) = (if \(k = 0\) then 0 else \(m\) div \(n\))
by (fact div-mult-mult1-if)

lemma numeral-times-minus-swap:
fixes x :: 'a::comm-ring-1
shows numeral w * -x = x * - numeral w
by (simp add: ac-simps)

ML-file "Tools/numeral-simprocs.ML"

simproc-setup semiring-assoc-fold
\((\langle a::'a::comm-semiring-1-cancel\rangle * b)\) =
"K Numeral-Simprocs.assoc-fold"
simproc-setup int-combine-numerals
\((i::\text{comm-ring-1}) + j \mid (i::\text{comm-ring-1}) - j) = \langle K \text{ Numeral-Simprocs.combine-numerals} \rangle\)

simproc-setup field-combine-numerals
\(((i::\text{field,ring-char-0}) + j \mid (i::\text{field,ring-char-0}) - j) = \langle K \text{ Numeral-Simprocs.field-combine-numerals} \rangle\)

simproc-setup inteq-cancel-numerals
\((l::\text{comm-ring-1}) + m = n \mid \langle \langle l::\text{comm-ring-1} \rangle + m = n \mid \langle l::\text{comm-ring-1} \rangle - m = n \mid \langle l::\text{comm-ring-1} \rangle = m + n \mid \langle l::\text{comm-ring-1} \rangle = m - n \mid \langle l::\text{comm-ring-1} \rangle = m \mid \langle l::\text{comm-ring-1} \rangle = - m \rangle = \langle K \text{ Numeral-Simprocs.eq-cancel-numerals} \rangle\)

simproc-setup intless-cancel-numerals
\((l::\text{linordered-idom}) + m < n \mid \langle \langle l::\text{linordered-idom} \rangle + m < n \mid \langle l::\text{linordered-idom} \rangle < m + n \mid \langle l::\text{linordered-idom} \rangle < m - n \mid \langle l::\text{linordered-idom} \rangle < m \mid \langle l::\text{linordered-idom} \rangle < - m \rangle = \langle K \text{ Numeral-Simprocs.le-cancel-numerals} \rangle\)

simproc-setup intle-cancel-numerals
\((l::\text{linordered-idom}) + m \leq n \mid \langle \langle l::\text{linordered-idom} \rangle + m \leq n \mid \langle l::\text{linordered-idom} \rangle \leq m + n \mid \langle l::\text{linordered-idom} \rangle < m \mid \langle l::\text{linordered-idom} \rangle < m \mid \langle l::\text{linordered-idom} \rangle \leq m \mid \langle l::\text{linordered-idom} \rangle \leq - m \rangle = \langle K \text{ Numeral-Simprocs.le-cancel-numerals} \rangle\)

simproc-setup ring-eq-cancel-numeral-factor
\((l::\text{idom,ring-char-0}) \ast m = n \mid \langle \langle l::\text{idom,ring-char-0} \rangle \ast m = n \mid \langle l::\text{idom,ring-char-0} \rangle = m + n \mid \langle l::\text{idom,ring-char-0} \rangle = m \mid \langle l::\text{idom,ring-char-0} \rangle = - m \rangle = \langle K \text{ Numeral-Simprocs.eq-cancel-numeral-factor} \rangle\)

simproc-setup ring-less-cancel-numeral-factor
\[ ((l::'a::linordered-idom) \ast m < n) \]
\[ |(l::'a::linordered-idom) < m \ast n) = \]
\[ \langle K \text{ Numeral-Simprocs.less-cancel-numeral-factor} \rangle \]

**simproc-setup** ring-le-cancel-numeral-factor

\[ ((l::'a::linordered-idom) \ast m <= n) \]
\[ |(l::'a::linordered-idom) <= m \ast n) = \]
\[ \langle K \text{ Numeral-Simprocs.le-cancel-numeral-factor} \rangle \]

**simproc-setup** int-div-cancel-numeral-factors

\[ (((l::'a::\{euclidean-semiring-cancel,comm-ring-1,ring-char-0\}) \ast m) \div n) \]
\[ |(l::'a::\{euclidean-semiring-cancel,comm-ring-1,ring-char-0\}) \div (m \ast n)) = \]
\[ \langle K \text{ Numeral-Simprocs.div-cancel-numeral-factor} \rangle \]

**simproc-setup** divide-cancel-numeral-factor

\[ (((l::'a::\{field,ring-char-0\}) \ast m) \div n) \]
\[ |(l::'a::\{field,ring-char-0\}) \div (m \ast n)) = \]
\[ \langle K \text{ Numeral-Simprocs.divide-cancel-numeral-factor} \rangle \]

**simproc-setup** ring-eq-cancel-factor

\[ (l::'a::\text{idom} \ast m = n | (l::'a::\text{idom}) = m \ast n) = \]
\[ \langle K \text{ Numeral-Simprocs.eq-cancel-factor} \rangle \]

**simproc-setup** linordered-ring-le-cancel-factor

\[ ((l::'a::linordered-idom) \ast m <= n) \]
\[ |(l::'a::linordered-idom) <= m \ast n) = \]
\[ \langle K \text{ Numeral-Simprocs.le-cancel-factor} \rangle \]

**simproc-setup** linordered-ring-less-cancel-factor

\[ ((l::'a::linordered-idom) \ast m < n) \]
\[ |(l::'a::linordered-idom) < m \ast n) = \]
\[ \langle K \text{ Numeral-Simprocs.less-cancel-factor} \rangle \]

**simproc-setup** int-div-cancel-factor

\[ (((l::'a::euclidean-semiring-cancel) \ast m) \div n) \]
\[ |(l::'a::euclidean-semiring-cancel) \div (m \ast n)) = \]
\[ \langle K \text{ Numeral-Simprocs.div-cancel-factor} \rangle \]

**simproc-setup** int-mod-cancel-factor

\[ (((l::'a::euclidean-semiring-cancel) \ast m) \mod n) \]
\[ |(l::'a::euclidean-semiring-cancel) \mod (m \ast n)) = \]
\[ \langle K \text{ Numeral-Simprocs.mod-cancel-factor} \rangle \]

**simproc-setup** dvd-cancel-factor

\[ (((l::'a::\text{idom}) \ast m) \dvd n) \]
\[ |(l::'a::\text{idom}) \dvd (m \ast n)) = \]
\[ \langle K \text{ Numeral-Simprocs.dvd-cancel-factor} \rangle \]
simproc-setup divide-cancel-factor
  \(((l::'a::field) * m) / n\)
  \|(l::'a::field) / (m * n)) =
  "K Numeral-Simprocs.divide-cancel-factor"

ML-file <Tools/nat-numeral-simprocs.ML>

simproc-setup nat-combine-numerals
  \((i::nat) + j | Suc (i + j)) =
  "K Nat-Numeral-Simprocs.combine-numerals"

simproc-setup nateq-cancel-numerals
  \((l::nat) + m = n | (l::nat) = m + n |
  (l::nat) * m = n | (l::nat) = m * n |
  Suc m = n | m = Suc n) =
  "K Nat-Numeral-Simprocs.eq-cancel-numerals"

simproc-setup natless-cancel-numerals
  \((l::nat) + m < n | (l::nat) < m + n |
  (l::nat) * m < n | (l::nat) < m * n |
  Suc m < n | m < Suc n) =
  "K Nat-Numeral-Simprocs.less-cancel-numerals"

simproc-setup natle-cancel-numerals
  \((l::nat) + m \leq n | (l::nat) \leq m + n |
  (l::nat) * m \leq n | (l::nat) \leq m * n |
  Suc m \leq n | m \leq Suc n) =
  "K Nat-Numeral-Simprocs.le-cancel-numerals"

simproc-setup natdiff-cancel-numerals
  \((l::nat) + m) - n | (l::nat) - (m + n) |
  (l::nat) * m - n | (l::nat) - m * n |
  Suc m - n | m - Suc n) =
  "K Nat-Numeral-Simprocs.diff-cancel-numerals"

simproc-setup nat-eq-cancel-numeral-factor
  \((l::nat) * m = n | (l::nat) = m * n) =
  "K Nat-Numeral-Simprocs.eq-cancel-numeral-factor"

simproc-setup nat-less-cancel-numeral-factor
  \((l::nat) * m < n | (l::nat) < m * n) =
  "K Nat-Numeral-Simprocs.less-cancel-numeral-factor"

simproc-setup nat-le-cancel-numeral-factor
  \((l::nat) * m \leq n | (l::nat) \leq m * n) =
  "K Nat-Numeral-Simprocs.le-cancel-numeral-factor"

simproc-setup nat-div-cancel-numeral-factor
THEORY “Semiring-Normalization”

((l::nat) * m) div n | (l::nat) div (m * n)) =
  ‹K Nat-Numeral-Simprocs.div-cancel-numeral-factor›

simproc-setup nat-dvd-cancel-numeral-factor
  ((l::nat) * m) dvd n | (l::nat) dvd (m * n)) =
  ‹K Nat-Numeral-Simprocs.dvd-cancel-numeral-factor›

simproc-setup nat-eq-cancel-factor
  ((l::nat) * m = n | (l::nat) = m * n) =
  ‹K Nat-Numeral-Simprocs.eq-cancel-factor›

simproc-setup nat-less-cancel-factor
  ((l::nat) * m < n | (l::nat) < m * n) =
  ‹K Nat-Numeral-Simprocs.less-cancel-factor›

simproc-setup nat-le-cancel-factor
  ((l::nat) * m <= n | (l::nat) <= m * n) =
  ‹K Nat-Numeral-Simprocs.le-cancel-factor›

simproc-setup nat-div-cancel-factor
  ((l::nat) * m) div n | (l::nat) div (m * n)) =
  ‹K Nat-Numeral-Simprocs.div-cancel-factor›

simproc-setup nat-dvd-cancel-factor
  ((l::nat) * m) dvd n | (l::nat) dvd (m * n)) =
  ‹K Nat-Numeral-Simprocs.dvd-cancel-factor›

declaration
  K (Lin-Arith.add-simprocs

end

59 Semiring normalization

theory Semiring-Normalization
imports Numeral-Simprocs
begin

Prelude

class comm-semiring-1-cancel-crossproduct = comm-semiring-1-cancel +
  assumes crossproduct-eq: \( w \cdot y + x \cdot z = w \cdot z + x \cdot y \iff w = x \lor y = z \)
begin

lemma crossproduct-noteq:
  \( a \neq b \land c \neq d \iff a \cdot c + b \cdot d \neq a \cdot d + b \cdot c \)
by (simp add: crossproduct-eq)

lemma add-scale-eq-noteq:
  \( r \neq 0 \Rightarrow a = b \land c \neq d \Rightarrow a + r \cdot c \neq b + r \cdot d \)
proof (rule notI)
  assume nz: \( r \neq 0 \) and cnd: \( a = b \land c \neq d \)
  have \( (0 \cdot d) + (r \cdot c) = (0 \cdot c) + (r \cdot d) \)
  using add-left-imp-eq eq mult-zero-left by (simp add: cnd)
  then show False using crossproduct-eq [of 0 d] nz cnd by simp
qed

lemma add-0-iff:
  \( b = b + a \iff a = 0 \)
using add-left-imp-eq [of b a 0] by auto

end

subclass (in idom) comm-semiring-1-cancel-crossproduct
proof
  fix \( w \, x \, y \, z \)
  show \( w \cdot y + x \cdot z = w \cdot z + x \cdot y \iff w = x \lor y = z \)
  proof
    assume \( w \cdot y + x \cdot z = w \cdot z + x \cdot y \)
    then have \( w \cdot y + x \cdot z - w \cdot z - x \cdot y = 0 \) by (simp add: algebra-simps)
    then have \( w \cdot (y - z) - x \cdot (y - z) = 0 \) by (simp add: algebra-simps)
    then have \( (y - z) \cdot (w - x) = 0 \) by (simp add: algebra-simps)
    then have \( y - z = 0 \lor w - x = 0 \) by (rule divisors-zero)
    then show \( w = x \lor y = z \) by auto
  qed (auto simp add: ac-simps)
qed

instance nat :: comm-semiring-1-cancel-crossproduct
proof
  fix \( w \, x \, y \, z :: \text{nat} \)
  have aux: \( \forall \, y \, z. \, y < z \implies w \cdot y + x \cdot z = w \cdot z + x \cdot y \implies w = x \)
  proof
    fix \( y \, z :: \text{nat} \)
    assume \( y < z \) then have \( \exists k. \, z = y + k \land k \neq 0 \) by (intro exI [of - z - y])
    auto
then obtain \( k \) where \( z = y + k \) and \( k \neq 0 \) by blast
assume \( w * y + x * z = w * z + x * y \)
then have \( (w * y + x * y) + x * k = (w * y + x * y) + w * k \) by (simp add:
\( \langle z = y + k \rangle \) algebra-simps)
then have \( x * k = w * k \) by simp
then show \( w = x \) using \( \langle k \neq 0 \rangle \) by simp
qed

Semiring normalization proper

ML-file \( \langle \text{Tools/semiring-normalizer.ML} \rangle \)

context comm-semiring-1
begin

lemma semiring-normalization-rules [no-atp]:
\( (a * m) + (b * m) = (a + b) * m \)
\( (a * m) + m = (a + 1) * m \)
\( m + (a * m) = (a + 1) * m \)
\( m + m = (1 + 1) * m \)
\( \theta + a = a \)
\( a + \theta = a \)
\( a * b = b * a \)
\( (a + b) * c = (a * c) + (b * c) \)
\( \theta * a = 0 \)
\( a * 0 = 0 \)
\( 1 * a = a \)
\( a * 1 = a \)
\( (lx * ly) * (rx * ry) = (lx * rx) * (ly * ry) \)
\( (lx * ly) * (rx * ry) = lx * (ly * (rx * ry)) \)
\( (lx * ly) * (rx * ry) = rx * ((lx * ly) * ry) \)
\( (lx * ly) * rx = (lx * rx) * ly \)
\( (lx * ly) * rx = lx * (ly * rx) \)
\( lx * (rx * ry) = (lx * rx) * ry \)
\( lx * (rx * ry) = rx * (lx * ry) \)
\( (a + b) + (c + d) = (a + c) + (b + d) \)
\( (a + b) + c = a + (b + c) \)
\( a + (c + d) = c + (a + d) \)
\( (a + b) + c = (a + c) + b \)
\( a + c = c + a \)
\( a + (c + d) = (a + c) + d \)
\( (x ^ p) * (x ^ q) = x ^ {p + q} \)
\( x * (x ^ q) = x ^ {Suc q} \)
\( (x ^ q) * x = x ^ {Suc q} \)
\( x * x = x ^ 2 \)
\( (x * y) ^ q = (x ^ q) * (y ^ q) \)
\( (x ^ p) ^ q = x ^ {p * q} \)
\(x \sim 0 = 1\)
\(x \sim 1 = x\)
\(x \sim (y + z) = (x \sim y) + (x \sim z)\)
\(x \sim (\text{Suc } q) = x \* (x \sim q)\)
\(x \sim (2\*n) = (x \sim n) \* (x \sim n)\)

by (simp-all add: algebra-simps power-add power2-eq-square
power-mult-distrib power-mult del: one-add-one)

local-setup (Semiring-Normalizer.declare @{thm comm-semiring-1-axioms})
{semiring = ([term}\(x + y\), term\(x \* y\), term\(x \sim n\), term\(0\), term\(1\)],
@{thms semiring-normalization-rules}},
ring = ([], []),
field = ([], []),
idom = [],
ideal = []}
)

end

context comm-ring-1
begin

lemma ring-normalization-rules [no-atp]:
\(- x = (-1) \* x\)
\(x - y = x + (-y)\)
by simp-all

local-setup (Semiring-Normalizer.declare @{thm comm-ring-1-axioms})
{semiring = ([term}\(x + y\), term\(x \* y\), term\(x \sim n\), term\(0\), term\(1\)],
@{thms semiring-normalization-rules}},
ring = ([term]\(-x\), term\(-x\)], @{thms ring-normalization-rules}),
field = ([], []),
idom = [],
ideal = []}
)

end

context comm-semiring-1-cancel-crossproduct
begin

local-setup (Semiring-Normalizer.declare @{thm comm-semiring-1-cancel-crossproduct-axioms})
{semiring = ([term}\(x + y\), term\(x \* y\), term\(x \sim n\), term\(0\), term\(1\)],
@{thms semiring-normalization-rules}},
ring = ([], []),
field = ([], []),

idom = @\{thms crossproduct-noteq add-scale-eq-noteq\},
ideal = []

end

context idom
begin
local-setup

Semiring-Normalizer.declare @\{thm idom-axioms\}
{semiring = (\[term x + y, term x * y, term x ^ n, term 0, term 1\],
  @\{thms semiring-normalization-rules\}),
ring = (\[term x - y, term (- x)\], @\{thms ring-normalization-rules\}),
field = (\[]),
idom = @\{thms crossproduct-noteq add-scale-eq-noteq\},
ideal = @\{thms right-minus-eq add-0-iff\}}

end

context field
begin
local-setup

Semiring-Normalizer.declare @\{thm field-axioms\}
{semiring = (\[term x + y, term x * y, term x ^ n, term 0, term 1\],
  @\{thms semiring-normalization-rules\}),
ring = (\[term x - y, term (- x)\], @\{thms ring-normalization-rules\}),
field = (\[term x / y, term (inverse x)\], @\{thms divide-inverse inverse-eq-divide\}),
idom = @\{thms crossproduct-noteq add-scale-eq-noteq\},
ideal = @\{thms right-minus-eq add-0-iff\}}

end

code-identifier

code-module Semiring-Normalization \rightarrow (SML) Arith and (OCaml) Arith and (Haskell) Arith
end

60 Groebner bases

theory Groebner-Basis
imports Semiring-Normalization Parity
begin
60.1 Groebner Bases

lemmas bool-simps = simp-thms(1–34) — FIXME move to HOL.HOL

lemma nnf-simps: — FIXME shadows fact binding in HOL.HOL

\[ \neg(P \land Q) = (\neg P \lor \neg Q) \quad (\neg P \lor \neg Q) = \neg(P \land Q) \]
\[ (P \imp Q) = (\neg P \lor Q) \quad (P \land Q) \lor (\neg P \land \neg Q) \land \neg(P) = P \]
by blast+

lemma dnf:

\[ (P \land (Q \lor R)) = ((P \land Q) \lor (P \land R)) \]
\[ ((Q \lor R) \land P) = ((Q \land P) \lor (R \land P)) \]
\[ (P \land Q) = (Q \land P) \quad (P \lor Q) = (Q \lor P) \]
by blast+

lemmas weak-dnf-simps = dnf bool-simps

lemma PFalse:

\[ P \equiv False \implies \neg P \]
\[ \neg P \implies (P \equiv False) \]
by auto

named-theorems algebra pre—simplification rules for algebraic methods

ML-file ‹Tools/groebner.ML›

method-setup algebra =

let

val addN = add
val delN = del
val any-keyword = keyword addN || keyword delN
val thms = Scan.repeats (Scan.unless any-keyword Attrib.multi-thm);

fun keyword k = Scan.lift (Args.arg wanted -- Args.colon) >> K ()

fun simplify (add-ths, del-ths) => fn ctxt =>
SIMPLE-METHOD' (Groebner.algebra-tac add-ths del-ths ctxt))

solve polynomial equations over (semi)rings and ideal membership problems using Groebner bases

declare dvd-def[algebra]
declare mod-eq-0-iff-dvd[algebra]
declare mod-div-trivial[algebra]
declare mod-mod-trivial[algebra]
declare div-by-0[algebra]
declare mod-by-0[algebra]
declare mult-div-mod-eq[algebra]
declare div-minus-minus [algebra]
declare mod-minus-minus [algebra]
declare div-minus-right [algebra]
declare mod-minus-right [algebra]
declare div-0 [algebra]
declare mod-0 [algebra]
declare mod-by-1 [algebra]
declare div-by-1 [algebra]
declare mod-minus1-right [algebra]
declare div-minus1-right [algebra]
declare mod-mult-self2-is-0 [algebra]
declare mod-mult-self1-is-0 [algebra]

lemma zmod-eq-0-iff [algebra]:
  \(<m \mod d = 0 \iff (\exists q. m = d \cdot q)>\) for m d :: int
  by (auto simp add: mod-eq-0-iff-dvd)

declare dvd-0-left-iff [algebra]
declare zdvd1-eq [algebra]
declare mod-eq-dvd-iff [algebra]
declare nat-mod-eq-iff [algebra]

context semiring-parity
begin

declare even-mult-iff [algebra]
declare even-power [algebra]
end

context ring-parity
begin

declare even-minus [algebra]
end

declare even-Suc [algebra]
declare even-diff-nat [algebra]
end

61 Set intervals

theory Set-Interval
imports Parity
begin
THEORY “Set-Interval”

lemma card-2-iff: card S = 2 ←→ (∃ x y. S = {x,y} ∧ x ≠ y)
  by (auto simp: card-Suc-eq numeral-eq-Suc)

lemma card-2-iff': card S = 2 ←→ (∃ x y∈S. x ≠ y ∧ (∀ z∈S. z = x ∨ z = y))
  by (auto simp: card-Suc-eq numeral-eq-Suc)

lemma card-3-iff: card S = 3 ←→ (∃ x y z. S = {x,y,z} ∧ x ≠ y ∧ y ≠ z ∧ x ≠ z)
  by (fastforce simp: card-Suc-eq numeral-eq-Suc)

corollary card-Suc-eq_nat:
  fixes n::nat
  assumes card_Suc_eq: card S = card (Suc S)
  shows card_Suc_n consisting of $\{1\}..<n\}$

context ord
begin

definition lessThan :: 'a => 'a set ((1 {..<})) where
{..<u} == {x. x < u}

definition atMost :: 'a => 'a set ((1 {..})) where
{..u} == {x. x ≤ u}

definition greaterThan :: 'a => 'a set ((1 {..<})) where
{l..<} == {x. l<x}

definition atLeast :: 'a => 'a set ((1 {...})) where
{..l} == {x. l≤x}

definition greaterThanLessThan :: 'a => 'a => 'a set ((1 {<..<})) where
{l..<u} == {l..<} Int {..<u}

definition atLeastLessThan :: 'a => 'a => 'a set ((1 {...<})) where
{1..<u} == {1..} Int {..<u}

definition greaterThanAtMost :: 'a => 'a => 'a set ((1 {<..}) where
{l<..} == {l..<} Int {..u}

definition atLeastAtMost :: 'a => 'a => 'a set ((1 {...}) where
{1..<u} == {1..} Int {..u}

end

A note of warning when using $\{..<n\}$ on type nat: it is equivalent to $\{0..<n\}$
but some lemmas involving \{m..<n\} may not exist in \{..<n\}-form as well.

**syntax (ASCII)**

-\(\text{-UNION-le}:: \ 'a => 'a => 'b set => 'b set\)

\((3\text{UN} -<.-./ -) [0, 0, 10]\)

-\(\text{-UNION-less}:: \ 'a => 'a => 'b set => 'b set\)

\((3\text{UN} -<.-./ -) [0, 0, 10] 10\)

-\(\text{-INTER-le}:: \ 'a => 'a => 'b set => 'b set\)

\((3\text{INT} -<.-./ -) [0, 0, 10]\)

-\(\text{-INTER-less}:: \ 'a => 'a => 'b set => 'b set\)

\((3\text{INT} -<.-./ -) [0, 0, 10] 10\)

**syntax (latex output)**

-\(\text{-UNION-le}:: \ 'a \Rightarrow 'a \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}\)

\([0, 0, 10] 10\)

-\(\text{-UNION-less}:: \ 'a \Rightarrow 'a \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}\)

\([0, 0, 10] 10\)

-\(\text{-INTER-le}:: \ 'a \Rightarrow 'a \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}\)

\([0, 0, 10] 10\)

-\(\text{-INTER-less}:: \ 'a \Rightarrow 'a \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}\)

\([0, 0, 10] 10\)

**translations**

\(\bigcup i \leq n. \ A = \bigcup i \in \{..n\}. \ A\)

\(\bigcup i < n. \ A = \bigcup i \in \{..<n\}. \ A\)

\(\bigcap i \leq n. \ A = \bigcap i \in \{..n\}. \ A\)

\(\bigcap i < n. \ A = \bigcap i \in \{..<n\}. \ A\)

**61.1 Various equivalences**

**lemma (in ord) lessThan-iff [iff]: \((i \in \text{lessThan} \ k) = (i<k)\)**

by (simp add: lessThan-def)

**lemma Compl-lessThan [simp]:**

\(!k:: 'a::linorder. \ -\text{lessThan} \ k = \text{atLeast} \ k\)

by (auto simp add: lessThan-def atLeast-def)

**lemma single-Diff-lessThan [simp]: !k:: 'a:: preorder. \ \{k\} - \text{lessThan} \ k = \{k\}\)

by auto

**lemma (in ord) greaterThan-iff [iff]: \((i \in \text{greaterThan} \ k) = (k<i)\)**

by (simp add: greaterThan-def)

**lemma Compl-greaterThan [simp]:**

\(!k:: 'a::linorder. \ -\text{greaterThan} \ k = \text{atMost} \ k\)

by (auto simp add: greaterThan-def atMost-def)
lemma Compl-atMost [simp]: \!\! k:: 'a::linorder. \- atMost k = greaterThan k
  by (metis Compl-greaterThan double-complement)

lemma (in ord) atLeast-iff [iff]: \( i \in \text{atLeast } k \) = \( k <\leq i \)
  by (simp add: atLeast-def)

lemma Compl-atLeast [simp]: \!\! k:: 'a::linorder. \- atLeast k = lessThan k
  by (auto simp add: lessThan-def atLeast-def)

lemma (in ord) atMost-iff [iff]: \( i \in \text{atMost } k \) = \( i <\leq k \)
  by (simp add: atMost-def)

lemma atMost-Int-atLeast: \!\! n:: 'a::order. atMost n \cap atLeast n = \{ n \}
  by (blast intro: order-antisym)

lemma (in linorder) lessThan-Int-lessThan: \{ a <.. \} \cap \{ b <.. \} = \{ \max a b <.. \}
  by auto

lemma (in linorder) greaterThan-Int-greaterThan: \{.. < a \} \cap \{.. < b \} = \{.. < \min a b \}
  by auto

61.2 Logical Equivalences for Set Inclusion and Equality

lemma atLeast-empty-triv [simp]: \{\}.. = UNIV
  by auto

lemma atMost-UNIV-triv [simp]: \{.. UNIV \} = UNIV
  by auto

lemma atLeast-subset-iff [iff]:
  \( \text{atLeast } x \subseteq \text{atLeast } y \) = \( y \leq (x::'a::preorder) \)
  by (blast intro: order-trans)

lemma atLeast-eq-iff [iff]:
  \( \text{atLeast } x = \text{atLeast } y \) = \( x = (y::'a::order) \)
  by (auto simp: equalityE)

lemma greaterThan-subset-iff [iff]:
  \( \text{greaterThan } x \subseteq \text{greaterThan } y \) = \( y \leq (x::'a::linorder) \)
  unfolding greaterThan-def by (auto simp: linorder-not-less [symmetric])

lemma greaterThan-eq-iff [iff]:
  \( \text{greaterThan } x = \text{greaterThan } y \) = \( x = (y::'a::linorder) \)
  by (auto simp: elim!: equalityE)

lemma atMost-subset-iff [iff]: \( \text{atMost } x \subseteq \text{atMost } y \) = \( x \leq (y::'a::preorder) \)
  by (blast intro: order-trans)
lemma atMost-eq-iff [iff]: \((atMost x = atMost y) = (x = (y::'a::order))\)
  by (blast intro: order-antisym order-trans)

lemma lessThan-subset-iff [iff]:
  \((lessThan x \subseteq lessThan y) = (x \leq (y::'a::linorder))\)
unfolding lessThan-def by (auto simp: linorder-not-less [symmetric])

lemma lessThan-eq-iff [iff]:
  \((lessThan x = lessThan y) = (x = (y::'a::linorder))\)
by (auto simp: elim!: equalityE)

lemma lessThan-strict-subset-iff:
  fixes \(m\) \(n\) :: 'a::linorder
  shows \(\{..<\,m\}\) \(\,<..<\,n\}\) \(\longleftrightarrow\) \(m\,<n\)
by (metis leD lessThan-subset-iff linorder-linear not-less-iff-gr-or-eq psubset-eq)

lemma (in linorder) Ici-subset-Ioi-iff:
  \(\{a ..\}\) \(\subseteq\) \(\{b <..\}\) \(\longleftrightarrow\) \(b < a\)
by auto

lemma (in linorder) Iic-subset-Iio-iff:
  \(\{..<a\}\) \(\subseteq\) \(\{..<b\}\) \(\longleftrightarrow\) \(a < b\)
by auto

lemma (in preorder) Ioi-le-Ico:
  \(\{a <..\}\) \(\subseteq\) \(\{a ..\}\)
by (auto intro: less-imp-le)

61.3 Two-sided intervals

context ord
begin

lemma greaterThanLessThan-iff [simp]: \(i \in \{l..<u\}\) = \((l < i \land i < u)\)
  by (simp add: greaterThanLessThan-def)

lemma atLeastLessThan-iff [simp]: \(i \in \{l..<u\}\) = \((l \leq i \land i < u)\)
  by (simp add: atLeastLessThan-def)

lemma greaterThanAtMost-iff [simp]: \(i \in \{l..<u\}\) = \((l < i \land i \leq u)\)
  by (simp add: greaterThanAtMost-def)

lemma atLeastAtMost-iff [simp]: \(i \in \{l..<u\}\) = \((l \leq i \land i \leq u)\)
  by (simp add: atLeastAtMost-def)

The above four lemmas could be declared as iffs. Unfortunately this breaks many proofs. Since it only helps blast, it is better to leave them alone.

lemma greaterThanLessThan-eq:
  \(\{a <..\,<b\}\) = \(\{a <..<b\}\) \cap \(..<b\) \}
by auto

lemma (in order) atLeastLessThan-eq-atLeastAtMost-diff:
\{a..<b\} = \{a..b\} - \{b\}
by (auto simp add: atLeastLessThan-def atLeastAtMost-def)

lemma (in order) greaterThanAtMost-eq-atLeastAtMost-diff:
\{a..<b\} = \{a..b\} - \{a\}
by (auto simp add: greaterThanAtMost-def atLeastAtMost-def)
end

61.3.1 Emptyness, singletons, subset
context preorder begin

lemma atLeastAtMost-empty-iff [simp]:
\{a..b\} = {} ←→ (¬ a ≤ b)
by auto (blast intro: order-trans)

lemma atLeastAtMost-empty-iff2 [simp]:
{} = \{a..b\} ←→ (¬ a ≤ b)
by auto (blast intro: order-trans)

lemma atLeastLessThan-empty-iff [simp]:
\{a..<b\} = {} ←→ (¬ a < b)
by auto (blast intro: le-less-trans)

lemma atLeastLessThan-empty-iff2 [simp]:
{} = \{a..<b\} ←→ (¬ a < b)
by auto (blast intro: le-less-trans)

lemma greaterThanAtMost-empty-iff [simp]:
\{k..<l\} = {} ←→ ¬ k < l
by auto (blast intro: less-le-trans)

lemma greaterThanAtMost-empty-iff2 [simp]:
{} = \{k..<l\} ←→ ¬ k < l
by auto (blast intro: less-le-trans)

lemma atLeastAtMost-subset-iff [simp]:
\{a..b\} ≤ \{c..d\} ←→ (¬ a ≤ b) ∨ c ≤ a ∧ b ≤ d

unfolding atLeastAtMost-def atLeast-def atMost-def
by (blast intro: order-trans)

lemma atLeastAtMost-psubset-iff:
\{a..b\} < \{c..d\} ←→
((¬ a ≤ b) ∨ c ≤ a ∧ b ≤ d ∧ (c < a ∨ b < d)) ∧ c ≤ d
by (simp add: psubset-eq set-eq-iff less-le-not-le)(blast intro: order-trans)

lemma atLeastAtMost-psubseteq-atLeastLessThan-iff:
\{a..b\} ⊆ \{c..<d\} ←→ (a ≤ b ⇒ c ≤ a ∧ b < d)
by auto (blast intro: local.order-trans local.le-less-trans elim: )+
lemma Icc-subset-Ici-iff\[simp\]:
\[
{\{l..h\}} \subseteq \{l'...\} = (\sim l \leq h \lor l \geq l')
\]
by (auto simp: subset-eq intro: order-trans)

lemma Icc-subset-Iic-iff\[simp\]:
\[
{\{l..h\}} \subseteq \{..h'\} = (\sim l \leq h \lor h \leq h')
\]
by (auto simp: subset-eq intro: order-trans)

lemma not-Ici-eq-empty\[simp\]:
\[
\{l..\} \neq \{\}\n\]
by (auto simp: set-eq-iff)

lemma not-Iic-eq-empty\[simp\]:
\[
\{..h\} \neq \{\}\n\]
by (auto simp: set-eq-iff)

lemmas not-empty-eq-Ici-eq-empty\[simp\] =
not-Ici-eq-empty\[symmetric\]

lemmas not-empty-eq-Iic-eq-empty\[simp\] =
not-Iic-eq-empty\[symmetric\]

end

context order
begin

lemma atLeastatMost-empty\[simp\]:
\[
b < a \Longrightarrow \{a..b\} = \{\}\n\]
and atLeastatMost-empty\[simp\]:
\[
\sim a \leq b \Longrightarrow \{a..b\} = \{\}\n\]
by (auto simp: atLeastAtMost-def atLeast-def atMost-def)

lemma atLeastLessThan-empty\[simp\]:
\[
b \leq a \Longrightarrow \{a..<b\} = \{\}\n\]
by (auto simp: atLeastLessThan-def)

lemma greaterThanAtMost-empty\[simp\]:
\[
l \leq k \Longrightarrow \{k..<l\} = \{\}\n\]
by (auto simp: greaterThanAtMost-def greaterThan-def atMost-def)

lemma greaterThanLessThan-empty\[simp\]:
\[
l \leq k \Longrightarrow \{k..<l\} = \{\}\n\]
by (auto simp: greaterThanLessThan-def greaterThan-less-def less-than-def)

lemma atLeastAtMost-singleton \[simp\]:
\[
\{a..a\} = \{a\}\n\]
by (auto simp add: atLeastAtMost-def atMost-def atLeast-def)

lemma atLeastAtMost-singleton\':
\[
a = b \Longrightarrow \{a..b\} = \{a\}\n\]
by simp

lemma Icc-eq-Icc\[simp\]:
\[
{\{l..h\}} = {\{l'..h'\}} = (l = l' \land h = h' \lor \sim l \leq h \land \sim l' \leq h')
\]
by (simp add: order-class.order.eq-iff) (auto intro: order-trans)

lemma (in linorder) Ico-eq-Ico:
\[
{\{l..<h\}} = {\{l'..<h'\}} = (l = l' \land h = h' \lor \sim l < h \land \sim l' < h')
\]
by (metis atLeastLessThan-empty-iff2 nle-le not-less ord.atLeastLessThan-iff2)
lemma atLeastAtMost-singleton-iff[simp]:
\{a .. b\} = \{c\} \iff a = b \land b = c

proof
  assume \{a..b\} = \{c\}
  hence *: \neg (\forall a \leq b) unfolding atLeastAtMost-empty-iff[symmetric] by simp
  with \{a..b\} = \{c\} have c \leq a \land b \leq c by auto
  with * show a = b \land b = c by auto
qed simp

end

context no-top
begin

lemma not-UNIV-le-Icc[simp]: \neg UNIV \subseteq \{l..h\}
using gt-ex[of h] by(auto simp: subset-eq less-le-not-le)

lemma not-UNIV-le-Iic[simp]: \neg UNIV \subseteq \{..h\}
using gt-ex[of h] by(auto simp: subset-eq less-le-not-le)

lemma not-Ici-le-Icc[simp]: \neg \{l..\} \subseteq \{l'..h'\}
using lt-ex[of l] by(auto simp: subset-eq less-le)(blast dest:antisym-conv intro: order-trans)

lemma not-Iic-le-Icc[simp]: \neg \{l..h\} \subseteq \{l'..h'\}
using lt-ex[of l'] by(auto simp: subset-eq less-le)(blast dest:antisym-conv intro: order-trans)

end

context no-bot
begin

lemma not-UNIV-le-Ici[simp]: \neg UNIV \subseteq \{l..\}
using lt-ex[of l] by(auto simp: subset-eq less-le-not-le)

lemma not-Iic-le-Icci[simp]: \neg \{..h\} \subseteq \{l'..\}
using lt-ex[of l'] by(auto simp: subset-eq less-le)(blast dest:antisym-conv intro: order-trans)

lemma not-Iic-le-Icci[simp]: \neg \{..h\} \subseteq \{l'..\}
using lt-ex[of l'] by(auto simp: subset-eq less-le)(blast dest:antisym-conv intro: order-trans)

end
context no-top
begin

lemma not-UNIV-eq-Icc[simp]: \( \neg \text{UNIV} = \{l'.h'\} \)
using gt-ex[of h'] by (auto simp: set-eq-iff less-le-not-le)

lemmas not-Icc-eq-UNIV[simp] = not-UNIV-eq-Icc[symmetric]

lemma not-UNIV-eq-Iic[simp]: \( \neg \text{UNIV} = \{..h'\} \)
using gt-ex[of h'] by (auto simp: set-eq-iff less-le-not-le)

lemmas not-Iic-eq-UNIV[simp] = not-UNIV-eq-Iic[symmetric]

lemma not-Icc-eq-Ici[simp]: \( \neg \{l..h\} = \{l'..\} \)
unfolding atLeastAtMost-def using not-Ici-le-Iic[of l'] by blast

lemmas not-Ici-eq-Icc[simp] = not-Icc-eq-Ici[symmetric]

end

context no-bot
begin

lemma not-UNIV-eq-Ici[simp]: \( \neg \text{UNIV} = \{l'..\} \)
using lt-ex[of a l'] by (auto simp: set-eq-iff less-le-not-le)

lemmas not-Ici-eq-UNIV[simp] = not-UNIV-eq-Ici[symmetric]

lemma not-Icc-eq-Iic[simp]: \( \neg \{l..h\} = \{..h'\} \)
unfolding atLeastAtMost-def using not-Iic-le-Ici[of h'] by blast

lemmas not-Iic-eq-Icc[simp] = not-Icc-eq-Iic[symmetric]

end

context dense-linorder
begin

lemma greaterThanLessThan-empty-iff[simp]:
\( \{ a <..< b \} = \{ \} \iff b \leq a \)
using dense[of a b] by (cases a < b) auto
lemma greaterThanLessThan-empty-iff2 [simp]:
\[ \{ a..<b \} \leftrightarrow b \leq a \]
using dense[of a b] by (cases a < b) auto

lemma atLeastLessThan-subseteq-atLeastAtMost-iff:
\[ \{ a..< b \} \subseteq \{ c .. d \} \leftrightarrow (a < b \rightarrow c \leq a \land b \leq d) \]
using dense[of a min c b] dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanAtMost-subseteq-atLeastLessThan-iff:
\[ \{ a..< b \} \subseteq \{ c ..< d \} \leftrightarrow (a < b \rightarrow c \leq a \land b < d) \]
using dense[of a min c b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanAtMost-subseteq-greaterThanLessThan-iff:
\[ \{ a..< b \} \subseteq \{ c..<d \} \leftrightarrow (a < b \rightarrow a \geq c \land b \leq d) \]
using dense[of a min c b] dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

end

context no-top
begin

lemma greaterThan-non-empty [simp]: \( \{ x..< \} \neq \{ \} \)
using gt-ex[of x] by auto

end
context no-bot

begin

lemma lessThan-non-empty[simp]: \{..< x\} \neq \{
    using lt-ex[of x] by auto

end

lemma (in linorder) atLeastLessThan-subset-iff:
    \{a..<b\} \subseteq \{c..<d\} \implies b \leq a \lor c \leq a \land b \leq d

proof (cases a < b)
    case True
    assume assm: \{a..<b\} \subseteq \{c..<d\}
    then have 1: c \leq a \land a \leq d
        using True by (auto simp add: subset-eq Ball-def)
    then have 2: b \leq d
        using assm by (auto simp add: subset-eq)
    from 1 2 show ?thesis
        by simp
    qed (auto)

lemma atLeastLessThan-inj:
    fixes a b c d :: 'a::linorder
    assumes eq: \{a..<b\} = \{c..<d\} and a < b c < d
    shows a = c b = d
    using assms by (metis atLeastLessThan-subset-iff eq less-le-not-le antisym_conv2 subset_refl+)

lemma atLeastLessThan-eq-iff:
    fixes a b c d :: 'a::linorder
    assumes a < b c < d
    shows \{a ..< b\} = \{c ..< d\} \iff a = c \land b = d
    using atLeastLessThan-inj assms by auto

lemma (in linorder) Ioc-inj:
    \langle a ..< b \rangle = \langle c ..< d \rangle \iff (b \leq a \land d \leq c) \lor a = c \land b = d\> (is \langle ?P \iff ?Q \rangle)

proof
    assume ?Q
    then show ?P
        by auto
next
    assume ?P
    then have \langle a < c \land x \leq b \iff c < x \land x \leq d \rangle for x
        by (simp add: set-eq-iff)
    from this [of a] this [of b] this [of c] this [of d] show ?Q
        by auto
    qed
lemma (in order) Iio-Int-singleton: {..<k} ∩ {x} = (if x < k then {x} else {})
  by auto

lemma (in linorder) Ioc-subset-iff: {a..<b} ⊆ {c..<d} ↔ (b ≤ c ∧ a ≤ d)
  by (auto simp: subset_eq Ball_def) (metis less_le not-less)

lemma (in order-bot) atLeast-eq-UNIV-iff: {x..} = UNIV ↔ x = bot
  by (auto simp: set-eq_iff intro: le-bot)

lemma (in order-top) atMost-eq-UNIV-iff: {..x} = UNIV ↔ x = top
  by (auto simp: set-eq_iff intro: top-le)

lemma (in bounded-lattice) atLeastAtMost-eq-UNIV-iff: {x..y} = UNIV ↔ x = bot ∧ y = top
  by (auto simp: set-eq_iff intro: top-le le-bot)

lemma Iio-eq-empty-iff: {..<n::'a::{linorder, order-bot}} = {} ↔ n = bot
  by (auto simp: set-eq_iff not_less le-bot)

lemma infinite-Ioo: assumes a < b shows ¬ finite {..<a..<b}
proof
  assume fin: finite {..<a..<b}
  moreover have ne: {..<a..<b} ≠ {} using fin by auto
  then obtain k where f k = m' m ≤ k by auto
  moreover have m' ≤ f m using f-image by auto
  ultimately show f m = m'
  using f-mono by (auto elim: monoE[where x=m and y=k])
qed

61.4 Infinite intervals

category dense-linorder
begin

lemma infinite-Ioo:
  assumes a < b
  shows ¬ finite {..<a..<b}
proof
  assume fin: finite {..<a..<b}
  moreover have ne: {..<a..<b} ≠ {} using fin by auto
  ultimately have a < Max {..<a..<b} Max {a..<b} < b
THEORY “Set-Interval”

using Max-in[of {a <..< b}] by auto
then obtain x where Max {a <..< b} < x x < b
  using dense[of Max {a<..<b} b] by auto
then have x ∈ {a <..< b}
  using <a < Max {a <..< b}, by auto
then have x ≤ Max {a <..< b}
  using fin by auto
with Max {a <..< b} < x, show False by auto
qed

lemma infinite-Icc: a < b =⇒ ¬ finite {a .. b}
  using greaterThanLessThan-subseteq-atLeastAtMost-iff[of a b a b]
  by (auto dest: finite-subset)

lemma infinite-Ico: a < b =⇒ ¬ finite {a ..< b}
  using greaterThanLessThan-subseteq-greaterThanAtMost-iff[of a b a b]
  by (auto dest: finite-subset)

lemma infinite-Ioc: a < b =⇒ ¬ finite {a <..< b}
  using greaterThanLessThan-subseteq-atLeastLessThan-iff[of a b a b]
  by (auto dest: finite-subset)

lemma infinite-Ioo: ¬ finite {..< a :: ⦃a :: ⦃no-bot, linorder⦄}
  proof
    assume finite {..< a}
    then have *: ∀x. x < a =⇒ Min {..< a} ≤ x
      by auto
    obtain x where x < a
      using lt-ex by auto
    obtain y where y < Min {..< a}
      using lt-ex by auto
also have $\text{Min} \{..< a\} \leq x$
using $(x < a)$ by fact
also note $(x < a)$
finally have $\text{Min} \{..< a\} \leq y$
by fact
with $(y < \text{Min} \{..< a\})$ show False by auto
qed

lemma infinite-Iic: $\neg$ finite \{.. a :: 'a :: {no-bot, linorder}\}
using infinite-Iio[of a] finite-subset[of \{..< a\} \{.. a\}]
by (auto simp: subset-eq less-imp-le)

lemma infinite-Ioi: $\neg$ finite \{a :: 'a :: {no-top, linorder} <..\}
proof
assume finite \{a <..\}
then have $(\forall x. a < x \implies x \leq \text{Max} \{a <..\})$
by auto
obtain $y$ where $\text{Max} \{a <..\} < y$
using gt-ex by auto

obtain $x$ where $x; a < x$
using gt-ex by auto
also from $x$ have $x \leq \text{Max} \{a <..\}$
by fact
also note $(\text{Max} \{a <..\} < y)$
finally have $y \leq \text{Max} \{a <..\}$
by fact
with $(\text{Max} \{a <..\} < y)$ show False by auto
qed

lemma infinite-Ici: $\neg$ finite \{a :: 'a :: {no-top, linorder} ..\}
using infinite-Ioi[of a] finite-subset[of \{a <..\} \{a ..\}]
by (auto simp: subset-eq less-imp-le)

61.4.1 Intersection
context linorder
begin

lemma Int-atLeastAtMost[simp]: \{a..b\} \text{Int} \{c..d\} = \{\text{max} a c .. \text{min} b d\}
by auto

lemma Int-atLeastAtMostR1[simp]: \{..b\} \text{Int} \{c..d\} = \{c .. \text{min} b d\}
by auto

lemma Int-atLeastAtMostR2[simp]: \{a..\} \text{Int} \{c..d\} = \{\text{max} a c .. d\}
by auto
lemma Int-atLeastAtMostL1 [simp]: \{a..b\} \cap \{c..d\} = \{a .. \min b d\}
by auto

lemma Int-atLeastAtMostL2 [simp]: \{a..b\} \cap \{c..\} = \{\max a c .. b\}
by auto

lemma Int-atLeastLessThan [simp]: \{a..<b\} \cap \{c..<d\} = \{\max a c..< \min b d\}
by auto

lemma Int-greaterThanAtMost [simp]: \{a<..b\} \cap \{c..<d\} = \{\max a c <.. b\}
by auto

lemma Int-greaterThanLessThan [simp]: \{a<..<b\} \cap \{c..<<d\} = \{\max a c <..< \min b d\}
by auto

lemma Int-atMost [simp]: \{..\} \cap \{..a\} = \{.. \min a b\}
by (auto simp: min-def)

lemma Int-atLeastLessThan [simp]: \{a..<b\} \cap \{c..<d\} = \{\max a c..< \min b d\}
by auto

lemma Ioc-disjoint: \{a..<b\} \cap \{c..<d\} = \{}
\iff b \leq a \lor d \leq c \lor b \leq c \lor d \leq a
by auto
end

context complete-lattice
begin

lemma shows Sup-atLeast [simp]: \text{Sup} \{x ..\} = \text{top}
and Sup-greaterThanAtLeast [simp]: x < \text{top} \implies \text{Sup} \{x ..\} = \text{top}
and Sup-atMost [simp]: \text{Sup} \{.. y\} = y
and Sup-atLeastAtMost [simp]: x \leq y \implies \text{Sup} \{x .. y\} = y
and Sup-greaterThanAtMost [simp]: x < y \implies \text{Sup} \{x .. y\} = y
by (auto intro!: Sup-eqI)

lemma shows Inf-atMost [simp]: \text{Inf} \{.. x\} = \text{bot}
and Inf-atMostLessThan [simp]: \text{top} < x \implies \text{Inf} \{.. x\} = \text{bot}
and Inf-atLeast [simp]: \text{Inf} \{x ..\} = x
and Inf-atLeastAtMost [simp]: x \leq y \implies \text{Inf} \{x .. y\} = x
and Inf-greaterThanLessThan [simp]: x < y \implies \text{Inf} \{x .. y\} = x
by (auto intro!: Inf-eqI)
end

lemma
fixes x y :: 'a :: \{complete-lattice, dense-linorder\}
THEORY “Set-Interval”

shows Sup-lessThan[simp]: Sup \{..< y\} = y
and Sup-atLeastLessThan[simp]: x < y \implies Sup \{ x..< y\} = y
and Sup-greaterThanLessThan[simp]: x < y \implies Sup \{ x..< y\} = y
and Inf-greaterThan[simp]: Inf \{x..<\} = x
and Inf-greaterThanAtMost[simp]: x < y \implies Inf \{x..< y\} = x
and Inf-greaterThanLessThan[simp]: x < y \implies Inf \{x..< y\} = x
by (auto intro: Inf-eqI Sup-eqI intro: dense-le dense-le-bounded dense-ge dense-ge-bounded)

61.5 Intervals of natural numbers

61.5.1 The Constant lessThan

lemma lessThan-0 [simp]: lessThan 0 = nat = \{
by (simp add: lessThan-def)

lemma lessThan-Suc: lessThan (Suc k) = insert k (lessThan k)
by (simp add: lessThan-def less-Suc-eq, blast)

The following proof is convenient in induction proofs where new elements get indices at the beginning. So it is used to transform \{..<Suc n\} to 0 and \{..<n\}.

lemma zero-notin-Suc-image [simp]: 0 \notin Suc A
by auto

lemma lessThan-Suc-eq-insert-0: \{..<Suc n\} = insert 0 (Suc \{..<n\})
by (auto simp: image-iff less-Suc-eq-0-disj)

lemma lessThan-Suc-atMost: lessThan (Suc k) = atMost k
by (simp add: lessThan-def atMost-def less-Suc-eq-le)

lemma atMost-Suc-eq-insert-0: \{.. Suc n\} = insert 0 (Suc \{.. n\})
unfolding lessThan-Suc-atMost[symmetric] lessThan-Suc-eq-insert-0[of Suc n]
..

lemma UN-lessThan-UNIV: \(\bigcup m::nat. \text{lessThan } m\) = UNIV
by blast

61.5.2 The Constant greaterThan

lemma greaterThan-0: greaterThan 0 = range Suc
unfolding greaterThan-def
by (blast dest: gr0-conv-Suc [THEN iffD1])

lemma greaterThan-Suc: greaterThan (Suc k) = greaterThan k - \{Suc k\}
unfolding greaterThan-def
by (auto elim: linorder-neqE)

lemma INT-greaterThan-UNIV: \(\bigcap m::nat. \text{greaterThan } m\) = \{
by blast
61.5.3 The Constant atLeast

**lemma atLeast-0 [simp]:** atLeast (0 :: nat) = UNIV 
by (unfold atLeast-def UNIV-def, simp)

**lemma atLeast-Suc:** atLeast (Suc k) = atLeast k - {k} 
unfolding atLeast-def by (auto simp: order-le-less Suc-le-eq)

**lemma atLeast-Suc-greaterThan:** atLeast (Suc k) = greaterThan k 
by (auto simp add: greaterThan-def atLeast-def less-Suc-eq-le)

**lemma UN-atLeast-UNIV:** (∪ m :: nat. atLeast m) = UNIV 
by blast

61.5.4 The Constant atMost

**lemma atMost-0 [simp]:** atMost (0 :: nat) = {0} 
by (simp add: atMost-def)

**lemma atMost-Suc:** atMost (Suc k) = insert (Suc k) (atMost k) 
unfolding atMost-def by (auto simp: less-Suc-eq order-le-less)

**lemma UN-atMost-UNIV:** (∪ m :: nat. atMost m) = UNIV 
by blast

61.5.5 The Constant atLeastLessThan

The orientation of the following 2 rules is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside, when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

**lemma atLeast0LessThan [code-abbrev]:** {0 :: nat..<n} = {..<n} 
by (simp add: lessThan-def atLeastLessThan-def)

**lemma atLeast0AtMost [code-abbrev]:** {0..n :: nat} = {..n} 
by (simp add: atMost-def atLeastAtMost-def)

**lemma lessThan-atLeast0:** {..<n} = {0 :: nat..<n} 
by (simp add: atLeast0LessThan)

**lemma atMost-atLeast0:** {..n} = {0 :: nat..n} 
by (simp add: atLeast0AtMost)

**lemma atLeastLessThan0:** {m..<0 :: nat} = {} 
by (simp add: atLeastLessThan-def)

**lemma atLeast0-lessThan-Suc:** {0..<Suc n} = insert n {0..<n} 
by (simp add: atLeast0LessThan lessThan-Suc)
THEORY “Set-Interval”

lemma atLeast0-lessThan-Suc-eq-insert-0: \{0..<\text{Suc } n\} = \text{insert } 0 \{0..<n\}
by (simp add: atLeast0LessThan lessThan-Suc-eq-insert-0)

61.5.6 The Constant atLeastAtMost

lemma Icc-eq-insert-lb-nat: \(m \leq n \Rightarrow \{m..n\} = \text{insert } m \{\text{Suc } m..n\}\)
by auto

lemma atLeast0-atMost-Suc:
\{0..\text{Suc } n\} = \text{insert } (\text{Suc } n) \{0..n\}
by (simp add: atLeast0AtMost atMost-Suc)

lemma atLeast0-atMost-Suc-eq-insert-0:
\{0..\text{Suc } n\} = \text{insert } 0 \{\text{Suc } \{0..n\}\}
by (simp add: atLeast0AtMost atMost-Suc-eq-insert-0)

61.5.7 Intervals of nats with Suc

Not a simp-rule because the RHS is too messy.

lemma atLeastLessThanSuc:
\{m..<\text{Suc } u\} = (\text{if } m \leq n \text{ then insert } n \{m..<n\} \text{ else } \{\})
by (auto simp add: atLeastLessThan-def)

lemma atLeastLessThan-singleton [simp]: \{m..<\text{Suc } m\} = \{m\}
by (auto simp add: atLeastLessThan-def)

lemma atLeastLessThan-atLeastAtMost: \{l..<\text{Suc } u\} = \{l..u\}
by (simp add: lessThan-Suc-atMost atLeastAtMost-def atLeastLessThan-def)

lemma atLeastSucAtMost-greaterThanAtMost: \{\text{Suc } l..u\} = \{l..<u\}
by (simp add: atLeast-Suc-greaterThan atLeastAtMost-def
greaterThanAtMost-def)

lemma atLeastSucLessThan-greaterThanLessThan: \{\text{Suc } l..<u\} = \{l..<u\}
by (simp add: atLeast-Suc-greaterThan atLeastLessThan-def
greaterThanLessThan-def)

lemma atLeastAtMostSuc-conv: \(m \leq \text{Suc } n \Rightarrow \{m..\text{Suc } n\} = \text{insert } (\text{Suc } n) \{m..n\}\)
by auto

lemma atLeastAtMost-insertL: \(m \leq n \Rightarrow \text{insert } m \{\text{Suc } m..n\} = \{m ..n\}\)
by auto

The analogous result is useful on int:

lemma atLeastAtMostPlus1-int-conv:
\(m \leq 1+n \Rightarrow \{m..1+n\} = \text{insert } (1+n) \{m..n::\text{int}\}\)
by (auto intro: set-eqI)
lemma atLeastLessThan-add-Un: \( i \leq j \implies \{ i..<j+k \} = \{ i..<j \} \cup \{ j..<j+k::\text{nat} \} \)
by (induct k) (simp-all add: atLeastLessThanSuc)

61.5.8 Intervals and numerals

lemma lessThan-nat-numeral: — Evaluation for specific numerals
\( \text{lessThan} (\text{numeral } k :: \text{nat}) = \text{insert} (\text{pred-numeral } k) \) (lessThan (pred-numeral k))
by (simp add: numeral-eq-Suc lessThan-Suc)

lemma atMost-nat-numeral: — Evaluation for specific numerals
\( \text{atMost} (\text{numeral } k :: \text{nat}) = \text{insert} (\text{numeral } k) \) (atMost (pred-numeral k))
by (simp add: numeral-eq-Suc atMost-Suc)

lemma atLeastLessThan-nat-numeral: — Evaluation for specific numerals
\( \text{atLeastLessThan} m (\text{numeral } k :: \text{nat}) = \)
if \( m \leq (\text{pred-numeral } k) \) then \( \text{insert} (\text{pred-numeral } k) \) (atLeastLessThan m (pred-numeral k))
else \{\}
by (simp add: numeral-eq-Suc atLeastLessThanSuc)

61.5.9 Image

context linordered-semidom begin

lemma image-add-atLeast[simp]: \( \text{plus } k \cdot \{ i..\} = \{ k + i..\} \)
proof
have \( n = k + (n - k) \) if \( i + k \leq n \) for \( n \)
proof
have \( n = (n - (k + i)) + (k + i) \) using that
by (metis add-commute le-add-diff-inverse)
then show \( n = k + (n - k) \)
by (metis local.add-diff-cancel-left' add-associative add-commute)
qed
then show \(?A \subseteq ?B \)
by (fastforce simp: add-le-imp-le-diff add-commute)
qed

lemma image-add-atLeastAtMost [simp]:
\( \text{plus } k \cdot \{ i..j \} = \{ i + k..j + k \} \) (is \(?A = ?B \))
proof
show \(?A \subseteq ?B \)
by (auto simp add: ac_simps)
next
show \(?B \subseteq ?A \)
proof
fix \( n \)
assume \( n \in ?B \)
then have \( i \leq n - k \)
  by (simp add: add-le-imp-le-diff)

have \( n = n - k + k \)
proof -
  from \( n \in \{B\} \) have \( n = n - (i + k) + (i + k) \)
  by simp
also have \( \ldots = n - k - i + i + k \)
  by (simp add: algebra-simps)
also have \( \ldots = n - k + k \)
  using \( i \leq n - k \) by simp
finally show \( \text{thesis} \).
qed

moreover have \( n - k \in \{i..j\} \)
using \( n \in \{B\} \)
by (auto simp: add-le-imp-le-diff add-le-add-imp-ifff-le)
ultimately show \( n \in \{A\} \)
by (simp add: ac-simps)
qed

end

context ordered-ab-group-add
begin

lemma image-add-atLeastAtMost' [simp]:
  \( \lambda n. n + k \) \( \cdot \{i..j\} = \{i + k..j + k\} \)
  by (simp add: add.commute [of \(- k\)])

lemma image-add-atLeastLessThan [simp]:
  plus \( k \) \( \cdot \{i..<j\} = \{i + k..<j + k\} \)
  by (simp add: image-set-diff atLeastLessThan-eq-atLeastAtMost-diff ac-simps)

lemma image-add-atLeastLessThan' [simp]:
  \( \lambda n. n + k \) \( \cdot \{i..<j\} = \{i + k..<j + k\} \)
  by (simp add: add.commute [of \(- k\)])

lemma image-add-greaterThanAtMost [simp]:
  \( (+) c \cdot \{a<..<b\} = \{c + a<..<c + b\} \)
  by (simp add: image-set-diff greaterThanAtMost-eq-atLeastAtMost-diff ac-simps)

end

context ordered-ab-group-add
begin

lemma
  fixes \( x \) :: 'a
  shows image-uminus-greaterThan [simp]: \( \text{uminus} \cdot \{x<..\} = \{..<-x\} \)
  and image-uminus-atLeast [simp]: \( \text{uminus} \cdot \{x..\} = \{..<-x\} \)
proof safe
  fix \( y \) assume \( y < -x \)
  hence \( x < -y \) using neg-less-iff-less[of \(- y\) \( x\)] by simp
  have \( - (y) \in \text{uminus} \cdot \{x<..\} \)
    by (rule image1) (simp add: *)
thus \( y \in \uminus \{ x <. \} \) by simp

next
fix \( y \) assume \( y \leq -x \)

have \(-y \in \uminus \{ x. \}\) by (rule imageI) (use \( y \leq -x \) in simp)

thus \( y \in \uminus \{ x. \} \) by simp

qed simp-all

lemma
fixes \( x :: 'a \)
shows image-uminus-lessThan[simp]: \( \uminus \{ x <. \} = \{ -x <. \} \)
and image-uminus-atMost[simp]: \( \uminus \{ x. \} = \{ -x. \} \)

proof –

have \( \uminus \{ x <. \} = \uminus \{ -x <. \} \)
and \( \uminus \{ x. \} = \uminus \{ -x. \} \) by simp-all

thus \( \uminus \{ x <. \} = \{ -x <. \} \) and \( \uminus \{ x. \} = \{ -x. \} \)

by (simp-all add: image-image
del: image-uminus-greaterThan image-uminus-atLeast)

qed

lemma
fixes \( x :: 'a \)
shows image-uminus-atLeastAtMost[simp]: \( \uminus \{ x..y \} = \{ -y..-x \} \)
and image-uminus-greaterThanAtMost[simp]: \( \uminus \{ x<y..y \} = \{ -y<y..-x \} \)
and image-uminus-greaterThanAtLeast[simp]: \( \uminus \{ x<y..y \} = \{ -y<y..-x \} \)

by (simp-all add: atLeastAtMost-def greaterThanAtMost-def atLeastLessThan-def greaterThanLessThan-def image-Int[OF inj-uminus] Int-commute)

lemma image-add-atMost[simp]: \( (+) \ c \ {..a} = \{..c + a \} \)

by (auto intro!: image-eqI [where \( x=x - c \) for \( x \)] simp: algebra-simps)

end

lemma image-Suc-atLeastAtMost [simp]:
\( \text{Suc } \{ i..j \} = \text{Suc } i..\text{Suc } j \)

using image-add-atLeastAtMost [of 1 i j]

by (simp only: plus-1-eq-Suc) simp

lemma image-Suc-atLeastLessThan [simp]:
\( \text{Suc } \{ i..<j \} = \text{Suc } i..<\text{Suc } j \)

using image-add-atLeastLessThan [of 1 i j]

by (simp only: plus-1-eq-Suc) simp

corollary image-Suc-atMost:
\( \text{Suc } \{ ..n \} = \{ 1..\text{Suc } n \} \)

by (simp add: atMost-atLeast0 atLeastLessThanSuc-atLeastAtMost)

corollary image-Suc-lessThan:
Suc \{..<n\} = \{1..n\} by (simp add: lessThan-atLeast0 atLeastLessThanSuc-atLeastAtMost)

lemma image-diff-atLeastAtMost [simp]:
  fixes d :: 'a::linordered-idom
  shows \((-\) d \{a..b\}) = \{d-b..d-a\}
proof
  show \{d - b..d - a\} \subseteq \(-\) d \{a..b\}
  proof
    fix x
    assume x \in \{d - b..d - a\}
    then have d - x \in \{a..b\} and x = d - (d - x)
      by auto
    then show x \in \(-\) d \{a..b\}
      by (rule rev-image-eqI)
  qed
qed

lemma image-diff-atLeastLessThan [simp]:
  fixes a b c :: 'a::linordered-idom
  shows \((-\) c \{a..<b\}) = \{c-b..<c-a\}
proof
  have \((-\) c \{a..<b\}) = (+) c \{a..<b\}
    unfolding image-image by simp
  also have \ldots = \{c-b..<c-a\} by simp
  finally show \?thesis by simp
qed

lemma image-minus-const-greaterThanAtMost[simp]:
  fixes a b c :: 'a::linordered-idom
  shows \((-\) c \{a..<\}) = \{c-b..<c-a\}
proof
  have \((-\) c \{a..<\}) = (+) c \{a..<\}
    unfolding image-image by simp
  also have \ldots = \{c-b..<c-a\} by simp
  finally show \?thesis by simp
qed

lemma image-minus-const-atLeast[simp]:
  fixes a :: 'a::linordered-idom
  shows \((-\) c \{a..\}) = \{..c-a\}
proof
  have \((-\) c \{a..\}) = (+) c \{a..\}
    unfolding image-image by simp
  also have \ldots = \{..c-a\} by simp
  finally show \?thesis by simp
qed

lemma image-minus-const-AtMost[simp]:
  fixes b :: 'a::linordered-idom
  shows \((-\) b \{..\}) = \{..c-b\}
shows \((-) c \cdot \{..b\} = \{c - b..\}

proof -
  have \((-) c \cdot \{..b\} = (+) c \cdot \uminus \cdot \{..b\}
    unfolding image-image by simp
  also have \ldots = \{c - b..\} by simp
  finally show \textit{thesis} by simp
qed

lemma \textit{image-minus-const-atLeastAtMost'} [simp]:
  \((\lambda t. t - d)\{a..b\} = \{a - d..b - d\}\) for \(d::\text{linordered-idom} \)
  by (metis (no-types, lifting) diff-conv-add-uminus image-add-atLeastAtMost' image-cong)

context \textit{linordered-field}
begin

lemma \textit{image-mult-atLeastAtMost} [simp]:
  \((\ast) d \cdot \{a..b\} = \{d \cdot a..d \cdot b\}\) if \(d > 0\)
  using that
  by (auto simp: field-simps mult-le-cancel-right intro: rev-image-eqI [where \(x=x/d\) for \(x\)])

lemma \textit{image-divide-atLeastAtMost} [simp]:
  \((\lambda c. c / d) \cdot \{a..b\} = \{a / d..b / d\}\) if \(d > 0\)
  proof -
    from that have inverse \(d > 0\)
    by simp
    with \textit{image-mult-atLeastAtMost} [of inverse \(d\) \(a\) \(b\)]
    have \((\ast)\) (inverse \(d\)) \cdot \{a..b\} = \{inverse \(d\) \cdot a..inverse \(d\) \cdot b\}
    by blast
    moreover have \((\ast)\) (inverse \(d\)) = \((\lambda c. c / d)\)
    by (simp add: fun-eq-iff field-simps)
    ultimately show \textit{thesis}
    by simp
  qed

lemma \textit{image-mult-atLeastAtMost-if}:
  \((\ast)\) \(c \cdot \{x..y\} = \)
  \(\) (if \(c > 0\) then \{c \cdot x .. c \cdot y\} else if \(x \leq y\) then \{c \cdot y .. c \cdot x\} else \{\})
  proof (cases \(c = 0 \lor x > y\))
    case True
    then show \textit{thesis}
    by auto
  next
    case False
    then have \(x \leq y\)
    by auto
    from False consider \(c < 0\) \(c > 0\)
    by (auto simp add: neq_iff)
then show \( ? \)thesis
proof cases
  case 1
  have \((\ast)\) \( c \cdot \{x..y\} = \{c \cdot y..c \cdot x\}\)
  proof (rule set-eqI)
    fix \( d \)
    from 1 have \( \text{inj} (\lambda z. z / c) \)
      by (auto intro: injI)
    then have \( d \in (\ast) \ c \cdot \{x..y\} \iff d / c \in (\lambda z. z \div c) \cdot (\ast) \ c \cdot \{x..y\} \)
      by (subst inj-image-mem-iff) simp-all
    also have \( \ldots \iff d / c \in \{x..y\} \)
      using 1 by (simp add: image-image)
    also have \( \ldots \iff d \in \{c \cdot y..c \cdot x\} \)
      by (auto simp add: field-simps 1)
    finally show \( d \in (\ast) \ c \cdot \{x..y\} \iff d \in \{c \cdot y..c \cdot x\} \).
      qed
  qed (simp add: mult-left-mono-neg)
  qed

lemma \texttt{image-affinity-atLeastAtMost-if'}:
\((\lambda x. x \cdot c) \cdot \{x..y\} = \)
  \((\text{if } x \leq y \text{ then if } c > 0 \text{ then } \{x \cdot c..y \cdot c\} \text{ else } \{y \cdot c..x \cdot c\} \text{ else } \})\)
using \texttt{image-affinity-atLeastAtMost-if [of c x y]} by (auto simp add: ac-simps)

lemma \texttt{image-affinity-atLeastAtMost}:
\((\lambda x. \, m \cdot x + c) \cdot \{a..b\} = \)
  \((\text{if } \{a..b\} = \{} \text{ then } \{} \)
  \hspace{0.5cm} else if \( 0 \leq m \) then \( \{m \cdot a + c..m \cdot b + c\} \)
  \hspace{0.5cm} else \( \{m \cdot b + c..m \cdot a + c\} \})
proof –
  have \( \ast : (\lambda x. \, m \cdot x + c) = ((\lambda x. x + c) \circ (\ast) \ m) \)
    by (simp add: fun-eq-iff)
  show ?thesis by (simp only: \* image-comp [symmetric] image-affinity-atLeastAtMost-if)
    (auto simp add: mult-le-cancel-left)
  qed

lemma \texttt{image-affinity-atLeastAtMost-diff}:
\((\lambda x. \, m \cdot x - c) \cdot \{a..b\} = \)
  \((\text{if } \{a..b\} = \{} \text{ then } \{} \)
  \hspace{0.5cm} else if \( 0 \leq m \) then \( \{m \cdot a - c..m \cdot b - c\} \)
  \hspace{0.5cm} else \( \{m \cdot b - c..m \cdot a - c\} \})
using \texttt{image-affinity-atLeastAtMost [of m -c a b]}
by simp

lemma \texttt{image-affinity-atLeastAtMost-div}:
\((\lambda x. \, x / m + c) \cdot \{a..b\} = \)
  \((\text{if } \{a..b\} = \{} \text{ then } \{} \)
  \hspace{0.5cm} else if \( 0 \leq m \) then \( \{a / m + c..b / m + c\} \)
  \hspace{0.5cm} else \( \{b / m + c..a / m + c\} \})
using \texttt{image-affinity-atLeastAtMost [of inverse m c a b]}

THEORY “Set-Interval”
THEORY “Set-Interval”

by (simp add: field-class.field-divide-inverse algebra-simps inverse-eq-divide)

lemma image-affinity-atLeastAtMost-div-diff:
((λx. x/m − c) ‘ {a..b}) = (if {a..b}={} then {} else if 0 ≤ m then {a/m − c .. b/m − c} else {b/m − c .. a/m − c})

using image-affinity-atLeastAtMost-div-diff [of inverse m c a b]
by (simp add: field-class.field-divide-inverse algebra-simps inverse-eq-divide)

end

lemma atLeast1-lessThan-eq-remove0:
{Suc 0..<n} = {..<n} − {0}
by auto

lemma atLeast1-atMost-eq-remove0:
{Suc 0..<n} = {..<n} − {0}
by auto

lemma image-add-int-atLeastLessThan:
(λx. x + (l::int)) ‘ {0..<u−l} = {l..<u}
by safe auto

lemma image-minus-const-atLeastLessThan-nat:
fixes c :: nat
shows (λi. i − c) ‘ {x..<y} = (if c < y then {x − c..<y − c} else if x < y then {0} else {}}
(is - = ?right)

proof safe
fix a assume a: a ∈ ?right
show a ∈ (λi. i − c) ‘ {x..<y}
proof cases
assume c < y with a show ?thesis
by (auto intro!: image-eqI[of - a + c])

next
assume ¬ c < y with a show ?thesis
by (auto intro!: image-eqI[of - x] split: if-split_asm)

qed

qed auto

lemma image-int-atLeastLessThan:
int ‘ {a..<b} = {int a..<int b}
by (auto intro!: image-eqI [where x = nat x for x])

lemma image-int-atLeastAtMost:
int ‘ {a..b} = {int a..int b}
by (auto intro!: image-eqI [where x = nat x for x])
61.5.10 Finiteness

**lemma finite-lessThan [iff]:** fixes k :: nat shows finite {..<k}
by (induct k) (simp-all add: lessThan-Suc)

**lemma finite-atMost [iff]:** fixes k :: nat shows finite {..<k}
by (induct k) (simp-all add: atMost-Suc)

**lemma finite-greaterThanLessThan [iff]:**
fixes l :: nat shows finite {l..<u}
by (simp add: greaterThanLessThan-def)

**lemma finite-atLeastLessThan [iff]:**
fixes l :: nat shows finite {l..<u}
by (simp add: atLeastLessThan-def)

**lemma finite-greaterThanAtMost [iff]:**
fixes l :: nat shows finite {l..<u}
by (simp add: greaterThanAtMost-def)

**lemma finite-atLeastAtMost [iff]:**
fixes l :: nat shows finite {l..u}
by (simp add: atLeastAtMost-def)

A bounded set of natural numbers is finite.

**lemma bounded-nat-set-is-finite:**
(∀ i∈N. i < (n::nat)) ⇒ finite N
by (rule finite-subset [OF finite-lessThan]) auto

A set of natural numbers is finite iff it is bounded.

**lemma finite-nat-set-iff-bounded:**
finite(N::nat set) = (∃ m. ∀ n∈N. n<m) (is ?F = ?B)
proof
assume f:?F show ?B
  using Max-ge[OF ?F, simplified less-Suc-eq-le[symmetric]] by blast
next
qed

**lemma finite-nat-set-iff-bounded-le:**
finite(N::nat set) = (∃ m. ∀ n∈N. n≤m)
unfolding finite-nat-set-iff-bounded
by (blast dest:less-imp-le-nat le-imp-less-Suc)

**lemma finite-less-ub:**
(∃ f::nat⇒nat. (∀ n. n ≤ f n) ⇒ finite {n. f n ≤ u})
by (rule finite-subset[of - {..u}])
(auto intro: order-trans)

**lemma bounded-Max-nat:**
fixes P :: nat ⇒ bool
assumes x: P x and M: ∃ x. P x ⇒ x ≤ M
obtains \( m \) where \( P \) holds for all \( x \) such that \( P x \Rightarrow x \leq m \)

**proof**

- have finite \( \{ x. P x \} \)
  - using \( M \) finite-nat-set-iff-bounded-le by auto

then have \( \operatorname{Max} \{ x. P x \} \in \{ x. P x \} \)

then show \?thesis
  - by (simp add: finite \{ x. P x \}\ that)

**qed**

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

**lemma** subset-card-intvl-is-intvl:

assumes \( A \subseteq \{ k..< k + \operatorname{card} A \} \)

shows \( A = \{ k..< k + \operatorname{card} A \} \)

**proof** (cases finite \( A \))

- case \( True \)
  - from this and assms show \?thesis

- case \( False \)
  - with assms show \?thesis by simp

**qed**

### 61.5.11 Proving Inclusions and Equalities between Unions

**lemma** UN-le-eq-Un0:

\( (\bigcup_{i \leq n \cdot \text{nat}}. M \ i) = (\bigcup_{i \in \{1..n\}}. M \ i) \cup M \ \emptyset \) (is \(?A = ?B\))

**proof**

show \(?A \subseteq ?B\)

proof
  fix \( x \) assume \( x \in ?A \)
  then obtain \( i \) where \( i \leq n \cdot x \in M \ i \) by auto
  show \( x \in ?B \)
  proof (cases \( i \))
    - case \( 0 \) with \( i \) show \?thesis by simp
    - case \( \text{Suc } j \) with \( i \) show \?thesis by auto
  qed

**qed**

next
```

show ?B ⊆ ?A by fastforce
qed

lemma UN-le-add-shift:
(⋃i≤n::nat. M(i+k)) = (⋃i∈{k..n+k}. M i) (is ?A = ?B)
proof
  show ?A ⊆ ?B by fastforce
next
  show ?B ⊆ ?A
    proof
      fix x assume x ∈ ?B
      then obtain i where i: i ∈ {k..n+k} x ∈ M(i) by auto
      hence i − k ≤ n ∧ x ∈ M((i − k) + k) by auto
      thus x ∈ ?A by blast
    qed
qed

lemma UN-le-add-shift-strict:
(⋃i<n::nat. M(i+k)) = (⋃i∈{k..<n+k}. M i)
proof
  show ?B ⊆ ?A
    proof
      fix x assume x ∈ ?B
      then obtain i where i: i ∈ {k..<n+k} x ∈ M(i) by auto
      then have i − k < n ∧ x ∈ M((i − k) + k) by auto
      then show x ∈ ?A using UN-le-add-shift by blast
    qed
qed

lemma UN-UN-finite-eq:
(⋃n::nat. ⋃i∈{0..<n}. A i) = (⋃n. A n)
by (auto simp add: atLeast0LessThan)

lemma UN-finite-subset:
(∀n::nat. (⋃i∈{0..<n}. A i) ⊆ C) ⇒ (⋃n. A n) ⊆ C
by (subst UN-UN-finite-eq [symmetric]) blast

lemma UN-finite2-subset:
assumes (∀n::nat. (⋃i∈{0..<n}. A i) ⊆ (⋃i∈{0..<n+k}. B i))
shows (⋃n. A n) ⊆ (⋃n. B n)
proof (rule UN-finite-subset, rule subsetI)
  fix n and a
  from assms have (⋃i∈{0..<n}. A i) ⊆ (⋃i∈{0..<n+k}. B i) .
  moreover assume a ∈ (⋃i∈{0..<n}. A i)
  ultimately have a ∈ (⋃i∈{0..<n+k}. B i) by blast
  then show a ∈ (⋃n. B i) by (auto simp add: UN-UN-finite-eq)
qed

lemma UN-finite2-eq:
assumes (∀n::nat. (⋃i∈{0..<n}. A i) = (⋃i∈{0..<n+k}. B i))
```
shows \((\bigcup n \ A n) = (\bigcup n \ B n)\)

proof (rule subset-antisym [OF UN-finite-subset UN-finite2-subset])

- fix \(n\)
  - show \(\bigcup (A ^\prime \{0..<n\}) \subseteq (\bigcup n \ B n)\)
    - using assms by auto
  next
    - fix \(n\)
      - show \(\bigcup (B ^\prime \{0..<n\}) \subseteq \bigcup (A ^\prime \{0..<n + k\})\)
        - using assms by (force simp add: atLeastLessThan-add-Un [of 0])
  qed

61.5.12 Cardinality

lemma card-lessThan [simp]: \(\text{card} \{..<u\} = u\)
  by (induct u, simp-all add: lessThan-Suc)

lemma card-atMost [simp]: \(\text{card} \{..u\} = \text{Suc} \ u\)
  by (simp add: lessThan-Suc-atMost [THEN sym])

lemma card-atLeastLessThan [simp]: \(\text{card} \{l..<u\} = u - l\)
  proof
    - have \((\lambda x. x + l) \ ^\prime \{..<u - l\} \subseteq \{l..<u\}\)
      - by auto
    moreover have \(\{l..<u\} \subseteq (\lambda x. x + l) \ ^\prime \{..<u - l\}\)
      - proof
        - fix \(x\)
          - assume \(\ast\): \(x \in \{l..<u\}\)
          - then have \(x - l \in \{..<u - l\}\)
            - by auto
          - then have \((x - l) + l \in (\lambda x. x + l) \ ^\prime \{..<u - l\}\)
            - by auto
          - then show \(x \in (\lambda x. x + l) \ ^\prime \{..<u - l\}\)
            - using \(\ast\) by auto
        qed
    - ultimately have \(\{l..<u\} = (\lambda x. x + l) \ ^\prime \{..<u - l\}\)
      - by auto
    - then have \(\text{card} \{l..<u\} = \text{card} \{..<u - l\}\)
      - by (simp add: card-image inj-on-def)
    - then show \(?thesis\)
      - by simp
  qed

lemma card-atLeastAtMost [simp]: \(\text{card} \{l..u\} = \text{Suc} \ u - l\)
  by (subst atLeastLessThanSuc-atLeastAtMost [THEN sym], simp)

lemma card-greaterThanAtMost [simp]: \(\text{card} \{l<..u\} = u - l\)
  by (subst atLeastSucAtMost-greaterThanAtMost [THEN sym], simp)

lemma card-greaterThanLessThan [simp]: \(\text{card} \{l<..<u\} = u - \text{Suc} \ l\)
  by (subst atLeastLessThanSuc-atLeastLessThan [THEN sym], simp)
by (subst atLeastSucLessThan-greaterThanLessThan [THEN sym], simp)

lemma subset-eq-atLeast0-lessThan-finite:
  fixes n :: nat
  assumes N ⊆ {0..<n}
  shows finite N
  using assms finite-atLeastLessThan by (rule finite-subset)

lemma subset-eq-atLeast0-atMost-finite:
  fixes n :: nat
  assumes N ⊆ {0..n}
  shows finite N
  using assms finite-atLeastAtMost by (rule finite-subset)

lemma ex-bij-betw-nat-finite:
  finite M =⇒ ∃ h. bij-betw h {0..<card M} M
  apply(drule finite-imp-nat-seg-image-inj-on)
  apply(auto simp:atLeast0LessThan[symmetric] lessThan-def[symmetric] card-image bij-betw-def)
  done

lemma ex-bij-betw-finite-nat:
  finite M =⇒ ∃ h. bij-betw h M {0..<card M}
  by (blast dest: ex-bij-betw-nat-finite bij-betw-inv)

lemma finite-same-card-bij:
  finite A =⇒ finite B =⇒ card A = card B =⇒ ∃ h. bij-betw h A B
  apply(drule ex-bij-betw-nat-finite)
  apply(drule ex-bij-betw-nat-finite)
  apply(auto intro!:bij-betw-trans)
  done

lemma ex-bij-betw-nat-finite-1:
  finite M =⇒ ∃ h. bij-betw h {1..<card M} M
  by (rule finite-same-card-bij auto)

lemma bij-betw-iff-card:
  assumes finite A finite B
  shows (∃ f. bij-betw f A B) ⇔ (card A = card B)
proof
  assume card A = card B
  moreover obtain f where bij-betw f A {0..< card A}
    using assms ex-bij-betw-nat-finite by blast
  moreover obtain g where bij-betw g {0..< card B} B
    using assms ex-bij-betw-nat-finite by blast
  ultimately have bij-betw (g ∘ f) A B
    by (auto simp: bij-betw-trans)
  thus (∃ f. bij-betw f A B) by blast
qed (auto simp: bij-betw-same-card)
lemma \textit{subset-eq-atLeast0-lessThan-card}:
  \begin{itemize}
  \item \texttt{proof}
  \item \texttt{fixes} \texttt{n :: nat} 
  \item \texttt{assumes} \texttt{N \subseteq \{0..<n\}} 
  \item \texttt{shows} \texttt{card N \leq n} 
  \item \texttt{proof} –
  \item \texttt{from asms finite-lessThan have card N \leq \text{card} \{0..<n\}} 
  \item \texttt{using card-mono by blast} 
  \item \texttt{then show ?thesis by simp} 
  \item \texttt{qed} 
  \end{itemize}

Relational version of \textit{card-inj-on-le}:

\begin{itemize}
  \item \texttt{lemma \text{card-le-if-inj-on-rel}}:
  \item \texttt{assumes} \texttt{finite B} 
  \item \texttt{\exists a, a \in A \implies \exists b, b \in B \land r a b} 
  \item \texttt{\exists a1 a2 b. [ a1 \in A; a2 \in A; b \in B; r a1 b; r a2 b ] \implies a1 = a2} 
  \item \texttt{shows} \texttt{card A \leq card B} 
  \item \texttt{proof} –
  \item \texttt{let} \texttt{?P = \lambda a b, b \in B \land r a b} 
  \item \texttt{let} \texttt{?f = \lambda a. SOME b. ?P a b} 
  \item \texttt{have 1: \textit{?f A \subseteq B} by (auto intro: someI2-ex[OF asms(2)])} 
  \item \texttt{have inj-on \textit{?f A}} 
  \item \texttt{unfolding inj-on-def} 
  \item \texttt{proof safe} 
  \item \texttt{fix a1 a2 assume asms: a1 \in A a2 \in A \textit{?f a1 = ?f a2}} 
  \item \texttt{have 0: \textit{?f a1 \in B} using 1 \textit{a1 \in A} by blast} 
  \item \texttt{have 1: r a1 (\textit{?f a1}) using someI-ex[OF asms(2)][OF \textit{a1 \in A}] by blast} 
  \item \texttt{have 2: r a2 (\textit{?f a1}) using someI-ex[OF asms(2)][OF \textit{a2 \in A}] asms(3) by auto} 
  \item \texttt{show a1 = a2 using asms(3)[OF asms(1,2) 0 1 2].} 
  \item \texttt{qed} 
  \item \texttt{with 1 show ?thesis using card-inj-on-le[OF \textit{?f A B} asms(1)] by simp} 
  \item \texttt{qed} 
  \end{itemize}

\begin{itemize}
  \item \texttt{lemma \textit{inj-on-funpow-least}:} 
  \item \texttt{\langle f \cdots k \rangle s \{0..<n\}} 
  \item \texttt{\langle f \cdots n \rangle s = s \langle \forall m. 0 < m \implies m < n \implies (f \cdots m) s \neq s \rangle} 
  \item \texttt{proof} –
  \item \texttt{\{ fix k l assume A: k < n l < n k \neq l (f \cdots k) s = (f \cdots l) s} 
  \item \texttt{define k' l' where k' = min k l and l' = max k l} 
  \item \texttt{with A have A': k' < l' (f \cdots k') s = (f \cdots l') s l' < n} 
  \item \texttt{by (auto simp: min-def max-def)} 
  \item \texttt{have s = (f \cdots ((n - l') + l')) s using that l' < n by simp} 
  \item \texttt{also have \ldots = (f \cdots (n - l')) (f \cdots l') s by (simp add: funpow-add)} 
  \item \texttt{also have \langle f \cdots l' \rangle s = (f \cdots k') s by (simp add: A')} 
  \item \texttt{also have \langle f \cdots (n - l') \rangle \ldots = (f \cdots (n - l' + k')) s by (simp add: funpow-add)} 
  \item \texttt{finally have \langle f \cdots (n - l' + k') \rangle s = s by simp} 
  \item \texttt{moreover have n - l' + k' < n 0 < n - l' + k' using A' by linarith+} 
  \end{itemize}
ultimately have False using that(2) by auto
}
then show ?thesis by (intro inj-onI) auto
qed

61.6  Intervals of integers

lemma atLeastLessThanPlusOne-atLeastAtMost-int: \{l..<\u+1\} = \{l..\u::int\}
  by (auto simp add: atLeastAtMost-def atLeastLessThan-def)

lemma atLeastPlusOneAtMost-greaterThanAtMost-int: \{l+1..\u\} = \{\<\u::int\}
  by (auto simp add: atLeastAtMost-def greaterThanAtMost-def)

lemma atLeastPlusOneLessThan-greaterThanLessThan-int:
  \{l+1..<\u\} = \{\<\u::int\}
  by (auto simp add: atLeastLessThan-def greaterThanLessThan-def)

61.6.1  Finiteness

lemma image-atLeastZeroLessThan-int:
  assumes 0 \leq \u
  shows \{(0::int)..\u\} = int ' \{\<\nat \u\}
  unfolding image-def lessThan-def
proof
  show \{0..<\u\} \subseteq \{y. \\\exists x\{x. x < \nat \u\}. y = int x\}
    by (auto simp add: zless-nat-eq-int-zless [THEN sym])
proof
  assume \(x \in \{0..<\u\}
  then have \(x = int \\(nat x\) \text{ and } \nat x < \nat \u\)
    by (auto simp add: zless-nat-eq-int-zless [THEN sym])
  then have \(\\exists x\{x. x < \nat \u\}. x = int x\)
    using \(\\times \{\times \\}\) by simp
  then show \(x \in \{y. \\exists x\{x. x < \nat \u\}. y = int x\}\)
    by simp
qed
qed (auto)

lemma finite-atLeastZeroLessThan-int: finite \{(0::int)..\u\}
proof (cases 0 \leq \u)
  case True
  then show ?thesis
    by (auto simp: image-atLeastZeroLessThan-int)
qed auto

lemma finite-atLeastLessThan-int [iff]: finite \{l..<\u::int\}
  by (simp only: image-add-int-atLeastLessThan [symmetric, af l] finite-imageI
    finite-atLeastZeroLessThan-int)

lemma finite-atLeastAtMost-int [iff]: finite \{l..\u::int\}

THEORY “Set-Interval”

by (subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym], simp)

lemma finite-greaterThanAtMost-int [iff]: finite {l..<u::int}
  by (subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp)

lemma finite-greaterThanLessThan-int [iff]: finite {l..<u::int}
  by (subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp)

61.6.2 Cardinality

lemma card-atLeastZeroLessThan-int [simp]: card {0..<u} = nat u
  proof (cases 0 ≤ u)
    case True
    then show ?thesis
      by (auto simp: image-atLeastZeroLessThan-int card-image inj-on-def)
  qed

lemma card-atLeastLessThan-int [simp]: card {l..<u} = nat (u - l)
  proof
    have card {l..<u} = card {0..<u - l}
      apply (subst image-add-int-atLeastLessThan [symmetric])
      apply (rule card-image)
      apply (simp add: inj-on-def)
    done
    then show ?thesis
      by (simp add: card-atLeastZeroLessThan-int)
  qed

lemma card-atLeastAtMost-int [simp]: card {l..u} = nat (u - l)
  apply (subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym])
  apply (auto simp add: algebra-simps)
  done

lemma card-greaterThanAtMost-int [simp]: card {l..<u} = nat (u - (l + 1))
  by (subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp)

lemma finite-M-bounded-by-nat: finite {k. P k ∧ k < (i::nat)}
  proof
    have {k. P k ∧ k < i} ⊆ {..<i} by auto
    with finite-lessThan[of i] show ?thesis by (simp add: finite-subset)
  qed

lemma card-less:
  assumes zero-in-M: 0 ∈ M
  shows card {k ∈ M. k < Suc i} ≠ 0
  proof —
from zero-in-M have \( \{ k \in M. k < \text{Suc } i \} \neq \{ \} \) by auto

with finite-M-bounded-by-nat show \(?\text{thesis}\) by (auto simp add: card-eq-0-iff)

qed

lemma card-less-Suc2:
assumes \( \emptyset \notin M \) shows \( \text{card } \{ k. \text{Suc } k \in M \land k < i \} = \text{card } \{ k \in M. k < \text{Suc } i \} \)

proof –
  have \( [[ j \in M; j < \text{Suc } i ]] \implies j - \text{Suc } 0 < i \land \text{Suc } (j - \text{Suc } 0) \in M \land \text{Suc } 0 \leq j \) for \( j \) by (cases \( j \)) (use assms in auto)
  show \(?\text{thesis}\) proof (rule card-bij-eq)
    show inj-on Suc \( \{ k. \text{Suc } k \in M \land k < i \} \)
      by force
    show inj-on (\( \lambda x. x - \text{Suc } 0 \)) \( \{ k \in M. k < \text{Suc } i \} \)
      by (rule inj-on-diff-nat) (use \( \star \) in blast)
  qed (use \( \star \) in auto)

qed

lemma card-less-Suc:
assumes \( \emptyset \in M \)
shows \( \text{Suc } (\text{card } \{ k. \text{Suc } k \in M \land k < i \}) = \text{card } \{ k \in M. k < \text{Suc } i \} \)

proof –
  have \( \ldots = \text{Suc } (\text{card } \{ k \in M - \{ \emptyset \}. k < \text{Suc } i \}) \)
  apply (subt card-less-Suc2)
  using assms by auto
  also have \( \ldots = \text{Suc } (\text{card } ([k \in M. k < \text{Suc } i] - \{ \emptyset \})) \)
  by (force intro: arg-cong [where \( f=\text{card} \)]
  also have \( \ldots = \text{card } (\text{insert } 0 ([k \in M. k < \text{Suc } i] - \{ \emptyset \})) \)
  by (simp add: card.insert-remove)
  also have \( \ldots = \text{card } \{ k \in M. k < \text{Suc } i \} \)
  using assms by (force simp add: intro: arg-cong [where \( f=\text{card} \)]
  finally show \(?\text{thesis}\).

qed

lemma card-le-Suc-Max: finite \( S \implies \text{card } S \leq \text{Suc } (\text{Max } S) \)
proof (rule classical)
  assume finite \( S \) and \( \neg \text{Suc } (\text{Max } S) \geq \text{card } S \)
  then have \( \text{Suc } (\text{Max } S) < \text{card } S \)
  by simp
  with \( \text{finite } S \) have \( S \subseteq \{ 0..\text{Max } S \} \)
  by auto
  hence \( \text{card } S \leq \text{card } \{ 0..\text{Max } S \} \)
  by (intro card-mono; auto)
thus \( \text{card } S \leq \text{Suc } (\text{Max } S) \)
by simp
qed

61.7 Lemmas useful with the summation operator sum

For examples, see Algebra/poly/UnivPoly2.thy

61.7.1 Disjoint Unions

Singletons and open intervals

lemma \texttt{isl-disj-un-singleton}:
\begin{align*}
\{l: a: \text{linorder}\} \cup \{l < \} &= \{l, ..\} \\
\{..u\} \cup \{u, '\}: \text{linorder}\} &= \{..u\} \\
(l: a: \text{linorder}) < u \Longrightarrow \{l\} \cup \{l <..u\} &= \{l < u\} \\
(l: a: \text{linorder}) < u \Longrightarrow \{l<..u\} \cup \{u\} &= \{l <..u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{l <..u\} \cup \{u, u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{l <..u\} \cup \{u\} &= \{l <..u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{l <..u\} \cup \{u, u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{l <..u\} \cup \{u\} &= \{l <..u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{l <..u\} \cup \{u, u\} &= \{l < u\} \\
by auto
\end{align*}

One- and two-sided intervals

lemma \texttt{isl-disj-un-one}:
\begin{align*}
(l: a: \text{linorder}) < u \Longrightarrow \{..l\} \cup \{l <..u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{..l\} \cup \{l <..u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{..l\} \cup \{l < u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{..l\} \cup \{l < u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{..l\} \cup \{l < u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{..l\} \cup \{l < u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq u \Longrightarrow \{..l\} \cup \{l <..u\} &= \{l < u\} \\
by auto
\end{align*}

Two- and two-sided intervals

lemma \texttt{isl-disj-un-two}:
\begin{align*}
(l: a: \text{linorder}) < m, m \leq u \Longrightarrow \{l <..m\} \cup \{m <..u\} &= \{l <..u\} \\
(l: a: \text{linorder}) \leq m, m < u \Longrightarrow \{l <..m\} \cup \{m <..u\} &= \{l <..u\} \\
(l: a: \text{linorder}) \leq m, m \leq u \Longrightarrow \{l <..m\} \cup \{m < u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq m, m < u \Longrightarrow \{l <..m\} \cup \{m < u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq m, m \leq u \Longrightarrow \{l <..m\} \cup \{m < u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq m, m < u \Longrightarrow \{l <..m\} \cup \{m < u\} &= \{l < u\} \\
(l: a: \text{linorder}) \leq m, m \leq u \Longrightarrow \{l <..m\} \cup \{m <..u\} &= \{l < u\} \\
by auto
\end{align*}

lemma \texttt{isl-disj-un-two-touch}:
\begin{align*}
(l: a: \text{linorder}) < m, m < u \Longrightarrow \{l <..m\} \cup \{m <..u\} &= \{l <..u\} \\
(l: a: \text{linorder}) \leq m, m < u \Longrightarrow \{l <..m\} \cup \{m <..u\} &= \{l <..u\} \\
(l: a: \text{linorder}) \leq m, m \leq u \Longrightarrow \{l <..m\} \cup \{m <..u\} &= \{l <..u\} \\
by auto
\end{align*}
THEORY "Set-Interval"

\[ (l::'a::linorder) \leq m; m \leq u \implies \{l..m\} \cup \{m..u\} = \{l..u\} \]
by auto

lemmas ivl-disj-un = ivl-disj-un-singleton ivl-disj-un-one ivl-disj-un-two ivl-disj-un-two-touch

61.7.2 Disjoint Intersections

One- and two-sided intervals

lemma ivl-disj-int-one:
\{..l::'a::order\} \inter \{l..<u\} = \{}
\{..<l\} \inter \{l..<u\} = \{}
\{..<l\} \inter \{l..<u\} = \{}
\{l..<u\} \inter \{u..<\} = \{}
\{l..<u\} \inter \{u..<\} = \{}
\{l..<u\} \inter \{..<u\} = \{}
by auto

Two- and two-sided intervals

lemma ivl-disj-int-two:
\{l::'a::order..<m\} \inter \{m..<u\} = \{}
\{l..<m\} \inter \{m..<u\} = \{}
\{l..<m\} \inter \{m..<u\} = \{}
\{l..<m\} \inter \{m..<u\} = \{}
\{l..<m\} \inter \{m..<u\} = \{}
\{l..<m\} \inter \{m..<u\} = \{}
\{l..<m\} \inter \{m..<u\} = \{}
by auto

lemmas ivl-disj-int = ivl-disj-int-one ivl-disj-int-two

61.7.3 Some Differences

lemma ivl-diff[simp]:
i \leq n \implies \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}
by(auto)

lemma (in linorder) lessThan-minus-lessThan [simp]:
\{..<n\} - \{..<m\} = \{m..<n\}
by auto

lemma (in linorder) atLeastAtMost-diff-ends:
\{a..b\} - \{a, b\} = \{a..<b\}
by auto
61.7.4 Some Subset Conditions

lemma `iid-subset [simp]: `{i..<j} ⊆ {m..<n}` = `{j ≤ i ∨ m ≤ i ∧ j ≤ (n :: `'a::linorder)}`
   using linorder_class.le_less_linear[of i n]
   by safe (force intro: leI)+

61.8 Generic big monoid operation over intervals

context semiring_char_0 begin

lemma `inj-on-of-nat [simp]:`
  `inj-on of-nat N` by (rule inj-onI) simp

lemma `bij-betw-of-nat [simp]:`
  `bij-betw of-nat N A ←→ of-nat ' N = A` by (simp add: bij-betw-def)

lemma `Nats-infinite: infinite (N :: 'a set)`
  `by (metis Nats-def infinite_UNIV_char_0 inj-on-of-nat)`
end

context comm_monoid_set begin

lemma atLeastLessThan-reindex:
  `F g `{h m..<h n} = F (g ◦ h) `{m..<n}`
  if bij_betw h `{m..<n}` `{h m..<h n}` for m n :: nat
  proof –
    from that have `inj_on h `{m..<n}` and `h `{m..<n} = `{h m..<h n}`
      by (simp_all add: bij_betw_def)
    then show ?thesis
      using reindex [of h `{m..<n}` g] by simp
  qed

lemma atLeastAtMost-reindex:
  `F g `{h m..h n} = F (g ◦ h) `{m..n}`
  if bij_betw h `{m..n}` `{h m..h n}` for m n :: nat
  proof –
    from that have `inj_on h `{m..n}` and `h `{m..n} = `{h m..h n}`
      by (simp_all add: bij_betw_def)
    then show ?thesis
      using reindex [of h `{m..n}` g] by simp
  qed

lemma atLeastLessThan-shift-bounds:
  `F g `{m + k..<n + k} = F (g ◦ plus k) `{m..<n}`
  for m n k :: nat
using atLeastLessThan-reindex [of plus k m n g]
by (simp add: ac-simps)

lemma atLeastAtMost-shift-bounds:
  \( F \circ g \{ m + k .. n + k \} = F (g \circ \text{plus } k) \{ m .. n \} \)
  for \( m n k :: \text{nat} \)
using atLeastAtMost-reindex [of plus k m n g]
by (simp add: ac-simps)

lemma atLeast-Suc-lessThan-Suc-shift:
  \( F \circ g \{ \text{Suc } m .. \text{Suc } n \} = F (g \circ \text{Suc}) \{ m .. \text{Suc } n \} \)
using atLeastLessThan-shift-bounds [of - - 1]
by (simp add: plus-1-eq-Suc)

lemma atLeast-Suc-atMost-Suc-shift:
  \( F \circ g \{ \text{Suc } m .. \text{Suc } n \} = F (g \circ \text{Suc}) \{ m .. \text{Suc } n \} \)
using atLeastAtMost-shift-bounds [of - - 1]
by (simp add: plus-1-eq-Suc)

lemma atLeast-atMost-pred-shift:
  \( F \circ (g \circ (\lambda n. n - \text{Suc } 0)) \{ \text{Suc } m .. \text{Suc } n \} = F \circ g \{ m .. n \} \)
unfolding atLeast-Suc-atMost-Suc-shift by simp

lemma atLeast-lessThan-pred-shift:
  \( F \circ (g \circ (\lambda n. n - \text{Suc } 0)) \{ \text{Suc } m .. \text{Suc } n \} = F \circ g \{ m .. \text{Suc } n \} \)
unfolding atLeast-Suc-lessThan-Suc-shift by simp

lemma atLeast-int-lessThan-int-shift:
  \( F \circ g \{ \text{int } m .. \text{int } n \} = F (g \circ \text{int}) \{ m .. \text{int } n \} \)
by (rule atLeastLessThan-reindex)
  (simp add: image-int-atLeastLessThan)

lemma atLeast-int-atMost-int-shift:
  \( F \circ g \{ \text{int } m .. \text{int } n \} = F (g \circ \text{int}) \{ m .. \text{int } n \} \)
by (rule atLeastAtMost-reindex)
  (simp add: image-int-atLeastAtMost)

lemma atLeast0-lessThan-Suc:
  \( F \circ g \{ 0 .. \text{Suc } n \} = F \circ g \{ 0 .. \text{Suc } n \} \star g \)
by (simp add: atLeast0-lessThan-Suc ac-simps)

lemma atLeast0-atMost-Suc:
  \( F \circ g \{ 0 .. \text{Suc } n \} = F \circ g \{ 0 .. \text{Suc } n \} \star g (\text{Suc } n) \)
by (simp add: atLeast0-atMost-Suc ac-simps)

lemma atLeast0-lessThan-Suc-shift:
  \( F \circ g \{ 0 .. \text{Suc } n \} = g \star F (g \circ \text{Suc}) \{ 0 .. \text{Suc } n \} \)
by (simp add: atLeast0-lessThan-Suc-eq-insert-0 atLeast-Suc-lessThan-Suc-shift)
lemma atLeast0-atMost-Suc-shift:
\[ F \cdot g \{ 0..\text{Suc } n \} = g \cdot 0 \ast F \cdot (g \circ \text{Suc}) \{ 0..n \} \]
by (simp add: atLeast0-atMost-Suc-eq-insert-0 atLeast-atMost-Suc-shift)

lemma atLeast-Suc-lessThan:
\[ F \cdot g \{ m..<n \} = g \cdot m \ast F \cdot g \{ \text{Suc } m..<n \} \]
if \( m < n \)
proof -
  from that have \{ m..<n \} = insert m \{ \text{Suc } m..<n \}
  by auto
then show \(?thesis by simp 
qed

lemma atLeast-Suc-atMost:
\[ F \cdot g \{ m..n \} = g \cdot m \ast F \cdot g \{ \text{Suc } m..n \} \]
if \( m \leq n \)
proof -
  from that have \{ m..n \} = insert m \{ \text{Suc } m..n \}
  by auto
then show \(?thesis by simp 
qed

lemma ivl-cong:
\[ a = c \Longrightarrow b = d \Longrightarrow (\forall x. c \leq x \Longrightarrow x < d \Longrightarrow g \cdot x = h \cdot x) \]
by (rule cong) simp-all

lemma atLeastLessThan-shift-0:
fixes m n p :: nat
shows \( F \cdot g \{ m..<n \} = F \cdot (g \circ \text{plus } m) \{ 0..<n - m \} \)
using atLeastLessThan-shift-bounds [of g 0 m n - m]
by simp-all

lemma atLeastAtMost-shift-0:
fixes m n p :: nat
assumes m \leq n
shows \( F \cdot g \{ m..n \} = F \cdot (g \circ \text{plus } m) \{ 0..n - m \} \)
using assms atLeastAtMost-shift-bounds [of g 0 m n - m] by simp

lemma atLeastLessThan-concat:
fixes m n p :: nat
shows \( m \leq n \Longrightarrow n \leq p \Longrightarrow F \cdot g \{ m..<\text{Suc } i \} \ast F \cdot g \{ \text{Suc } i..<p \} = F \cdot g \{ \text{Suc } i..<p \} \)
by (simp add: union-disjoint [symmetric] ivl-disj-un)

lemma atLeastLessThan-rev:
\[ F \cdot g \{ n..<m \} = F \cdot (\lambda i. g \cdot (m + n - \text{Suc } i)) \{ n..<m \} \]
by (rule reindex-bij-witness [where i=\lambda i. m + n - \text{Suc } i and j=\lambda i. m + n - \text{Suc } i], auto)

lemma atLeastAtMost-rev:
fixes n m :: nat
shows $F \ g \ \{n..m\} = F (\lambda i. \ g \ (m + n - i)) \ \{n..m\}$

by (rule reindex-bij-witness [where $i = \lambda i. \ m + n - i$ and $j = \lambda i. \ m + n - i$])

auto

lemma atLeastLessThan-rev-atLeast-Suc-atMost:
$F \ g \ \{n..<m\} = F (\lambda i. \ g \ (m + n - i)) \ \{\text{Suc} \ n..m\}$

unfolding atLeastLessThan-rev [of $g \ n \ m$]

by (cases $m$) (simp-all add: atLeast-Suc-atMost-Suc-shift atLeastLessThanSuc-atLeastAtMost)

end

61.9 Summation indexed over intervals

syntax (ASCII)

- from-to-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b (SUM - = -,-/ -) [0,0,0,10] 10
- from-upto-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b (SUM - = -,-,-/- -) [0,0,0,10] 10
- apt-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((SUM -,-/ -) [0,0,10] 10)
- upto-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((SUM -,-/- -) [0,0,10] 10)

syntax (latex-sum output)

- from-to-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\sumx = \_/- \_) [0,0,0,10] 10)
- from-upto-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\sumx = \_/- \_) [0,0,0,10] 10)
- apt-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\sumx < -/- \_) [0,0,10] 10)
- upto-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\sumx \_<_/-/- \_) [0,0,10] 10)

syntax

- from-to-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\sumx = \_/- \_) [0,0,0,10] 10)
- from-upto-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\sumx = \_/- \_) [0,0,0,10] 10)
- apt-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\sumx < -/- \_) [0,0,10] 10)
- upto-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\sumx \_<_/-/- \_) [0,0,10] 10)

translations

\[ \sum x =. a..b. \ t = \text{CONST} \ \text{sum} \ (\lambda x. \ t) \ \{a..b\} \]
\[ \sum x =. a..<b. \ t = \text{CONST} \ \text{sum} \ (\lambda x. \ t) \ \{a..<b\} \]
\[ \sum i \leq n. \ t = \text{CONST} \ \text{sum} \ (\lambda i. \ t) \ \{..n\} \]
\[ \sum i < n. \ t = \text{CONST} \ \text{sum} \ (\lambda i. \ t) \ \{..<n\} \]

The above introduces some pretty alternative syntaxes for summation over intervals:
The left column shows the term before introduction of the new syntax, the middle column shows the new (default) syntax, and the right column shows a special syntax. The latter is only meaningful for latex output and has to be activated explicitly by setting the print mode to latex-sum (e.g. via `mode = latex-sum` in antiquotations). It is not the default \LaTeX output because it only works well with italic-style formulae, not tt-style.

Note that for uniformity on `nat` it is better to use $\sum x = 0..<n$. \textit{e} rather than $\sum x < n$. \textit{e}: \texttt{sum} may not provide all lemmas available for \{\texttt{m..<n}\} also in the special form for \{\texttt{..<e}\}.

This congruence rule should be used for sums over intervals as the standard theorem \texttt{sum.cong} does not work well with the simplifier who adds the unsimplified premise $x \in B$ to the context.

```plaintext
context comm-monoid-set
begin

lemma zero-middle:
assumes 1 \leq p k \leq p
shows $F (\lambda j. \text{if } j < k \text{ then } g j \text{ else if } j = k \text{ then } 1 \text{ else } h (j - Suc 0)) \{..p\}$
  = $F (\lambda j. \text{if } j < k \text{ then } g j \text{ else } h j) \{..p - Suc 0\}$ (is ?lhs = ?rhs)
proof -
  have [simp]: \{..p - Suc 0\} \cap \{j. j < k\} = \{..<k\} \{..p - Suc 0\} \cap - \{j. j < k\}
  using assms by auto
  have ?lhs = $F g \{..<k\} \bigstar F (\lambda j. \text{if } j = k \text{ then } 1 \text{ else } h (j - Suc 0)) \{k..p\}$
    using union-disjoint [of \{..<k\} \{k..p\}] assms
  by (simp add: ivl-disj-int-one ivl-disj-un-one)
  also have \ldots = $F g \{..<k\} \bigstar F (\lambda j. h (j - Suc 0)) \{Suc k..p\}$
    by (simp add: atLeast-Suc-atMost [of k p] assms)
  also have \ldots = $F g \{..<k\} \bigstar F h \{k..p - Suc 0\}$
    using reindex [of Suc \{k..p - Suc 0\}] assms by simp
  also have \ldots = ?rhs
    by (simp add: If-cases)
  finally show ?thesis.
qed

lemma atMost-Suc [simp]:
$F g \{..Suc n\} = F g \{..n\} \bigstar g (Suc n)$
by (simp add: atMost-Suc ac-simps)

lemma lessThan-Suc [simp]:
$F g \{..<Suc n\} = F g \{..<n\} \bigstar g n$
by (simp add: lessThan-Suc ac-simps)

lemma cl-ivl-Suc [simp]:
$F g \{m..Suc n\} = (\text{if } Suc n < m \text{ then } 1 \text{ else } F g \{m..n\} \bigstar g(Suc n))$
by (auto simp: ac-simps atLeastAtMostSuc-conv)
```
THEORY “Set-Interval”

lemma op-ivl-Suc [simp]:
  \( F \ g \ {m..<\text{Suc} \ n} = (\text{if} \ n < m \ \text{then} \ 1 \ \text{else} \ F \ g \ {m..<n} \ * \ g(n)) \)
by (auto simp: ac-simps atLeastLessThanSuc)

lemma head:
  fixes \( n :: \text{nat} \)
  assumes mn: \( m \leq n \)
  shows \( F \ g \ {m..n} = g \ m * F \ g \ {m..<n} \)
proof -
  from mn have \( \{m..n\} = \{m\} \cup \{m..<n\} \)
  by (auto intro: ivl-disj-un-singleton)
  hence \( ?\text{lhs} = F \ g \ (\{m\} \cup \{m..<n\}) \)
  by (simp add: atLeast0LessThan)
  also have \( \ldots = ?\text{rhs} \) by simp
  finally show \( ?\text{thesis} \).
qed

lemma last-plus:
  fixes \( n :: \text{nat} \)
  shows \( m \leq n \implies F \ g \ {m..n} = g \ n * F \ g \ {m..<n} \)
by (cases \( n \)) (auto simp: atLeastLessThanSuc-atLeastAtMost commute)

lemma head-if:
  fixes \( n :: \text{nat} \)
  shows \( F \ g \ {m..n} = (\text{if} \ n < m \ \text{then} \ 1 \ \text{else} \ F \ g \ {m..<n} * g(n)) \)
by (simp add: commute last-plus)

lemma ub-add-nat:
  assumes \( (m::\text{nat}) \leq n + 1 \)
  shows \( F \ g \ {m..n+p} = F \ g \ {m..n} * F \ g \ {n+1..n+p} \)
proof -
  have \( \{m..n+p\} = \{m..n\} \cup \{n+1..n+p\} \) using \( m \leq n+1 \) by auto
  thus \( ?\text{thesis} \) by (auto simp: ivl-disj-int union-disjoint atLeastSucAtMost-greaterThanAtMost)
qed

lemma nat-group:
  fixes \( k::\text{nat} \)
  shows \( F \ (\lambda m. F \ g \ {m* k..<m*k + k}) \ {..<n} = F \ g \ {..< n * k} \)
proof (cases \( k \))
  case (Suc \( l \))
  then have \( k > 0 \)
  by auto
  then show \( ?\text{thesis} \)
  by (induct \( n \)) (simp-all add: atLeastLessThan-concat add.commute atLeast0LessThan[symmetric])
qed auto

lemma triangle-reindex:
  fixes \( n :: \text{nat} \)
  shows \( F \ (\lambda(i,j). g \ i \ j) \ {(i,j). i+j < n} = F \ (\lambda k. F \ (\lambda i. g \ i (k - i)) \ {..<k}) \ {..<n} \)
apply (simp add: Sigma)
apply (rule reindex-bij-witness[where \( j = \lambda(i, j). (i+j, i) \) and \( i = \lambda(k, i). (i, k-i) \)])
apply auto
done

lemma triangle-reindex-eq:
fixes \( n :: \text{nat} \)
shows \( F (\lambda(i,j). g i j) \{ (i, j). i+j \leq n \} = F (\lambda k. F (\lambda i. g i (k-i)) \{ ..k \}) \{ ..n \} \)
using triangle-reindex [of \( g \) \( \text{Suc} \) \( n \)]
by (simp only: Nat.less-Suc-eq-le lessThan-Suc-atMost)

lemma shift-bounds-nat-ivl:
\( F g \{ m+k..<n+k \} = F (\lambda(i). g(i+k)) \{ m..<n::\text{nat} \} \)
by (induct \( n \), auto simp: atLeastLessThanSuc)

lemma shift-bounds-cl-nat-ivl:
\( F g \{ m+k..n+k \} = F (\lambda(i). g(i+k)) \{ m..n::\text{nat} \} \)
by (rule reindex-bij-witness[where \( i = \lambda i. n - \text{Suc} i \) and \( j = \lambda i. n - \text{Suc} i \)] auto)

corollary shift-bounds-cl-Suc-ivl:
\( F g \{ \text{Suc} m .. \text{Suc} n \} = F (\lambda i. g(\text{Suc} i)) \{ m..n \} \)
by (simp add: shift-bounds-cl-nat-ivl[where \( k = \text{Suc} 0 \), simplified])

corollary Suc-reindex-ivl: \( m \leq n \implies F g \{ m..n \} * g (\text{Suc} n) = g m * F (\lambda i. g (\text{Suc} i)) \{ m..n \} \)
by (simp add: assoc atLeast-Suc-atMost flip: shift-bounds-cl-Suc-ivl)

corollary shift-bounds-Suc-ivl:
\( F g \{ \text{Suc} m..<\text{Suc} n \} = F (\lambda i. g(\text{Suc} i)) \{ m..<n \} \)
by (simp add: shift-bounds-nat-ivl[where \( k = \text{Suc} 0 \), simplified])

lemma atMost-Suc-shift:
shows \( F g \{ ..\text{Suc} n \} = g 0 * F (\lambda i. g (\text{Suc} i)) \{ ..n \} \)
proof (induct \( n \))
case 0 show ?case by simp
next
case (\( \text{Suc} n \)) note IH = this
have \( F g \{ ..\text{Suc} (\text{Suc} n) \} = F g \{ ..\text{Suc} n \} * g (\text{Suc} (\text{Suc} n)) \)
  by (rule atMost-Suc)
also have \( F g \{ ..\text{Suc} n \} = g 0 * F (\lambda i. g (\text{Suc} i)) \{ ..n \} \)
  by (rule IH)
also have \( g 0 * F (\lambda i. g (\text{Suc} i)) \{ ..n \} * g (\text{Suc} (\text{Suc} n)) = g 0 * (F (\lambda i. g (\text{Suc} i)) \{ ..n \} * g (\text{Suc} (\text{Suc} n))) \)
  by (rule assoc)
also have \( F (\lambda i. g (\text{Suc} i)) \{ ..n \} * g (\text{Suc} (\text{Suc} n)) = F (\lambda i. g (\text{Suc} i)) \{ ..\text{Suc} n \} \)
THEORY "Set-Interval"

\begin{enumerate}
\item \textbf{lemma} \texttt{lessThan-Suc-shift:}
\begin{quote}
\texttt{F g \{..<\text{Suc } n\} = g 0 \ast \text{F (} \lambda i. \text{g (Suc } i) \text{)} \{..<\text{n}\}}
\end{quote}
by (\texttt{induction n}) (\texttt{simp-all add: ac-simps})
\item \textbf{lemma} \texttt{atMost-shift:}
\begin{quote}
\texttt{F g \{..\text{n}\} = g 0 \ast \text{F (} \lambda i. \text{g (Suc } i) \text{)} \{..<\text{n}\}}
\end{quote}
by (\texttt{metis atLeast0AtMost atLeast0LessThan atLeastLessThanSuc-atLeastAtMost atLeastSucAtMost-greaterThanAtMost le0 head shift-bounds-Suc-ivl})
\item \textbf{lemma} \texttt{nested-swaps:}
\begin{quote}
\texttt{F (} \lambda i. \text{F (} \lambda j. \text{a } i \ j) \{0..<i\} \}} \{..\text{n}\} = \text{F (} \lambda j. \text{F (} \lambda i. \text{a } i \ j) \{\text{Suc } j..\text{n}\} \} \{..<\text{n}\}
\end{quote}
by (\texttt{induction n}) (\texttt{auto simp: distrib})
\item \textbf{lemma} \texttt{nested-swaps':}
\begin{quote}
\texttt{F (} \lambda i. \text{F (} \lambda j. \text{a } i \ j) \{..\text{i}\} \}} \{..\text{n}\} = \text{F (} \lambda j. \text{F (} \lambda i. \text{a } i \ j) \{\text{Suc } j..\text{n}\} \} \{..<\text{n}\}
\end{quote}
by (\texttt{induction n}) (\texttt{auto simp: distrib})
\item \textbf{lemma} \texttt{atLeast1-atMost-eq:}
\begin{quote}
\texttt{F g \{\text{Suc } 0..\text{n}\} = \text{F (} \lambda k. \text{g (Suc } k) \text{)} \{..<\text{n}\}}
\end{quote}
\begin{proof}
\begin{itemize}
\item have \texttt{F g \{Suc 0..n\} = F g \{Suc \ ' \{..<n\}}}
by (\texttt{simp add: image-Suc-lessThan})
\item also have \texttt{... = F (} \lambda k. \text{g (Suc } k) \text{)} \{..<\text{n}\}
by (\texttt{simp add: reindex})
\end{itemize}
finally show \texttt{?thesis}.
\end{proof}
\item \textbf{lemma} \texttt{atLeastLessThan-Suc: a \leq b \implies F g \{a..<\text{Suc } b\} = F g \{a..<b\} \ast g b}
by (\texttt{simp add: atLeastLessThanSuc commute})
\item \textbf{lemma} \texttt{nat-ivl-Suc':}
\begin{quote}
\texttt{assumes m \leq \text{Suc } n}
shows \texttt{F g \{m..\text{Suc } n\} = g (\text{Suc } n) \ast F g \{m..n\}}
\end{quote}
\begin{proof}
\begin{itemize}
\item from \texttt{assms have \{m..Suc } n\} = \texttt{insert (Suc } n\) \{m..n\} by \texttt{auto}
\item also have \texttt{F g ... = g (Suc } n\) \ast F g \{m..n\} by \texttt{simp}
\end{itemize}
finally show \texttt{?thesis}.
\end{proof}
\item \textbf{lemma} \texttt{in-pairs: F g \{2\ast m..\text{Suc}(2\ast n)\} = F (} \lambda i. \text{g(2\ast i)} \ast g(\text{Suc}(2\ast i))) \{m..n\}
\begin{proof}
\texttt{(induction } n\text{)}
\begin{itemize}
\item \texttt{case } 0
\item show \texttt{?case}
\end{itemize}
\end{proof}
\end{enumerate}
THEORY “Set-Interval”

by (cases m=0) auto

next
  case (Suc n)
  then show ?case
    by (auto simp: assoc split: if-split-asm)

qed

lemma in-pairs-0: \( F g \{..Suc(2*n)\} = F (\lambda i. g(2*i) \cdot g(Suc(2*i))) \{..n\} \)
  using in-pairs [of - 0 n] by (simp add: atLeast0AtMost)

end

lemma card-sum-le-nat-sum: \( \sum\{0..<\text{card } S\} \leq \sum S \)
proof (cases finite S)
  case True
  then show ?thesis
    proof (induction card S arbitrary: S)
      case (Suc x)
      then have Max S \( \geq \) x
        using card-le-Suc-Max by fastforce
      let \(?S' = S - \{\text{Max } S\}\)
      from Suc have Max S \( \in \) S by (auto intro: Max-in)
      hence \( \text{cards: card } S = \text{Suc (card } ?S') \)
        using \( \text{finite } S \) by (intro card.remove; auto)
      hence \( \sum\{0..<\text{card } ?S'\} \leq \sum ?S' \)
        using Suc by (intro Suc; auto)
      hence \( \sum\{0..<\text{card } ?S'\} + x \leq \sum ?S' + \text{Max } S \)
        using \( \text{Max } S \geq x \) by simp
      also have ... = \( \sum S \)
        using sum.remove[OF \( \text{finite } S \); \( \text{Max } S \in S \)], where g=\( \lambda x. x \)
        by simp
      finally show ?case
        using cards Suc by auto
      qed simp

qed simp

lemma sum-natinterval-diff:
  fixes f :: nat \Rightarrow ('a::ab-group-add)
  shows \( \sum (\lambda k. f k - f(k + 1)) \{\{m::nat\}..n\} = \)
    \( \text{if } m \leq n \text{ then } f m - f(n + 1) \text{ else } 0 \)
  by (induct n, auto simp add: algebra-simps not-le le-Suc-eq)

lemma sum-diff-nat-id:
  fixes f :: nat \Rightarrow ('a::ab-group-add)
  shows \([m \leq n; n \leq p] \Rightarrow \sum f \{m..<p\} - \sum f \{m..<n\} = \sum f \{n..<p\}\)
  using sum.atLeastLessThan-concat [of m n p f, symmetric]
  by (simp add: ac-simps)

lemma sum-diff-distrib: \( \forall x. Q x \leq P x \Rightarrow (\sum x<n. P x) - (\sum x<n. Q x) = \)
\[
\sum_{x<n} P x - Q x :: \text{nat}
\]
by (subst sum-subtractf-nat) auto

### 61.9.1 Shifting bounds

**context** `comm-monoid-add`

**begin**

**context**

- `fixes f :: nat ⇒ 'a`
- `assumes f 0 = 0`

**begin**

**lemma** `sum-shift-lb-Suc0-0-upt`:
- `\sum f \{\text{Suc } 0..<k\} = \sum f \{0..<k\}`

**proof** (cases `k`)

- **case** `0`
  then show `?thesis`
  by simp

- **next**
  **case** `(Suc k)`
  moreover have `{0..< Suc k} = insert 0 \{Suc 0..<Suc k\}`
  by auto
  ultimately show `?thesis`
  using `f 0 = 0` by simp

**qed**

**lemma** `sum-shift-lb-Suc0-0`:
- `\sum f \{\text{Suc } 0..k\} = \sum f \{0..k\}`

**proof** (cases `k`)

- **case** `0`
  with `f 0 = 0` show `?thesis`
  by simp

- **next**
  **case** `(Suc k)`
  moreover have `{0..< Suc k} = insert 0 \{Suc 0..<Suc k\}`
  by auto
  ultimately show `?thesis`
  using `f 0 = 0` by simp

**qed**

**end**

**end**

**lemma** `sum-Suc-diff`:

- `fixes f :: nat ⇒ 'a::ab-group-add`
- `assumes m ≤ Suc n`
- `shows (∑ i = m..n. f(Suc i) - f i) = f (Suc n) - f m`

**using** `assms` by (induct `n`) (auto simp: le-Suc-eq)
lemma `sum-Suc-diff`:
fixes f :: `nat => 'a::ab-group-add`
assumes m ≤ n
shows `\( \sum i = m..<n. f (Suc i) - f i \) = f n - f m`
using assms by (induct n) (auto simp: le-Suc-eq)

lemma `sum-diff-split`:
fixes f :: `nat => 'a::ab-group-add`
assumes m ≤ n
shows `\( \sum i \leq n. f i \) - `\( \sum i < m. f i \) = `\( \sum i \leq n - m. f (n - i) \)`
proof
  have `\( \wedge. i \leq n - m \implies \exists k \geq m. k \leq n \land i = n - k \)`
  by (metis Nat.le-diff-conv2 add.commute `\langle m \leq n \rangle`)
  then have eq: `{..n-m} = (-)n `\{m..n\}`
  by force
  have inj: `inj-on ((-)n) `\{m..n\}`
  by (auto simp: inj-on-def)
  have `\( \sum i \leq n - m. f (n - i) \) = `\( \sum i = m..n. f i \)`
  by (simp add: eq sum.reindex-cong[OF inj])
  also have `\ldots = `\( \sum i \leq n. f i \) - `{i < m. f i}`
  by (simp only: atLeast0AtMost atLeast0LessThan atLeastLessThanSuc-atLeastAtMost)
  finally show `\( ?thesis \)` by metis
qed

61.9.2 Telescoping sums

lemma `sum-telescope`:
fixes f :: `nat => 'a::ab-group-add`
shows `\( \sum (\lambda i. f i - f (Suc i)) \{..i\} = f 0 - f (Suc i) \)`
by (induct i) simp-all

lemma `sum-telescope"`:
assumes m ≤ n
shows `\( \sum k \in \{Suc m..n\}. f k - f (k - 1) \) = f n - (f m :: 'a::ab-group-add)`
by (rule dec-induct[OF assms]) (simp-all add: algebra-simps)

lemma `sum-lessThan-telescope`:
`\( \sum n < m. f (Suc n) - f n :: 'a::ab-group-add \) = f m - f 0`
by (induction m) (simp-all add: algebra-simps)

lemma `sum-lessThan-telescope"`:
`\( \sum n < m. f n - f (Suc n) :: 'a::ab-group-add \) = f 0 - f m`
by (induction m) (simp-all add: algebra-simps)

61.9.3 The formula for geometric sums

lemma `sum-power2`:
`\( \sum i = 0..<k. (2::nat)^i \) = 2^k - 1`
by (induction k) (auto simp: mult-2)

lemma geometric-sum:
assumes \( x \neq 1 \)
shows \( (\sum_{i<n} x \cdot i) = (x^n - 1) / (x - 1) \cdot \text{field} \)
proof
- from assms obtain y where \( y = x - 1 \) and \( y \neq 0 \) by simp-all
moreover have \( (\sum_{i<n} (y + 1) \cdot i) = ((y + 1)^n - 1) / y \)
by (induct n) (simp-all add: field-simps \( y \neq 0 \cdot \))
ultimately show \( \text{thesis} \) by simp
qed

lemma geometric-sum-less:
assumes \( 0 < x \) and \( x < 1 \) finite \( S \)
show \( (\sum_{i \in S} x \cdot i) < 1 / (1 - x) \cdot \text{field} \cdot \text{linordered-field} \)
proof
- define \( n \) where \( n = \text{Suc} \cdot (\text{Max} S) \)
also have \( (\sum_{i \in S} x \cdot i) \leq (\sum_{i<n} x \cdot i) \)
unfolding \( n \)-def using assms by (fastforce intro: sum-mono2 le-imp-less-Suc)
also have \( \ldots \) = \( (1 - x^\cdot n) / (1 - x) \)
using assms by (simp add: geometric-sum field-simps)
also have \( \ldots < 1 / (1 - x) \)
using assms by (simp add: field-simps power-Suc-less)
finally show \( \text{thesis} \).
qed

lemma diff-power-eq-sum:
fixes \( y \cdot \text{a} : \{\text{comm-ring,monoid-mult}\} \)
shows \( x \cdot (\text{Suc} n) - y \cdot (\text{Suc} n) = (x - y) \cdot (\sum_{p<\text{Suc} n} x \cdot p) \cdot y \cdot (n - p) \)
proof (induct n)
case \( \text{Suc} n \)
also have \( x \cdot (\text{Suc} n) - y \cdot (\text{Suc} n) = x \cdot (x \cdot n) - y \cdot (y \cdot y \cdot n) \)
also have \( \ldots \) = \( y \cdot (x \cdot (\text{Suc} n) - y \cdot (\text{Suc} n)) + (x - y) \cdot (x \cdot n) \)
also have \( \ldots \) = \( y \cdot ((x - y) \cdot (\sum_{p<\text{Suc} n} x \cdot p) \cdot y \cdot (n - p)) + (x - y) \cdot (x \cdot n) \)
also have \( \ldots \) = \( (x - y) \cdot (\sum_{p<\text{Suc} n} x \cdot p) \cdot y \cdot (n - p) + (x - y) \cdot (x \cdot n) \)
also have \( \ldots \) = \( (x - y) \cdot (\sum_{p<\text{Suc} n} x \cdot p) \cdot y \cdot (n - p) + (x - y) \cdot (x \cdot n) \)
by (simp only: mult.left-commute)
also have \( \ldots \) = \( (x - y) \cdot (\sum_{p<\text{Suc} n} x \cdot p) \cdot y \cdot (n - p) + (x - y) \cdot (x \cdot n) \)
by (simp only: mult.left-commute)
also have \( \ldots \) = \( (x - y) \cdot (\sum_{p<\text{Suc} n} x \cdot p) \cdot y \cdot (n - p) + (x - y) \cdot (x \cdot n) \)
by (simp only: mult.left-commute)
finally show \( \text{case} \).
qed simp

corollary power-diff-sumr2: — \text{COMPLEX-POLYFUN} in HOL Light
fixes $x :: 'a::{comm-ring,monoid-mult}$
shows $x^n - y^n = (x - y) \cdot (\sum_{i<n} y^{n-i})$
using diff-power-eq-sum[of $x - y$]
by (cases $n = 0$) (simp-all add: field-simps)

lemma power-diff-1-eq:
fixes $x :: 'a::{comm-ring,monoid-mult}$
shows $x^n - 1 = (x - 1) \cdot (\sum_{i<n} (x^{n-i}))$
using diff-power-eq-sum[of $x - 1$]
by (cases $n$) auto

lemma one-diff-power-eq:
fixes $x :: 'a::{comm-ring,monoid-mult}$
shows $1 - x^n = (1 - x) \cdot (\sum_{i<n} (x^{n-i}))$
using diff-power-eq-sum[of $1 - x$]
by (cases $n$) auto

lemma one-diff-power-eq':
fixes $x :: 'a::{comm-ring,monoid-mult}$
shows $(1 - x) \cdot (\sum_{i<n} (x^{n-i})) = 1 - x^{\text{Suc}$ n}$
by (metis one-diff-power-eq lessThan-Suc-atMost)

lemma sum-gp-basic:
fixes $x :: 'a::{comm-ring,monoid-mult}$
shows $(1 - x) \cdot (\sum_{i\leq n} (x^{i})) = 1 - x^{\text{Suc}$ n}$
by (simp only: one-diff-power-eq lessThan-Suc-atMost)

lemma sum-power-shift:
fixes $x :: 'a::{comm-ring,monoid-mult}$
assumes $m \leq n$
shows $(\sum_{i=m..n} x^{i}) = x^m \cdot (\sum_{i=m..n} x^{i-m})$
proof -
have $(\sum_{i=m..n} x^{i}) = x^m \cdot (\sum_{i=m..n} x^{i-m})$
by (simp add: sum-distrib-left power-add [symmetric])
also have $(\sum_{i=m..n} x^{i-m}) = (\sum_{i\leq n-m} x^{i})$
using $(m \leq n)$ by (intro sum.reindex-bij-witness[where $j=\lambda i. \ i - m$ and $i=\lambda i. \ i + m$]) auto
finally show $?thesis$.

qed

lemma sum-gp-multiplied:
fixes $x :: 'a::{comm-ring,monoid-mult}$
assumes $m \leq n$
shows $(1 - x) \cdot (\sum_{i=m..n} x^{i}) = x^m \cdot (1 - x) \cdot (\sum_{i\leq n-m} x^{i})$
proof -
have $(1 - x) \cdot (\sum_{i=m..n} x^{i}) = x^m \cdot (1 - x) \cdot (\sum_{i\leq n-m} x^{i})$
by (metis mult.assoc mult.commute assms sum-power-shift)
also have $(1 - x) \cdot (\sum_{i=m..n} x^{i}) = (1 - x)^{\text{Suc$(n-m)$}}$
by (metis mult.assoc sum-gp-basic)
also have \( \ldots = x^m - x^{\text{Suc } n} \)
using assms
by (simp add: algebra-simps) (metis le-add-diff-inverse power-add)
finally show \( ?\text{thesis} \).
qed

lemma sum-gp:
fixes \( x :: 'a::{\text{comm-ring,division-ring}} \)
sows \( \big( \sum_{i=m..n} x^i \big) = \)
\begin{align*}
& (\text{if } n < m \text{ then } 0) \\
& \text{else if } x = 1 \text{ then of-nat}((n + 1) - m) \\
& \text{else } (x^m - x^{\text{Suc } n}) / (1 - x)
\end{align*}

proof (cases \( n < m \))
case False 
assume \( \ast \): \( \neg n < m \)
then show \( ?\text{thesis} \)
proof (cases \( x = 1 \))
case False 
assume \( x \neq 1 \)
then have \( \text{not-zero: } 1 - x \neq 0 \)
by auto 
have \( (1 - x) * (\sum_{i=m..n} x^i) = x^m - x * x^n \)
using sum-gp-multiplied \( \text{of } m \text{ } n \text{ } x \) * by auto 
then have \( (\sum_{i=m..n} x^i) = (x^m - x * x^n) / (1 - x) \)
using nonzero-divide-eq-eq \( \text{mult.commute } \text{not-zero} \)
by metis 
then show \( ?\text{thesis} \)
by auto 
qed (auto)
qed (auto)

61.9.4 Geometric progressions

lemma sum-gp0:
fixes \( x :: 'a::{\text{comm-ring,division-ring}} \)
sows \( \big( \sum_{i \leq n} x^i \big) = \)
\begin{align*}
& (\text{if } x = 1 \text{ then of-nat}(n + 1) \text{ else } (1 - x^{\text{Suc } n}) / (1 - x)) \\
\end{align*}

using sum-gp-basic \( \text{of } n \text{ } x \) by (simp add: mult.commute field-split-simps)

lemma sum-power-add:
fixes \( x :: 'a::{\text{comm-ring,monoid-mult}} \)
sows \( \big( \sum_{i \in I} x^{(m+i)} \big) = x^m * \big( \sum_{i \in I} x^i \big) \)
by (simp add: sum-distrib-left power-add)

lemma sum-gp-offset:
fixes \( x :: 'a::{\text{comm-ring,division-ring}} \)
sows \( \big( \sum_{i=m..m\text{ }n} x^i \big) = \)
\begin{align*}
& (\text{if } x = 1 \text{ then of-nat } n + 1 \text{ else } x^m * (1 - x^{\text{Suc } n}) / (1 - x)) \\
\end{align*}
using sum-gp [of x m m+n]
by (auto simp: power-add algebra-simps)

lemma sum-gp-strict:
  fixes x :: 'a::{comm_ring,division-ring}
  shows \( \sum_{i<n} x^i = (if x = 1 then of-nat n else (1 - x^n) / (1 - x)) \)
  by (induct n) (auto simp: algebra-simps field-split-simps)

61.9.5 The formulae for arithmetic sums

context comm-semiring-1
begin

lemma double-gauss-sum:  
  \( 2 * (\sum_{i=0..n} of-nat i) = of-nat n * (of-nat n + 1) \)
  by (induct n) (simp-all add: sum.atLeast0-atMost-Suc algebra-simps left-add-twice)

lemma double-gauss-sum-from-Suc-0:  
  \( 2 * (\sum_{i=Suc 0..n} of-nat i) = of-nat n * (of-nat n + 1) \)
  proof 
  have \( \sum of-nat \{Suc 0..n\} = sum of-nat \{insert 0 \{Suc 0..n\}\} \)
    by simp
  also have \( \ldots = sum of-nat \{0..n\} \)
    by (cases n) (simp-all add: atLeast0-atMost-Suc-eq-insert-0)
  finally show \( \text{thesis} \)
    by (simp add: double-gauss-sum)
qed

lemma double-arith-series:  
  \( 2 * (\sum_{i=0..n} a + of-nat i * d) = (of-nat n + 1) * (2 * a + of-nat n * d) \)
  proof 
  have \( \sum i = 0..n. a + of-nat i * d = ((\sum i = 0..n. a) + (\sum i = 0..n. of-nat i * d)) \)
    by (rule sum.distrib)
  also have \( \ldots = (of-nat (Suc n) * a + d * (\sum i = 0..n. of-nat i)) \)
    by (simp add: sum-distrib-left algebra-simps)
  finally show \( \text{thesis} \)
    by (simp add: algebra-simps double-gauss-sum left-add-twice)
qed

end

context linordered-euclidean-semiring
begin

lemma gauss-sum:  
  \( (\sum_{i=0..n} of-nat i) = of-nat n * (of-nat n + 1) \div 2 \)
  using double-gauss-sum [of n, symmetric] by simp
THEORY “Set-Interval”

lemma gauss-sum-from-Suc-0:
\( \sum i = Suc 0 \ldots n \) of-nat i = of-nat n \* (of-nat n + 1) div 2
using double-gauss-sum-from-Suc-0 [of n, symmetric] by simp

lemma arith-series:
\( \sum i = 0 \ldots n \) a + of-nat i \* d = (of-nat n + 1) \* (2 \* a + of-nat n \* d) div 2
using double-arith-series [of a d n, symmetric] by simp

end

lemma gauss-sum-nat:
\( \sum \{0 \ldots n\} = (n \* Suc n) \) div 2
using gauss-sum [of n, where \(?a = nat\)] by simp

lemma arith-series-nat:
\( \sum i = 0 \ldots n \) a + i \* d = Suc n \* (2 \* a + n \* d) div 2
using arith-series [of a d n] by simp

lemma Sum-Icc-int:
\( \sum \{m \ldots n\} = (n * (n + 1) - m * (m - 1)) \) div 2
if m \leq n for m n :: int
using that proof (induct i \equiv nat (n - m) arbitrary; m n)
case 0
then have m = n
  by arith
then show \?case
  by (simp add: algebra-simps mult-2 [symmetric])
next
case (Suc i)
have 0: i = nat((n - 1) - m) m \leq n - 1 using Suc(2,3) by arith+
have \( \sum \{m \ldots n\} = \sum \{m \ldots 1\} + n \) using \( \{m \leq n\} \)
  by subst atLeastAtMostPlus1-int-cone simp-all
also have \ldots = ((n - 1) \* (n - 1) + n) \* (n - 1) div 2 + n
  by (simp add: Suc(1)[OF 0])
also have \ldots = ((n - 1) \* (n + 1) - m \* (m - 1)) \* (n - 1) + 2 \* n \) \* (n - 1) \* (m - 1) \) div 2
  by simp
also have \ldots = ((n - 1) \* (n + 1) - m \* (m - 1) \) \* (n - 1) \* (m - 1) \) div 2
  by (simp add: algebra-simps mult-2-right)
finally show \?case .
qed

lemma Sum-Icc-nat:
\( \sum \{m \ldots n\} = (n * (n + 1) - m * (m - 1)) \) div 2 for m n :: nat
proof (cases m \leq n)
case True
then have \*: m * (m - 1) \leq n * (n + 1)
  by (meson diff-le-self order-trans le-add1 mult-le mono)
have \( int (\sum \{m \ldots n\}) = \sum \{int m \ldots int n\} \)
  by (simp add: sum.atLeast-int-atMost-int-shift)
also have .. = (int n * (int n + 1) - int m * (int m - 1)) div 2
using \langle m \leq n \rangle by (simp add: Sum-Icc-int)
also have .. = int \((n * (n + 1) - m * (m - 1)) \div 2\)
using le-square * by (simp add: algebra-simps of-nat-div of-nat-diff)
finally show \(?thesis
by (simp only: of-nat-eq-iff)
next
case False
then show \(?thesis
by (auto dest: less-imp-Suc-add simp add: not-le algebra-simps)
qed

lemma Sum-Ico-nat:
\(\sum \{m..<n\} = (n * (n - 1) - m * (m - 1)) \div 2\) for \(m n ::\) nat
by (cases n) (simp-all add: atLeastLessThanSuc-atLeastAtMost Sum-Icc-nat)

61.9.6 Division remainder

lemma range-mod:
fixes \(n ::\) nat
assumes \(n > 0\)
shows range \((\lambda m. m \mod n) = \{0..<n\} \) \(\text{is } ?A = ?B\)
proof (rule set-eqI)
fix \(m\)
show \(m \in ?A \iff m \in ?B\)
proof
assume \(m \in ?A\)
with assms show \(m \in ?B\)
by auto
next
assume \(m \in ?B\)
moreover have \(m \mod n \in ?A\)
by (rule rangeI)
ultimately show \(m \in ?A\)
by simp
qed
qed

61.10 Products indexed over intervals

syntax (ASCII)
- from-to-prod :: idt \(\Rightarrow\) 'a \(\Rightarrow\) 'a \(\Rightarrow\) 'b \(\Rightarrow\) 'b \(\Rightarrow\) '((PROD - = -../= -) [0,0,0,10] 10)
- from-upto-prod :: idt \(\Rightarrow\) 'a \(\Rightarrow\) 'a \(\Rightarrow\) 'b \(\Rightarrow\) 'b \(\Rightarrow\) '((PROD - = -..</= -) [0,0,0,10] 10)
- up-to-prod :: idt \(\Rightarrow\) 'a \(\Rightarrow\) 'b \(\Rightarrow\) 'b \(\Rightarrow\) '((PROD - <=/= -) [0,0,0,10] 10)

syntax (latex-prod output)
- from-to-prod :: idt \(\Rightarrow\) 'a \(\Rightarrow\) 'a \(\Rightarrow\) 'b \(\Rightarrow\) 'b
\(\langle \prod_{= -} \rangle [0,0,0,10] 10\)
61.10.1 Telescoping products

**Lemma prod-telescope:**

**Proof**

- **Cases** \(i \leq j\)
  - **Case** True
    - Then show \(\neg \text{thesis}\)
      - By (metis le_iff_add prod-int-plus-eq)
  - **Case** False
    - Then show \(\neg \text{thesis}\)
      - By auto

qed
THEORY “Set-Interval”

fixes f::nat ⇒ 'a::field
assumes ∨ i. i≤n ⇒ f i ≠ 0
shows (∏ i<n. f (Suc i) / f i) = f n / f 0
using assms by (induction n) auto

lemma prod-lessThan-telescope':
fixes f::nat ⇒ 'a::field
assumes ∨ i. i≤n ⇒ f i ≠ 0
shows (∏ i<n. f i / f (Suc i)) = f 0 / f n
using assms by (induction n) auto

61.11 Efficient folding over intervals

function fold-atLeastAtMost-nat where
|simp del|: fold-atLeastAtMost-nat f a (b::nat) acc =
(if a > b then acc else fold-atLeastAtMost-nat f (a+1) b (f a acc))
by pat-completeness auto
termination by (relation measure (λ(-,-,b,). Suc b − a)) auto

lemma fold-atLeastAtMost-nat:
assumes comp-fun-commute f
shows fold-atLeastAtMost-nat f a b acc = Finite-Set.fold f acc {a..b}
using assms
proof (induction f a b acc rule: fold-atLeastAtMost-nat.induct, goal-cases)
case 1 f a b acc
interpret comp-fun-commute f by fact
show ?case
proof (cases a > b)
case True
thus ?thesis by (subst fold-atLeastAtMost-nat.simps) auto
next
case False
with 1 show ?thesis
by (subst fold-atLeastAtMost-nat.simps)
(auto simp: atLeastAtMost-insertL[ symmetric] fold-fun-left-comm)
qed
qed

lemma sum-atLeastAtMost-code:
sum f {a..b} = fold-atLeastAtMost-nat (λa acc. f a + acc) a b 0
proof –
have comp-fun-commute (λa. (+) (f a))
  by unfold-locales (auto simp: o-def add-ac)
thus ?thesis
  by (simp add: sum.eq-fold fold-atLeastAtMost-nat o-def)
qed

lemma prod-atLeastAtMost-code:
prod f {a..b} = fold-atLeastAtMost-nat (λa acc. f a * acc) a b 1
proof

have comp-fun-commute (λa. (♯) (f a))
  by unfold-locales (auto simp: o-def mult-ac)
thus thesis
  by (simp add: prod.eq-fold-fold-atLeastAtMost-nat o-def)
qed

lemma pairs-le-eq-Sigma: \{ (i, j). i + j ≤ m \} = Sigma (atMost m) (λr. atMost (m − r))
  for m :: nat
by auto

lemma sum-up-index-split: (∑ k≤m + n. f k) = (∑ k≤m. f k) + (∑ k = Suc m..m + n. f k)
by (metis atLeast0AtMost Suc-eq-plus1 le0 sum.ub-add-nat)

lemma Sigma-interval-disjoint: (SIGMA i:A. {..v i}) ∩ (SIGMA i:A. {v i <..w})
  = {}
  for w :: 'a::order
by auto

lemma product-atMost-eq-Un: A × {..m} = (SIGMA i:A. {..m − i}) ∪ (SIGMA i:A. {m − i <..m})
  for m :: nat
by auto

lemma polynomial-product:
  fixes x :: 'a::idom
  assumes m: ∃i. i > m ⇒ a i = 0
  and n: ∃j. j > n ⇒ b j = 0
  shows (∑ i≤m. (a i) * x ^ i) * (∑ j≤n. (b j) * x ^ j) =
  (∑ r≤m + n. (∑ k≤r. (a k) * (b (r − k)))) * x ^ r
proof

  have \( \bigwedge i. j. [m + n − i < j; a i ≠ 0] ⇒ b j = 0 \)
    by (meson le-add-diff leI le-less-trans m n)
  then have $\mathrel{\vDash} \bigwedge (i,j) \in (\Sigma i. m + n - i < n). a i * x ^ i * (b j * x ^ j) = 0$
    by (clarsimp simp add: sum-Un Sigma-interval-disjoint intro: sum.neutral)
  have (∑ i≤m. (a i) * x ^ i) * (∑ j≤n. (b j) * x ^ j) = (∑ i≤m. ∑ j≤n. (a i * x ^ i) * (b j * x ^ j))
    by (rule sum-product)
  also have $\mathrel{\ldots} = (∑ i≤m + n. ∑ j≤n + m. a i * x ^ i * (b j * x ^ j))$
    using assms by (auto simp: sum-up-index-split)
  also have $\mathrel{\ldots} = (∑ r≤m + n. ∑ j≤m + n − r. a r * x ^ r * (b j * x ^ j))$
    by (simp add: add-ac sum Sigma product-atMost-eq-Un sum Sigma interval-disjoint)

  $\mathrel{\vDash}$
  also have $\mathrel{\ldots} = (∑ (i,j) \in \{ (i,j). i + j ≤ m + n \}. (a i * x ^ i) * (b j * x ^ j))$
by (auto simp: pairs-le-eq-Sigma sum.Sigma)
also have ... = (∑ k≤m + n. ∑ i≤k. a i * x ^ i * (b * x ^ (k - i)))
by (rule sum.triangle-reindex-eq)
also have ... = (∑ r≤m + n. (∑ k≤r. (a k) * (b * x ^ (r - k))) * x ^ r)
by (auto simp: algebra-simps sum-distrib-left simp flip: power-add intro: sum.cong)
finally show thesis .
qed
end

62 Decision Procedure for Presburger Arithmetic

theory Presburger
imports Groebner-Basis Set-Interval
keywords try0 :: diag
begin

ML-file ⟨Tools/Qelim/qelim.ml⟩
ML-file ⟨Tools/Qelim/cooper-procedure.ml⟩

62.1 The −∞ and +∞ Properties

lemma minf:
[∃ (z ::'a::linorder).∀ x<z. P x = P' x; ∃ z.∀ x<z. Q x = Q' x]
⟹ ∃ z.∀ x<z. (P x ∧ Q x) = (P' x ∧ Q' x)
[∃ (z ::'a::linorder).∀ x<z. P x = P' x; ∃ z.∀ x<z. Q x = Q' x]
⟹ ∃ z.∀ x<z. (P x ∨ Q x) = (P' x ∨ Q' x)
∃ (z ::'a::linorder).∀ x<z.(x = t) = False
∃ (z ::'a::linorder).∀ x<z.(x ≠ t) = True
∃ (z ::'a::linorder).∀ x<z.(x < t) = True
∃ (z ::'a::linorder).∀ x<z.(x > t) = False
∃ (z ::'a::linorder).∀ x<z.(x ≥ t) = False
∃ z.∀ x.(x:b::linorder,plus,Rings.dvd)<z. (d dvd x + s) = (d dvd x + s)
∃ z.∀ x.(x:b::linorder,plus,Rings.dvd)<z. (~ d dvd x + s) = (~ d dvd x + s)
∃ z.∀ x<z. F = F

proof safe
  fix z1 z2
  assume ∀ x<z1. P x = P' x and ∀ x<z2. Q x = Q' x
  then have ∀ x < min z1 z2. (P x ∧ Q x) = (P' x ∧ Q' x)
    by simp
  then show ∃ z. ∀ x<z. (P x ∧ Q x) = (P' x ∧ Q' x)
    by blast
next
  fix z1 z2
  assume ∀ x<z1. P x = P' x and ∀ x<z2. Q x = Q' x
  then have ∀ x < min z1 z2. (P x ∨ Q x) = (P' x ∨ Q' x)
    by simp
  then show ∃ z. ∀ x<z. (P x ∨ Q x) = (P' x ∨ Q' x)
by blast
next
have \forall x < t. \ x \leq t
  by fastforce
then show \exists z. \forall x < z. \ (x \leq t) = True
  by auto
next
have \forall x < t. \neg t < x
  by fastforce
then show \exists z. \forall x < z. \ (t < x) = False
  by auto
next
have \forall x < t. \neg t \leq x
  by fastforce
then show \exists z. \forall x < z. \ (t \leq x) = False
  by auto
qed auto

lemma pinf:
\[ \exists (z ::'a::linorder). \forall x > z. \ P x = P' x; \exists z. \forall x > z. \ Q x = Q' x \] 
\[ \Rightarrow \exists z. \forall x > z. \ (P x \land Q x) = (P' x \land Q' x) \] 
\[ \exists (z ::'a::linorder). \forall x > z. \ Q x = Q' x \]
\[ \Rightarrow \exists z. \forall x > z. \ (P x \lor Q x) = (P' x \lor Q' x) \]
\[ \exists (z ::'a::linorder). \forall x > z. (x = t) = False \]
\[ \exists (z ::'a::linorder). \forall x > z. (x \neq t) = True \]
\[ \exists (z ::'a::linorder). \forall x > z. (x < t) = False \]
\[ \exists (z ::'a::linorder). \forall x > z. (x \leq t) = False \]
\[ \exists (z ::'a::linorder). \forall x > z. (x > t) = True \]
\[ \exists (z ::'a::linorder). \forall x > z. (x \geq t) = True \]
\[ \exists z. \forall (x ::'b::linorder, plus, Rings, dvd). \forall x > z. (\neg (d dvd x + s)) = (\neg (d dvd x + s)) \]
\[ \exists z. \forall (x ::'b::linorder, plus, Rings, dvd). \forall x > z. (\neg (d dvd x + s)) = (\neg (d dvd x + s)) \]
\[ \exists z. \forall x > z. \ F = F \]

proof safe
fix z1 z2
assume \forall x > z1. \ P x = P' x and \forall x > z2. \ Q x = Q' x
then have \forall x > max z1 z2. \ (P x \land Q x) = (P' x \land Q' x)
  by simp
then show \exists z. \forall x > z. \ (P x \land Q x) = (P' x \land Q' x)
  by blast
next
fix z1 z2
assume \forall x > z1. \ P x = P' x and \forall x > z2. \ Q x = Q' x
then have \forall x > max z1 z2. \ (P x \lor Q x) = (P' x \lor Q' x)
  by simp
then show \exists z. \forall x > z. \ (P x \lor Q x) = (P' x \lor Q' x)
  by blast
next
have \forall x > t. \neg x < t
  by fastforce
```
then show \( \exists z. \forall x > z. x < t = False \)
  by blast
next
have \( \forall x > t. \neg x \leq t \)
  by fastforce
then show \( \exists z. \forall x > z. x \leq t = False \)
  by blast
next
have \( \forall x > t. t \leq x \)
  by fastforce
then show \( \exists z. \forall x > z. t \leq x = True \)
  by blast
qed auto

lemma inf-period:
[\[ \forall x. P x = P (x - k*D); \forall x. Q x = Q (x - k*D) \]\]

[\[ \forall x. (P x \land Q x) = (P (x - k*D) \land Q (x - k*D)) \]\]

[\[ \forall x. Q x = Q (x - k*D) \]\]

(d::"comm-ring,Rings.dvd") dvd D \[\rightarrow\] \forall x k. (d dvd x + t) = (d dvd (x - k*D) + t)

(d::"comm-ring,Rings.dvd") dvd D \[\rightarrow\] \forall x k. (\neg d dvd x + t) = (\neg d dvd (x - k*D) + t)

\forall x k. F = F
apply (auto elim!: dvdE simp add: algebra-simps)
unfolding mult.assoc [symmetric] distrib-right [symmetric] left-diff-distrib [symmetric]
unfolding dvd-def mult.commute [of d]
by auto
```
\(x - D > t\))
\[D > 0 : t - 1 \in B \implies (\forall (x :: \text{int}). (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (x \geq t) \implies (x - D \geq t))\]
\[d \vdash d \in D \implies (\forall (x :: \text{int}). (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (d \vdash d + x + t) \implies (d \vdash d (x - D + t))\]
\[d \vdash d \in D \implies (\forall (x :: \text{int}). (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (\neg d \vdash d + x + t) \implies (\neg d \vdash d (x - D + t))\]
\[\forall x. (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies F \implies F\]

proof
\[(\text{blast, blast})\]

assume dp; \(D > 0\) and \(tB; t \in B\)

show \((\forall x. (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (x = t) \implies (x - D = t))\)

apply \((\text{rule allI, rule impI, erule ballE[where } x = t\text{], erule ballE[where } x = t - 1])\)

apply algebra using dp tB by simp-all

next

assume dp; \(D > 0\) and \(tB; t \in B\)

show \((\forall x. (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (x = t) \implies (x - D = t))\)

apply \((\text{rule allI, rule impI, erule ballE[where } x = D\text{], erule ballE[where } x = t])\)

apply algebra

using dp tB by simp-all

next

assume dp; \(D > 0\) thus \((\forall x. (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (x < t) \implies (x - D < t))\) by arith

next

assume dp; \(D > 0\) thus \((\forall x. (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (x \leq t) \implies (x - D \leq t))\) by arith

next

assume dp; \(D > 0\) and \(tB; t \in B\)

\{ fix x assume nob; \(\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j\) and \(g; x > t\) and \(ng; \neg (x - D) > t\) \\
hence \(x - t \leq D\) and \(1 \leq x - t\) by simp+ \\
hence \(\exists j \in \{1 .. D\}. x - t = j\) by auto \\
hence \(\exists j \in \{1 .. D\}. x = t + j\) by \((\text{simp add: algebra-simps})\) \\
with \(\text{nob tB have False by simp}\) \\
thus \((\forall x. (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (x > t) \implies (x - D > t))\) by blast

next

assume dp; \(D > 0\) and \(tB; t \in B\)

\{ fix x assume nob; \(\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j\) and \(g; x > t\) and \(ng; \neg (x - D) \geq t\) \\
hence \(x - (t - 1) \leq D\) and \(1 \leq x - (t - 1)\) by simp+ \\
hence \(\exists j \in \{1 .. D\}. x - (t - 1) = j\) by auto \\
hence \(\exists j \in \{1 .. D\}. x = (t - 1) + j\) by \((\text{simp add: algebra-simps})\) \\
with \(\text{nob tB have False by simp}\) \\
thus \((\forall x. (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (x \geq t) \implies (x - D \geq t))\) by blast

next

assume dp; \(d \vdash d \in D\)

\{ fix x assume H; \(d \vdash d + x + t\) with \(d \vdash d \in D\) and \(t\) by algebra\}

thus \((\forall (x :: \text{int}). (\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (d \vdash d + x + t) \implies (d \vdash d + x + t + t)\) by simp
next
assume d: d dvd D
{fix x assume H: ¬(d dvd x + t) with d have ¬ d dvd (x - D) + t
  by (clarsimp simp add: dvd-def,erule-tac x= ka + k in allE,simp add: algebra_simps)
} thus ∀(x::int).(∀j∈{1 .. D}. ∀ b∈B. x ≠ b + j) → (¬ d dvd x+t) → (¬ d dvd (x - D) + t) by auto
qed blast

lemma aset:
[∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → P x → P(x + D) ;
 ∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → Q x → Q(x + D)] ⇒
∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (P x ∧ Q x) → (P(x + D) ∧ Q (x + D))
[D>0; t + 1 ∈ A] ⇒ (∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (x = t) → (x + D = t))
[D>0; t ∈ A] ⇒ (∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (x ≠ t) → (x + D ≠ t))
[D>0; t + 1 ∈ A] ⇒ (∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (x < t) → (x + D < t))
[D>0; t ∈ A] ⇒ (∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (x ≤ t) → (x + D ≤ t))
D>0 ⇒ (∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (x ≥ t) → (x + D ≥ t))
d dvd D ⇒ (∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (d dvd x+t) → (d dvd (x + D) + t))
d dvd D ⇒ (∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (¬ d dvd x+t) → (¬ d dvd (x + D) + t))
∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → F → F
proof (blast, blast)
assume dp: D > 0 and tA: t + 1 ∈ A
show (∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (x = t) → (x + D = t))
  apply (rule allI, rule impI,erule ballE[where x=1],erule ballE[where x=t + 1])
  using dp tA by simp-all
next
assume dp: D > 0 and tA: t ∈ A
show (∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (x ≠ t) → (x + D ≠ t))
  apply (rule allI, rule impI,erule ballE[where x=D],erule ballE[where x=t])
  using dp tA by simp-all
next
assume dp: D > 0 thus (∀ x.(∀ j∈{1 .. D}. ∀ b∈A. x ≠ b - j) → (x > t) → (x + D > t)) by arith
next
  assume dp: \(D > 0\) thus \(\forall x. (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \rightarrow (x \geq t) \rightarrow (x + D \geq t)\) by arith
next
  assume dp: \(D > 0\) and \(\forall A. t \in A\)
  \{ fix \(x\) assume \(\exists j \in \{1 .. D\}. \forall b \in A. x \neq b - j\) and \(g: x < t\) and \(ng: \neg(x + D) < t\)
  hence \((t + 1) - x \leq D\) and \(1 \leq (t + 1) - x\) by simp+
  hence \(\exists j \in \{1 .. D\}. t - x = j\) by auto
  hence \(\exists j \in \{1 .. D\}. x = (t + 1) - j\) by (auto simp add: algebra-simps)
  with \(\forall A. t \in A\) have False by simp\}
  thus \(\forall x. (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \rightarrow (x < t) \rightarrow (x + D < t)\) by blast
next
  assume dp: \(D > 0\) and \(\forall A. t + 1 \in A\)
  \{ fix \(x\) assume \(\exists j \in \{1 .. D\}. \forall b \in A. x \neq b - j\) and \(g: x \leq t\) and \(ng: \neg(x + D) \leq t\)
  hence \((t + 1) - x \leq D\) and \(1 \leq (t + 1) - x\) by simp-all add: algebra-simps
  hence \(\exists j \in \{1 .. D\}. (t + 1) - x = j\) by auto
  hence \(\exists j \in \{1 .. D\}. x = (t + 1) - j\) by (auto simp add: algebra-simps)
  with \(\forall A. t \in A\) have False by simp\}
  thus \(\forall x. (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \rightarrow (x \leq t) \rightarrow (x + D \leq t)\) by blast
next
  assume d: \(d \ dvd D\)
  have \(\exists x. d \ dvd x + t \implies \ d \ dvd x + D + t\)
  proof -
    fix \(x\)
    assume H: \(d \ dvd x + t\)
    then obtain \(k\) where \(x + t = d * k\)
    unfolding dvd-def by blast
    moreover from \(d\) obtain \(k\) where \(*:D = d * k\)
    unfolding dvd-def by blast
    ultimately have \(x + d * k + t = d * (ka + k)\)
    by (simp add: algebra-simps)
    then show \(d \ dvd (x + D) + t\)
    using \(*\) unfolding dvd-def by blast
  qed
  thus \(\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \rightarrow (d \ dvd x+t) \rightarrow (d \ dvd (x + D) + t)\) by simp
next
  assume d: \(d \ dvd D\)
  \{ fix \(x\) assume H: \(\neg(d \ dvd x + t)\) with \(d\) have \(\neg d \ dvd (x + D) + t\)
    by (clarsimp simp add: dvd-def,erule-tac \(x::= ka - k\) in allE,simp add: algebra-simps)\}
  thus \(\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \rightarrow (\neg d \ dvd x+t) \rightarrow (\neg d \ dvd (x + D) + t)\) by auto
qed blast
62.3 Cooper’s Theorem \(-\infty\) and \(+\infty\) Version

62.3.1 First some trivial facts about periodic sets or predicates

**Lemma** periodic-finite-ex:

- **Assumes** dpos: \((0 :: \text{int}) < d\) and modd: \(\forall x. P x = P(x - k \cdot d)\)
- **Shows** \((\exists x. P x) = (\exists j \in \{1 \ldots d\}. P j)\)

**Proof**

- Assume \(?LHS\)
- Then obtain \(x\) where \(P x\) ..
  - Have \(x \mod d = x - (x \div d) \cdot d\) by (simp add: mult-div-mod-eq [symmetric] ac-simps eq-diff-eq)
  - Hence \(P mod: P x = P(x \mod d)\) using modd by simp
- Show \(?RHS\)
  - Proof (cases)
    - Assume \(x \mod d = 0\)
      - Hence \(P 0\) using \(P P mod\) by simp
      - Moreover have \(P 0 = P(0 - (-1) \cdot d)\) using modd by blast
      - Ultimately have \(P d\) by simp
      - Moreover have \(d \in \{1 \ldots d\}\) using dpos by simp
      - Ultimately show \(?RHS\) ..
  - Next
    - Assume not0: \(x \mod d \neq 0\)
      - Have \(P(x \mod d)\) using dpos \(P P mod\) by simp
      - Moreover have \(x \mod d \in \{1 \ldots d\}\)
      - Proof –
        - From dpos have \(0 \leq x \mod d\) by (rule pos-mod-sign)
        - Moreover from dpos have \(x \mod d < d\) by (rule pos-mod-bound)
        - Ultimately show \(?thesis\) using not0 by simp
    - Qed
  - Ultimately show \(?RHS\) ..
  - Qed
  - Qed auto

62.3.2 The \(-\infty\) Version

**Lemma** decrec-lemma: \(0 < (d :: \text{int}) \Rightarrow x - (|x - z| + 1) \cdot d < z\)

by (induct rule: int-gr-induct) (simp-all add: int-distrib)

**Lemma** increc-lemma: \(0 < (d :: \text{int}) \Rightarrow z < x + (|x - z| + 1) \cdot d\)

by (induct rule: int-gr-induct) (simp-all add: int-distrib)

**Lemma** decrec-mult-lemma:

- Assumes dpos: \((0 :: \text{int}) < d\) and minus: \(\forall x. P x \rightarrow P(x - d)\) and knneg: \(0 \leq k\)
- Shows \(\forall x. P x \rightarrow P(x - k \cdot d)\)
  - Using knneg
  - Proof (induct rule:int-ge-induct)
  - Case base thus \(?case\) by simp
next
  case (step i)
    { fix x
      have P x → P (x - i * d) using step.hyps by blast
      also have ... → P(x - (i + 1) * d) using minus[THEN spec, of x - i * d]
        by (simp add: algebra-simps)
      ultimately have P x → P(x - (i + 1) * d) by blast }
    thus ?case ..
  qed

lemma minusinfinity:
  assumes dpos: 0 < d and
    P1eqP1: ∀ x k. P1 x = P1(x - k* d) and ePeqP1: ∃ z::int. ∀ x. x < z → (P x = P1 x)
  shows (∃ x. P1 x) → (∃ x. P x)
proof
  assume eP1: ∃ x. P1 x
  then obtain x where P1: P1 x ..
  from ePeqP1 obtain z where P1eqP: ∀ x. x < z → (P x = P1 x) ..
  let w = x - ([|x - z|] + 1) * d
  from dpos have w: ?w < z by (rule decr-lemma)
  have P1 x = P1 ?w using P1eqP1 by blast
  also have P(?w) using w P1eqP by blast
  finally have P ?w using P1 by blast
  thus ∃ x. P x ..
  qed

lemma cpmi:
  assumes dp: 0 < D and p1:∃ z. ∀ x< z. P x = P' x
  and nb:∀ x.(∀ j ∈ {1..D}. ∀(b::int) ∈ B. x ≠ b+j) → P (x) → P (x - D)
  and pd: ∀ x k. P' x = P'(x - k* D)
  shows (∃ x. P x) = (∃ j ∈ {1..D}. P' j) ∨ (∃ j ∈ {1..D}. ∃ b ∈ B. P (b+j))
  (is ?L = (?R1 ∨ ?R2))
proof
  { assume ?R2 hence ?L by blast }
moreover
  { assume H:?R1 hence ?L using minusinfinity[OF dp pd p1] periodic-finite-ex[OF dp pd] by simp }
moreover
  { fix x
    assume P: P x and H: ¬ ?R2
      { fix y assume ¬ (∃ j ∈ {1..D}. ∃ b∈B. P (b + j)) and P: P y
          hence ¬(∃ j::int) ∈ {1..D}. ∃(b::int) ∈ B. y = b+j) by auto
            with nb P have P(y - D) by auto 
          hence ∀ x. ¬(∃ j::int) ∈ {1..D}. ∃(b::int) ∈ B. P(b+j)) → P (x) → P (x - D)
            by blast
            with H P have th: ∀ x. P x → P (x - D) by auto
          from p1 obtain z where z: ∀ x. x < z → (P x = P1 x) by blast
          let ?y = x - ([|x - z|] + 1)*D

 THEORY "Presburger"
have \( z < 0 \rightarrow (|x - z| + 1) \) by \textit{arith}.

from \( dp \) have \( y < z \) using \textit{decr-lemma}[OF \( dp \)] by \textit{simp}

have \( \theta_2 : P' \) by \textit{auto}

with \textit{periodic-finite-ex}[OF \( dp \)]

have \( P \) by \textit{blast}

ultimately show \( \theta \) by \textit{blast}

\textbf{62.3.3 The} \( +\infty \) \textbf{Version}

\textbf{lemma} \textit{plusinfinity}:\n\begin{itemize}
\item \textit{dpos}; \( (0::\text{int}) < d \) and
\end{itemize}

\begin{align*}
P1eqP1 & : \forall x k. 
P' x = P'(x - k*d) \text{ and } ePeqP1 : \exists z. \forall x > z. 
P x = P' x 
\end{align*}

shows \( \exists x. P' x \rightarrow (\exists x. P x) \)

\textbf{proof}
\begin{itemize}
\item assume \( eP1 : \exists x. P' x \)
\item then obtain \( x \) where \( P1 : P' x \).
\item from \( eP1 \) obtain \( z \) where \( P1eqP : \forall x > z. 
P x = P' x \).
\end{itemize}

let \( ?w' = x + (|x - z| + 1) * d \)

let \( ?w = x - (|x - z| + 1) * d \)

have \( \textit{ww}'[\textit{simp}]: ?w = ?w' \) by (\textit{simp add: algebra-simps})

from \( \textit{dpos} \) have \( w : ?w > z \)

by (\textit{simp only; \textit{ww'} incr-lemma})

hence \( P' x = P' ?w \) using \( P1eqP1 \) by \textit{blast}

also have \( \ldots = P(?w) \) using \textit{w P1eqP} by \textit{blast}

finally have \( P ?w \) using \( P1 \) by \textit{blast}

thus \( \exists x. P x \).

\textbf{qed}

\textbf{lemma} \textit{incr-mult-lemma}:
\begin{itemize}
\item \textit{dpos}; (0::int) < d and \textit{plus}; \forall x::\text{int}. 
P x \rightarrow P(x + d) \text{ and } \textit{kneg}; 0 <= k
\end{itemize}

shows \( \forall x. P x \rightarrow P(x + k*d) \)

\textbf{using} \textit{kneg}

\textbf{proof} (\textit{induct rule: int-ge-induct})

\textbf{case} \textit{base} \textbf{thus} \( \?case \) by \textit{simp}

\textbf{next}

\textbf{case} (\textit{step i})

\textbf{fix} \( x \)

\textbf{have} \( P x \rightarrow P(x + i * d) \) \textbf{using} \textit{step.hyps} by \textit{blast}

\textbf{also have} \( \ldots \rightarrow P(x + (i + 1) * d) \) \textbf{using} \textit{plus}[THEN \textit{spec}, \textit{of} \( x + i * d \)]

\textbf{by} (\textit{simp add: int-distrib ac-simps})

\textbf{ultimately have} \( P x \rightarrow P(x + (i + 1) * d) \) \textbf{by} \textit{blast}

\textbf{thus} \( \?case \).

\textbf{qed}

\textbf{lemma} \textit{cppi}:
\begin{itemize}
\item \textit{dpos}; \( 0 < D \) and \( p1 : \exists z. \forall x > z. \ P x = P' x \)
\item \textit{and} \( \textit{ub} ; \forall x. (\forall j \in \{1..D\}. \ \forall (b::\text{int}) \in A. \ x \neq b - j) \rightarrow P(x) \rightarrow P(x + D) \)
\end{itemize}
and pd: ∀ x k. P' x = P' (x - k ∗ D)
shows (∃ x. P x) = ((∃ j ∈ {1..D} . P' j) ∨ (∃ j ∈ {1..D}. b ∈ A. P (b - j)))
is ?L = (?R1 ∨ ?R2)
proof
{ assume ?R2 hence ?L by blast }
moreover
moreover
{ fix x 
  assume P: P x and H: ¬ ?R2
  { fix y assume ¬ (∃ j ∈ {1..D}. \exists b ∈ A. P (b - j)) and P: P y
    hence ¬ (∃ (j::int) ∈ {1..D}. (b::int) ∈ A. y = b - j) by auto
    with \nb P have P (y + D) by auto }
  hence \forall x. ¬ (∃ (j::int) ∈ {1..D}. (b::int) ∈ A. P(b - j)) → P x → P (x + D) by blast
  with H P have th: \forall x. P x → P (x + D) by auto
  from p1 obtain z where z: \forall x. x > z → (P x = P' x) by blast
  let \?y = x + ((|x| - 1)| + 1) ∗ D
  have \zp: \theta ≤ ((|x| - 1)| + 1) by arith
  from dp have \yz: \?y > z using incr-lemma[OF dp] by simp
  from \yz[rule-format, OF dp] incr-mult-lemma[OF dp th \zp, rule-format, OF P] have th2: \?y \by auto
  with periodic-finite-ex[OF dp pd]
  have ?R1 by blast }
ultimately show \?thesis by blast
qed

lemma simp-from-to: \{i..j::int\} = \{if \ j < \ i then \{\} else insert \ i \ \{i+1..j\}\}
apply(simp add:atLeastAtMost Def atLeast-def atMost-def)
apply(.fastforce)
done

theorem unity-coef-ex: (∃ (x::'a::{semiring_0,Rings.dvd})). P (l * x) \equiv (∃ x. l dvd (x + 0) ∩ P x)
unfolding dvd-def by (rule eq-reflection, rule iffI) auto

lemma zdvd-mono:
  fixes k m t :: int
  assumes k ≠ 0
  shows m dvd t \equiv k * m dvd k * t
  using assms by simp

lemma uminus-dvd-conv:
  fixes d t :: int
  shows d dvd t \equiv - d dvd t and d dvd t \equiv d dvd - t
  by simp-all

Theorems for transforming predicates on nat to predicates on int
**THEORY** “Presburger”

**lemma** *zdif-int-split*: \(P (\text{int } (x - y)) = ((y \leq x \implies P (\text{int } x - \text{int } y)) \land (x < y \implies P \theta))\)

by (cases \(y \leq x\)) (simp-all add: of-nat-diff)

Specific instances of congruence rules, to prevent simplifier from looping.

**theorem** *imp-le-cong*:

\[ [\{ x = x'; 0 \leq x' \implies P = P' \}] \implies (0 \leq (x::\text{int}) \implies P) = (0 \leq x' \implies P') \]

by simp

**theorem** *conj-le-cong*:

\[ [\{ x = x'; 0 \leq x' \implies P = P' \}] \implies (0 \leq (x::\text{int}) \land P) = (0 \leq x' \land P') \]

by (simp cong: conj-cong)

**ML-file** (Tools/Qelim/cooper.ML)

**method-setup** presburger = :

let

\(\text{fun keyword } k = \text{Scan.lift } (\text{Args.$$ k -- Args.colon}) \gg K ()}\)

\(\text{fun simple-keyword } k = \text{Scan.lift } (\text{Args.$$ k}) \gg K ()}\)

val addN = add
val delN = del
val elimN = elim
val any-keyword = keyword addN || keyword delN || simple-keyword elimN
val thms = Scan.repeats (Scan.unless any-keyword Attrib.multi-thm)
in

\(\text{Scan.optional } (\text{simple-keyword } elimN \gg K \text{ false true -- --})\)

\(\text{Scan.optional } (\text{keyword } addN \|-- \text{thms}) \|-- --\)

\(\text{Scan.optional } (\text{keyword } delN \|-- \text{thms}) \|-- >\)

(fun ((elim, add-ths), del-ths) => fn ctxt =>
   SIMPLE-METHOD' (Cooper.tac elim add-ths del-ths ctxt))
end

Cooper’s algorithm for Presburger arithmetic

**declare** mod-eq-0-iff-dvd [presburger]
**declare** mod-by-Suc-0 [presburger]
**declare** mod-0 [presburger]
**declare** mod-by-1 [presburger]
**declare** mod-self [presburger]
**declare** div-by-0 [presburger]
**declare** mod-by-0 [presburger]
**declare** mod-div-trivial [presburger]
**declare** mult-div-mod-eq [presburger]
**declare** div-mult-mod-eq [presburger]
**declare** mod-mult-self1 [presburger]
**declare** mod-mult-self2 [presburger]
**declare** mod2-Suc-Suc [presburger]
**declare** not-mod-2-eq-0-eq-1 [presburger]
**declare** nat-zero-less-power-iff [presburger]
lemma [presburger, algebra]: \( m \mod 2 = (1::nat) \iff \neg 2 \, \text{dvd} \, m \) by presburger

lemma [presburger, algebra]: \( m \mod 2 = Suc 0 \iff \neg 2 \, \text{dvd} \, m \) by presburger

lemma [presburger, algebra]: \( m \mod (Suc (Suc 0)) = (1::nat) \iff \neg 2 \, \text{dvd} \, m \) by presburger

lemma [presburger, algebra]: \( m \mod 2 = (1::nat) \iff \neg 2 \, \text{dvd} \, m \) by presburger

context semiring-parity
begin

declare even-mult-iff [presburger]

declare even-power [presburger]

lemma [presburger]:
\[
even(a + b) \iff even \, a \land even \, b \lor odd \, a \land odd \, b
\]
by auto

end

context ring-parity
begin

declare even-minus [presburger]

end

context linordered-idom
begin

declare zero-le-power-eq [presburger]

declare zero-less-power-eq [presburger]

declare power-less-zero-eq [presburger]

declare power-le-zero-eq [presburger]

end

declare even-Suc [presburger]

lemma [presburger]:
\[
Suc \, n \, \text{div} \, Suc \, (Suc \, 0) = n \, \text{div} \, Suc \, (Suc \, 0) \iff even \, n
\]
by presburger

declare even-diff-nat [presburger]
lemma [presburger]:
    fixes k :: int
    shows \((k + 1) \div 2 = k \div 2 \iff \text{even } k\)
    by presburger

decision

lemma [presburger]:
    fixes k :: int
    shows \((k + 1) \div 2 = k \div 2 + 1 \iff \text{odd } k\)
    by presburger

lemma [presburger]:
    even n \iff \text{even } (\text{int n})
    by simp

62.4 Nice facts about division by \(4::'a\)

lemma even-even-mod-4-iff:
    even \((n::nat) \iff \text{even } (n \mod 4)\)
    by presburger

lemma odd-mod-4-div-2:
    \(n \mod 4 = (3::nat) \implies \text{odd } ((n - \text{Suc 0}) \div 2)\)
    by presburger

lemma even-mod-4-div-2:
    \(n \mod 4 = \text{Suc 0} \implies \text{even } ((n - \text{Suc 0}) \div 2)\)
    by presburger

62.5 Try0

ML-file ⟨Tools/try0.ML⟩

end

63 Bindings to Satisfiability Modulo Theories (SMT)
solvers based on SMT-LIB 2

theory SMT
  imports Numeral-Simprocs
  keywords smt-status :: diag
begin

63.1 A skolemization tactic and proof method

lemma ex-iff-push: \((\exists y. P \iff Q y) \iff (P \implies (\exists y. Q y)) \land ((\forall y. Q y) \implies P)\)
  by metis

ML <
fun moura-tac ctxt = 
  TRY o Atomize-Elim.atomize-elim-tac ctxt THEN'
  REPEAT o EqSubst.eqsubst-tac ctxt [0]
  @{thms choice-iff[symmetric] bchoice-iff[symmetric]} THEN'
  TRY o Simplifier.asm-full-simp-tac
  (clear-simps ctxt addsimps @{thms all-simps ex-simps ex-iff-push}) THEN-ALL-NEW
  Metis-Tactic.metis-tac (take 1 ATP-Proof-Reconstruct.partial-type-encs)
  ATP-Proof-Reconstruct.default-metis-lam-trans ctxt []

method-setup moura = （
  Scan.succeed (SIMPLE-METHOD' o moura-tac)
  › solve skolemization goals, especially those arising from Z3 proofs

hide-fact (open) ex-iff-push

63.2 Triggers for quantifier instantiation

Some SMT solvers support patterns as a quantifier instantiation heuristics. Patterns may either be positive terms (tagged by 'pat') triggering quantifier instantiations – when the solver finds a term matching a positive pattern, it instantiates the corresponding quantifier accordingly – or negative terms (tagged by 'nopat') inhibiting quantifier instantiations. A list of patterns of the same kind is called a multipattern, and all patterns in a multipattern are considered conjunctively for quantifier instantiation. A list of multipatterns is called a trigger, and their multipatterns act disjunctively during quantifier instantiation. Each multipattern should mention at least all quantified variables of the preceding quantifier block.

typedec 'a symb-list

consts 
  Symb-Nil :: 'a symb-list
  Symb-Cons :: 'a ⇒ 'a symb-list ⇒ 'a symb-list

typedec pattern

consts
  pat :: 'a ⇒ pattern
  nopat :: 'a ⇒ pattern

definition trigger :: pattern symb-list symb-list ⇒ bool ⇒ bool where
  trigger - P = P

63.3 Higher-order encoding

Application is made explicit for constants occurring with varying numbers of arguments. This is achieved by the introduction of the following constant.
definition fun-app :: 'a ⇒ 'a where fun-app f = f

Some solvers support a theory of arrays which can be used to encode higher-order functions. The following set of lemmas specifies the properties of such (extensional) arrays.

lemmas array-rules = ext fun-upd-apply fun-upd.same fun-upd-other fun-upd-upd fun-app-def

63.4 Normalization

lemma case-bool-if[abs-def]: case-bool x y P = (if P then x else y)
  by simp

lemmas Ex1-def-raw = Ex1-def[abs-def]
lemmas Ball-def-raw = Ball-def[abs-def]
lemmas Bex-def-raw = Bex-def[abs-def]
lemmas abs-if-raw = abs-if[abs-def]
lemmas min-def-raw = min-def[abs-def]
lemmas max-def-raw = max-def[abs-def]

lemma nat-zero-as-int:
  0 = nat 0
  by simp

lemma nat-one-as-int:
  1 = nat 1
  by simp

lemma nat-numeral-as-int:
  numeral = (λi. nat (numeral i)) by simp

lemma nat-less-as-int:
  < = (λa b. int a < int b) by simp

lemma nat-leq-as-int:
  ≤ = (λa b. int a ≤ int b) by simp

lemma Suc-as-int:
  Suc = (λa. nat (int a + 1)) by (rule ext) simp

lemma nat-plus-as-int:
  + = (λa b. nat (int a + int b)) by (rule ext)+ simp

lemma nat-minus-as-int:
  − = (λa b. nat (int a − int b)) by (rule ext)+ simp

lemma nat-times-as-int:
  ∗ = (λa b. nat (int a ∗ int b)) by (simp add: nat-mult-distrib)

lemma nat-mod-as-int:
  mod = (λa b. nat (int a mod int b)) by (simp add: nat-mod-distrib)

lemma nat-int-comparison:
  fixes a b :: nat
  shows (a = b) = (int a = int b)
  and (a < b) = (int a < int b)
  and (a ≤ b) = (int a ≤ int b)
  by simp-all
lemma \textit{int-ops}: 
  \begin{align*}
    \textit{fixes} \ a \ b \ :: \ \text{n}\text{at} \\
    \textit{shows} \ & \ \textit{int} \ 0 = 0 \\
    & \ \text{and} \ \textit{int} \ 1 = 1 \\
    & \ \text{and} \ \textit{int} \ (\text{numeral} \ n) = \text{numeral} \ n \\
    & \ \text{and} \ \textit{int} \ (\text{Suc} \ a) = \textit{int} \ a + 1 \\
    & \ \text{and} \ \textit{int} \ (a + b) = \textit{int} \ a + \textit{int} \ b \\
    & \ \text{and} \ \textit{int} \ (a - b) = (\text{if} \ \textit{int} \ a < \textit{int} \ b \ \text{then} \ 0 \ \text{else} \ \textit{int} \ a - \textit{int} \ b) \\
    & \ \text{and} \ \textit{int} \ (a \ * \ b) = \textit{int} \ a \ * \ \textit{int} \ b \\
    & \ \text{and} \ \textit{int} \ (a \ \text{div} \ b) = \textit{int} \ a \ \text{div} \ \textit{int} \ b \\
    & \ \text{and} \ \textit{int} \ (a \ \text{mod} \ b) = \textit{int} \ a \ \text{mod} \ \textit{int} \ b \\
    \textit{by} \ (\text{auto \ intro: \ zdiv-int \ zmod-int})
  \end{align*}

lemma \textit{int-if}: 
  \begin{align*}
    \textit{fixes} \ a \ b \ :: \ \text{n}\text{at} \\
    \textit{shows} \ & \ \textit{int} \ (\text{if} \ P \ \text{then} \ a \ \text{else} \ b) = (\text{if} \ P \ \text{then} \ \textit{int} \ a \ \text{else} \ \textit{int} \ b) \\
    \textit{by} \ \text{simpl}
  \end{align*}

63.5 Integer division and modulo for Z3

The following Z3-inspired definitions are overspecified for the case where \( l = 0 \). This Schönheitsfehler is corrected in the \textit{div-as-z3div} and \textit{mod-as-z3mod} theorems.

\begin{align*}
  \text{definition} \ \textit{z3div} :: \ \text{int} \Rightarrow \ \text{int} \Rightarrow \ \text{int} \ \text{where} \\
  \textit{z3div} \ k \ l = \ (\text{if} \ l \geq 0 \ \text{then} \ \textit{k \ div} \ l \ \text{else} \ - (\textit{k \ div} \ - l))
\end{align*}

\begin{align*}
  \text{definition} \ \textit{z3mod} :: \ \text{int} \Rightarrow \ \text{int} \Rightarrow \ \text{int} \ \text{where} \\
  \textit{z3mod} \ k \ l = \ k \ \text{mod} \ (\text{if} \ l \geq 0 \ \text{then} \ l \ \text{else} \ - l)
\end{align*}

\begin{align*}
  \text{lemma} \ \textit{div-as-z3div}: \\
  & \forall \ k \ l. \ \textit{k \ div} \ l = (\text{if} \ l = 0 \ \text{then} \ 0 \ \text{else} \ \text{if} \ l > 0 \ \text{then} \ \textit{z3div} \ k \ l \ \text{else} \ \textit{z3div} \ (\textit{− k}) \ (\textit{− l})) \\
  \textit{by} \ (\text{simpl \ add: \ z3div-def})
\end{align*}

\begin{align*}
  \text{lemma} \ \textit{mod-as-z3mod}: \\
  & \forall \ k \ l. \ \textit{k \ mod} \ l = (\text{if} \ l = 0 \ \text{then} \ k \ \text{else} \ \text{if} \ l > 0 \ \text{then} \ \textit{z3mod} \ k \ l \ \text{else} \ \textit{z3mod} \ (\textit{− k}) \ (\textit{− l})) \\
  \textit{by} \ (\text{simpl \ add: \ z3mod-def})
\end{align*}

63.6 Extra theorems for veriT reconstruction

\begin{align*}
  \text{lemma} \ \textit{verit-sko-forall}: \langle \forall \ x. \ P \ x \rangle \longleftrightarrow \ P \ (\text{SOME} \ x. \ \neg P \ x) \\
  \textit{using} \ \textit{someI} [\text{of} \ \langle \lambda x. \ \neg P \ x \rangle] \\
  \textit{by} \ \text{auto}
\end{align*}

\begin{align*}
  \text{lemma} \ \textit{verit-sko-forall′}: \ \neg P \ (\text{SOME} \ x. \ \neg P \ x) = A \rightarrow \ (\forall \ x. \ P \ x) = A \\
  \textit{by} \ (\text{subst \ verit-sko-forall})
\end{align*}
lemma verit-sko-forall': \langle B = A \implies (\text{SOME } x. \ P x) \rangle = A \equiv (\text{SOME } x. \ P x) = B \rangle
  by auto

lemma verit-sko-forall-indirect: \langle x = (\text{SOME } x. \ \neg P x) \implies (\forall x. \ P x) \iff P x \rangle
  using someI[of \langle \lambda x. \ \neg P x \rangle]
  by auto

lemma verit-sko-forall-indirect2: 
\langle x = (\text{SOME } x. \ \neg P x) \implies (\forall x. \ P x = P' x) \iff P x \rangle
  using someI[of \langle \lambda x. \ \neg P x \rangle]
  by auto

lemma verit-sko-ex: \langle \exists x. \ P x \rangle \iff P (\text{SOME } x. \ P x) \rangle
  using someI[of \langle \lambda x. \ P x \rangle]
  by auto

lemma verit-sko-ex': \langle P \ (\text{SOME } x. \ P x) = A \implies (\exists x. \ P x) = A \rangle
  by (subst verit-sko-ex)

lemma verit-sko-ex-indirect: \langle x = (\text{SOME } x. \ P x) \implies (\exists x. \ P x) \iff P x \rangle
  using someI[of \langle \lambda x. \ P x \rangle]
  by auto

lemma verit-sko-ex-indirect2: \langle x = (\text{SOME } x. \ P x) \implies (\forall x. \ P x = P' x) \iff P x \rangle
  using someI[of \langle \lambda x. \ P x \rangle]
  by auto

lemma verit-Pure-trans: \langle P \equiv Q \implies Q \equiv P \rangle
  by auto

lemma verit-if-cong:
  assumes \langle b \equiv c \rangle
  and \langle c \implies x \equiv u \rangle
  and \langle \neg c \implies y \equiv v \rangle
  shows \langle (\text{if } b \text{ then } x \text{ else } y) \equiv (\text{if } c \text{ then } u \text{ else } v) \rangle
  using assms if-cong[of \langle b \ c \ u \rangle]
  by auto

lemma verit-if-weak-cong': \langle b \equiv c \implies (\text{if } b \text{ then } x \text{ else } y) \equiv (\text{if } c \text{ then } x \text{ else } y) \rangle
  by auto

lemma verit-or-neg:
\langle (A \implies B) \implies B \lor \neg A \rangle,
\langle (\neg A \implies B) \implies B \lor A \rangle
  by auto
lemma verit-subst-bool: ‹P ⇒ f True ⇒ f P›
  by auto

lemma verit-and-pos:
  ‹(a ∧ b ∧ c) ∨ A) ⇒ (a ∧ b ∧ c) ∨ A›
  by blast+

lemma verit-farkas:
  ‹(¬a ⇒ A) ⇒ a ∨ A›
  by blast+

lemma verit-or-pos:
  ‹(A ∧ A') = (c ∧ A) ∨ (¬c ∧ A')›
  by blast+

lemma verit-la-generic:
  ‹(a::int) ≤ x ∨ a = x ∨ a ≥ x›
  by linarith

lemma verit-bfun-elim:
  ‹(if b then P True else P False) = P b›
  ‹(∀b. P b) = (P False ∧ P True)›
  ‹(∃b. P b) = (P False ∨ P True)›
  by (cases b) (auto simp: all-bool-eq ex-bool-eq)

lemma verit-eq-true-simplify:
  ‹(P = True) ≡ P›
  by auto

lemma verit-and-neg:
  ‹(a ∧ b) ⇒ (a ∧ b) ∨ A›
  by blast+

lemma verit-forall-inst:
  ‹(a ↔ B) ⇒ ¬A ∨ B›
  ‹¬A ↔ B ⇒ A ∨ B›
  ‹A ↔ ¬B ⇒ B ∨ A›
  ‹A → B ⇒ A ∨ B›
  ‹¬A → B ⇒ A ∨ B›
  by blast+

lemma verit-eq-transitive:
THEORY “SMT”

\[
\begin{align*}
A = B & \implies B = C \implies A = C, \\
A = B & \implies C = B \implies A = C, \\
B = A & \implies B = C \implies A = C, \\
B = A & \implies C = B \implies A = C.
\end{align*}
\]

by auto

lemma verit-bool-simplify:

\[
\begin{align*}
\neg(P \implies Q) & \iff P \land \neg Q, \\
\neg(P \lor Q) & \iff \neg P \land \neg Q, \\
\neg(P \land Q) & \iff \neg P \lor \neg Q, \\
(P \implies (Q \implies R)) & \iff ((P \land Q) \implies R), \\
(P \implies Q) \implies Q & \iff P \lor Q, \\
(Q \iff (P \lor Q)) & \iff (P \implies Q) \quad \text{— This rule was inverted}
\end{align*}
\]

by auto

We need the last equation for \(\neg (\forall a\ b. \neg a \land b)\)

lemma verit-connective-def: \(\text{— the definition of XOR is missing as the operator is not generated by Isabelle}\)

\[
\begin{align*}
(A = B) & \iff ((A \implies B) \land (B \implies A)), \\
(\text{If } A B C) & = ((A \implies B) \land (\neg A \implies C)), \\
(\exists x. P x) & \iff \neg(\forall x. \neg P x), \\
(\neg(\exists x. P x)) & \iff (\forall x. \neg P x)
\end{align*}
\]

by auto

lemma verit-ite-simplify:

\[
\begin{align*}
(\text{If } True B C) & = B, \\
(\text{If } False B C) & = C, \\
(\text{If } A' B B) & = B, \\
(\text{If } \neg A' B C) & = (\text{If } A' C B), \\
(\text{If } c (\text{If } c A B) C) & = (\text{If } c A C), \\
(\text{If } c C (\text{If } c A B)) & = (\text{If } c C B), \\
(\text{If } A' True False) & = A', \\
(\text{If } A' False True) & \iff \neg A', \\
(\text{If } A' True B') & \iff A' \lor B', \\
(\text{If } A' B' False) & \iff A' \land B', \\
(\text{If } A' False B') & \iff \neg A' \land B', \\
(\text{If } A' B' True) & \iff \neg A' \lor B', \\
x \land True & \iff x, \\
x \lor False & \iff x
\end{align*}
\]

for \(B C :: 'a\) and \(A' B' C' :: bool\)

by auto

lemma verit-and-simplify1:

\[
\begin{align*}
True \land b & \iff b \land True \iff b, \\
False \land b & \iff False \land False \iff False
\end{align*}
\]
\( (c \land \neg c) \leftrightarrow \text{False} \) \( (\neg c \land c) \leftrightarrow \text{False} \)
\( \neg \neg a = a \)  
by auto

lemmas verit-and-simplify = conj-\text{ac} \ de-Morgan-conj \ disj-not1

\textbf{lemma} verit-or-simplify-1:
\( (\text{False} \lor b \leftrightarrow b) \) \( (b \lor \text{False} \leftrightarrow b) \)
\( b \lor \neg b \)  
\( \neg b \lor b \)  
by auto

lemmas verit-or-simplify = disj-\text{ac}

\textbf{lemma} verit-not-simplify:
\( \neg \neg b \leftrightarrow b \) \( \neg \text{True} \leftrightarrow \text{False} \) \( \neg \text{False} \leftrightarrow \text{True} \)
by auto

\textbf{lemma} verit-implies-simplify:
\( (\neg a \rightarrow \neg b) \leftrightarrow (b \rightarrow a) \)
\( (\text{False} \rightarrow a) \leftrightarrow \text{True} \)
\( (a \rightarrow \text{True}) \leftrightarrow \text{True} \)
\( (\text{True} \rightarrow a) \leftrightarrow a \)
\( (a \rightarrow \text{False}) \leftrightarrow \neg a \)
\( (a \rightarrow a) \leftrightarrow \text{True} \)
\( (\neg a \rightarrow a) \leftrightarrow a \)
\( (a \rightarrow \neg a) \leftrightarrow \neg a \)
\( ((a \rightarrow b) \rightarrow b) \leftrightarrow a \lor b \)  
by auto

\textbf{lemma} verit-equiv-simplify:
\( ((\neg a) = (\neg b)) \leftrightarrow (a = b) \)
\( (a = a) \leftrightarrow \text{True} \)
\( (a = (\neg a)) \leftrightarrow \text{False} \)
\( ((\neg a) = a) \leftrightarrow \text{False} \)
\( (\text{True} = a) \leftrightarrow a \)
\( (a = \text{True}) \leftrightarrow a \)
\( (\text{False} = a) \leftrightarrow \neg a \)
\( (a = \text{False}) \leftrightarrow \neg a \)
\( \neg \neg a \leftrightarrow a \)
\( (\neg \text{False}) = \text{True} \)  
for a b :: bool
by auto

lemmas verit-eq-simplify =
semiring-char-0-class.eq-numeral-simps eq-refl zero-neq-one num.simps
neg-equal-zero equal-neg-zero one-neq-zero neg-equal-iff-equal
lemma verit-minus-simplify:
\((a :: 'a :: cancel-comm-monoid-add) - a = 0)\)
\((a :: 'a :: cancel-comm-monoid-add) - 0 = a)\)
\((0 - (b :: 'b :: {group-add})) = -b)\)
\((- ( (b :: 'b :: group-add)) = b)\)
by auto

lemma verit-sum-simplify:
\((a :: 'a :: cancel-comm-monoid-add) + 0 = a)\)
by auto

lemmas verit-prod-simplify =

mult-1
mult-1-right

lemma verit-comp-simplify1:
\((a :: 'a :: order) < a \longleftrightarrow False)\)
\((a \leq a)\)
\((\neg (b' \leq a') \longleftrightarrow (a' :: 'b :: linorder) < b')\)
by auto

lemmas verit-comp-simplify =
verit-comp-simplify1
le-numeral-simps
le-num-simps
less-numeral-simps
less-num-simps
zero-less-one
zero-le-one
less-neg-numeral-simps

lemma verit-la-disequality:
\((a :: 'a :: linorder) = b \lor \neg a \leq b \lor \neg b \leq a)\)
by auto

context
begin

For the reconstruction, we need to keep the order of the arguments.

named-theorems smt-arith-multiplication (Theorems to reconstruct arithmetic theorems.)

named-theorems smt-arith-combine (Theorems to reconstruct arithmetic theorems.)

named-theorems smt-arith-simplify (Theorems to combine theorems in the LA procedure.)
lemmas [smt-arith-simplify] =
  div-add dvd-numeral-simp dvdmod-steps less-num-simps le-num-simps if-True if-False dvdmod-cancel
  dvd-mul dvd-mul2 less-irrefl prod.case numeral-plus-one dvdmod-step-def order.refl le-zero-eq
  le-numeral-simps less-numeral-simps mult.right-neutral simp-thms divides-aux-eq
  mult-nonneg-nonneg dvd-imp-mod-0 dvd-add zero-less-one mod-mult-self4 numeral-mod-numeral
  dvdmod-trivial prod.sel mult.left-neutral div-pos-pos-trivial arith-simps div-add
  div-mult-self1
  add-le-cancel-left add-le-same-cancel2 not-one-le-zero le-numeral-simps add-le-same-cancel1
  zero-neq-one zero-le-one le-num-simps add-Suc mod-div-trivial nat.distinct mult-minus-right
  add.inverse-inverse distrib-left-numeral mult-num-simps numeral-times-numeral
  add-num-simps
  dvdmod-steps rel-simps if-True if-False numeral-div-numeral dvdmod-cancel prod.case
  add-num-simps one-plus-numeral fst-conv arith-simps sub-num-simps dbl-inc-simps
  dbl-simps mult-1 add-le-cancel-right left-diff-distrib-numeral add-uminus-conv-diff
  zero-neq-one
  zero-le-one One-nat-def add-Suc mod-div-trivial nat.distinct of-int-1 numerals
  numeral-One
  of-int-numeral add-uminus-conv-diff zle-diff1-eq add-less-same-cancel2 minus-add-distrib
  add-diff-cancel-left' add-diff-eq ring-distribs mult-minus-left minus-diff-eq

lemma [smt-arith-simplify]:
  \(\neg (a' :: 'a :: linorder) < b' \iff b' \leq a'\)
  \(\neg (a' :: 'a :: linorder) \leq b' \iff b' < a'\)
  \((c::int) \mod \text{Numeral1} = 0\)
  \((a::nat) \mod \text{Numeral1} = 0\)
  \((c::int) \div \text{Numeral1} = c\)
  \((a \div \text{Numeral1} = a)\)
  \((c::int) \mod 1 = 0\)
  \((a \mod 1 = 0)\)
  \((c::int) \div 1 = c\)
  \((a \div 1 = a)\)
  \(\neg(a' \neq b') \iff a' = b'\)
  by auto

lemma div-mod-decomp: \(A = (A \div n) \ast n + (A \mod n)\) for \(A :: nat\)
by auto

lemma div-less-mono:
  fixes \(A B :: nat\)
  assumes \(A < B\) \(0 < n\) and
  mod: \(A \mod n = 0B \mod n = 0\)
  shows \((A \div n) < (B \div n)\)
proof –
show ?thesis
  using assms(1)
  apply (subst (asm) div-mod-decomp[of A n])
  apply (subst (asm) div-mod-decomp[of B n])
  unfolding mod
  by (use assms(2,3) in (auto simp: ac-simps))
qed

lemma verit-le-mono-div:
  fixes A B :: nat
  assumes A < B 0 < n
  shows (A div n) + (if B mod n = 0 then 1 else 0) <= (B div n)
  by (auto simp: ac-simps Suc-leI assms less-mult-imp-div-less div-le-mono less-imp-le-nat)

lemmas [smt-arith-multiplication] =
  verit-le-mono-div[THEN mult-le-mono1, unfolded add-mult-distrib]
  div-le-mono[THEN mult-le-mono2, unfolded add-mult-distrib]

lemma div-mod-decomp-int: A = (A div n) * n + (A mod n) for A :: int
  by auto

lemma zdiv-mono-strict:
  fixes A B :: int
  assumes A < B 0 < n and
  mod: A mod n = 0B mod n = 0
  shows (A div n) < (B div n)
proof -
  show ?thesis
    using assms(1)
    apply (subst (asm) div-mod-decomp-int[of A n])
    apply (subst (asm) div-mod-decomp-int[of B n])
    unfolding mod
    by (use assms(2,3) in (auto simp: ac-simps))
qed

lemma verit-le-mono-div-int:
  (A div n + (if B mod n = 0 then 1 else 0) <= B div n)
  if (A < B) 0 < n
  for A B n :: int
proof -
  from (A < B) 0 < n have (A div n <= B div n)
    by (auto intro: zdiv-mono1)
  show ?thesis
    proof (cases (n dvd B))
      case False
      with (A div n <= B div n) show ?thesis
        by auto
    next
      case True
then obtain $C$ where $B = n \cdot C$; ..
then have $B \div n = C$,
  using $(0 < n)$ by simp
from $(0 < n)$ have $(A \mod n \geq 0)$
  by simp
have $(A \div n < C)$
proof (rule ccontr)
  assume $(\neg A \div n < C)$
  then have $(C \leq A \div n)$
    by simp
  with $(B \div n = C)$ $(A \div n \leq B \div n)$
  have $(A \div n = C)$
    by simp
  moreover from $(A < B)$ have $(n \cdot (A \div n) + A \mod n < B)$
    by simp
  ultimately have $(n \cdot C + A \mod n < n \cdot C)$
    using $(B = n \cdot C)$ by simp
  moreover have $(A \mod n \geq 0)$
    using $(0 < n)$ by simp
  ultimately show False
    by simp
  qed
with $(n \text{ dvd } B)$ $(B \div n = C)$ show ?thesis
    by simp
qed

lemma verit-less-mono-div-int2:
  fixes $A$, $B$ :: int
  assumes $A \leq B$ $0 < -n$
  shows $(A \div n) \geq (B \div n)$
  using assms(1) assms(2) zdiv-mono1-neg by auto

lemmas [smt-arith-multiplication] =
  verit-le-mono-div-int[THEN mult-left-mono, unfolded int-distrib]
  zdiv-mono1[THEN mult-left-mono, unfolded int-distrib]

lemmas [smt-arith-multiplication] =
  arg-cong[of - - $(\lambda a :: \text{nat}. \ a \div n \cdot p)$ for $n \ p :: \text{nat}$, THEN sym]
  arg-cong[of - - $(\lambda a :: \text{int}. \ a \div n \cdot p)$ for $n \ p :: \text{int}$, THEN sym]

lemma [smt-arith-combine]:
  $a < b \Longrightarrow c < d \Longrightarrow a + c + 2 \leq b + d$
  $a < b \Longrightarrow c \leq d \Longrightarrow a + c + 1 \leq b + d$
  $a \leq b \Longrightarrow c < d \Longrightarrow a + c + 1 \leq b + d$ for $a \ b \ c :: \text{int}$
  by auto

lemma [smt-arith-combine]:
  $a < b \Longrightarrow c < d \Longrightarrow a + c + 2 \leq b + d$
\[
a < b \implies c \leq d \implies a + c + 1 \leq b + d
\]
\[
a \leq b \implies c < d \implies a + c + 1 \leq b + d \text{ for } a, b, c :: \text{nat}
\]
by \text{auto}

\text{lemmas} \ [\text{smt-arith-combine}] =
\begin{align*}
& \text{add-strict-mono} \\
& \text{add-less-le-mono} \\
& \text{add-mono} \\
& \text{add-le-less-mono}
\end{align*}

\text{lemma} \ [\text{smt-arith-combine}];
\begin{align*}
& \langle m < n \implies c = d \implies m + c < n + d \rangle \\
& \langle m \leq n \implies c = d \implies m + c \leq n + d \rangle \\
& \langle c = d \implies m < n \implies m + c < n + d \rangle \\
& \langle c = d \implies m \leq n \implies m + c \leq n + d \rangle \\
& \text{for } m :: \langle 'a :: \text{ordered-cancel-ab-semigroup-add} \rangle \\
& \text{by (auto intro: ordered-cancel-ab-semigroup-add-class.add-strict-right-mono} \\
& \text{ordered-ab-semigroup-add-class.add-right-mono)}
\end{align*}

\text{lemma} \ \text{verit-negate-coefficient};
\begin{align*}
& \langle a \leq (b :: 'a :: \{\text{ordered-ab-group-add}\}) \implies -a \geq -b \rangle \\
& \langle a < b \implies -a > -b \rangle \\
& \langle a = b \implies -a = -b \rangle \\
& \text{by \text{auto}}
\end{align*}

end

\text{lemma} \ \text{verit-ite-intro};
\begin{align*}
& \langle (\text{if } a \text{ then } \text{P} (\text{if } a \text{ then } a' \text{ else } b') \text{ else } Q) \longleftrightarrow (\text{if } a \text{ then } \text{P'} \text{ else } \text{Q'}) \rangle \\
& \langle (\text{if } a \text{ then } \text{P'} \text{ else } \text{Q'} (\text{if } a \text{ then } a' \text{ else } b')) \longleftrightarrow (\text{if } a \text{ then } \text{P'} \text{ else } \text{Q'} b') \rangle \\
& \langle A = f (\text{if } a \text{ then } R \text{ else } S) \longleftrightarrow (\text{if } a \text{ then } A = f R \text{ else } A = f S) \rangle \\
& \text{by \text{auto}}
\end{align*}

\text{lemma} \ \text{verit-ite-if-cong};
\begin{align*}
& \text{fixes } x, y :: \text{bool} \\
& \text{assumes } b = c \\
& \text{and } c \equiv \text{True} \implies x = u \\
& \text{and } c \equiv \text{False} \implies y = v \\
& \text{shows } (\text{if } b \text{ then } x \text{ else } y) \equiv (\text{if } c \text{ then } u \text{ else } v) \\
& \text{proof} \\
& \text{have } H: (\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v) \\
& \text{using assms by (auto split: if-splits)} \\
& \text{show } (\text{if } b \text{ then } x \text{ else } y) \equiv (\text{if } c \text{ then } u \text{ else } v) \\
& \text{by (subst } H \text{) \text{auto}}
\end{align*}
qed
63.7 Setup

ML-file <Tools/SMT/smt-util.ML>
ML-file <Tools/SMT/smt-failure.ML>
ML-file <Tools/SMT/smt-config.ML>
ML-file <Tools/SMT/smt-builtin.ML>
ML-file <Tools/SMT/smt-datatypes.ML>
ML-file <Tools/SMT/smt-normalize.ML>
ML-file <Tools/SMT/smt-translate.ML>
ML-file <Tools/SMT/smtlib.ML>
ML-file <Tools/SMT/smtlib-interface.ML>
ML-file <Tools/SMT/smtlib-proof.ML>
ML-file <Tools/SMT/smtlib-isar.ML>
ML-file <Tools/SMT/z3-proof.ML>
ML-file <Tools/SMT/z3-isar.ML>
ML-file <Tools/SMT/smt-solver.ML>
ML-file <Tools/SMT/cvc-interface.ML>
ML-file <Tools/SMT/lethe-proof.ML>
ML-file <Tools/SMT/lethe-isar.ML>
ML-file <Tools/SMT/lethe-proof-parse.ML>
ML-file <Tools/SMT/cvc-proof-parse.ML>
ML-file <Tools/SMT/conj-disj-perm.ML>
ML-file <Tools/SMT/smt-replay-methods.ML>
ML-file <Tools/SMT/smt-replay.ML>
ML-file <Tools/SMT/smt-replay-arith.ML>
ML-file <Tools/SMT/z3-interface.ML>
ML-file <Tools/SMT/z3-replay-rules.ML>
ML-file <Tools/SMT/z3-replay-methods.ML>
ML-file <Tools/SMT/z3-replay.ML>
ML-file <Tools/SMT/lethe-replay-methods.ML>
ML-file <Tools/SMT/cvc5-replay-methods.ML>
ML-file <Tools/SMT/verit-replay-methods.ML>
ML-file <Tools/SMT/verit-strategies.ML>
ML-file <Tools/SMT/verit-replay.ML>
ML-file <Tools/SMT/cvc5-replay.ML>
ML-file <Tools/SMT/smt-systems.ML>

63.8 Configuration

The current configuration can be printed by the command `smt-status`, which shows the values of most options.

63.9 General configuration options

The option `smt-solver` can be used to change the target SMT solver. The possible values can be obtained from the `smt-status` command.

\texttt{declare \{smt-solver = z3\]}
Since SMT solvers are potentially nonterminating, there is a timeout (given in seconds) to restrict their runtime.

\[\text{declare } \{\text{smt-timeout} = 0\}\]

SMT solvers apply randomized heuristics. In case a problem is not solvable by an SMT solver, changing the following option might help.

\[\text{declare } \{\text{smt-random-seed} = 1\}\]

In general, the binding to SMT solvers runs as an oracle, i.e, the SMT solvers are fully trusted without additional checks. The following option can cause the SMT solver to run in proof-producing mode, giving a checkable certificate. This is currently implemented only for veriT and Z3.

\[\text{declare } \{\text{smt-oracle} = \text{false}\}\]

Each SMT solver provides several command-line options to tweak its behaviour. They can be passed to the solver by setting the following options.

\[\begin{align*}
\text{declare } \{\text{cvc4-options} = \}\ & \\
\text{declare } \{\text{cvc5-options} = \}\ & \\
\text{declaration } \{\text{cvc5-proof-options} = \text{--proof-format-mode=alethe --proof-granularity=dsl-rewrite}\}\ & \\
\text{declare } \{\text{verit-options} = \}\ & \\
\text{declare } \{\text{z3-options} = \}\ &
\end{align*}\]

The SMT method provides an inference mechanism to detect simple triggers in quantified formulas, which might increase the number of problems solvable by SMT solvers (note: triggers guide quantifier instantiations in the SMT solver). To turn it on, set the following option.

\[\text{declare } \{\text{smt-infer-triggers} = \text{false}\}\]

Enable the following option to use built-in support for datatypes, codatatypes, and records in CVC4 and cvc5. Currently, this is implemented only in oracle mode.

\[\text{declare } \{\text{cvc-extensions} = \text{false}\}\]

Enable the following option to use built-in support for div/mod, datatypes, and records in Z3. Currently, this is implemented only in oracle mode.

\[\text{declare } \{\text{z3-extensions} = \text{false}\}\]

## 63.10 Certificates

By setting the option \textit{smt-certificates} to the name of a file, all following applications of an SMT solver a cached in that file. Any further application of the same SMT solver (using the very same configuration) re-uses the cached certificate instead of invoking the solver. An empty string disables caching certificates.
The filename should be given as an explicit path. It is good practice to use the name of the current theory (with ending *.certs instead of *.thy) as the certificates file. Certificate files should be used at most once in a certain theory context, to avoid race conditions with other concurrent accesses.

\texttt{declare \{smt-certificates = \}}

The option \texttt{smt-read-only-certificates} controls whether only stored certificates should be used or invocation of an SMT solver is allowed. When set to \texttt{true}, no SMT solver will ever be invoked and only the existing certificates found in the configured cache are used; when set to \texttt{false} and there is no cached certificate for some proposition, then the configured SMT solver is invoked.

\texttt{declare \{smt-read-only-certificates = false\]}

### 63.11 Tracing

The SMT method, when applied, traces important information. To make it entirely silent, set the following option to \texttt{false}.

\texttt{declare \{smt-verbose = true\]}

For tracing the generated problem file given to the SMT solver as well as the returned result of the solver, the option \texttt{smt-trace} should be set to \texttt{true}.

\texttt{declare \{smt-trace = false\]}

### 63.12 Schematic rules for Z3 proof reconstruction

Several proof rules of Z3 are not very well documented. There are two lemma groups which can turn failing Z3 proof reconstruction attempts into succeeding ones: the facts in \texttt{z3-rule} are tried prior to any implemented reconstruction procedure for all uncertain Z3 proof rules; the facts in \texttt{z3-simp} are only fed to invocations of the simplifier when reconstructing theory-specific proof steps.

\texttt{lemmas \{z3-rule\] =}

\texttt{refl eq-commute conj-commute disj-commute simp-thms nnf-simps}

\texttt{ring-distribr field-simps times-divide-eq-right times-divide-eq-left}

\texttt{if-True if-False not-not}

\texttt{NO-MATCH-def}

\texttt{lemma \{z3-rule\]:}

\texttt{(P \land Q) = (\lnot (\lnot P \lor \lnot Q))}

\texttt{(P \land Q) = (\lnot (\lnot Q \lor \lnot P))}

\texttt{(\lnot P \land Q) = (\lnot (P \lor \lnot Q))}

\texttt{(\lnot P \land Q) = (\lnot (Q \lor \lnot P))}

\texttt{(P \land \lnot Q) = (\lnot (\lnot P \lor Q))}

\texttt{(P \land \lnot Q) = (\lnot (Q \lor \lnot P))}
\((\neg P \land \neg Q) = (\neg (P \lor Q))\)
\((\neg P \land Q) = (\neg (Q \lor P))\)
by auto

**lemma [z3-rule]:**
\((P \rightarrow Q) = (Q \lor \neg P)\)
\((\neg P \rightarrow Q) = (P \lor Q)\)
\((\neg P \rightarrow Q) = (Q \lor P)\)
\((\text{True} \rightarrow P) = P\)
\((P \rightarrow \text{True}) = \text{True}\)
\((\text{False} \rightarrow P) = \text{True}\)
\((P \rightarrow P) = \text{True}\)
\((\neg (A \leftrightarrow B)) \leftrightarrow (A \leftrightarrow B)\)
by auto

**lemma [z3-rule]:**
\(((P = Q) \rightarrow R) = (R \lor (Q = (\neg P)))\)
by auto

**lemma [z3-rule]:**
\((\neg \text{True}) = \text{False}\)
\((\neg \text{False}) = \text{True}\)
\((x = x) = \text{True}\)
\((P = \text{True}) = P\)
\((\text{True} = P) = P\)
\((P = \neg P) = (\neg P)\)
\((\neg P = P) = \text{False}\)
\((P = (\neg P)) = \text{False}\)
\((\neg (\neg P) = (\neg Q)) = (P = Q)\)
\((\neg (P = (\neg Q)) = (P = Q)\)
\((\neg (\neg P) = Q) = (P = Q)\)
\((P \neq Q) = (Q = (\neg P))\)
\((P = Q) = ((\neg P \lor Q) \land (P \lor (\neg Q)))\)
\((P \neq Q) = ((\neg P \lor (\neg Q)) \land (P \lor Q))\)
by auto

**lemma [z3-rule]:**
\((\text{if } P \text{ then } P \text{ else } \neg P) = \text{True}\)
\((\text{if } \neg P \text{ then } \neg P \text{ else } P) = \text{True}\)
\((\text{if } P \text{ then } \text{True} \text{ else } \text{False}) = P\)
\((\text{if } P \text{ then } \text{False} \text{ else } \text{True}) = (\neg P)\)
\((\text{if } P \text{ then } Q \text{ else } \text{True}) = ((\neg P) \lor Q)\)
\((\text{if } P \text{ then } Q \text{ else } \text{True}) = (Q \lor (\neg P))\)
\((\text{if } P \text{ then } Q \text{ else } \neg Q) = (P = Q)\)
\((\text{if } P \text{ then } Q \text{ else } \neg Q) = (Q = P)\)
\((\text{if } P \text{ then } \neg Q \text{ else } Q) = (P = (\neg Q))\)
\((\text{if } P \text{ then } \neg Q \text{ else } Q) = ((\neg Q) = P)\)
\((\neg P \text{ then } x \text{ else } y) = (\text{if } P \text{ then } y \text{ else } x)\)
THEORY “SMT”

(if $P$ then (if $Q$ then $x$ else $y$) else $x$) = (if $P$ $\land$ ($\neg$ $Q$) then $y$ else $x$)
(if $P$ then (if $Q$ then $x$ else $y$) else $x$) = (if ($\neg$ $Q$) $\land$ $P$ then $y$ else $x$)
(if $P$ then (if $Q$ then $x$ else $y$) else $y$) = (if $P$ $\land$ $Q$ then $x$ else $y$)
(if $P$ then (if $Q$ then $x$ else $y$) else $y$) = (if $Q$ $\land$ $P$ then $x$ else $y$)
(if $P$ then $x$ else if $P$ then $y$ else $z$) = (if $P$ then $x$ else $z$)
(if $P$ then $x$ else if $Q$ then $x$ else $y$) = (if $P$ $\lor$ $Q$ then $x$ else $y$)
(if $P$ then $x$ else if $Q$ then $x$ else $y$) = (if $Q$ $\lor$ $P$ then $x$ else $y$)
(if $P$ then $x$ = $y$ else $z$ = $y$) = (if $P$ then $x$ else $z$)

lemma [z3-rule]:

$0 + (x :: int) = x$
$x + 0 = x$
$x + x = 2 \ast x$
$0 \ast x = 0$
$1 \ast x = x$
$x + y = y + x$

by auto

lemma [z3-rule]:

$P = Q \lor P \lor Q$
$P = Q \lor \neg P \lor \neg Q$
($\neg$ $P$) = $Q \lor \neg P \lor Q$
($\neg$ $P$) = $Q \lor P \lor \neg Q$
$P = ($\neg$ $Q$) $\lor$ $\neg P \lor Q$
$P = ($\neg$ $Q$) $\lor$ $P \lor \neg Q$
$P \neq Q \lor P \lor \neg Q$
$P \neq Q \lor \neg P \lor Q$
($\neg$ $P$) $\neq Q \lor P \lor Q$
$P \lor Q \lor P \neq ($\neg$ $Q$)
$P \lor Q \lor (\neg P) \neq Q$
$P \lor \neg Q \lor P \neq Q$
$\neg P \lor Q \lor P \neq Q$
$P \lor y = (if $P$ then $x$ else $y$)
$P \lor (if P then x else y) = y$
$\neg P \lor x = (if P then x else y)$
$\neg P \lor (if P then x else y) = x$
$P \lor R \lor \neg (if P then Q else R)$
$\neg P \lor Q \lor \neg (if P then Q else R)$
$\neg (if P then Q else R) \lor \neg P \lor Q$
$\neg (if P then Q else R) \lor P \lor R$
$(if P then Q else R) \lor \neg P \lor \neg Q$
$(if P then Q else R) \lor \neg P \lor R$
$(if P then Q else R) \lor \neg P \lor Q$
$(if P then Q else R) \lor P \lor R$

by auto
hide-type (open) symb-list pattern
hide-const (open) Symb-Nil Symb-Cons trigger pat nopat fun-app z3div z3mod

end

64 Sledgehammer: Isabelle–ATP Linkup

theory Sledgehammer
imports Presburger SMT
keywords
  sledgehammer :: diag and
  sledgehammer-params :: thy-decl
begin

ML-file ‹ Tools/ATP/system-on-tptp.ML ‹
ML-file ‹ Tools/Sledgehammer/async-manager-legacy.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-util.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-fact.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-proof-methods.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-isar-annotate.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-isar-proof.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-isar-preplay.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-isar-compress.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-isar-minimize.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-isar.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-atp-systems.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-prover.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-prover-atp.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-prover-smt.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-prover-minimize.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-mepo.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-mash.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-commands.ML ‹
ML-file ‹ Tools/Sledgehammer/sledgehammer-tactics.ML ‹

end

65 Setup for Lifting/Transfer for the set type

theory Lifting-Set
imports Lifting Groups-Big
begin

65.1 Relator and predicator properties

lemma rel-setD1: \[ \rel-set R A B; \ x \in A \implies \exists y \in B. \ R \ x \ y \]
and  \( \text{rel-setD2: } \{ \text{rel-set } R \ A \ B; \ y \in B \} \implies \exists x \in A. \ R \ x \ y \) 
\text{by (simp-all add: rel-set-def)}

\text{lemma rel-set-conversep [simp]: rel-set } A^{-1}^{-1} = (\text{rel-set } A)^{-1}^{-1} 
\text{unfolding rel-set-def by auto}

\text{lemma rel-set-eq [relator-eq]: rel-set } (\_ =) = (\_ =) 
\text{unfolding rel-set-def fun-eq-iff by auto}

\text{lemma rel-set-mono [relator-mono]:} 
\text{assumes } A \leq B 
\text{shows rel-set } A \leq \text{rel-set } B 
\text{using assms unfolding rel-set-def by blast}

\text{lemma rel-set-OO [relator-distr]: rel-set } R \ \text{OO } \text{rel-set } S = \text{rel-set } (R \ \text{OO } S) 
\text{apply (rule sym)} 
\text{apply (intro ext)} 
\text{subgoal for } X \ Z 
\text{apply (rule iffI)} 
\text{apply (rule relcomppI [where } b=\{y. \ (\exists x \in X. \ R \ x \ y) \land (\exists z \in Z. \ S \ y \ z)\}\]} 
\text{apply (simp add: rel-set-def, fast)+} 
\text{done}

\text{lemma Domainp-set [relator-domain]:} 
\text{Domainp (rel-set } T) = (\lambda A. \text{ Ball } A (\text{Domainp } T)) 
\text{unfolding rel-set-def Domainp-iff [abs-def]} 
\text{apply (intro ext)} 
\text{apply (rule iffI)} 
\text{apply blast} 
\text{subgoal for } A \text{ by (rule exI [where } x=\{y. \ (\exists x \in X. \ T \ x \ y)\}\]} \text{ fast} 
\text{done}

\text{lemma left-total-rel-set [transfer-rule]:} 
\text{left-total } A \implies \text{left-total } (\text{rel-set } A) 
\text{unfolding left-total-def rel-set-def} 
\text{apply safe} 
\text{subgoal for } X \text{ by (rule exI [where } x=\{y. \ (\exists x \in X. \ A \ x \ y)\}\]} \text{ fast} 
\text{done}

\text{lemma left-unique-rel-set [transfer-rule]:} 
\text{left-unique } A \implies \text{left-unique } (\text{rel-set } A) 
\text{unfolding left-unique-def rel-set-def} 
\text{by fast}

\text{lemma right-total-rel-set [transfer-rule]:} 
\text{right-total } A \implies \text{right-total } (\text{rel-set } A) 
\text{using left-total-rel-set[of } A^{-1}^{-1}] \text{ by simp}
lemma right-unique-rel-set [transfer-rule]:
right-unique A ⇒ right-unique (rel-set A)
unfolding right-unique-def rel-set-def by fast

lemma bi-total-rel-set [transfer-rule]:
bi-total A ⇒ bi-total (rel-set A)
by(simp add: bi-total-alt-def left-total-rel-set right-total-rel-set)

lemma bi-unique-rel-set [transfer-rule]:
bi-unique A ⇒ bi-unique (rel-set A)
unfolding bi-unique-def rel-set-def by fast

lemma set-relator-eq-onp [relator-eq-onp]:
rel-set (eq-onp P) = eq-onp (λA. Ball A P)
unfolding fun-eq-iff rel-set-def eq-onp-def Ball-def by fast

lemma bi-unique-rel-set-lemma:
assumes bi-unique R and rel-set R X Y
obtains f where Y = image f X and inj-on f X and ∀ x ∈ X. R x (f x)
proof
  define f where f x = (THE y. R x y) for x
  { fix x assume x ∈ X
    with ⟨rel-set R X Y⟩ ⟨bi-unique R⟩ have R x (f x)
    by (simp add: bi-unique-def rel-set-def f-def) (metis theI)
    with assms ⟨x ∈ X⟩
    have R x (f x) ∀ x′ ∈ X. R x′ (f x) −→ x = x′ ∀ y ∈ Y. R x y −→ y = f x f x ∈ Y
    by (fastforce simp add: bi-unique-def rel-set-def) + }
  note * = this
  moreover
  { fix y assume y ∈ Y
    with ⟨rel-set R X Y⟩ ⟨3⟩ ⟨y ∈ Y⟩ have ∃ x ∈ X. y = f x
      by (fastforce simp: rel-set-def) }
  ultimately show ∀ x ∈ X. R x (f x) Y = image f X inj-on f X
    by (auto simp: inj-on-def image-iff)
  qed

65.2 Quotient theorem for the Lifting package

lemma Quotient-set[quot-map]:
assumes Quotient R Abs Rep T
shows Quotient (rel-set R) (image Abs) (image Rep) (rel-set T)
using assms unfolding Quotient-alt-def4
apply (simp add: rel-set-OO[symmetric])
apply (simp add: rel-set-def)
apply fast
done
65.3 Transfer rules for the Transfer package

65.3.1 Unconditional transfer rules

context includes lifting-syntax
begin

lemma empty-transfer [transfer-rule]: (rel-set A) {} {}
  unfolding rel-set-def by simp

lemma insert-transfer [transfer-rule]:
  (A ===> rel-set A ===> rel-set A) insert insert
  unfolding rel-fun-def rel-set-def by auto

lemma union-transfer [transfer-rule]:
  (rel-set A ===> rel-set A ===> rel-set A) union union
  unfolding rel-fun-def rel-set-def by auto

lemma Union-transfer [transfer-rule]:
  (rel-set (rel-set A) ===> rel-set A) Union Union
  unfolding rel-fun-def rel-set-def by simp fast

lemma image-transfer [transfer-rule]:
  ((A ===> B) ===> rel-set A ===> rel-set B) image image
  unfolding rel-fun-def rel-set-def by simp fast

lemma UNION-transfer [transfer-rule]:— TODO deletion candidate
  (rel-set A ===> (A ===> rel-set B) ===> rel-set B) (λA f. ∪(f ' A)) (λA f. ∪(f ' A))
  by transfer-prover

lemma Ball-transfer [transfer-rule]:
  (rel-set A ===> (A ===> (=)) ===> (=)) Ball Ball
  unfolding rel-set-def rel-fun-def by fast

lemma Bex-transfer [transfer-rule]:
  (rel-set A ===> (A ===> (=)) ===> (=)) Bex Bex
  unfolding rel-set-def rel-fun-def by fast

lemma Pow-transfer [transfer-rule]:
  (rel-set A ===> rel-set (rel-set A)) Pow Pow
  apply (rule rel-funI)
  apply (rule rel-setI)
  subgoal for X Y X'
    apply (rule rev-bexI [where x = {y∈Y. ∃x∈X'. A x y}] )
    apply clarsimp
    apply (simp add: rel-set-def)
    apply fast
    done
  subgoal for X Y Y'

apply (rule rev-bexI [where x={x∈X. ∃ y∈Y'. A x y}])
apply clarsimp
apply (simp add: rel-set-def)
apply fast
done
done

lemma rel-set-transfer [transfer-rule]:
((A ==>) B ==>) (=) ==>) rel-set A ==>) rel-set B ==>) (=) rel-set
unfolding rel-fun-def rel-set-def by fast

lemma bind-transfer [transfer-rule]:
(rel-set A ==>) (A ==>) rel-set B ==>) rel-set B) Set.bind Set.bind
unfolding bind-UNION [abs-def] by transfer-prover

lemma INF-parametric [transfer-rule]: — TODO deletion candidate
(rel-set A ==>) (A ==>) HOL.eq ==>) HOL.eq) (λA f. Inf (f * A)) (λA f. Inf (f * A))
by transfer-prover

lemma SUP-parametric [transfer-rule]: — TODO deletion candidate
(rel-set R ==>) (R ==>) HOL.eq ==>) HOL.eq) (λA f. Sup (f * A)) (λA f. Sup (f * A))
by transfer-prover

65.3.2 Rules requiring bi-unique, bi-total or right-total relations

lemma member-transfer [transfer-rule]:
assumes bi-unique A
shows (A ==>) rel-set A ==>) (=)) (∈) (∈)
using assms unfolding rel-fun-def rel-set-def bi-unique-def by fast

lemma right-total-Collect-transfer [transfer-rule]:
assumes right-total A
shows ((A ==>) (=)) ==>) rel-set A) (AP. Collect (λx. P x ∧ Domainp A x)) Collect
using assms unfolding right-total-def rel-set-def rel-fun-def Domainp-iff by fast

lemma Collect-transfer [transfer-rule]:
assumes bi-total A
shows ((A ==>) (=)) ==>) rel-set A Collect Collect
using assms unfolding rel-fun-def rel-set-def bi-total-def by fast

lemma inter-transfer [transfer-rule]:
assumes bi-unique A
shows (rel-set A ==>) rel-set A ==>) rel-set A) inter inter
using assms unfolding rel-fun-def rel-set-def bi-unique-def by fast
lemma Diff-transfer [transfer-rule]:
assumes bi-unique A
shows (rel-set A ===> rel-set A ===> rel-set A) (-) (-)
using assms unfolding rel-fun-def rel-set-def bi-unique-def
unfolding Ball-def Bex-def Diff-eq
by (safe, simp, metis, simp, metis)

lemma subset-transfer [transfer-rule]:
assumes bi-unique A
shows (rel-set A ===> rel-set A ===> (=)) (⊆) (⊆)
unfolding subset-eq [abs-def] by transfer-prover

context
  includes lifting-syntax
begin

lemma strict-subset-transfer [transfer-rule]:
assumes bi-unique A
shows (rel-set A ===> rel-set A ===> (=)) (⊂) (⊂)
unfolding subset-not-subset-eq by transfer-prover

end

declare right-total-UNIV-transfer[transfer-rule]

lemma UNIV-transfer [transfer-rule]:
assumes bi-total A
shows (rel-set A) UNIV UNIV
using assms unfolding rel-set-def bi-total-def by simp

lemma right-total-Compl-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A and [transfer-rule]: right-total A
shows (rel-set A ===> rel-set A) (λS. uminus S ∩ Collect (Domainp A)) uminus
unfolding Compl-eq [abs-def]
by (subst Collect-conj-eq[symmetric]) transfer-prover

lemma Compl-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A and [transfer-rule]: bi-total A
shows (rel-set A ===> rel-set A) uminus uminus
unfolding Compl-eq [abs-def] by transfer-prover

lemma right-total-Inter-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A and [transfer-rule]: right-total A
shows (rel-set (rel-set A) ===> rel-set A) (λS. ∩ S ∩ Collect (Domainp A)) Inter
unfolding Inter-eq[abs-def]
by (subst Collect-conj-eq[symmetric]) transfer-prover
lemma \textit{Inter-transfer}\ [\textit{transfer-rule}]
  assumes \textit{transfer-rule}: bi-unique \(A\) and \textit{transfer-rule}: bi-total \(A\)
  shows \((\text{rel-set}\ (\text{rel-set}\ A)\implies\text{rel-set}\ A)\) \textit{Inter} Inter
  unfolding \textit{Inter-eq} \[\textit{abs-def}\] by \textit{transfer-prover}

lemma \textit{filter-transfer}\ [\textit{transfer-rule}]
  assumes \textit{transfer-rule}: bi-unique \(A\)
  shows \(((\((\text{rel-set}\ A)\implies\text{rel-set}\ A)\implies\text{rel-set}\ A)\)) \textit{Set.filter} Set.filter
  unfolding \textit{Set.filter-def}[\textit{abs-def}] \textit{rel-fun-def} \textit{rel-set-def} by \textit{blast}

lemma \textit{finite-transfer}\ [\textit{transfer-rule}]
  bi-unique \(A\) \implies (rel-set \(A\) \implies \text{finite}) \textit{finite}
  by (\textit{rule rel-funI, erule \((1)\) bi-unique-rel-set-lemma})
  (auto dest: \textit{finite-imageD})

lemma \textit{card-transfer}\ [\textit{transfer-rule}]
  bi-unique \(A\) \implies (rel-set \(A\) \implies \text{card}) \textit{card}
  by (\textit{rule rel-funI, erule \((1)\) bi-unique-rel-set-lemma})
  (simp add: \textit{card-image})

context
  includes lifting-syntax
begin

lemma \textit{vimage-right-total-transfer}[\textit{transfer-rule}]
  assumes \textit{transfer-rule}: bi-unique \(B\) right-total \(A\)
  shows \(((\((\text{rel-set}\ B)\implies\text{rel-set}\ A)\implies\text{rel-set}\ A)\)) \textit{vimage}
  proof
  let \(?vimage\ = \((\lambda f. \{ x. f\ x\in B\wedge Domainp A\ x\})\))
  have \(((\((\text{rel-set}\ B)\implies\text{rel-set}\ A)\implies\text{rel-set}\ A)\)) \textit{vimage vimage}
  unfolding \textit{vimage-def}
  by \textit{transfer-prover}
  also have \textit{vimage} = \((\lambda f. \{ x. f\ =\ X\cap Collect\ (Domainp A)\})\)
  by auto
  finally show \(?\text{thesis}\).
  qed

end

lemma \textit{vimage-parametric}\ [\textit{transfer-rule}]
  assumes \textit{transfer-rule}: bi-total \(A\) bi-unique \(B\)
  shows \(((\((\text{rel-set}\ B)\implies\text{rel-set}\ A)\implies\text{rel-set}\ A)\)) \textit{vimage vimage}
  unfolding \textit{vimage-def}[\textit{abs-def}] by \textit{transfer-prover}

lemma \textit{Image-parametric}\ [\textit{transfer-rule}]
  assumes bi-unique \(A\)
  shows \((\text{rel-set}\ (\text{rel-prod}\ A\ B)\implies\text{rel-set}\ A)\implies\text{rel-set}\ B\)) \textit{‘}\textit{‘}
  by (\textit{intro rel-funI rel-setI})
(force dest: rel-setD1 bi-uniqueDr[OF assms], force dest: rel-setD2 bi-uniqueDr[OF assms])

lemma inj-on-transfer[transfer-rule]:
  \((A \bowtie= B) \bowtie= rel-set A \bowtie= (=))\) inj-on inj-on
if [transfer-rule]: bi-unique A bi-unique B
unfolding inj-on-def
by transfer-prover
end

lemma (in comm-monoid-set) F-parametric [transfer-rule]:
  fixes A :: 'b ⇒ 'c ⇒ bool
  assumes bi-unique A
  shows rel-fun (rel-fun A (=)) (rel-fun (rel-set A) (=)) F F
proof (rule rel-funI)+
  fix f :: 'b ⇒ 'a and g S T
  assume rel-f: \((R \bowtie= B) \bowtie= rel-set A \bowtie B\)
  with \(\text{bi-unique A}\) obtain i where bij-betw i S T \(\forall x \in S \Rightarrow f x = g (i x)\)
    by (auto elim: bi-unique-rel-set-lemma simp: rel-fun-def bij-betw-def)
  then show \(F f S = F g T\)
    by (simp add: reindex-bij-betw)
qed

lemmas sum-parametric = sum.F-parametric
lemmas prod-parametric = prod.F-parametric

lemma rel-set-UNION:
  assumes [transfer-rule]: rel-set Q A B rel-fun Q (rel-set R) f g
  shows rel-set R (\(\bigcup (f ' A)\)) (\(\bigcup (g ' B)\))
by transfer-prover

context
  includes lifting-syntax
begin

lemma fold-graph-transfer[transfer-rule]:
  assumes bi-unique R right-total R
  shows \((R \bowtie= B) \bowtie= rel-set R \bowtie= (=)\) \(\Rightarrow\) (=) \(\Rightarrow\) (=) \(\Rightarrow\) rel-set R \(\Rightarrow\) (=)
proof intro rel-funI)
  fix f1 :: 'a ⇒ 'c ⇒ 'c
  assume rel-f: \((R \bowtie= B) \bowtie= rel-set R \bowtie A\) \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
  fix z1 z2 :: 'c
  assume [simp]: z1 = z2
  fix A1 A2 assume rel-A: rel-set R A1 A2
  fix y1 y2 :: 'c
  assume [simp]: y1 = y2
from \(\text{bi-unique R} \bowtie\text{right-total R}\) have The-y: \(\forall y. \exists! x. R x y\)
  unfolding bi-unique-def right-total-def by auto
define r where \( r \equiv \lambda y. \text{THE } x. \, R \, x \, y \)

from The-y have r-y: \( R \, (r \, y) \) \( y \) \( \text{for } y \)
  unfolding r-def using the-equality by fastforce
with assms rel-A have inj-on r A2 A1 = r \('\) A2
  unfolding r-def rel-set-def inj-on-def bi-unique-def
  apply(auto simp: image-iff) by metis+
with bi-unique R rel-f r-y have (f1 o r) \( y \) = f2 \( y \) \( \text{for } y \)
  unfolding bi-unique-def rel-fun-def by auto
then have (f1 o r) = f2
  by blast
then show fold-graph f1 z1 A1 y1 = fold-graph f2 z2 A2 y2
  by (fastforce simp: fold-graph-image[of inj-on r A2\] \( A1 = r \,' A2\])
qed

lemma fold-transfer[transfer-rule]:
  assumes [transfer-rule]: bi-unique R right-total R
  shows \((R \Longrightarrow (=) \Longrightarrow (=)) \Longrightarrow (=) \Longrightarrow (=) \Longrightarrow \text{rel-set } R \Longrightarrow (=)\)
Finite-Set.fold Finite-Set.fold
  unfolding Finite-Set.fold-def
  by transfer-prover

end

end

66 The datatype of finite lists

theory List
imports Sledgehammer Lifting-Set
begin

datatype (set: 'a) list =
  Nil ([])
| Cons (hd: 'a) (tl: 'a list) (infixr \# 65)
for
  map: map
  rel: list-all2
  pred: list-all
where
  tl [] = []
datatype-compat list

lemma [case-names Nil Cons, cases type: list]:
  — for backward compatibility – names of variables differ
  \( (y = [] \Longrightarrow P) \Longrightarrow (\bigwedge \text{a list. } y = a \# \text{list } \Longrightarrow P) \Longrightarrow P \)
  by (rule list.exhaust)
lemma [case-names Nil Cons, induct type: list]:
— for backward compatibility — names of variables differ
\( P \ [] \implies (\forall \text{a list. } P \text{ list} \implies P (\text{a} \# \text{list})) \implies P \text{ list} \)
by (rule list.induct)

Compatibility:
setup (Sign.mandatory-path list)

lemmas inducts = list.induct
lemmas recs = list.rec
lemmas cases = list.case

setup (Sign.parent-path)

lemmas set-simps = list.set

syntax
— list Enumeration
-list :: "args =⇒ 'a list ([(\))]"

translations
\[ x, xs] == x\#[xs]\n\[ x] == x\#[]

66.1 Basic list processing functions

primrec (nonexhaustive) last :: 'a list ⇒ 'a where
  last (x \# xs) = (if xs = [] then x else last xs)

primrec butlast :: 'a list ⇒ 'a list where
  butlast [] = []
  butlast (x \# xs) = (if xs = [] then [] else x \# butlast xs)

lemma set-rec: set xs = rec-list {} (λx -. insert x) xs
by (induct xs) auto

definition coset :: 'a list ⇒ 'a set where
[simp]: coset xs = − set xs

primrec append :: 'a list ⇒ 'a list ⇒ 'a list (infixr @ 65) where
append-Nil: [] @ ys = ys
append-Cons: (x#xs) @ ys = x \# xs @ ys

primrec rev :: 'a list ⇒ 'a list where
rev [] = []
rev (x \# xs) = rev xs @ [x]

primrec filter:: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list where
THEORY “List”

filter P [] = [] |
filter P (x # xs) = (if P x then x # filter P xs else filter P xs)

Special input syntax for filter:
syntax (ASCII)
- filter :: [pttrn, 'a list, bool] => 'a list ((I[-<->. -]))
syntax
- filter :: [pttrn, 'a list, bool] => 'a list ((I[-<->. -]))
translations
[x<xs . P] -> CONST filter (λx. P) xs

primrec fold :: ('a => 'b => 'b) => 'a list => 'b where
  fold-Nil: fold f [] = id |
  fold-Cons: fold f (x # xs) = fold f xs o f x

primrec foldr :: ('a => 'b => 'b) => 'a list => 'b where
  foldr-Nil: foldr f [] = id |
  foldr-Cons: foldr f (x # xs) = f x o foldr f xs

primrec foldl :: ('b => 'a => 'b) => 'a list => 'b where
  foldl-Nil: foldl f a [] = a |
  foldl-Cons: foldl f a (x # xs) = foldl f (f a x) xs

primrec concat:: 'a list list => 'a list where
concat [] = [] |
concat (x # xs) = x @ concat xs

primrec drop:: nat => 'a list => 'a list where
  drop-Nil: drop n [] = [] |
  drop-Cons: drop n (x # xs) = (case n of 0 => x # xs | Suc m => drop m xs)
  Warning: simpset does not contain this definition, but separate theorems for n = 0 and n = Suc k

primrec take:: nat => 'a list => 'a list where
  take-Nil: take n [] = [] |
  take-Cons: take n (x # xs) = (case n of 0 => [] | Suc m => take m xs)
  Warning: simpset does not contain this definition, but separate theorems for n = 0 and n = Suc k

primrec (nonexhaustive) nth :: 'a list => nat => 'a (infixl ! 100) where
  nth-Cons: (x # xs) ! n = (case n of 0 => x | Suc k => xs ! k)
  Warning: simpset does not contain this definition, but separate theorems for n = 0 and n = Suc k

primrec list-update :: 'a list => nat => 'a => 'a list where
  list-update [] i v = [] |
  list-update (x # xs) i v = (case i of 0 => v # xs | Suc j => x # list-update xs j v)
nonterminal lupdbinds and lupdbind

syntax
-lupdbind:: [′a, ′a] => lupdbind  ((2:: /= -))
   :: lupdbind => lupdbinds  (-)
-lupdbinds :: [lupdbind, lupdbinds] => lupdbinds  (-/ -)
-LUpdate :: [′a, lupdbinds] => ′a  (-/[-]) [1000,0] 900

translations
-LUpdate xs (-lupdbinds b bs) == -LUpdate (-LUpdate xs b) bs
xs[i:=x] == CONST list-update xs i x

primrec takeWhile :: (′a⇒ bool) ⇒ ′a list ⇒ ′a list where
takeWhile P [] = [] |
takeWhile P (x # xs) = (if P x then x # takeWhile P xs else [])

primrec dropWhile :: (′a⇒ bool) ⇒ ′a list ⇒ ′a list where
dropWhile P [] = [] |
dropWhile P (x # xs) = (if P x then dropWhile P xs else x # xs)

primrec zip :: ′a list ⇒ ′b list ⇒ (′a × ′b) list where
zip xs [] = [] |
zip-Cons: zip xs (y # ys) =
   (case xs of [] ⇒ [] | z # zs ⇒ (z, y) # zip zs ys)
— Warning: simpset does not contain this definition, but separate theorems for xs = [] and xs = z # zs

abbreviation map2 :: (′a⇒ ′b⇒ ′c) ⇒ ′a list ⇒ ′b list ⇒ ′c list where
map2 f xs ys ≡ map (λ(x,y). f x y) (zip xs ys)

primrec product :: ′a list ⇒ ′b list ⇒ (′a × ′b) list where
product [] - = [] |
product (x#xs) ys = map (Pair x) ys @ product xs ys

hide-const (open) product

primrec product-lists :: ′a list list ⇒ ′a list list where
product-lists [] = [[]] |
product-lists (xs # xss) = concat (map (λx. map (Cons x) (product-lists xss)) xs)

primrec upt :: nat ⇒ nat ⇒ nat list ((1[..<]/-)) where
upt-0: [i..<0] = [] |
upt-Suc: [i..<(Suc j)] = (if i ≤ j then [i..<j] @ [j] else [])

definition insert :: ′a ⇒ ′a list ⇒ ′a list where
insert x xs = (if x ∈ set xs then xs else x # xs)

definition union :: ′a list ⇒ ′a list ⇒ ′a list where
union = fold insert
hide-const (open) insert union
hide-fact (open) insert-def union-def

primrec find :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a option where
find - [] = None | find P (x#xs) = (if P x then Some x else find P xs)

In the context of multisets, count-list is equivalent to count ◦ mset and it is advisable to use the latter.

primrec count-list :: 'a list ⇒ 'a ⇒ nat where
count-list [] y = 0 | count-list (x#xs) y = (if x=y then count-list xs y + 1 else count-list xs y)

definition extract :: ('a ⇒ bool) ⇒ 'a list ⇒ ('a list * 'a * 'a list) option where
extract P xs = (case dropWhile (Not ◦ P) xs of [] ⇒ None | y#ys ⇒ Some(takeWhile (Not ◦ P) xs, y, ys))

hide-const (open) extract

primrec those :: 'a option list ⇒ 'a list option where
those [] = Some [] | those (x#xs) = (case x of None ⇒ None | Some y ⇒ map-option (Cons y) (those xs))

primrec remove1 :: 'a ⇒ 'a list ⇒ 'a list where
remove1 x [] = [] | remove1 x (y#xs) = (if x = y then xs else y#remove1 x xs)

primrec removeAll :: 'a ⇒ 'a list ⇒ 'a list where
removeAll x [] = [] | removeAll x (y#xs) = (if x = y then removeAll xs else y#removeAll x xs)

primrec distinct :: 'a list ⇒ bool where
distinct [] ←→ True | distinct (x#xs) ←→ x /∈ set xs ∧ distinct xs

fun successively :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool where
successively P [] = True | successively P [x] = True | successively P (x # y # xs) = (P x y ∧ successively P (y#xs))

definition distinct-adj where
distinct-adj = successively (≠)
theory "List"

primrec remdups :: 'a list ⇒ 'a list where
remdups [] = [] |
remdups (x # xs) = (if x ∈ set xs then remdups xs else x # remdups xs)

fun remdups-adj :: 'a list ⇒ 'a list where
remdups-adj [] = [] |
remdups-adj [x] = [x] |
remdups-adj (x # y # xs) = (if x = y then remdups-adj (x # xs) else x # remdups-adj (y # xs))

primrec replicate :: nat ⇒ 'a ⇒ 'a list where
replicate-0: replicate 0 x = [] |
replicate-Suc: replicate (Suc n) x = x # replicate n x

function size is overloaded for all datatypes. Users may refer to the list version as length.

abbreviation length :: 'a list ⇒ nat where
length ≡ size

definition enumerate :: nat ⇒ 'a list ⇒ (nat × 'a) list where
enumerate-eq-zip: enumerate n xs = zip [n..<n + length xs] xs

primrec rotate1 :: 'a list ⇒ 'a list where
rotate1 [] = [] |
rotate1 (x # xs) = xs @ [x]

definition rotate :: nat ⇒ 'a list ⇒ 'a list where
rotate n = rotate1 ^^ n

definition nths :: 'a list ⇒ nat set ⇒ 'a list where
nths xs A = map fst (filter (λp. snd p ∈ A) (zip xs [0..<size xs]))

primrec subseqs :: 'a list ⇒ 'a list list where
subseqs [] = [[]] |
subseqs (x#xs) = (let xss = subseqs xs in map (Cons x) xss @ xss)

primrec n-lists :: nat ⇒ 'a list ⇒ 'a list list where
n-lists 0 xs = [[]] |
n-lists (Suc n) xs = concat (map (λys. map (λy # ys) xs) (n-lists n xs))

hide-const (open) n-lists

function splice :: 'a list ⇒ 'a list ⇒ 'a list where
splice [] ys = ys |
splice (x#xs) ys = x # splice ys xs
by pat-completeness auto

termination
by (relation measure (λ(xs, ys). size xs + size ys)) auto

function shuffles where
  shuffles [] ys = {ys}
| shuffles xs [] = {xs}
| shuffles (x # xs) (y # ys) = (#) x ' shuffles xs (y # ys) ∪ (#) y ' shuffles (x # xs) ys
  by pat-completeness simp-all
termination by lexicographic-order

Use only if you cannot use Min instead:

fun min-list :: 'a::ord list ⇒ 'a where
min-list (x # xs) = (case xs of [] ⇒ x | - ⇒ min x (min-list xs))

Returns first minimum:

fun arg-min-list :: ('a ⇒ ('b::linorder)) ⇒ 'a list ⇒ 'a where
arg-min-list f [] = x
| arg-min-list f (x # y # zs) = (let m = arg-min-list f (y # zs) in if f x ≤ f m then x else m)

Figure 1 shows characteristic examples that should give an intuitive understanding of the above functions.

The following simple sort(ed) functions are intended for proofs, not for efficient implementations.

A sorted predicate w.r.t. a relation:

fun sorted-wrt :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool where
sorted-wrt P [] = True |
| sorted-wrt P (x # y # zs) = ((∀ y ∈ set ys. P x y) ∧ sorted-wrt P ys)

A class-based sorted predicate:

class linorder
begin

abbreviation sorted :: 'a list ⇒ bool where
  sorted ≡ sorted-wrt (≤)

lemma sorted-simps: sorted [] = True sorted (x # ys) = ((∀ y ∈ set ys. x≤y) ∧ sorted ys)
  by auto

lemma strict-sorted-simps: sorted-wrt (<) [] = True sorted-wrt (<) (x # ys) = ((∀ y ∈ set ys. x<y) ∧ sorted-wrt (<) ys)
  by auto

primrec insort-key :: ('b ⇒ 'a) ⇒ 'b ⇒ 'b list ⇒ 'b list where
  insort-key f x [] = [x] |
[a, b] @ [c, d] = [a, b, c, d]
length [a, b, c] = 3
set [a, b, c] = \{a, b, c\}
map f [a, b, c] = [f a, f b, f c]
rev [a, b, c] = [c, b, a]
hd [a, b, c, d] = a
tl [a, b, c, d] = [b, c, d]
last [a, b, c, d] = d
butlast [a, b, c, d] = [a, b, c]
filter (\(\lambda n::\text{nat}. n < 2\)) [0, 2, 1] = [0, 1]
concat [[a, b], [c, d, e], [], [f]] = [a, b, c, d, e, f]
fold f [a, b, c] x = f c (f b (f a x))
foldr f [a, b, c] x = f a (f b (f c x))
foldl f x [a, b, c] = f (f (f x a) b) c
successively (\(\neq\)) [True, False, True, False]
zip [a, b, c] [x, y, z] = [(a, x), (b, y), (c, z)]
zip [a, b] [x, y, z] = [(a, x), (b, y)]
enumerate 3 [a, b, c] = [(3, a), (4, b), (5, c)]
List.product [a, b] [c, d] = [(a, c), (a, d), (b, c), (b, d)]
product-lists [a, b] [c, d] = [[a, c, d], [a, c, e], [b, c, e]]
splice [a, b, c] [x, y, z] = [a, x, b, y, c, z]
splice [a, b, c, d] [x, y, z] = [a, x, b, y, c, d]
shuffles [a, b] [c, d] = [[a, b, c, d], [a, c, b, d], [c, a, b, d], [c, a, b, d], [c, c, a, b, d], [c, c, a, b, d], [c, d, a, b], [c, d, a, b]]
take 2 [a, b, c, d] = [a, b]
take 6 [a, b, c, d] = [a, b, c, d]
drop 2 [a, b, c, d] = [c, d]
drop 6 [a, b, c, d] = []
takeWhile (\(\lambda n. n < 3\)) [1, 2, 3, 0] = [1, 2]
dropWhile (\(\lambda n. n < 3\)) [1, 2, 3, 0] = [3, 0]
distinct [2, 0, 1]
remdups [2, 0, 2, 1, 2] = [0, 1, 2]
remdups-adj [2, 2, 3, 1, 1, 2, 1] = [2, 3, 1, 2, 1]
List.insert 2 [0, 1, 2] = [0, 1, 2]
List.insert 3 [0, 1, 2] = [3, 0, 1, 2]
List.union [2, 3, 4] [0, 1, 2] = [4, 3, 0, 1, 2]
find ((<) 0) [0, 0] = None
find ((<) 0) [0, 1, 0, 2] = Some 1
count-list [0, 1, 0, 2] 0 = 2
List.extract ((<) 0) [0, 0] = None
List.extract ((<) 0) [0, 1, 0, 2] = Some ([0], [0, 2])
remove 1 [2, 0, 2, 1, 2] = [0, 2, 1, 2]
removeAll 2 [2, 0, 2, 1, 2] = [0, 1]
[a, b, c, d] 1 2 = c
[a, b, c, d] 2 := x = [a, b, x, d]
nths [a, b, c, d] 0 = [a, c, d]
subsets [a, b] = [[a, b], [a], [b], []]
List.n-lists 2 [a, b, c] = [[a, a], [b, a], [c, a], [a, b], [b, b], [c, b], [a, c], [b, c], [c, c]]
rotate 1 [a, b, c, d] = [b, c, d, a]
rotate 3 [a, b, c, d] = [d, a, b, c]
replicate 4 a = [a, a, a, a]
[2..5] = [2, 3, 4]
min-list [3, 1, -2] = -2
arg-min-list (\(\lambda i. i * i\)) [3, -1, 1, -2] = -1
insort-key \ f \ x \ (y#ys) = 
(if \ f \ x \ \leq \ f \ y \ \text{then} \ (x#y#ys) \ \text{else} \ y#(\text{insort-key} \ f \ x \ ys))

**definition** sort-key :: (′b ⇒ ′a) ⇒ ′b list ⇒ ′b list where
sort-key \ f \ xs = \text{foldr} \ (\text{insort-key} \ f) \ xs []

**definition** insort-insert-key :: (′b ⇒ ′a) ⇒ ′b ⇒ ′b list ⇒ ′b list where
insort-insert-key \ f \ x \ xs = 
(if \ f \ x \ \in \ f \ ' \ \text{set} \ xs \ \text{then} \ xs \ \text{else} \ \text{insort-key} \ f \ x \ xs)

**abbreviation** sort ≡ sort-key (λx. x)

**abbreviation** insort ≡ insort-key (λx. x)

**abbreviation** insort-insert ≡ insort-insert-key (λx. x)

**definition** stable-sort-key :: (′b ⇒ ′a) ⇒ ′b list ⇒ ′b list ⇒ bool where
stable-sort-key \ sk \ = 
(∀f \ xs \ k. \ \text{filter} \ (λy. \ f \ y \ = \ k) \ (sk \ f \ xs) = \text{filter} \ (λy. \ f \ y \ = \ k) \ xs)

**lemma** strict-sorted-iff: sorted-wrt (<) l ←→ sorted l ∧ distinct l
by (induction l) (auto iff: antisym-con1)

**lemma** strict-sorted-imp-sorted: sorted-wrt (<) xs ⇒ sorted xs
by (auto simp: strict-sorted-iff)

**end**

### 66.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: [(x,y). x ← xs, y ← ys, x ≠ y], the list of all pairs of distinct elements from xs and ys. The syntax is as in Haskell, except that | becomes a dot (like in Isabelle’s set comprehension): [e. x ← xs, ...] rather than [e| x ← xs, ...].

The qualifiers after the dot are

**generators** p ← xs, where p is a pattern and xs an expression of list type, or

**guards** b, where b is a boolean expression.

Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon output. Note that the translation of [e. x ← xs] is optimized to map (λx. e) xs.

It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.
THEORY "List"

nonterminal lc-qual and lc-quals

syntax
-listcompr :: 'a ⇒ lc-qual ⇒ lc-quals ⇒ 'a list ([- . -])
-lc-gen :: 'a ⇒ 'a list ⇒ lc-qual (- ← -)
-lc-test :: bool ⇒ lc-qual (-)

-lc-end :: lc-quals ()
-lc-quals :: lc-qual ⇒ lc-quals ⇒ lc-quals (, -)

syntax (ASCII)
-lc-gen :: 'a ⇒ 'a list ⇒ lc-qual (← -)

parse-translation <
let
val NilC = Syntax.const const-syntax Nil;
val ConsC = Syntax.const const-syntax Cons;
val mapC = Syntax.const const-syntax map;
val concatC = Syntax.const const-syntax concat;
val IfC = Syntax.const const-syntax If;
val dummyC = Syntax.const const-syntax Pure.dummy-pattern;
fun single x = ConsC $ x $ NilC;

fun pat-tr ctxt p e opti = (∗ %x. case x of p ⇒ e | - ⇒ [] ∗)
let (∗ FIXME proper name context!? ∗)
val x = Free (singleton (Name.variant-list (fold Term.add-free-names [p, e] [])) x, dummyT);
val e = if opti then single e else e;
val case1 = Syntax.const syntax-const (-case1) $ p $ e;
val case2 = Syntax.const syntax-const (-case1) $ dummyC $ NilC;
val cs = Syntax.const syntax-const (-case2) $ case1 $ case2;
in Syntax-Trans.abs-tr [x, Case-Translation.case-tr false ctxt [x, cs]] end;

fun pair-pat-tr (x as Free -) e = Syntax-Trans.abs-tr [x, e] | pair-pat-tr (- $ p1 $ p2) e = Syntax.const const-syntax case-prod $ pair-pat-tr p1 (pair-pat-tr p2 e) | pair-pat-tr dummy e = Syntax-Trans.abs-tr [Syntax.const -idtdummy, e]

fun pair-pat ctxt (Const (const-syntax Pair,-) $ s $ t) = pair-pat ctxt s andalso pair-pat ctxt t | pair-pat ctxt (Free (s,-)) = let
val thy = Proof-Context.theory-of ctxt;
val s’ = Proof-Context.intern-const ctxt s;
in not (Sign.declared-const thy s’) end
fun abs-tr ctxt p e opti =  
  let val p = Term.Position.strip-positions p  
  in if pair-pat ctxt p  
    then (pair-pat-tr p e, true)  
    else (pat-tr ctxt p e opti, false)  
  end

fun lc-tr ctxt [e, Const (syntax-const: -lc-test), -] $ b, qs] =  
  let
    val res =  
      (case qs of  
        Const (syntax-const (-lc-end), -) => single e  
        | Const (syntax-const (-lc-quals), -) =>  
          (case abs-tr ctxt p e true of  
            (f, true) => mapC $ f $ es  
            | (f, false) => concatC $ (mapC $ f $ es))
      )
  end

in [((syntax-const (-listcompr), lc-tr)) end]

ML-val (  
  let
    val read = Syntax.read-term context o Syntax.implode-input;  
    fun check s1 s2 =  
      read s1 aconv read s2 orelse  
      error (Check failed: ~  
        quote (#1 (Input.source-content s1)) ~ Position here list [Input.pos-of s1,  
        Input.pos-of s2]));  
  in
    check [[(x,y,z), b]] (if b then [(x, y, z)] else []);
    check [[(x,y,z), (x,-,y)→xs]] (map (λ(x,-,y). (x, y, z)) xs);
    check [[(x,y), (x, e,y)→ys]] (concat (map (λ(x, e,y). map (λy. e x y) ys) xs));
    check [[(x,y,z), x<0, x>b]] (if x < a then if b < x then [(x, y, z)] else [] else []);
    check [[(x,y,z), x→xs, x>b]] (concat (map (λx. if b < x then [(x, y, z)] else []))  
      xs);
    check [[(x,y), x<0, x→xs]] (if x < a then map (λx. (x, y, z)) xs else []);
    check [[(x,y). Cons True x ← xs]}

| pair-pat - t = (t = dummyC);

fun abs-tr ctxt p e opti =  
  let val p = Term.Position.strip-positions p  
  in if pair-pat ctxt p  
    then (pair-pat-tr p e, true)  
    else (pat-tr ctxt p e opti, false)  
  end

fun lc-tr ctxt [e, Const (syntax-const: -lc-test), -] $ b, qs] =  
  let
    val res =  
      (case qs of  
        Const (syntax-const (-lc-end), -) => single e  
        | Const (syntax-const (-lc-quals), -) =>  
          (case abs-tr ctxt p e true of  
            (f, true) => mapC $ f $ es  
            | (f, false) => concatC $ (mapC $ f $ es))
      )
  end

in [((syntax-const (-listcompr), lc-tr)) end]

ML-val (  
  let
    val read = Syntax.read-term context o Syntax.implode-input;  
    fun check s1 s2 =  
      read s1 aconv read s2 orelse  
      error (Check failed: ~  
        quote (#1 (Input.source-content s1)) ~ Position here list [Input.pos-of s1,  
        Input.pos-of s2]));  
  in
    check [[(x,y,z), b]] (if b then [(x, y, z)] else []);
    check [[(x,y,z), (x,-,y)→xs]] (map (λ(x,-,y). (x, y, z)) xs);
    check [[(x,y), (x, e,y)→ys]] (concat (map (λ(x, e,y). map (λy. e x y) ys) xs));
    check [[(x,y,z), x<0, x>b]] (if x < a then if b < x then [(x, y, z)] else [] else []);
    check [[(x,y,z), x→xs, x>b]] (concat (map (λx. if b < x then [(x, y, z)] else []))  
      xs);
    check [[(x,y), x<0, x→xs]] (if x < a then map (λx. (x, y, z)) xs else []);
    check [[(x,y). Cons True x ← xs]}

| pair-pat - t = (t = dummyC);


fun all-but-last-exists-conv cv ctxt ct =
    (case Thm.term_of ct of
        Const (const-name (Ex), _) $ Abs _ =>
        Conv.arg-conv (Conv.abs-conv (all-exists-conv cv o #2) ctxt) ct
    | _ => cv ctxt ct)

fun all-exists-conv cv ctxt ct =
    (case Thm.term_of ct of
        Const (const-name (Ex), _) $ Abs (_, _, Const (const-name (Ex), _) $ _) =>
        Conv.arg-conv (Conv.abs-conv (all-but-last-exists-conv cv o #2) ctxt) ct

ML

(* Simproc for rewriting list comprehensions applied to List.set to set
   comprehension. *)

signature LIST-TO-SET-COMPREHENSION =
sig
    val proc: Simplifier.proc
end

structure List-to-Set-Comprehension : LIST-TO-SET-COMPREHENSION = struct

(* conversion *)

fun all-exists-conv cv ctxt ct =
    (case Thm.term_of ct of
        Const (const-name (Ex), _) $ Abs _ =>
        Conv.arg-conv (Conv.abs-conv (all-exists-conv cv o #2) ctxt) ct
    | _ => cv ctxt ct)

fun all-but-last-exists-conv cv ctxt ct =
    (case Thm.term_of ct of
        Const (const-name (Ex), _) $ Abs (_, _, Const (const-name (Ex), _) $ _) =>
        Conv.arg-conv (Conv.abs-conv (all-but-last-exists-conv cv o #2) ctxt) ct

THEORY "List"
fun Collect-conv cv ctxt ct =
  (case Thm.term_of ct of
    Const (const-name (Collect), -) $ Abs - => Conv.arg-conv (Conv.abs-conv cv ctxt) ct
    | - => raise CTERM (Collect-conv, [ct])

fun rewr-conv' th = Conv.rewr-conv (mk-meta-eq th)

fun conjunct-assoc-conv ct =
  Conv.try-conv
  (rewr-conv' @{thm conj-assoc} then-conv HLogic.conj-conv Conv.all-conv conjunct-assoc-conv) ct

fun right-hand-set-comprehension-conv conv ctxt =
  HLogic.Trueprop-conv (HLogic.eq-conv Conv.all-conv
    (Collect-conv (all-exists-conv conv o #2) ctxt))

datatype termlets = If | Case of typ * int

local

val set-Nil-I = (@{lemma set [] = \x. False} by (simp add: empty-def [symmetric]))
val set-singleton = @{lemma set [a] = \x. x = a} by simp
val inst-Collect-mem-eq = @{lemma set A = \x. x \in set A} by simp
val del-refl-eq = @{lemma \t \equiv P \equiv P by simp}

fun mk-set T = Const (const-name set, HLogic.listT T ---> HLogic.mk-setT T)

fun dest-set (Const (const-name set, -) $ xs) = xs

fun dest-singleton-list (Const (const-name Cons, -) $ t $ (Const (const-name Nil, -))) = t
  | dest-singleton-list t = raise TERM (dest-singleton-list, [t])

(* We check that one case returns a singleton list and all other cases return [], and return the index of the one singleton list case.*)

fun possible-index-of-singleton-case cases =
  let
    fun check (i, case-t) s =
      (case strip-abs-body case-t of
        (Const (const-name Nil, -)) => s
      | - => (case s of SOME NONE => SOME (SOME i) | - => NONE))
  in
    fold-index check cases (SOME NONE) | > the-default NONE
  end
end

(*returns condition continuing term option*)
fun dest-if (Const (const-name $If\cdot\cdot$) $\&$ cond $\&$ then-t $\&$ Const (const-name $\langle\text{Nil},\cdot\rangle$)) =
  SOME (cond, then-t)
| dest-if - = NONE

(*returns (case-expr type index chosen-case constr-name) option*)
fun dest-case ctxt case-term =
  let
  val (case-const, args) = strip-comb case-term
  in
  (case try dest-Const case-const of
    SOME (c, T) =>
    (case Ctr-Sugar.ctr-sugar-of-case ctxt c of
      SOME {ctrs, ...} =>
      (case possible-index-of-singleton-case (fst (split-last args)) of
        SOME i =>
        let
          val constr-names = map (fst o dest-Const) ctrs
          val (Ts, -) = strip-type T
          val T' = List.last Ts
        in
          SOME (List.last args, T', i, nth args i, nth constr-names i) end
        | NONE =>> NONE)
      | NONE =>> NONE)
      | NONE =>> NONE)
    end
  end

fun tac ctxt [] =
  resolve-tac ctxt [set-singleton] 1 ORELSE
  resolve-tac ctxt [inst-Collect-mem-eq] 1
| tac ctxt (If :: cont) =
  Splitter.split-tac ctxt @{thms if-split} 1
  THEN resolve-tac ctxt @{thms conjI} 1
  THEN resolve-tac ctxt @{thms impI} 1
  THEN Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
    CONVERSION (right-hand-set-comprehension-conv (K
      (HOLogic.conj-conv (Conv.rewr-conv (List.last prems RS @{thm Eq-TrueI})))
    Conv.all-conv
    then-conv
    rewr-conv' @{lemma (True $\land$ P) = P by simp}}) ctxt') 1) ctxt 1
  THEN tac ctxt cont
  THEN resolve-tac ctxt @{thms impI} 1
  THEN Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
    CONVERSION (right-hand-set-comprehension-conv (K
      (HOLogic.conj-conv (Conv.rewr-conv (List.last prems RS @{thm Eq-FalseI})))
    Conv.all-conv
    then-conv rewr-conv' @{lemma (False $\land$ P) = False by simp}}) ctxt') 1)
THEORY "List"

ctxt 1
THEN resolve-tac ctxt [set-Nil-I] 1
| tac ctxt (Case (T, i :: cont) =
  let
    val SOME {injects, distincts, case-thms, split, ...} =
      Ctr-Sugar.ctr-sugar-of ctxt (fst (dest-Type T))
  in
    (* do case distinction *)
    Splitter.split-tac ctxt [split] 1
    THEN EVERY (map-index (fn (i', -) =>
        if i' < length case-thms - 1 then resolve-tac ctxt @\{thms conjI\} 1 else
        all-tac)
      THEN resolve-tac ctxt @\{thms allI\} 1
    THEN (if i' = i then
      (* continue recursively *)
      Subgoal.FOCUS (fn \{prems, context = ctxt', \} =>
        CONVERSION (Thm.eta_conversion then_conv right-hand-set-comprehension-conv
          (K
            ((HOLogic.conj-conv
              (HOLogic.eq-conv Conv.all-conv (rewr-conv' (List.last prems))))
            then-conv
              (Conv.try-conv (Conv.rewr-conv (map mk-meta-eq injects)))
            Conv.all-conv)
            then-conv
              (Conv.try-conv (Conv.rewr-conv del-refl-eq)
            then-conv conjunct-assoc-conv) ctxt'))
            then-conv
            (HOLogic.Trueprop-conv
              (HOLogic.eq-conv Conv.all-conv (Collect-conv (fn (-, ctxt'') =>
                Conv.repeat-conv
                (all-but-last-exists-conv
                 (K (rewr-conv'
                   @\{\{lemma (\(\exists x. x = t \land P x) = P t by simp\}\} ctxt''))))
                  ) ctxt'')))) 1) ctxt 1
    THEN tac ctxt cont
    else
      Subgoal.FOCUS (fn \{prems, context = ctxt', \} =>
        CONVERSION
        (right-hand-set-comprehension-conv (K
          (HOLogic.conj-conv
            ((HOLogic.eq-conv Conv.all-conv
              (rewr-conv' (List.last prems))) then-conv
              (Conv.rewr-conv (map (fn th => th RS @\{thm Eq-FalseI\})
            distincts))))
            Conv.all-conv then-conv
            (rewr-conv' @\{\{lemma (False \land P) = False by simp\}\}) ctxt')
            then-conv
            HOLogic.Trueprop-conv
            (HOLogic.eq-conv Conv.all-conv

(Collect-conv (fn (-, ctxt\")) =>
Conv.repeat-conv
(Conv.bottom-conv
(K (rewr-conv' @{lemma (\exists x. P = P by simp)}) ctxt\"))
ctxt\)) 1) ctxt 1
THEN resolve-tac ctxt [set-Nil-l 1] case-thms)
end

in

fun proc ctxt redex =
  let
  fun make-inner-eqs bound-vs Tis eqs t =
    (case dest-case ctxt t of
      SOME (x, T, i, cont, constr-name) =>
        let
          val (vs, body) = strip-abs (Envir.eta-long (map snd bound-vs) cont)
          val x' = incr-boundvars (length vs) x
          val eqs' = map (incr-boundvars (length vs)) eqs
          val constr-t =
            list-comb
            (Const (constr-name, map snd vs --> T), map Bound (((length vs) - 1) downto 0))
          val constr-eq = Const (const-name〈HOL.eq〉, T --> T --> typ〈bool〉) $ constr-t $ x'
          in
            body
          end
        | NONE =>
          (case dest-if t of
            SOME (condition, cont) => make-inner-eqs bound-vs (If :: Tis) (condition :: eqs') cont
          )
        | NONE =>
          if null eqs then NONE (*no rewriting, nothing to be done*)
          else
            let
              val Type (type-name〈list〉, [rT]) = fastype-of1 (map snd bound-vs, t)
              val pat-eq =
                (case try dest-singleton-list t of
                  SOME t' =>
                    Const (const-name〈HOL.eq〉, rT --> rT --> typ〈bool〉) $ Bound (length bound-vs) $ t'
                  | NONE =>
                    Const (const-name〈Set.member〉, rT --> HOLogic.mksetT rT --> typ〈bool〉) $ Bound (length bound-vs) $ (mk-set rT $ t))
              val reverse-bounds = curry subst-bounds
              ((map Bound (((length bound-vs) - 1) downto 0)) @$ (Bound (length

THEORY "List"

bound-vs

val eqs' = map reverse-bounds eqs
val pat-eq' = reverse-bounds pat-eq
val inner-t = fold (fn (T, T) => fn t => HOLogic.exists-const T $ absdummy T t) (rev bound-vs) (fold (curry HOLogic.mk-conj) eqs' pat-eq')
val rhs = HOLogic.mk-Collect (x, rT, inner-t)
val rewrite-rule-t = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs))
in SOME ((Goal.prove ctxt [] rewrite-rule-t (fn {context = ctxt', ...} => tac ctxt' (rev Tis))) RS @{thm eq-reflection})

\end{quote}

\begin{quote}
\textbf{simproc-setup} list-to-set-comprehension (set xs) =
\texttt{\langle K List-to-Set-Comprehension.proc\rangle}
\end{quote}

\textbf{code-datatype} set coset
\textbf{hide-const} (open) coset

\textbf{66.1.2} [] and (#)

\textbf{lemma} not-Cons-self [simp]:
\texttt{xs \neq x # xs}
\textbf{by} (induct xs) auto

\textbf{lemma} not-Cons-self2 [simp]: \texttt{x # xs \neq xs}
\textbf{by} (rule not-Cons-self [symmetric])

\textbf{lemma} neq-Nil-conv: \texttt{(xs \neq []) = (\exists ys. xs = y # ys)}
\textbf{by} (induct xs) auto

\textbf{lemma} tl-Nil: \texttt{tl xs = [] \leftrightarrow xs = [] \lor (\exists x. xs = [x])}
\textbf{by} (cases xs) auto

\textbf{lemmas} Nil-\texttt{tl} = tl-Nil[THEN eq-iff-swap]

\textbf{lemma} length-induct:
\texttt{(\forall xs. \forall ys. length ys < length xs \rightarrow P ys \rightarrow P xs) \rightarrow P xs}
by \,(\text{fact measure-induct})

\textbf{lemma} \, \text{induct-list012}:
\[ P \emptyset; \forall x. P [x]; \forall y z. [ P z; P (y # z) ] \Rightarrow P (x # y # z) \]\(\Rightarrow P x\)
by \,(\text{induction-schema} \, \text{\,(pal-completeness, lexicographic-order})

\textbf{lemma} \, \text{list-nonempty-induct} \,[\text{consumes 1, case-names single cons}]:
\[ x \neq \emptyset; \forall x. P [x]; \forall x z. z \neq \emptyset \Rightarrow P x \Rightarrow P (x # z) \]\(\Rightarrow P x\)
by \,(\text{induction \,x\, rule: \,\text{induct-list012}}) \,\text{auto}

\textbf{lemma} \, \text{inj-split-Cons} \,[\text{simp}]: \text{inj-on} \,(\lambda (xs, n). n#xs) X
by \,(\text{auto intro!: inj-onI})

\textbf{lemma} \, \text{inj-on-Cons1} \,[\text{simp}]: \text{inj-on} \,(#) A
by \,(\text{simp add: inj-on-def})

\subsection{66.1.3 \, length}
Needs to come before \@ because of theorem \text{append-eq-append-conv}.

\textbf{lemma} \, \text{length-append} \,[\text{simp}]: \text{length} \,(xs @ ys) = \text{length} \,xs + \text{length} \,ys
by \,(\text{induct \,xs}) \,\text{auto}

\textbf{lemma} \, \text{length-map} \,[\text{simp}]: \text{length} \,(\text{map} \, f \,xs) = \text{length} \,xs
by \,(\text{induct \,xs}) \,\text{auto}

\textbf{lemma} \, \text{length-rev} \,[\text{simp}]: \text{length} \,(\text{rev} \,xs) = \text{length} \,xs
by \,(\text{induct \,xs}) \,\text{auto}

\textbf{lemma} \, \text{length-tl} \,[\text{simp}]: \text{length} \,\text{tl} \,xs = \text{length} \,xs - 1
by \,(\text{cases \,xs}) \,\text{auto}

\textbf{lemma} \, \text{length-0-conv \,[iff]}: \,(\text{length} \,xs = 0) = (xs = [])
by \,(\text{induct \,xs}) \,\text{auto}

\textbf{lemma} \, \text{length-greater-0-conv \,[iff]}: \,(0 < \text{length} \,xs) = (xs \neq [])
by \,(\text{induct \,xs}) \,\text{auto}

\textbf{lemma} \, \text{length-pos-if-in-set} : x \in \text{set} \,xs \Rightarrow \text{length} \,xs > 0
by \,\text{auto}

\textbf{lemma} \, \text{length-Suc-conv}: \,(\text{length} \,xs = \text{Suc} \,n) = (\exists y \,ys. \,xs = y # ys \land \text{length} \,ys = n)
by \,(\text{induct \,xs}) \,\text{auto}

\textbf{lemmas} \, \text{Suc-length-conv = length-Suc-cone[THEN eq-iff-swap]}

\textbf{lemma} \, \text{Suc-le-length-iff}:
\,(\text{Suc} \,n \leq \text{length} \,xs) = (\exists y \,ys. \,xs = x # ys \land n \leq \text{length} \,ys)
by \,(\text{metis Suc-le-D[of \,n] Suc-le-mono[of \,n] Suc-length-cone[of \,- \,xs])}

\textbf{THEORY} \,\text{“List”}
lemma impossible-Cons: length xs ≤ length ys ⟹ xs = x # ys = False
by (induct xs) auto

lemma list-induct2 [consumes 1, case-names Nil Cons]:
length xs = length ys ⟹ P [] [] ⟹
(∀ x xs y ys. length xs = length ys ⟹ P xs ys ⟹ P (x # xs) (y # ys))
⟹ P xs ys
proof (induct zs arbitrary: ys)
case (Cons x xs) then show ⟨case by (cases ys) simp-all
qed simp

lemma list-induct3 [consumes 2, case-names Nil Cons]:
length xs = length ys = length zs ⟹ P [] [] [] ⟹
(∀ x xs y ys z zs. length xs = length ys = length zs ⟹ P xs ys zs ⟹
P (x # xs) (y # ys) (z # zs))
⟹ P xs ys zs
proof (induct zs arbitrary: ys zs)
case Nil then show ⟨case by simp
next
case (Cons x xs ys zs) then show ⟨case by (cases ys, simp-all)
(cases zs, simp-all)
qed

lemma list-induct4 [consumes 3, case-names Nil Cons]:
length xs = length ys = length zs = length ws ⟹ P [] [] [] [] ⟹
(∀ x xs y ys z zs w ws. length xs = length ys = length zs = length ws ⟹ P xs ys zs ws ⟹
P (x # xs) (y # ys) (z # zs) (w # ws))
⟹ P xs ys zs ws
proof (induct ws arbitrary: ys zs)
case Nil then show ⟨case by simp
next
case (Cons x xs ys zs ws) then show ⟨case by ((cases ys, simp-all), (cases zs, simp-all)) (cases ws, simp-all)
qed

lemma list-induct2':
[ P [] [] ];
∧ x xs. P (x # xs) [];
∧ y ys. P [] (y # ys);
∧ x xs y ys. P xs ys ⟹ P (x # xs) (y # ys) [] ⟹ P xs ys
by (induct xs arbitrary: ys) (case-tac x, auto)+

lemma list-all2-iff:
list-all2 P xs ys ⟷ length xs = length ys ∧ (∀ (x, y) ∈ set (zip xs ys). P x y)
by (induct xs ys rule: list-induct2') auto

lemma neq-if-length-neq: length xs ≠ length ys ⟹ (xs = ys) == False
by (rule Eq-FalseI) auto

66.1.4 @ – append

global-interpretation append: monoid append Nil

proof
  fix xs ys zs :: 'a list
  show (xs @ ys) @ zs = xs @ (ys @ zs)
    by (induct xs) simp-all

show xs @ [] = xs
  by (induct xs) simp-all
qed simp

lemma append-assoc [simp]: (xs @ ys) @ zs = xs @ (ys @ zs)
  by (fact append.assoc)

lemma append-Nil2: xs @ [] = xs
  by (fact append.right-neutral)

lemma append-is-Nil-conv [iff]: (xs @ ys = []) = (xs = [] ∧ ys = [])
  by (induct xs) auto

lemmas Nil-is-append-conv [iff] = append-is-Nil-conv THEN eq-iff-swap

lemma append-self-conv [iff]: (xs @ ys = xs) = (ys = [])
  by (induct xs) auto

lemmas self-append-conv [iff] = append-self-conv THEN eq-iff-swap

lemma append-eq-append-conv [simp]:
  length xs = length ys ∨ length us = length vs
    ⇒ (xs@us = ys@us) = (xs=ys ∧ us=us)
  by (induct xs arbitrary: ys; case-tac ys; force)

lemma append-eq-append-conv2: (xs @ ys = zs @ ts) =
  (∃ us. xs = zs @ us ∧ us @ ys = ts ∨ xs @ us = zs ∧ ys = us @ ts)
proof (induct xs arbitrary: ys zs ts)
  case (Cons x xs)
  then show ?case
    by (cases zs) auto
qed fastforce

lemma same-append-eq [iff, induct-simp]: (xs @ ys = xs @ zs) = (ys = zs)
  by simp

lemma append1-eq-conv [iff]: (xs @ [x] = ys @ [y]) = (xs = ys ∧ x = y)
  by simp

lemma append-same-eq [iff, induct-simp]: (ys @ xs = zs @ xs) = (ys = zs)
by simp

lemma append-self-conv2 [iff]: (xs @ ys = ys) = (xs = [])
using append-same-eq [of - - []] by auto

lemmas self-append-conv2 [iff] = append-self-conv2[THEN eq_iff_swap]

lemma hd-Cons-tl: xs ≠ [] --- hd xs # tl xs = xs
by (fact list.collapse)

lemma hd-append: hd (xs @ ys) = (if xs = [] then hd ys else hd xs)
by (induct xs) auto

lemma hd-append2 [simp]: xs ≠ [] --- hd (xs @ ys) = hd xs
by (simp add: hd-append split: list.split)

lemma tl-append: tl (xs @ ys) = (case xs of [] ⇒ tl ys | z#zs ⇒ zs @ ys)
by (simp split: list.split)

lemma tl-append2 [simp]: xs ≠ [] --- tl (xs @ ys) = tl xs @ ys
by (simp add: tl-append split: list.split)

lemma tl-append-if: tl (xs @ ys) = (if xs = [] then tl ys else tl xs @ ys)
by (simp)

lemma Cons-eq-append-conv: x#xs = ys@zs =
(ys = [] ∧ x#xs = zs ∨ (∃ys'. x#ys' = ys ∧ zs = ys'@zs))
by (cases ys) auto

lemma append-eq-Cons-conv: (ys@zs = x#xs) =
(ys = [] ∧ zs = x#xs ∨ (∃ys'. ys = x#ys' ∧ ys'@zs = xs))
by (cases ys) auto

lemma longest-common-prefix:
∃ps xs' ys'. xs = ps @ xs' ∧ ys = ps @ ys'
∧ (xs' = [] ∨ ys' = [] ∨ hd xs' ≠ hd ys')
by (induct xs ys rule: list_induct2')
(blast, blast, blast,
metis (no_types, opaque_lifting) append-Cons append-Nil list.sel(1))

Trivial rules for solving @-equations automatically.

lemma eq-Nil-appendI: xs = ys --- xs = [] @ ys
by simp

lemma Cons-eq-appendI: [x # xsI = ys; xs = xsI @ zs] --- x # xs = ys @ zs
by auto

lemma append-eq-appendI: [xs @ xsI = zs; ys = xsI @ us] --- xs @ ys = zs @ us
by auto

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.

simproc-setup list-eq ((xs::'a list) = ys) = 
let 
  fun last (cons as Const (const-name (Cons, -) $ - $ xs) = 
    (case xs of Const (const-name (Nil, -) => cons | - => last xs) 
      | last (Const(const-name (append, -) $ - $ ys) = last ys 
      | last t = t; 
  fun list1 (Const(const-name (Cons, -) $ - $ Const(const-name (Nil, -))) = 
false 
  | list1 = true 
  | list1 = false; 
  fun butlast ((cons as Const(const-name (Cons, -) $ x) $ xs) = 
    (case xs of Const (const-name (Nil, -) => xs | - => cons $ butlast xs) 
      | butlast ((app as Const (const-name (append, -) $ x) $ ys) = app $ butlast ys 
      | butlast xs = Const(const-name (Nil, -), fastype-of xs); 
  val rearr-ss = 
    simpset-of (put-simpset HOL-basic-ss context 
              addsimps [@{thm append-assoc}, @{thm append-Nil}, @{thm append-Cons}]); 
  fun list-eq ctxt (F as (eq as Const(-,eqT)) $ lhs $ rhs) = 
    let 
      val lastl = last lhs and lastr = last rhs; 
      fun rearr conv = 
        let 
          val lhs1 = butlast lhs and rhs1 = butlast rhs; 
          val Type(-,listT::-) = eqT 
          val appT = [listT,listT] => listT 
          val app = Const(const-name (append,appT) 
          val F2 = eq $ (app$lhs$lastl) $ (app$rhs$lastr) 
          val eq = HO Logic.mk-Trueprop (HO Logic.mk-eq (F,F2)); 
          val thm = Goal.prove ctxt [] [] eq 
            (K (simp-tac (put-simpset rearr-ss ctxt) 1)); 
        in SOME ((conv RS (thm RS trans)) RS eq-reflection) end; 
      in 
        if list1 lastl andalso list1 lastr then rearr @{thm append1-eq-conv} 
        else if list1 aconv lastr then rearr @{thm append-same-eq} 
        else NONE 
      end; 
    in K (fn ctxt => fn ct => list-eq ctxt (Thm.term-of ct)) end
  end;
66.1.5 map

**lemma** hd-map: \( xs \neq [] \implies \text{hd} \ (\text{map} \ f \ xs) = f \ (\text{hd} \ xs) \)
by (cases \( xs \)) simp-all

**lemma** map-tl: \( \text{map} \ f \ (\text{tl} \ xs) = \text{tl} \ (\text{map} \ f \ xs) \)
by (cases \( xs \)) simp-all

**lemma** map-ext: \( (\forall x. x \in \text{set} \ xs \implies f x = g x) \implies \text{map} \ f \ xs = \text{map} \ g \ xs \)
by (induct \( xs \)) simp-all

**lemma** map-ident [simp]: \( \text{map} \ (\lambda x. x) = (\lambda xs. xs) \)
by (rule ext, induct-tac \( xs \)) auto

**lemma** map-append [simp]: \( \text{map} \ f \ (xs \ @ \ ys) = \text{map} \ f \ xs \ @ \ \text{map} \ f \ ys \)
by (induct \( xs \)) auto

**lemma** map-map [simp]: \( \text{map} \ f \ (\text{map} \ g \ xs) = \text{map} \ (f \circ g) \ xs \)
by (induct \( xs \)) auto

**lemma** map-comp-map [simp]: \( ((\text{map} \ f) \circ (\text{map} \ g)) = \text{map} (f \circ g) \)
by (rule ext) simp

**lemma** rev-map: \( \text{rev} \ (\text{map} \ f \ xs) = \text{map} \ f \ (\text{rev} \ xs) \)
by (induct \( xs \)) auto

**lemma** map-eq-conv [simp]: \( (\text{map} \ f \ xs = \text{map} \ g \ xs) = (\forall x \in \text{set} \ xs. f x = g x) \)
by (induct \( xs \)) auto

**lemma** map-cong [fundef-cong]:
\[
xs = ys \implies (\forall x \in \text{set} \ ys \implies f x = g x) \implies \text{map} \ f \ xs = \text{map} \ g \ ys
\]
by simp

**lemma** map-is-Nil-conv [iff]: \( (\text{map} \ f \ xs = []) = (xs = []) \)
by (rule list.map-disc-iff)

**lemmas** Nil-is-map-conv [iff] = map-is-Nil-conv THEN eq-iff-swap

**lemma** map-eq-Cons-conv:
\[
(\text{map} \ f \ xs = y \# ys) = (\exists z \ zs. xs = z \# zs \land f z = y \land \text{map} \ f \ zs = ys)
\]
by (cases \( xs \)) auto

**lemma** Cons-eq-map-conv:
\[
(x \# xs = \text{map} \ f \ ys) = (\exists z \ zs. ys = z \# zs \land x = f z \land xs = \text{map} \ f \ zs)
\]
by (cases \( ys \)) auto

**lemmas** map-eq-Cons-D = map-eq-Cons-conv [THEN iffD1]
**lemmas** Cons-eq-map-D = Cons-eq-map-conv [THEN iffD1]
**declare** map-eq-Cons-D [dest!], Cons-eq-map-D [dest!]
lemma \textit{ex-map-conv}:
\[(\exists \, \text{xs} \cdot \text{ys} = \text{map } f \text{ xs}) = (\forall \, y \in \text{set } \text{ys} \cdot \exists \, x \cdot y = f \, x)\]
by (induct \text{ys}, auto simp add: Cons-eq-map-conv)

lemma \textit{map-eq-imp-length-eq}:
assumes \[
\text{map } f \text{ xs} = \text{map } g \text{ ys} \]
shows \[
\text{length } \text{xs} = \text{length } \text{ys} \]
using assms
proof (induct \text{ys} arbitrary: \text{xs})
  case Nil then show \text{?case} by simp
next
  case (Cons \, y \, ys) then obtain \text{z} \, zs where \text{xs}: \text{xs} = \text{z} \# \text{zs}
    by (auto)
  from Cons \text{xs} have \text{map } f \text{ zs} = \text{map } g \text{ ys}
    by simp
  with \text{xs} show \text{?case} by simp
qed

lemma \textit{map-inj-on}:
assumes \[
\text{map } \text{xs} = \text{map } \text{ys} \text{ and } \text{inj-on } \text{f} \text{ (set } \text{xs Un set } \text{ys}) \]
shows \[
\text{xs} = \text{ys} \]
using map-eq-imp-length-eq [OF map] assms
proof (induct rule: list-induct2)
  case (Cons \text{x} \text{xs} \text{y} \text{ys})
  then show \text{?case} by (auto intro: sym)
qed auto

lemma \textit{inj-on-map-eq-map}:
\text{inj-on } \text{f} \text{ (set } \text{xs Un set } \text{ys}) \implies (\text{map } f \text{ xs} = \text{map } f \text{ ys}) = (\text{xs} = \text{ys})
by (blast dest: map-inj-on)

lemma \textit{map-injective}:
\[
\text{map } f \text{ xs} = \text{map } f \text{ ys} \implies \text{inj } f \implies \text{xs} = \text{ys} \]
by (induct ys arbitrary: \text{xs}) (auto dest!: injD)

lemma \textit{inj-map-eq-map\{simp\}}: \text{inj } f \implies (\text{map } f \text{ xs} = \text{map } f \text{ ys}) = (\text{xs} = \text{ys})
by (blast dest: map-injective)

lemma \textit{inj-mapI}: \text{inj } f \implies (\text{map } f)
by (iprover dest: map-injective injD intro: inj-on I)

lemma \textit{inj-mapD}: \text{inj } (\text{map } f) \implies \text{inj } f
by (metis (no-types, opaque-lifting) injI list.inject list.simps(9) the-inv-f-f)

lemma \textit{inj-map\{iff\}}: \text{inj } (\text{map } f) = \text{inj } f
by (blast dest: inj-mapD intro: inj-mapI)

lemma \textit{inj-on-mapI}: \text{inj-on } f \text{ (} \bigcup \, (\text{set } \text{A}) \text{)} \implies \text{inj-on } (\text{map } f) \text{ A}
by (blast intro: inj-onI dest: inj-onD map-inj-on)
lemma map-idI: (∀x. x ∈ set xs → f x = x) → map f xs = xs
by (induct xs, auto)

lemma map-fun-upd [simp]: y ∉ set xs → map (f(y:=v)) xs = map f xs
by (induct xs) auto

lemma map-fst-zip [simp]:
  length xs = length ys → map fst (zip xs ys) = xs
by (induct rule: list-induct2, simp-all)

lemma map-snd-zip [simp]:
  length xs = length ys → map snd (zip xs ys) = ys
by (induct rule: list-induct2, simp-all)

lemma map2-map-map: map2 h (map f xs) (map g xs) = map (λx. h (f x) (g x)) xs
by (induction xs) (auto)

functor map: map
by (simp-all add: id-def)

declare map.id [simp]

66.1.6  rev

lemma rev-append [simp]: rev (xs @ ys) = rev ys @ rev xs
by (induct xs) auto

lemma rev-rev-ident [simp]: rev (rev xs) = xs
by (induct xs) auto

lemma rev-swap: (rev xs = ys) = (xs = rev ys)
by auto

lemma rev-is-Nil-conv [iff]: (rev xs = []) = (xs = [])
by (induct xs) auto

lemmas Nil-is-rev-conv [iff] = rev-is-Nil-conv[THEN eq-iff-swap]

lemma rev-singleton-conv [simp]: (rev xs = [x]) = (xs = [x])
by (cases xs) auto

lemma singleton-rev-conv [simp]: ([x] = rev xs) = ([x] = xs)
by (cases xs) auto

lemma rev-is-rev-conv [iff]: (rev xs = rev ys) = (xs = ys)
proof (induct xs arbitrary: ys)
  case Nil
  then show ?case by force
next
  case Cons
  then show ?case by (cases ys) auto
qed

lemma inj-on-rev [iff]: inj-on rev A
by (simp add: inj-on-def)

lemma rev-induct [case-names Nil snoc]:
  assumes P [] and \( \forall xs. P \Rightarrow P (xs @ [x]) \)
  shows P xs
proof
  have P (rev (rev xs))
    by (rule-tac list = rev xs in list.induct, simp-all add: assms)
  then show ?thesis by simp
qed

lemma rev-exhaust [case-names Nil snoc]:
  (xs = [] \( \Rightarrow P \) \( \Rightarrow (\forall ys. P \Rightarrow P (ys @ [y])) \) \( \Rightarrow P \))
by (induct xs rule: rev-induct) auto

lemmas rev-cases = rev-exhaust

lemma rev-nonempty-induct [consumes 1, case-names single snoc]:
  assumes xs \( \neq [] \)
  and single: \( \forall x. P [x] \)
  and snoc': \( \forall xs. xs \neq [] \Rightarrow P xs \Rightarrow P (xs@[x]) \)
  shows P xs
using (xs \( \neq [] \)), proof (induct xs rule: rev-induct)
  case (snoc x xs) then show ?case
  proof (cases xs)
    case Nil thus ?thesis by (simp add: single)
  next
    case Cons with snoc show ?thesis by (fastforce intro!: snoc')
  qed
qed simp

lemma rev-eq-Cons-iff [iff]: (rev xs = y#ys) = (xs = rev ys @ [y])
by (rule rev-cases[of xs]) auto
lemma length-Suc-conv-rev: \( (\text{length } xs = \text{Suc } n) = (\exists y ys. xs = ys @ [y] \land \text{length } ys = n) \)
by (induct xs rule: rev-induct) auto

66.1.7 set

declare list.set[code-post] — pretty output

lemma finite-set [iff]: finite (set xs)
by (induct xs) auto

lemma set-append [simp]: set (xs @ ys) = (set xs \cup set ys)
by (induct xs) auto

lemma hd-in-set[simp]: xs \neq [] \implies hd xs \in set xs
by (cases xs) auto

lemma set-subset-Cons: set xs \subseteq set (x # xs)
by auto

lemma set-ConsD: y \in set (x # xs) \implies y=x \lor y \in set xs
by auto

lemma set-empty [iff]: (set xs = {}) = (xs = [])
by (induct xs) auto

lemmas set-empty2 [iff] = set-empty[THEN eq-iff-swap]

lemma set-rev [simp]: set (rev xs) = set xs
by (induct xs) auto

lemma set-map [simp]: set (map f xs) = f'(set xs)
by (induct xs) auto

lemma set-filter [simp]: set (filter P xs) = \{x. x \in set xs \land P x\}
by (induct xs) auto

lemma set-upt [simp]: set[i..<j] = \{i..<j\}
by (induct j) auto

lemma split-list: x \in set xs \implies \exists ys zs. xs = ys @ x # zs
proof (induct xs)
  case Nil thus ?case by simp
next
  case Cons thus ?case by (auto intro: Cons-eq-appendI)
qed

lemma in-set-conv-decomp: x \in set xs \iff (\exists ys zs. xs = ys @ x # zs)
by (auto elim: split-list)

lemma split-list-first: $x \in \text{set } xs \Longrightarrow \exists ys \ zs. \ xs = ys @ x \# zs \land x \notin \text{set } ys$
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons a xs)
  show ?case
  proof cases
    assume $x = a$ thus ?case using Cons by fastforce
  next
    assume $x \neq a$ thus ?case using Cons (fastforce intro: Cons-eq-appendI)
  qed
qed

lemma in-set-conv-decomp-first:
  $(x \in \text{set } xs) = (\exists ys \ zs. \ xs = ys @ x \# zs \land x \notin \text{set } ys)$
by (auto dest!: split-list-first)

lemma split-list-last: $x \in \text{set } xs \Longrightarrow \exists ys \ zs. \ xs = ys @ x \# zs \land x \notin \text{set } zs$
proof (induct xs rule: rev-induct)
  case Nil thus ?case by simp
next
  case (snoc a xs)
  show ?case
  proof cases
    assume $x = a$ thus ?case using snoc by (auto intro!: exI)
  next
    assume $x \neq a$ thus ?case using snoc by fastforce
  qed
qed

lemma in-set-conv-decomp-last:
  $(x \in \text{set } xs) = (\exists ys \ zs. \ xs = ys @ x \# zs \land x \notin \text{set } zs)$
by (auto dest!: split-list-last)

lemma split-list-prop: $\exists x \in \text{set } xs. \ P \ x \Longrightarrow \exists ys \ zs. \ xs = ys @ x \# zs \land P \ x$
proof (induct xs)
  case Nil thus ?case by simp
next
  case Cons thus ?case
  by (simp add: Bex_def) (metis append-Cons append.simps(1))
qed

lemma split-list-propE:
  assumes $\exists x \in \text{set } xs. \ P \ x$
  obtains $ys \ zs$ where $zs = ys @ x \# zs$ and $P \ x$
using split-list-prop [OF assms] by blast
lemma split-list-first-prop:
\[ \exists x \in \text{set} \; xs. \; P \; \Rightarrow \; \exists y \in \text{set} \; ys. \; \forall y \in \text{set} \; ys. \; \neg P \; y \]
proof (induct xs)
  case Nil thus ?case by simp
next
case (Cons x xs)
  show ?case
  proof cases
    assume P x
    hence \[ x \# xs = \emptyset @ x \# xs \land P \; x \land (\forall y \in \text{set} \; \emptyset. \; \neg P \; y) \] by simp
    thus \( \simthesis \) by fast
  next
    assume \( \neg P \; x \)
    hence \( \exists x \in \text{set} \; xs. \; P \; x \) using Cons(2) by simp
    thus \( \simthesis \) using (\( \neg P \; x \)) Cons(1) by (metis append-Cons set-ConsD)
  qed
qed

lemma split-list-first-propE:
  assumes \( \exists x \in \text{set} \; xs. \; P \; x \)
  obtains y \; x \; zs where \( xs = y @ x \# zs \land P \; x \land (\forall y \in \text{set} \; ys. \; \neg P \; y) \)
  using split-list-first-prop \[ \text{OF assms} \] by blast

lemma split-list-first-prop-iff:
\( (\exists x \in \text{set} \; xs. \; P \; x) \Longleftrightarrow (\exists y \in \text{set} \; ys. \; \neg P \; y) \)
by (rule, erule split-list-first-prop) auto

lemma split-list-last-prop:
\[ \exists x \in \text{set} \; xs. \; P \; x \Rightarrow \exists y \in \text{set} \; ys. \; \forall y \in \text{set} \; ys. \; \neg P \; y \]
proof (induct xs rule:rev-induct)
  case Nil thus ?case by simp
next
case (snoc x xs)
  show ?case
  proof cases
    assume P x thus \( \simthesis \) by (auto intro: exI)
  next
    assume \( \neg P \; x \)
    hence \( \exists x \in \text{set} \; xs. \; P \; x \) using snoc(2) by simp
    thus \( \simthesis \) using (\( \neg P \; x \)) snoc(1) by fastforce
  qed
qed

lemma split-list-last-propE:
  assumes \( \exists x \in \text{set} \; xs. \; P \; x \)
  obtains y \; x \; zs where \( xs = y @ x \# zs \land P \; x \land (\forall z \in \text{set} \; zs. \; \neg P \; z) \)
using split-list-last-prop [OF assms] by blast

lemma split-list-last-prop-iff:
  \((\exists x \in \text{set} \ xs. P x) \iff (\exists ys x zs. xs = ys @ x # zs \land P x \land (\forall z \in \text{set} \ zs. \neg P z))\)
by rule (erule split-list-last-prop, auto)

lemma finite-list: finite A \implies \exists xs. \text{set} \ xs = A
by (erule finite-induct) (auto simp add: list.set[symmetric] simp del: list.set)

lemma card-length: \text{card} (\text{set} \ xs) \leq \text{length} \ xs
by (induct xs) (auto simp add: card-insert-if)

lemma set-minus-filter-out: 
\text{set} \ xs - \{y\} = \text{set} (\text{filter} (\lambda x. \neg (x = y)) \ xs)
by (induct xs) auto

lemma append-Cons-eq-iff:
\[ x \notin \text{set} \ xs; x \notin \text{set} \ ys \] \implies 
xs @ x # ys = xs' @ x # ys' \iff (xs = xs' \land ys = ys')
by (auto simp: append-eq-Cons-conv Cons-eq-append-conv append-eq-append-conv2)

66.1.8 concat

lemma concat-append [simp]: concat (xs @ ys) = concat xs @ concat ys
by (induct xs) auto

lemma concat-eq-Nil-conv [simp]: (concat xss = []) = (\forall xs \in \text{set} \ xss. xs = [])
by (induct xss) auto

lemmas Nil-eq-concat-conv = concat-eq-Nil-conv[THEN eq-iff-swap]

lemma set-concat [simp]: set (concat xs) = (\bigcup x \in \text{set} \ xs. set x)
by (induct xs) auto

lemma concat-map-singleton[simp]: concat(map (%x. [f x]) xs) = map f xs
by (induct xs) auto

lemma map-concat: map f (concat xs) = concat (map (map f) xs)
by (induct xs) auto

lemma rev-concat: rev (concat xs) = concat (map rev (rev xs))
by (induct xs) auto

lemma length-concat-rev[simp]: length (concat (rev xs)) = length (concat xs)
by (induction xs) auto

lemma concat-eq-concat-iff: \forall (x, y) \in \text{set} (\text{zip} \ xs \ ys). length x = length y \implies
length xs = length ys ⇒ (concat xs = concat ys) = (xs = ys)

proof (induct xs arbitrary: ys)
  case (Cons x xs ys)
  thus ?case by (cases ys) auto
qed (auto)

lemma concat-injective: concat xs = concat ys ⇒ length xs = length ys ⇒ ∀(x, y) ∈ set (zip xs ys). length x = length y ⇒ xs = ys
  by (simp add: concat-eq-concat-iff)

lemma concat-eq-appendD:
  assumes concat xss = ys @ zs xss ≠ []
  shows ∃xss1 xs xs' xss2. xss = xss1 @ (xs @ xs') ≠ xss2 ∧ ys = concat xss1 @ xs ∧ zs = xs' @ concat xss2
  using assms
proof (induction xss arbitrary: ys)
  case (Cons xs xss)
  from Cons.prems consider
  us where xs @ us = ys concat xss = us @ zs |
  us where zs = ys @ us us @ concat xss = zs
  by (auto simp add: append-eq-append-conv2)
then show ?case
proof cases
  case 1
  then show ?thesis using Cons.IH[OF 1(2)]
    by (cases xss) (auto intro: exI[where x=[]], metis append.assoc append-Cons concat.simps(2))
qed (auto intro: exI[where x=[]])
qed simp

lemma concat-eq-append-conv:
  concat xss = ys @ zs \iff (if xss = [] then ys = [] ∧ zs = [] 
   else ∃xss1 xs xs' xss2. xss = xss1 @ (xs @ xs') ≠ xss2 ∧ ys = concat xss1 @ xs ∧ zs = xs' @ concat xss2)
  by (auto dest: concat-eq-appendD)

lemma hd-concat: [\exists x: x ≠ []; hd x ≠ []] \implies hd (concat xs) = hd (hd xs)
  by (metis concat.simps(2) hd-Cons tl hd-append2)

simproc-setup list-neq ((xs::'a list) = ys) = (*/
Reduces xs=ys to False if xs and ys cannot be of the same length.
This is the case if the atomic sublists of one are a submultiset
of those of the other list and there are fewer Cons's in one than the other.*/)

let
fun len (Const(const-name(Nil,-))) acc = acc
  | len (Const(const-name(Cons,-)) $ xs) (ts,n) = len xs (ts,n+1)
  | len (Const(const-name(append,-)) $ xs $ ys) acc = len xs (len ys acc)
  | len (Const(const-name(rev,-)) $ xs) acc = len xs acc
  | len (Const(const-name(map,-)) $ xs) acc = len xs acc
  | len (Const(const-name(concat,T)) $ (Const(const-name(rev,-)) $ xss)) acc = len (Const(const-name(concat,T)) $ xss) acc
  | len t (ls,n) = (t::ts,n);

val ss = simpset-of context;

fun list-neq ctxt ct =
  let
  val (Const(eqT) $ lhs $ rhs) = Thm.term-of ct;
  val (ls,m) = len lhs ([],0) and (rs,n) = len rhs ([],0);
  fun prove-neq() =
    let
      val Type(-,listT::-) = eqT;
      val size = HOLogic.size-const listT;
      val eq-len = HOLogic.mk_eq (size $ lhs, size $ rhs);
      val neq-len = HOLogic.mk_Trueprop (HOLogic.Not $ eq-len);
      val thm = Goal.prove ctxt [] [] neq-len
        (K (simp-tac (put-simpset ss ctxt) 1));
    in SOME (thm RS @{thm neq-if-length-neq}) end
  in
    if m < n andalso submultiset (op aconv) (ls,rs) orelse
      n < m andalso submultiset (op aconv) (rs,ls)
    then prove-neq() else NONE
  end;
  in K list-neq end

66.1.9 filter

lemma filter-append [simp]: filter P (xs @ ys) = filter P xs @ filter P ys
  by (induct xs) auto

lemma rev-filter: rev (filter P xs) = filter P (rev xs)
  by (induct xs) simp-all

lemma filter-filter [simp]: filter P (filter Q xs) = filter (λx. Q x ∧ P x) xs
  by (induct xs) auto

lemma filter-concat: filter p (concat xs) = concat (map (filter p) xs)
  by (induct xs) auto

lemma length-filter-le [simp]: length (filter P xs) ≤ length xs
  by (induct xs) (auto simp add: le-SucI)
lemma sum-length-filter-compl:
  length(filter P xs) + length(filter (λx. ¬P x) xs) = length xs
by (induct xs) simp-all

lemma filter-True [simp]: ∀x ∈ set xs. P x ⇒ filter P xs = xs
by (induct xs) auto

lemma filter-False [simp]: ∀x ∈ set xs. ¬P x ⇒ filter P xs = []
by (induct xs) auto

lemma filter-empty-conv: (filter P xs = []) = (∀x∈set xs. ¬P x)
by (induct xs) simp-all

lemmas empty-filter-conv = filter-empty-conv[THEN eq-iff-swap]

lemma filter-id-conv: (filter P xs = xs) = (∀x∈set xs. P x)
proof (induct xs)
case (Cons x xs)
then show ?case using length-filter-le
  by (simp add: impossible-Cons)
qed auto

lemma filter-map: filter P (map f xs) = map f (filter (P ◦ f) xs)
by (induct xs) simp-all

lemma length-filter-map[simp]:
  length (filter P (map f xs)) = length(filter (P ◦ f) xs)
by (simp add: filter-map)

lemma filter-is-subset [simp]: set (filter P xs) ≤ set xs
by auto

lemma length-filter-less:
  [ x ∈ set xs; ¬ P x ] ⇒ length(filter P xs) < length xs
proof (induct xs)
case Nil thus ?case by simp
next
case (Cons x xs) thus ?case
  using Suc-le-eq by fastforce
qed

lemma length-filter-conv-card:
  length(filter p xs) = card{ i. i < length xs ∧ p(xs!i)}
proof (induct xs)
case Nil thus ?case by simp
next
case (Cons x xs)
let \(?S = \{ i. i < \text{length } xs \land p(xs[i]) \}\)

have fin: finite \(?S\) by (fast intro: bounded-nat-set-is-finite)

show \(?\text{case (is } \forall i = \text{card } ?S')\)
proof (cases)
  assume \(p x\)
  hence eq: \(?S' = \text{insert } 0 (\text{Suc } ?S)\)
    by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
  have length (filter \(p\) (\(x \# xs\))) = \text{card } ?S
    using Cons \((p x)\) by simp
  also have \(. . . = \text{Suc(card}\(\text{Suc } ?S)\)) using fin
    by (simp add: card-image)
  also have \(. . . = \text{card } ?S'\) using eq fin
    by (simp add:card-insert-if)
  finally show ?thesis.
next
  assume \(\neg p x\)
  hence eq: \(?S' = \text{Suc } ?S\)
    by (auto simp add: image-def split:nat.split elim:lessE)
  have length (filter \(p\) (\(x \# xs\))) = \text{card } ?S
    using Cons \((\neg p x)\) by simp
  also have \(. . . = \text{card}\(\text{Suc } ?S)\) using fin
    by (simp add: card-image)
  also have \(. . . = \text{card } ?S'\) using eq fin
    by (simp add:card-insert-if)
  finally show ?thesis.
qed

lemma Cons-eq-filterD:
\(x \# xs = \text{filter } P\) \(ys \implies \exists us vs. ys = us \at x \# vs \land (\forall u \in \text{set } us. \neg P\) u) \land P x \land xs = \text{filter } P\) vs
(is - \implies \exists us vs. \(\neg P\) ys us vs)
proof (induct \(ys\))
case Nil thus ?case by simp
next
case \((\text{Cons } y\) ys)\)
show ?case (is \(\exists x. \ ?Q\) x)\)
proof cases
  assume Py: \(P y\)
  show ?thesis
  proof cases
    assume \(x = y\)
    with Py Cons.prems have \(?Q\) \[ by simp
    then show ?thesis ..
  next
    assume \(x \neq y\)
    with Py Cons.prems show ?thesis by simp
  qed
next
THEORY “List”

assume ¬ P y
with Cons obtain us vs where ?P (y#ys) (y#us) vs by fastforce
then have ?Q (y#us) by simp
then show ?thesis ..
qed
qed

lemma filter-eq-ConsD:
filter P ys = x#xs ⇒
∃ us vs. ys = us @ x # vs ∧ (∀ u ∈ set us. ¬ P u) ∧ P x ∧ xs = filter P vs
by (rule Cons-eq-filterD) simp

lemma filter-eq-Cons-iff:
(filter P ys = x#xs) =
(∃ us vs. ys = us @ x # vs ∧ (∀ u ∈ set us. ¬ P u) ∧ P x ∧ xs = filter P vs)
by (auto dest: filter-eq-ConsD)

lemmas Cons-eq-filter-iff = filter-eq-Cons-iff[THEN eq-iff-swap]

lemma inj-on-filter-key-eq:
assumes inj-on f (insert y (set xs))
shows filter (λx. f y = f x) xs = filter (HOL.eq y) xs
using assms by (induct xs) auto

lemma filter-cong[fundef-cong]:
xs = ys ⇒ (∀ x ∈ set ys. P x) ⇒ filter P xs = filter P ys
by (induct ys arbitrary: xs) auto

66.1.10 List partitioning

primrec partition :: (′a ⇒ bool) ⇒ ′a list ⇒ ′a list × ′a list where
partition P [] = ([], []) |
partition P (x # xs) =
(let (yes, no) = partition P xs
    in if P x then (x # yes, no) else (yes, x # no))

lemma partition-filter1: fst (partition P xs) = filter P xs
by (induct xs) (auto simp add: Let-def split-def)

lemma partition-filter2: snd (partition P xs) = filter (Not o P) xs
by (induct xs) (auto simp add: Let-def split-def)

lemma partition-P:
assumes partition P xs = (yes, no)
shows (∀ p ∈ set yes. P p) ∧ (∀ p ∈ set no. ¬ P p)
proof
from assms have yes = fst (partition P xs) and no = snd (partition P xs)
by simp-all
then show ?thesis by (simp-all add: partition-filter1 partition-filter2)
lemma partition-set:
  assumes partition P xs = (yes, no)
  shows set yes \cup set no = set xs
proof –
  from assms have yes = fst (partition P xs) and no = snd (partition P xs)
  by simp-all
  then show ?thesis by (auto simp add: partition-filter1 partition-filter2)
qed

lemma partition-filter-conv [simp]:
  partition f xs = (filter f xs, filter (Not \circ f) xs)
proof
  by simp
next
  unfolding partition-filter2 [symmetric]
next
  unfolding partition-filter1 [symmetric] by simp
qed

declare partition.simps [simp del]

66.1.11 (!)

lemma nth-Cons-0 [simp, code]: (x # xs)!0 = x
  by auto

lemma nth-Cons-Suc [simp, code]: (x # xs)!Suc n = xs!n
  by auto

declare nth.simps [simp del]

lemma nth-Cons-pos [simp]: 0 < n \Rightarrow (x#xs)!n = xs!(n - 1)
proof
  by (auto simp: Nat.gr0_conv_Suc)

lemma nth-append:
  (xs @ ys)!n = (if n < length xs then xs!n else ys!(n - length xs))
proof
  (induct xs arbitrary: n)
  case (Cons x xs)
  then show ?case
    using less-Suc-eq-0-disj by auto
qed simp

lemma nth-append-length [simp]: (xs @ x # ys)!length xs = x
proof
  (induct xs)
next
lemma nth-append-length-plus [simp]: (xs @ ys)! (length xs + n) = ys!n
proof
  (induct xs)
next
lemma nth-map [simp]: n < length xs \Rightarrow (map f xs)!n = f(xs!n)
proof
  (induct xs arbitrary: n)
  case (Cons x xs)
  then show ?case
using less-Suc-eq-0-disj by auto

qed simp

lemma nth-tl: \( n < \text{length} (tl \ xs) \implies tl \ xs ! n = xs ! Suc n \)
  by (induction xs) auto

lemma hd-conv-nth: \( xs \neq [] \implies hd \ xs = xs!0 \)
  by (cases xs) simp-all

lemma list-eq-iff-nth-eq:
  \((xs = ys) = (\text{length} \ xs = \text{length} \ ys \land (\forall i < \text{length} \ xs. xs!i = ys!i))\)
proof (induct xs arbitrary: ys)
  case (Cons x xs ys)
  show ?case
  proof (cases ys)
    case (Cons y ys)
    with Cons.hyps show ?thesis by fastforce
  qed simp
  qed force

lemma map-equality-iff:
  \( \text{map} \ f \ xs = \text{map} \ g \ ys \longleftrightarrow \text{length} \ xs = \text{length} \ ys \land (\forall i < \text{length} \ ys. f (xs!i) = g (ys!i)) \)
  by (fastforce simp: list-eq-iff-nth-eq)

lemma set-conv-nth:
  \( \text{set} \ xs = \{xs!i \mid i. \ i < \text{length} \ xs\} \)
proof (induct xs)
  case (Cons x xs)
  have insert x \{xs!i \mid i. \ i < \text{length} \ xs\} = \{(x # xs)!i \mid i. \ i < Suc (\text{length} \ xs)\}
  (is ?L=?R)
  proof
    show \?L \subseteq \?R
      by force
    show \?R \subseteq \?L
      using less-Suc-eq-0-disj by auto
  qed
  with Cons show ?case
  proof simp
  qed simp

lemma in-set-conv-nth: \( (x \in \text{set} \ xs) = (\exists i < \text{length} \ xs. xs!i = x) \)
  by(auto simp:set-conv-nth)

lemma nth-equal-first-eq:
  assumes x /\in set xs
  assumes n \leq \text{length} \ xs
  shows \( (x \# xs)!n = x \longleftrightarrow n = 0 \) (is ?lhs \longleftrightarrow ?rhs)
proof
  assume ?lhs
show \(?rhs\)
proof (rule ccontr)
  assume \(n \neq 0\)
  then have \(n > 0\) by simp
  with \(?lhs\) have \(xs ! (n - 1) = x\) by simp
moreover from \(n > 0\) and \(n \leq \text{length} \, xs\) have \(n - 1 < \text{length} \, xs\) by simp
ultimately have \(\exists \, i < \text{length} \, xs. \, xs ! i = x\) by auto
with \(x \notin \text{set} \, xs\) in-set-conv-nth [of \(x \, xs\)] show \(\text{False}\) by simp
qed

next
  assume \(?rhs\) then show \(?lhs\) by simp
qed

lemma nth-non-equal-first-eq:
  assumes \(x \neq y\)
  shows \((x \# xs) ! n = y \iff xs ! (n - 1) = y \land n > 0\) (is \(?lhs \iff \?rhs\))
proof
  assume \(?lhs\) with \(\text{assms}\) have \(n > 0\) by (cases \(n\)) simp-all
  with \(?lhs\) show \(?rhs\) by simp
next
  assume \(?rhs\) then show \(?lhs\) by simp
qed

lemma list-ball-nth:
  \([\forall i < \text{length} \, xs. \, \forall x \in \text{set} \, xs. \, P \, x] \implies P (xs ! n)\)
by (auto simp add: set-conv-nth)

lemma nth-mem [simp]: \(n < \text{length} \, xs \implies xs ! n \in \text{set} \, xs\)
by (auto simp add: set-conv-nth)

lemma all-nth-imp-all-set:
  \([\forall i < \text{length} \, xs. \, P (xs ! i); \, x \in \text{set} \, xs]\) \implies P \, x
by (auto simp add: set-conv-nth)

lemma all-set-conv-all-nth:
  (\(\forall x \in \text{set} \, xs. \, P \, x\)) = (\(\forall i. \, i < \text{length} \, xs \implies P (xs ! i)\))
by (auto simp add: set-conv-nth)

lemma rev-nth:
  \(n < \text{size} \, xs \implies \text{rev} \, xs ! n = xs ! (\text{length} \, xs - Suc \, n)\)
proof (induct \(xs\) arbitrary: \(n\))
  case Nil thus \(?case\) by simp
next
  case (Cons \(x \, xs\))
hence \(n < Suc \, (\text{length} \, xs)\) by simp
moreover
  { assume \(n < \text{length} \, xs\)
  with \(n\) obtain \(n'\) where \(n' : \text{length} \, xs - n = Suc \, n'\)
  by (cases \(\text{length} \, xs - n\), auto)
  moreover

from $n'$ have $\text{length } xs - \text{Suc } n = n'$ by simp
ultimately
have $xs \mapsto (\text{length } xs - \text{Suc } n) = (x \neq xs) \mapsto (\text{length } xs - n)$ by simp
\}
ultimately
show ?case by (clarsimp simp add: Cons nth-append)
qed

lemma Skolem-list-nth:
$(\forall i < k. \exists x. P i x) = (\exists xs. \text{size } xs = k \wedge (\forall i < k. P i (xs!i)))$
(is - = $(\exists xs. ?P k xs))
proof (induct $k$
  case 0 show ?case by simp
next
  case (Suc $k$
    show ?case by simp add: nth-append less-Suc-eq)
  next
    assume ?R thus ?L using Suc by auto
  qed
qed

66.1.12 list-update

lemma length-list-update [simp]: $\text{length}(xs[i:=x]) = \text{length } xs$
by (induct $xs$ arbitrary: $i$) (auto split: nat.split)

lemma nth-list-update:
i < $\text{length } xs$ \imp $(xs[i:=x])[j] = (if i = j then x else xs!j)$
by (induct $xs$ arbitrary: $i$ $j$) (auto simp add: nth-Cons split: nat.split)

lemma nth-list-update-eq [simp]: $i < \text{length } xs \imp (xs[i:=x])!i = x$
by (simp add: nth-list-update)

lemma nth-list-update-neq [simp]: $i \neq j \imp xs[i:=x]!j = xs!j$
by (induct $xs$ arbitrary: $i$ $j$) (auto simp add: nth-Cons split: nat.split)

lemma list-update-id[simp]: $xs[i := xs!i] = xs$
by (induct $xs$ arbitrary: $i$) (simp-all split: nat.split)

lemma list-update-beyond[simp]: $\text{length } xs \leq i \imp xs[i:=x] = xs$
proof (induct $xs$ arbitrary: $i$)
  case (Cons $x$ $xs$ $i$)
  then show ?case by (metis leD length-list-update list-eq-iff-nth-eq nth-list-update-neq)
**THEORY "List"**

```plaintext
qed simp

lemma list-update-nonempty[simp]: xs[k:=x] = [] <-> xs=[]
  by (simp only: length-0-cone[symmetric] length-list-update)

lemma list-update-same-conv:
  i < length xs ==> (xs[i:=x] = xs) = (xs!i = x)
  by (induct xs arbitrary: i) (auto split: nat.split)

lemma list-update-appendI:
  i < size xs ==> (xs @ ys)[i:=x] = xs[i:=x] @ ys
  by (induct xs arbitrary: i) (auto split: nat.split)

lemma list-update-append: (xs @ ys) [n:=x] =
  (if n < length xs then xs[n:= x] @ ys else xs @ (ys [n-length xs:= x]))
  by (induct xs arbitrary: n) (auto split: nat.splits)

lemma list-update-length [simp]:
  (xs @ x # ys)[length xs := y] = (xs @ y # ys)
  by (induct xs, auto)

lemma map-update: map f (xs[k:=y]) = (map f xs)[k := f y]
  by (induct xs arbitrary: k) (auto split: nat.splits)

lemma rev-update:
  k < length xs ==> rev (xs[k:=y]) = (rev xs)[length xs - k - 1 := y]
  by (induct xs arbitrary: k) (auto simp: list-update-append split: nat.splits)

lemma update-zip:
  (zip xs ys)[i:=xys] = zip (xs[i:=fst xys]) (ys[i:=snd xys])
  by (induct ys arbitrary: i xys) (auto, case_tac xs, auto split: nat.split)

lemma set-update-subset-insert: set(xs[i:=x]) <= insert x (set xs)
  by (induct xs arbitrary: i) (auto split: nat.split)

lemma set-update-subsetI: [set xs ⊆ A; x ∈ A] ==> set(xs[i := x]) ⊆ A
  by (blast dest!: set-update-subset-insert [THEN subsetD])

lemma set-update-memberI: n < length xs ==> x ∈ set (xs[n := x])
  by (induct xs arbitrary: n) (auto split: nat.splits)

lemma list-update-overwrite[simp]:
  xs [i := x, i := y] = xs [i := y]
  by (induct xs arbitrary: i) (simp-all split: nat.split)

lemma list-update-swap:
  i ≠ i' ==> xs [i := x, i' := x'] = xs [i' := x', i := x]
  by (induct xs arbitrary: i i') (simp-all split: nat.split)
```
lemma list-update-code [code]:
\[
\begin{align*}
\&[[i := y] = [] \\
\&(x \# xs)[0 := y] = y \# xs \\
\&(x \# xs)[\text{Suc } i := y] = x \# xs[i := y]
\end{align*}
\]
by simp-all

66.1.13 last and butlast

lemma hd-Nil-eq-last: hd Nil = last Nil
unfolding hd-def last-def by simp

lemma last-snoc [simp]: last (xs @ [x]) = x
by (induct xs) auto

lemma butlast-snoc [simp]: butlast (xs @ [x]) = xs
by (induct xs) auto

lemma last-ConsL: xs = [] \implies last(x#xs) = x
by simp

lemma last-ConsR: xs \neq [] \implies last(x#xs) = last xs
by simp

lemma last-append: last(xs @ ys) = (if ys = [] then last xs else last ys)
by (induct xs) (auto)

lemma last-appendL[simp]: ys = [] \implies last(xs @ ys) = last xs
by (simp add:last-append)

lemma last-appendR[simp]: ys \neq [] \implies last(xs @ ys) = last ys
by (simp add:last-append)

lemma last-tl: xs = [] \lor tl xs \neq [] \implies last(tl xs) = last xs
by (induct xs) simp-all

lemma butlast-tl: butlast (tl xs) = tl (butlast xs)
by (induct xs) simp-all

lemma hd-rev: hd(rev xs) = last xs
by (metis hd-Cons-tl hd-Nil-eq-last last-snoc rev-eq-Cons-iff rev-isNil-conv)

lemma last-rev: last(rev xs) = hd xs
by (metis hd-rev rev-swap)

lemma last-in-set[simp]: as \neq [] \implies last as \in set as
by (induct as) auto

lemma length-butlast [simp]: length (butlast xs) = length xs - 1
by (induct xs rule: rev-induct) auto

lemma butlast-append:
butlast (xs @ ys) = (if ys = [] then butlast xs else xs @ butlast ys)
by (induct xs arbitrary: ys) auto

lemma append-butlast-last-id [simp]:
xs ≠ [] ⟹ butlast xs @ [last xs] = xs
by (induct xs) auto

lemma in-set-butlastD:
x ∈ set (butlast xs)
shows x ∈ set xs
by (induct xs) (auto split: if-split-asm)

lemma in-set-butlast-appendI:
x ∈ set (butlast xs) ∨ x ∈ set (butlast ys)
shows x ∈ set (butlast (xs @ ys))
by (auto dest: in-set-butlastD simp add: butlast-append)

lemma last-drop [simp]:
n < length xs ⟹ last (drop n xs) = last xs
by (induct xs arbitrary: n) (auto simp add: neq-Nil-conv)

lemma butlast-conv-take:
butlast xs = take (length xs - 1) xs
by (induction xs rule: induct-list012) simp-all

lemma last-list-update:
x ≠ [] ⟹ last (xs[k:=x]) = (if k = size xs - 1 then x else last xs)
by (auto simp: last-cone-nth)

lemma butlast-list-update:
butlast(xs[k:=x]) =
(if k = size xs - 1 then butlast xs else (butlast xs)[k:=x])
by (cases xs rule: rev-cases) (auto simp: list-update-append split: nat.splits)

lemma last-map: xs ≠ [] ⟹ last (map f xs) = f (last xs)
by (cases xs rule: rev-cases) simp-all

lemma map-butlast: map f (butlast xs) = butlast (map f xs)
by (induct xs) simp-all
lemma snoc-eq-iff-butlast:
\[
xs \circ [x] = ys \iff (ys \neq [] \land \text{butlast } ys = xs \land \text{last } ys = x)
\]
by fastforce

corollary longest-common-suffix:
\[
\exists ss \, xs' \, ys'. \, xs = xs' \circ ss \land \text{ys} = ys' \circ ss \\
\land (xs' \neq [] \lor ys' \neq [] \lor \text{last } xs' \neq \text{last } ys')
\]
using longest-common-prefix[of rev xs rev ys]
unfolding rev-swap rev-append by (metis last-rev rev-is-Nil-conv)

lemma butlast-rev [simp]: \text{butlast} (rev xs) = rev (tl xs)
by (cases xs) simp-all

66.1.14 \text{take and drop}

lemma take-0: \text{take} 0 xs = []
by (induct xs) auto

lemma drop-0: \text{drop} 0 xs = xs
by (induct xs) auto

lemma take0[simp]: \text{take} 0 = (\lambda xs. [])
by (rule ext) (rule take-0)

lemma drop0[simp]: \text{drop} 0 = (\lambda x. x)
by (rule ext) (rule drop-0)

lemma take-Suc-Cons [simp]: \text{take} (Suc n) (x \# xs) = x \# \text{take} n xs
by simp

lemma drop-Suc-Cons [simp]: \text{drop} (Suc n) (x \# xs) = \text{drop} n xs
by simp

declare take-Cons [simp del] and drop-Cons [simp del]

lemma take-Suc: xs \neq [] \Longrightarrow \text{take} (Suc n) xs = \text{hd} xs \# \text{take} n (tl xs)
by (clarsimp simp add: neq-Nil-conv)

lemma drop-Suc: \text{drop} (Suc n) xs = \text{drop} n (tl xs)
by (cases xs, simp-all)

lemma hd-take[simp]: j > 0 \Longrightarrow \text{hd} (\text{take} j xs) = \text{hd} xs
by (metis gr0-cone-Suc list.sel(1) take.simps(1) take-Suc)

lemma take-tl: \text{take} n (tl xs) = tl (\text{take} (Suc n) xs)
by (induct xs arbitrary: n) simp-all

lemma drop-tl: \text{drop} n (tl xs) = tl (\text{drop} n xs)
by (induct xs arbitrary: n, simp-add: drop-Cons drop-Suc split: nat.split)

lemma tl-take: tl (take n xs) = take (n - 1) (tl xs)
  by (cases n, simp, cases xs, auto)

lemma tl-drop: tl (drop n xs) = drop n (tl xs)
  by (simp only: drop-tl)

lemma nth-via-drop: drop n xs = y#ys ==> xs!n = y
  by (induct xs arbitrary: n, simp-auto simp: drop-Cons nth-Cons split: nat.split)

lemma take-Suc-conv-app-nth:
  i < length xs ==> take (Suc i) xs = take i xs @ [xs!i]
proof (induct xs arbitrary: i)
  case Nil
  then show ?case by simp
next
  case Cons
  then show ?case by (cases i) auto
qed

lemma Cons-nth-drop-Suc:
  i < length xs ==> (xs!i) # (drop (Suc i) xs) = drop i xs
proof (induct xs arbitrary: i)
  case Nil
  then show ?case by simp
next
  case Cons
  then show ?case by (cases i) auto
qed

lemma length-take [simp]: length (take n xs) = min (length xs) n
  by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma length-drop [simp]: length (drop n xs) = (length xs - n)
  by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma take-all [simp]: length xs <= n ==> take n xs = xs
  by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma drop-all [simp]: length xs <= n ==> drop n xs = []
  by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma take-all-iff [simp]: take n xs = xs <= length xs <= n
  by (metis length-take min.order-iff take-all)

lemma take-eq-Nil[simp]: (take n xs = []) = (n = 0 ∨ xs = [])
by (induct xs arbitrary: n) (auto simp: take-Cons split: nat.split)

lemmas take-eq-Nil2 [simp] = take-eq-Nil [THEN eq_iff_swap]

lemmas drop-eq-Nil2 [simp] = drop-eq-Nil [THEN eq_iff_swap]

lemma take-append [simp]:
  take n (xs @ ys) = (take n xs @ take (n - length xs) ys)
by (induct n arbitrary: xs) (auto, case_tac xs, auto)

lemma drop-append [simp]:
  drop n (xs @ ys) = drop n xs @ drop (n - length xs) ys
by (induct n arbitrary: xs) (auto, case_tac xs, auto)

lemma take-take [simp]:
  take n (take m xs) = take (min n m) xs
proof (induct m arbitrary: xs n)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs; cases n) simp_all
qed

lemma drop-drop [simp]:
  drop n (drop m xs) = drop (n + m) xs
proof (induct m arbitrary: xs)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs) simp_all
qed

lemma take-drop: take n (drop m xs) = drop m (take (n + m) x)
proof (induct m arbitrary: xs n)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs; cases n) simp_all
qed

lemma drop-take: drop n (take m xs) = take (m - n) (drop n xs)
by (induct xs arbitrary: m n) (auto simp: take-Cons drop-Cons split: nat.split)

lemma append-take-drop-id [simp]:
  take n xs @ drop n xs = xs
proof (induct n arbitrary: xs)
lemma take-map: take n (map f xs) = map f (take n xs)
proof (induct n arbitrary: xs)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs) simp-all
qed

lemma drop-map: drop n (map f xs) = map f (drop n xs)
proof (induct n arbitrary: xs)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs) simp-all
qed

lemma rev-take: rev (take i xs) = drop (length xs - i) (rev xs)
proof (induct xs arbitrary: i)
  case Nil
  then show ?case by simp
next
  case Cons
  then show ?case by (cases i) auto
qed

lemma rev-drop: rev (drop i xs) = take (length xs - i) (rev xs)
proof (induct xs arbitrary: i)
  case Nil
  then show ?case by simp
next
  case Cons
  then show ?case by (cases i) auto
qed

lemma drop-rev: drop n (rev xs) = rev (take (length xs - n) xs)
  by (cases length xs < n) (auto simp: rev-take)

lemma take-rev: take n (rev xs) = rev (drop (length xs - n) xs)
  by (cases length xs < n) (auto simp: rev-drop)
lemma nth-take [simp]: \( i < n \Rightarrow (\text{take } n \, xs)!i = xs!i \)
proof (induct xs arbitrary: \( i \, n \))
  case Nil
  then show \(?case by simp\)
next
  case Cons
  then show \(?case by (cases n; cases i) simp-all\)
qed

lemma nth-drop [simp]:
\( n \leq \text{length } xs \Rightarrow (\text{drop } n \, xs)!i = xs!(n + i) \)
proof (induct \( n \) arbitrary: \( xs \))
  case 0
  then show \(?case by simp\)
next
  case Suc
  then show \(?case by (cases \( xs \)) simp-all\)\)
 qed

lemma butlast-take:
\( n \leq \text{length } xs \Rightarrow \text{butlast} (\text{take } n \, xs) = \text{take } (n - 1) \, xs \)
by (simp add: butlast-conv-take)

lemma butlast-drop: \( \text{butlast} (\text{drop } n \, xs) = \text{drop } n \, (\text{butlast } xs) \)
by (simp add: butlast-conv-take drop-take ac-simps)

lemma take-butlast: \( n < \text{length } xs \Rightarrow \text{take } n \, (\text{butlast } xs) = \text{take } n \, xs \)
by (simp add: butlast-conv-take)

lemma drop-butlast: \( \text{drop } n \, (\text{butlast } xs) = \text{butlast} (\text{drop } n \, xs) \)
by (simp add: butlast-conv-take drop-take ac-simps)

lemma butlast-power: \( (\text{butlast} ^^ n) \, xs = \text{take} (\text{length } xs - n) \, xs \)
by (induct \( n \)) (auto simp: butlast-take)

lemma hd-drop-conv-nth: \( n < \text{length } xs \Rightarrow \text{hd}(\text{drop } n \, xs) = xs!n \)
by (simp add: hd-cone-nth)

lemma set-take-subset-set-take:
\( m \leq n \Rightarrow \text{set} (\text{take } m \, xs) \subseteq \text{set} (\text{take } n \, xs) \)
proof (induct \( xs \) arbitrary: \( m \, n \))
  case (Cons \( x \, m \, n \) \) then show \(?case by (cases \( n \)) (auto simp: take-Cons)\)
qed simp

lemma set-take-subset: \( \text{set} (\text{take } n \, xs) \subseteq \text{set} \, xs \)
by (induct \( xs \) arbitrary: \( n \))(auto simp: take-Cons split:nat.split)

lemma set-drop-subset: \( \text{set} (\text{drop } n \, xs) \subseteq \text{set} \, xs \)
by \((\text{induct } xs \text{ arbitrary: } n) (\text{auto simp: drop-Cons split: nat.split})\)

**Lemma** set-drop-subset-set-drop:
\(m \geq n \implies \text{set}(\text{drop } m \; xs) \subseteq \text{set}(\text{drop } n \; xs)\)

**Proof** (\(\text{induct } xs \text{ arbitrary: } m \; n\))

**Case** (\(\text{Cons } x \; xs \; m \; n\))

**Then show** ?case

by (\(\text{clarsimp simp: drop-Cons split: nat.split}\) ) (\(\text{metis set-drop-subset subset-iff}\) )

qed simp

**Lemma** in-set-takeD:
\(x \in \text{set}(\text{take } n \; xs) = \implies x \in \text{set} \; xs\)

using set-take-subset by fast

**Lemma** in-set-dropD:
\(x \in \text{set}(\text{drop } n \; xs) = \implies x \in \text{set} \; xs\)

using set-drop-subset by fast

**Lemma** append-eq-conv-conj:
\((xs @ ys = zs) = (xs = \text{take} (\text{length } xs) \; zs \land ys = \text{drop} (\text{length } xs) \; zs)\)

**Proof** (\(\text{induct } xs \text{ arbitrary: } zs\))

**Case** (\(\text{Cons } x \; xs \; zs\))

**Then show** ?case

by (\(\text{cases } zs, \auto\) )

qed auto

**Lemma** map-eq-append-conv:
\(\text{map } f \; xs = ys @ zs \iff (\exists \; us \; vs. \; xs = us @ vs \land ys = \text{map } f \; us \land zs = \text{map } f \; vs)\)

**Proof** –

**Have** \(\text{map } f \; xs \neq ys @ zs \land \text{map } f \; xs \neq ys @ zs \lor \text{map } f \; xs \neq ys @ zs \lor \text{map } f \; xs = ys @ zs \land \)

\((\exists bs \; bsa. \; xs = bs @ bsa \land ys = \text{map } f \; bs \land zs = \text{map } f \; bsa)\)

by (\(\text{metis append-eq-conj append-take-drop-id drop-map take-map}\) )

**Then show** ?thesis

using map-append by blast

qed

**Lemmas** append-eq-map-conv = map-eq-append-conv[THEN eq-iff-swap]

**Lemma** take-add:
\(\text{take } (i+j) \; xs = \text{take } i \; xs \; @ \; \text{take } j \; (\text{drop } i \; xs)\)

**Proof** (\(\text{induct } xs \text{ arbitrary: } i\))

**Case** (\(\text{Cons } x \; xs \; i\))

**Then show** ?case

by (\(\text{cases } i, \auto\) )

qed auto

**Lemma** append-eq-append-conv-if:
\((xs_1 @ xs_2 = ys_1 @ ys_2) = \)

\((\text{if } \text{size } xs_1 \leq \text{size } ys_1 \; \text{then } \text{take } (\text{size } xs_1) \; ys_1 \land xs_2 = \text{drop} (\text{size } xs_1) \; ys_1 @ ys_2 \; \text{else } \text{take } (\text{size } ys_1) \; xs_1 = ys_1 \land \text{drop} (\text{size } ys_1) \; xs_1 @ xs_2 = ys_2)\)

**Proof** (\(\text{induct } xs_1 \text{ arbitrary: } ys_1\))
THEORY “List”

case (Cons a xs y₁) then show ?case
  by (cases y₁, auto)
qed auto

lemma take-hd-drop:
  n < length xs \implies take n xs @ [hd (drop n xs)] = take (Suc n) xs
by (induct xs arbitrary: n) (simp-all add: drop-Cons split: nat.split)

lemma id-take-nth-drop:
i < length xs \implies xs = take i xs @ xs![i # drop (Suc i) xs
proof –
  assume si: i < length xs
  hence xs = take (Suc i) xs @ drop (Suc i) xs by auto
moreover
  from si have take (Suc i) xs = take i xs @ [xs!i]
    using take-Suc-conv-app-nth by blast
ultimately show ?thesis by auto
qed

lemma take-update-cancel[simp]: n ≤ m \implies take n (xs[m := y]) = take n xs
by (simp add: list-eq-iff-nth-eq)

lemma drop-update-cancel[simp]: n < m \implies drop m (xs[n := x]) = drop m xs
by (simp add: list-eq-iff-nth-eq)

lemma upd-conv-take-nth-drop:
i < length xs \implies xs[i:=a] = take i xs @ a # drop (Suc i) xs
proof –
  assume i: i < length xs
  have xs[i:=a] = (take i xs @ xs![i # drop (Suc i) xs[i:=a]
    by (rule arg-cong[OF id-take-nth-drop[OF i]])
  also have ... = take i xs @ a # drop (Suc i) xs
    using i by (simp add: list-update-append)
  finally show ?thesis .
qed

lemma take-update-swap: take m (xs[n := x]) = (take m xs)[n := x]
proof (cases n ≥ length xs)
  case False
  then show ?thesis
    by (simp add: upd-conv-take-nth-drop take-Cons drop-take min-def diff-Suc split: nat.split)
qed auto

lemma drop-update-swap:
  assumes m ≤ n shows drop m (xs[n := x]) = (drop m xs)[n - m := x]
proof (cases n ≥ length xs)
  case False
    with assms show ?thesis
by (simp add: upd-conv-take-nth-drop drop-take)

qed auto

lemma nth-image: \( l \leq \text{size} \, xs \implies \text{nth} \, xs \, \{0..<l\} = \text{set}(\text{take} \, l \, xs) \)
  by (simp add: set-conv-nth) force

66.1.15 \hspace{1cm} \text{takeWhile} \textbf{ and } \text{dropWhile}

lemma length-takeWhile-le: \( \text{length} \, (\text{takeWhile} \, P \, xs) \leq \text{length} \, xs \)
  by (induct xs) auto

lemma takeWhile-dropWhile-id [simp]: \( \text{takeWhile} \, P \, xs @ \text{dropWhile} \, P \, P \, xs = xs \)
  by (induct xs) auto

lemma takeWhile-append1 [simp]:
  \( \forall x \in \text{set} \, xs. \neg P(x) \implies \text{takeWhile} \, P \, (xs @ ys) = \text{takeWhile} \, P \, xs \)
  by (induct xs) auto

lemma takeWhile-append2 [simp]:
  \( (\forall x. x \in \text{set} \, xs \implies P \, x) \implies \text{takeWhile} \, P \, (xs @ ys) = xs @ \text{takeWhile} \, P \, ys \)
  by (induct xs) auto

lemma takeWhile-append:
  \( \text{takeWhile} \, P \, (xs @ ys) = (\text{if} \ \forall x \in \text{set} \, xs. P \, x \ \text{then} \, xs @ \text{takeWhile} \, P \, ys \ \text{else} \, \text{takeWhile} \, P \, xs) \)
  using takeWhile-append1[of - xs P ys] takeWhile-append2[of xs P ys] by auto

lemma takeWhile-tail: \( \neg P \, x \implies \text{takeWhile} \, P \, (xs @ (x\#l)) = \text{takeWhile} \, P \, xs \)
  by (induct xs) auto

lemma takeWhile-eq-Nil-iff: \( \text{takeWhile} \, P \, xs = [] \iff xs = [] \lor \neg P \, (\text{hd} \, xs) \)
  by (cases xs) auto

lemma takeWhile-nth: \( j < \text{length} \, (\text{takeWhile} \, P \, xs) \implies \text{takeWhile} \, P \, xs ! j = xs ! j \)
  by (metis nth-append takeWhile-dropWhile-id)

lemma takeWhile-takeWhile: \( \text{takeWhile} \, Q \, (\text{takeWhile} \, P \, xs) = \text{takeWhile} \, (\lambda x. P \, x \land Q \, x) \, xs \)
  by(induct xs) simp-all

lemma dropWhile-nth: \( j < \text{length} \, (\text{dropWhile} \, P \, xs) \implies \text{dropWhile} \, P \, xs ! j = xs ! (j + \text{length} \, (\text{takeWhile} \, P \, xs)) \)
  by (metis add.commute nth-append-length-plus takeWhile-dropWhile-id)

lemma length-dropWhile-le: \( \text{length} \, (\text{dropWhile} \, P \, xs) \leq \text{length} \, xs \)
  by (induct xs) auto

lemma dropWhile-append1 [simp]:
\[ x \in \text{set } xs; \neg P(x) \implies \text{dropWhile } P (xs @ ys) = (\text{dropWhile } P xs)@ys \]
by (induct xs) auto

**lemma** dropWhile-append2 [simp]:
\[(\forall x. x \in \text{set } xs \implies P(x)) \implies \text{dropWhile } P (xs @ ys) = \text{dropWhile } P ys \]
by (induct xs) auto

**lemma** dropWhile-append3:
\[\neg P y \implies \text{dropWhile } P (xs @ y @ ys) = \text{dropWhile } P ys \]
by (induct xs) auto

**lemma** dropWhile-append:
\[\text{dropWhile } P (xs @ ys) = (\text{if } \forall x \in \text{set } xs. P x \text{ then } \text{dropWhile } P ys \text{ else } \text{dropWhile } P xs @ ys) \]
using dropWhile-append1[of - xs P ys] dropWhile-append2[of xs P ys] by auto

**lemma** dropWhile-last:
\[x \in \text{set } xs \implies \neg P x \implies \text{last } (\text{dropWhile } P xs) = \text{last } xs \]
by (auto simp add: dropWhile-append3 in-set-conv-decomp)

**lemma** set-dropWhileD: \[x \in \text{set } (\text{dropWhile } P xs) \implies x \in \text{set } xs \]
by (induct xs) (auto split: if-split-asm)

**lemma** set-takeWhileD: \[x \in \text{set } (\text{takeWhile } P xs) \implies x \in \text{set } xs \wedge P x \]
by (induct xs) (auto split: if-split-asm)

**lemma** takeWhile-eq-all-conv [simp]:
\[(\text{takeWhile } P xs = xs) = (\forall x \in \text{set } xs. P x) \]
by (induct xs, auto)

**lemma** dropWhile-eq-Nil-conv [simp]:
\[(\text{dropWhile } P xs = []) = (\forall x \in \text{set } xs. P x) \]
by (induct xs, auto)

**lemma** dropWhile-eq-Cons-conv:
\[(\text{dropWhile } P xs = y @ ys) = (xs = \text{takeWhile } P xs @ y @ ys \wedge \neg P y) \]
by (induct xs, auto)

**lemma** dropWhile-eq-self-iff: \[\text{dropWhile } P xs = xs \iff xs = [] \lor \neg P (hd xs) \]
by (cases xs) (auto simp: dropWhile-eq-Cons-conv)

**lemma** dropWhile-dropWhile1: \[(\forall x. Q x \implies P x) \implies \text{dropWhile } Q (\text{dropWhile } P xs) = \text{dropWhile } P xs \]
by (induct xs) simp-all

**lemma** dropWhile-dropWhile2: \[(\forall x. P x \implies Q x) \implies \text{takeWhile } P (\text{takeWhile } Q xs) = \text{takeWhile } P xs \]
by (induct xs) simp-all
lemma dropWhile-takeWhile:
\((\forall x. P x \implies Q x) \implies \text{dropWhile } P \ (\text{takeWhile } Q \ x) = \text{takeWhile } Q \ (\text{dropWhile } P \ x)\)
by (induction x) auto

lemma distinct-takeWhile[simp]: \text{distinct } x = \implies \text{distinct } (\text{takeWhile } P \ x)
by (induct x) (auto dest: set-takeWhileD)

lemma distinct-dropWhile[simp]: \text{distinct } x = \implies \text{distinct } (\text{dropWhile } P \ x)
by (induct x) auto

lemma takeWhile-map: \text{takeWhile } P \ (\text{map } f \ x) = \text{map } f \ (\text{takeWhile } (P \circ f) \ x)
by (induct x) auto

lemma dropWhile-map: \text{dropWhile } P \ (\text{map } f \ x) = \text{map } f \ (\text{dropWhile } (P \circ f) \ x)
by (induct x) auto

lemma takeWhile-eq-take: \text{takeWhile } P \ x = \text{take } (\text{length } (\text{takeWhile } P \ x)) \ x
by (induct x) auto

lemma dropWhile-eq-drop: \text{dropWhile } P \ x = \text{drop } (\text{length } (\text{takeWhile } P \ x)) \ x
by (induct x) auto

lemma hd-dropWhile: \text{dropWhile } P \ x \neq [] \implies \neg P \ (\text{hd } (\text{dropWhile } P \ x))
by (induct x) auto

lemma takeWhile-eq-filter:
assumes \(\forall x. x \in \text{set } (\text{dropWhile } P \ x) \implies \neg P x\)
shows \(\text{takeWhile } P \ x = \text{filter } P \ x\)
proof
have A: \text{filter } P \ x = \text{filter } P \ (\text{takeWhile } P \ x @ \text{dropWhile } P \ x)
by simp
have B: \text{filter } P \ (\text{dropWhile } P \ x) = []
unfolding \text{filter-empty-conv using assms by blast}
have \text{filter } P \ x = \text{takeWhile } P \ x
unfolding A filter-append B
by (auto simp add: filter-id-conv dest: set-takeWhileD)
thus \?thesis ..
qed

lemma takeWhile-eq-take-P-nth:
\([\forall i. [i < n \implies P (xs ! i) \implies n < length x \implies \neg P (xs ! n)]\] \implies \text{takeWhile } P \ x = \text{take } n \ x
proof (induct x arbitrary: n)
case Nil
thus \?case by simp
next
case (Cons x x)
show ?case
proof (cases n)
  case 0
  with Cons show ?thesis by simp
next
  case [simp]: (Suc n)
  have $P \cdot x$ using Cons.prems(1)[of 0] by simp
  moreover have $\text{takeWhile } P \cdot xs = \text{take } n' \cdot xs$
  proof (rule Cons.hyps)
    fix $i$
    assume $i < n' \cdot i < \text{length } xs$
    thus $P \cdot (xs ! i)$ using Cons.prems(1)[of Suc $i$] by simp
  next
    assume $n' < \text{length } xs$
    thus $\neg P \cdot (xs ! n')$ using Cons by auto
  qed
  ultimately show ?thesis by simp
qed

lemma nth-length-takeWhile:
  $\text{length } (\text{takeWhile } P \cdot xs) < \text{length } xs \Rightarrow \neg (xs!\text{length } (\text{takeWhile } P \cdot xs))$
by (induct xs) auto

lemma length-takeWhile-less-P-nth:
  assumes all: $\forall i. \ i < j \Rightarrow P \cdot (xs!i)$ and $j \leq \text{length } xs$
  shows $j \leq \text{length } (\text{takeWhile } P \cdot xs)$
proof (rule classical)
  assume $\neg ?\text{thesis}$
  hence $\text{length } (\text{takeWhile } P \cdot xs) < \text{length } xs$ using assms by simp
  thus $?\text{thesis}$ using all ($\neg ?\text{thesis}$) nth-length-takeWhile[of P xs] by auto
qed

Lemma takeWhile-neq-rev: $\left[distinct \cdot xs; \ x \in \text{set } xs\right] \Rightarrow \text{takeWhile } (\lambda y. y \neq x) \cdot (\text{rev } xs) = \text{rev } (\text{tl } (\text{dropWhile } (\lambda y. y \neq x) \cdot xs))$
by (induct xs) (auto simp: takeWhile-tail[where l=[]])

lemma dropWhile-neq-rev: $\left[distinct \cdot xs; \ x \in \text{set } xs\right] \Rightarrow \text{dropWhile } (\lambda y. y \neq x) \cdot (\text{rev } xs) = x \# \text{rev } (\text{takeWhile } (\lambda y. y \neq x) \cdot xs)$
proof (induct xs)
  case (Cons a xs)
  then show ?case
  by (auto, subst dropWhile-append2, auto)
qed simp

lemma takeWhile-not-last:
  $\text{distinct } xs \Rightarrow \text{takeWhile } (\lambda y. y \neq last xs) \cdot xs = \text{butlast } xs$
by (induction xs rule: induct-list012) auto
lemma takeWhile-cong [fundef-cong]:
\[ l = k; \forall x. x \in \text{set } l \implies P x = Q x \]
\implies \text{takeWhile } P \ l = \text{takeWhile } Q \ k

by (induct \ k \ arbitrary: \ l) (simp-all)

lemma dropWhile-cong [fundef-cong]:
\[ l = k; \forall x. x \in \text{set } l \implies P x = Q x \]
\implies \text{dropWhile } P \ l = \text{dropWhile } Q \ k

by (induct \ k \ arbitrary: \ l, simp-all)

lemma takeWhile-idem [simp]:
\[ \text{takeWhile } P \ (\text{takeWhile } P \ xs) = \text{takeWhile } P \ xs \]
by (induct xs) auto

lemma dropWhile-idem [simp]:
\[ \text{dropWhile } P \ (\text{dropWhile } P \ xs) = \text{dropWhile } P \ xs \]
by (induct xs) auto

66.1.16 zip

lemma zip-Nil [simp]: zip [] ys = []
by (induct ys) auto

lemma zip-Cons-Cons [simp]: zip (x # xs) (y # ys) = (x, y) # zip xs ys
by simp

declare zip-Cons [simp del]

lemma [code]:
zip [] ys = []
zip xs [] = []
zip (x # xs) (y # ys) = (x, y) # zip xs ys
by (fact zip-nil zip.simps(1) zip-Cons-Cons)+

lemma zip-Cons1:
\[ \text{zip} \ (x\#xs) \ ys = (\text{case } ys \ \text{of} \ [ ] \Rightarrow [ ] \mid y\#ys \Rightarrow (x, y) \# \text{zip } xs \ ys) \]
by (auto split:list.split)

lemma length-zip [simp]:
\[ \text{length } (\text{zip } xs \ ys) = \text{min } (\text{length } xs) (\text{length } ys) \]
by (induct xs ys rule:list-induct2') auto

lemma zip-obtain-same-length:
\[ \forall zs \ ws. \text{length } zs = \text{length } ws \Rightarrow n = \text{min } (\text{length } xs) (\text{length } ys) \]
\Rightarrow \text{take } n \ xs = \text{take } n \ ys \Rightarrow P (\text{zip } zs \ ws)

shows P (zip xs ys)

proof –
let \ ?n = \text{min } (\text{length } xs) (\text{length } ys)
have P (zip (\text{take } ?n \ xs) (\text{take } ?n \ ys))
by (rule assms) simp-all
moreover have zip xs ys = zip (take \?n xs) (take \?n ys)
proof (induct xs arbitrary: ys)
  case Nil then show \?case by simp
next
  case (Cons x xs) then show \?case by (cases ys) simp-all
qed
ultimately show \?thesis by simp
qed

lemma zip-append1:
zip (xs @ ys) zs =
zip xs (take (length xs) zs) @ zip (drop (length xs) zs) ys
by (induct xs zs rule: list-induct2') auto

lemma zip-append2:
zip xs (ys @ zs) =
zip (take (length ys) xs) ys @ zip (drop (length ys) xs) zs
by (induct xs ys rule: list-induct2') auto

lemma zip-append [simp]:
[length xs = length us] \Rightarrow
zip (xs@ys) (us@vs) = zip xs us @ zip ys vs
by (simp add: zip-append1)

lemma zip-rev:
length xs = length ys \Rightarrow zip (rev xs) (rev ys) = rev (zip xs ys)
by (induct rule: list-induct2, simp-all)

lemma zip-map-map:
zip (map f xs) (map g ys) = map (\lambda (x, y). (f x, g y)) (zip xs ys)
proof (induct xs arbitrary: ys)
  case (Cons x xs) note Cons-x-xs = Cons.hyps
  show \?case
  proof (cases ys)
    case (Cons y ys')
    show \?thesis unfolding Cons using Cons-x-xs by simp
  qed simp
qed simp

lemma zip-map1:
zip (map f xs) ys = map (\lambda (x, y). (f x, y)) (zip xs ys)
using zip-map-map[of f xs \lambda x. x ys] by simp

lemma zip-map2:
zip xs (map f ys) = map (\lambda (x, y). (x, f y)) (zip xs ys)
using zip-map-map[of \lambda x. x xs f ys] by simp

lemma map-zip-map:
map \ f \ (\text{zip} \ (\text{map} \ g \ \text{xs}) \ \text{ys}) = \text{map} \ (%(x,y). f(x, y)) \ (\text{zip} \ \text{xs} \ \text{ys}) \\
\text{by} \ (\text{auto simp: zip-map1})

\text{lemma \ map-zip-map2:} \\
map \ f \ (\text{zip} \ \text{xs} \ (\text{map} \ g \ \text{ys})) = \text{map} \ (%(x,y). f(x, y)) \ (\text{zip} \ \text{xs} \ \text{ys}) \\
\text{by} \ (\text{auto simp: zip-map2})

\text{Courtesy of Andreas Lochbihler:}

\text{lemma \ zip-same-conv-map:} \ \text{zip} \ \text{xs} \ \text{xs} = \text{map} \ (\lambda x. (x, x)) \ \text{xs} \\
\text{by} \ (\text{induct \ xs}) \ \text{auto}

\text{lemma \ nth-zip \ [simp]:} \\
[i < \text{length} \ \text{xs}; \ i < \text{length} \ \text{ys}] \Rightarrow (\text{zip} \ \text{xs} \ \text{ys})!i = (\text{xs}!i, \ \text{ys}!i) \\
\text{proof} \ (\text{induct} \ \text{ys} \ \text{arbitrary:} \ i \ \text{xs}) \\
\text{case \ (Cons} \ y \ \text{ys}) \\
\text{then show ?case} \\
\text{by} \ (\text{cases} \ \text{xs}; \ \text{auto}) \ (\text{simp-all add: nth.simps split: nat.split}) \ \text{qed \ auto}

\text{lemma \ set-zip:} \\
\text{set} \ (\text{zip} \ \text{xs} \ \text{ys}) = \{ (\text{x}!i, \ \text{ys}!i) \mid i. \ i < \text{min} \ (\text{length} \ \text{xs}) \ (\text{length} \ \text{ys}) \} \\
\text{by} \ (\text{simp add: set-conv-nth cong: rev-conj-cong})

\text{lemma \ zip-same:} \ ((a, b) \in \text{set} \ (\text{zip} \ \text{xs} \ \text{xs})) = (a \in \text{set} \ \text{xs} \ \land \ a = b) \\
\text{by} \ (\text{induct \ xs}) \ \text{auto}

\text{lemma \ zip-update:} \ \text{zip} \ (\text{xs}[i:=x]) \ (\text{ys}[i:=y]) = (\text{zip} \ \text{xs} \ \text{ys})[i:=(x,y)] \\
\text{by} \ (\text{simp add: update-zip})

\text{lemma \ zip-replicate \ [simp]:} \\
\text{zip} \ (\text{replicate} \ i \ \text{x}) \ (\text{replicate} \ j \ \text{y}) = \text{replicate} \ (\text{min} \ i \ j) \ (x,y) \\
\text{proof} \ (\text{induct} \ i \ \text{arbitrary:} \ j) \\
\text{case \ Suc} \ i \ \\
\text{then show ?case} \\
\text{by} \ (\text{cases} \ j; \ \text{auto}) \ \text{qed \ auto}

\text{lemma \ zip-replicate1:} \ \text{zip} \ (\text{replicate} \ n \ \text{x}) \ \text{ys} = \text{map} \ (\text{Pair} \ x) \ (\text{take} \ n \ \text{ys}) \\
\text{by} \ (\text{induction} \ \text{ys} \ \text{arbitrary:} \ n)(\text{case-tac \ [2] } \ n, \ \text{simp-all})

\text{lemma \ take-zip:} \ \text{take} \ n \ (\text{zip} \ \text{xs} \ \text{ys}) = \text{zip} \ (\text{take} \ n \ \text{xs}) \ (\text{take} \ n \ \text{ys}) \\
\text{proof} \ (\text{induct} \ n \ \text{arbitrary:} \ \text{xs} \ \text{ys}) \\
\text{case 0} \\
\text{then show ?case by simp} \\
\text{next} \\
\text{case \ Suc} \\
\text{then show ?case by (cases \ xs; \ cases \ ys) simp-all} \ \text{qed}
lemma drop-zip: \( \text{drop } n \ (\text{zip } xs \ ys) = \text{zip } (\text{drop } n \ xs) \ (\text{drop } n \ ys) \)
proof (induct \( n \) arbitrary: \( xs \ ys \))
  case 0
  then show \(?case\) by simp
next
  case \( \text{Suc} \)
  then show \(?case\) by (cases \( xs \); cases \( ys \)) simp-all
qed

lemma zip-takeWhile-fst: \( \text{zip } (\text{takeWhile } P \ xs) \ ys = \text{takeWhile } (P \circ \text{fst}) \ (\text{zip } xs \ ys) \)
proof (induct \( xs \) arbitrary: \( ys \))
  case \( \text{Nil} \)
  then show \(?case\) by simp
next
  case \( \text{Cons} \)
  then show \(?case\) by (cases \( ys \)) auto
qed

lemma zip-takeWhile-snd: \( \text{zip } xs \ (\text{takeWhile } P \ ys) = \text{takeWhile } (P \circ \text{snd}) \ (\text{zip } xs \ ys) \)
proof (induct \( xs \) arbitrary: \( ys \))
  case \( \text{Nil} \)
  then show \(?case\) by simp
next
  case \( \text{Cons} \)
  then show \(?case\) by (cases \( ys \)) auto
qed

lemma set-zip-leftD: \( (x, y) \in \text{set } (\text{zip } xs \ ys) \implies x \in \text{set } xs \)
  by (induct \( xs \ ys \) rule: list-induct2) auto

lemma set-zip-rightD: \( (x, y) \in \text{set } (\text{zip } xs \ ys) \implies y \in \text{set } ys \)
  by (induct \( xs \ ys \) rule: list-induct2) auto

lemma in-set-zipE:
  \( (x, y) \in \text{set } (\text{zip } xs \ ys) \implies \left( \exists \ n. \ x = \text{fst } p \land y = \text{snd } p \land n = \text{length } xs \land n < \text{length } ys \land [x, y] \in R \right) \implies R \)
  by (blast dest: set-zip-leftD set-zip-rightD)

lemma zip-map-fst-snd: \( \text{zip } (\text{map } \text{fst} \ zs) \ (\text{map } \text{snd} \ zs) = zs \)
  by (induct \( zs \)) simp-all

lemma zip-eq-conv:
  \( \text{length } xs = \text{length } ys \implies \text{zip } xs \ ys = zs \iff \text{map } \text{fst} \ zs = xs \land \text{map } \text{snd} \ zs = ys \)
  by (auto simp add: zip-map-fst-snd)

lemma in-set-zip:
  \( p \in \text{set } (\text{zip } xs \ ys) \iff \exists \ n. \ x = \text{fst } p \land y = \text{snd } p \land n = \text{length } xs \land n < \text{length } ys \)
by (cases p) (auto simp add: set-zip)

lemma in-set-impl-in-set-zip1:
  assumes length xs = length ys
  assumes x ∈ set xs
  obtains y where (x, y) ∈ set (zip xs ys)
proof –
  from assms have x ∈ set (map fst (zip xs ys)) by simp
  from this that show ?thesis by fastforce
qed

lemma in-set-impl-in-set-zip2:
  assumes length xs = length ys
  assumes y ∈ set ys
  obtains x where (x, y) ∈ set (zip xs ys)
proof –
  from assms have y ∈ set (map snd (zip xs ys)) by simp
  from this that show ?thesis by fastforce
qed

lemma zip-eq-Nil-iff [simp]:
  zip xs ys = [] ↔ xs = [] ∨ ys = []
by (cases xs; cases ys) simp-all

lemmas Nil-eq-zip-iff [simp] = zip-eq-Nil-iff [THEN eq-iff-swap]

lemma zip-eq-ConsE:
  assumes zip xs ys = xy # xys
  obtains x xs' y ys' where xs = x # xs'
  and ys = y # ys' and xy = (x, y)
  and xys = zip xs' ys'
proof –
  from assms have xs ≠ [] and ys ≠ []
    using zip-eq-Nil-iff [of xs ys] by simp-all
  then obtain x xs' y ys' where xs: xs = x # xs'
  and ys: ys = y # ys'
    by (cases xs; cases ys) auto
  with assms have xy = (x, y) and xys = zip xs' ys'
    by simp-all
  with xs ys show ?thesis ..
qed

lemma semilattice-map2:
  semilattice (map2 (•)) if semilattice (•)
  for f (infixl • 70)
proof –
  from that interpret semilattice f .
  show ?thesis
  proof
show \( \text{map2}(\ast)(\text{map2}(\ast)\,xs\,ys)\,zs = \text{map2}(\ast)\,xs\,(\text{map2}(\ast)\,ys\,zs) \)

for \( xs\,ys\,zs :: 'a\) list

proof (induction \( \text{zip}\,xs\,(\text{zip}\,ys\,zs)\) arbitrary: \( xs\,ys\,zs \))

\begin{description}
\item[case Nil]
from Nil \([\text{symmetric}]\) show ?case
\item[by auto]
\end{description}

next
\begin{description}
\item[case (Cons \( xyz\,xyzs)\)]
from Cons.hyps(2) \([\text{symmetric}]\) show ?case
\item[by (rule \text{zip-eq-ConsE} \,(erule \text{zip-eq-ConsE},
\text{auto intro: Cons.hyps(1) simp add: ac-simps})]
\end{description}

qed

show \( \text{map2}(\ast)\,xs\,ys = \text{map2}(\ast)\,ys\,xs \)

for \( xs\,ys :: 'a\) list

proof (induction \( \text{zip}\,xs\,ys\) arbitrary: \( xs\,ys \))

\begin{description}
\item[case Nil]
then show ?case
\item[by auto]
\end{description}

next
\begin{description}
\item[case (Cons \( xy\,xys)\)]
from Cons.hyps(2) \([\text{symmetric}]\) show ?case
\item[by (rule \text{zip-eq-ConsE} \,(auto intro: Cons.hyps(1) simp add: ac-simps)]
\end{description}

qed

show \( \text{map2}(\ast)\,xs\,xs = xs \)

for \( xs :: 'a\) list

by (induction \( xs\) simp-all)

qed

\begin{lem}
\begin{ass}
\text{map\,fst\,xs} = \text{map\,fst\,ys} \\text{and} \ \text{map\,snd\,xs} = \text{map\,snd\,ys} \\
\text{shows} \ \text{xs} = \text{ys}
\end{ass}

\begin{proof}
\begin{assms}(1) \text{have} \ \text{length\,xs} = \text{length\,ys} \text{by (rule map-eq-imp-length-eq)}
\begin{assms}
\item[from this \text{assms} show ?thesis]
\item[by (induct \( xs\,ys\) \text{rule: list-induct2} \,(simp-all add: prod-eqI)]
\end{assms}
\end{assms}
\end{proof}
\end{lem}

\begin{lem}
<\text{hd\,(zip\,xs\,ys)} = <\text{(hd\,xs,\,hd\,ys)}> \item[if <\text{xs} \neq []> \text{and} <\text{ys} \neq []>]
\begin{using}
\item[by (cases \( xs\); cases \( ys\) \,simp-all)]
\end{using}
\end{lem}

\begin{lem}
<\text{last\,(zip\,xs\,ys)} = <\text{(last\,xs,\,last\,ys)}> \item[if <\text{xs} \neq []> \text{and} <\text{ys} \neq []>]
\begin{and}
\item[\text{length\,xs} = \text{length\,ys}]
\item[\text{using} \,by (cases \( xs\) \text{rule: rev-cases}; cases \( ys\) \text{rule: rev-cases} \,simp-all]
\end{and}
\end{lem}
THEORY “List” 1315

66.1.17  list-all2

lemma list-all2-lengthD [intro?]:
  list-all2 P xs ys ⇒ length xs = length ys
  by (simp add: list-all2-iff)

lemma list-all2-Nil [iff, code]: list-all2 P [] ys = (ys = [])
  by (simp add: list-all2-iff)

lemma list-all2-Nil2 [iff, code]: list-all2 P xs [] = (xs = [])
  by (simp add: list-all2-iff)

lemma list-all2-Cons [iff, code]:
  list-all2 P (x # xs) (y # ys) = (P x y ∧ list-all2 P xs ys)
  by (auto simp add: list-all2-iff)

lemma list-all2-Cons1:
  list-all2 P (x # xs) ys = (∃ zs. ys = z # zs ∧ P x z ∧ list-all2 P zs)
  by (cases ys) auto

lemma list-all2-Cons2:
  list-all2 P xs (y # ys) = (∃ zs. xs = z # zs ∧ P z y ∧ list-all2 P zs)
  by (cases xs) auto

lemma list-all2-induct [consumes 1, case-names Nil Cons, induct set: list-all2]:
  assumes P: list-all2 P xs ys
  assumes Nil: R [] []
  assumes Cons: ∀ x xs y ys. [P x y; list-all2 P xs ys; R xs ys] ⇒ R (x # xs) (y # ys)
  shows R xs ys
  using P
  by (induct xs arbitrary: ys) (auto simp add: list-all2-Cons1 Nil Cons)

lemma list-all2-rev [iff]:
  list-all2 P (rev xs) (rev ys) = list-all2 P xs ys
  by (simp add: list-all2-iff zip-rev cong: conj-cong)

lemma list-all2-rev1:
  list-all2 P (rev xs) ys = list-all2 P xs (rev ys)
  by (subst list-all2-rev [symmetric]) simp

lemma list-all2-append1:
  list-all2 P (xs @ ys) zs =
  (∃ us vs. zs = us @ vs ∧ length us = length xs ∧ length vs = length ys ∧
  list-all2 P xs us ∧ list-all2 P ys vs) (lhs = rhs)
  proof
  assume ?lhs
  then show ?rhs
    apply (rule-tac x = take (length xs) zs in ext)

apply (rule-tac x = drop (length xs) zs in exI)
apply (force split: nat-diff-split simp add: list-all2-iff zip-append1)
done

next assume ?rhs then show ?lhs by (auto simp add: list-all2-iff)

qed

lemma list-all2-append2:
list-all2 P xs (ys @ zs) =
(∃ us vs xs = us @ vs ∧ length us = length ys ∧ length vs = length zs ∧
  list-all2 P us ys ∧ list-all2 P vs zs) (is ?lhs = ?rhs)

proof assume ?lhs then show ?rhs
  apply (rule-tac x = take (length ys) xs in exI)
  apply (rule-tac x = drop (length ys) xs in exI)
  apply (force split: nat-diff-split simp add: list-all2-iff zip-append2)
done

next assume ?rhs then show ?lhs by (auto simp add: list-all2-iff)

qed

lemma list-all2-append:
  length xs = length ys =⇒
  list-all2 P (xs @ us) (ys @ vs) = (list-all2 P xs ys ∧ list-all2 P us vs)
  by (induct rule: list-induct2, simp-all)

lemma list-all2-appendI [intro?, trans]:
  [ list-all2 P a b; list-all2 P c d ] =⇒ list-all2 P (a @ c) (b @ d)
  by (simp add: list-all2-append list-all2-lengthD)

lemma list-all2-conv-all-nth:
  list-all2 P xs ys =
  (length xs = length ys ∧ (∀ i < length xs. P (xs ! i) (ys ! i)))
  by (force simp add: list-all2-iff set-zip)

lemma list-all2-trans:
  assumes tr: !!a b c. P1 a b =⇒ P2 b c =⇒ P3 a c
  shows !!bs cs. list-all2 P1 as bs =⇒ list-all2 P2 bs cs =⇒ list-all2 P3 as cs
  (is !!bs cs. PROP ?Q as bs cs)
  proof (induct as)
    fix x xs bs assume I1: !!bs cs. PROP ?Q x x bs cs
    show !!cs. PROP ?Q (x ≠ x) bs cs
      proof (induct bs)
        fix y ys cs assume I2: !!cs. PROP ?Q (x ≠ x) y cs
      end
    end
  qed
THEORY “List”

show PROP \(?Q (x \# xs) (y \# ys) cs\)
  by (induct cs) (auto intro: tr I1 I2)
qed simp

lemma list-all2-all-nthI [intro?]:
  \(\text{length } a = \text{length } b \implies (\\forall n. \text{size } a < \text{size } b \implies P (a!n) (b!n)) \implies \text{list-all2 } P a b\)
  by (simp add: list-all2-conv-all-nth)

lemma list-all2I:
  \(\forall x \in \text{set } \text{zip } a b. \text{case-prod } P x \implies \text{length } a = \text{length } b \implies \text{list-all2 } P a b\)
  by (simp add: list-all2-iff)

lemma list-all2-nthD:
  \([\text{list-all2 } P xs ys; p < \text{size } xs]\] \implies P (xs!p) (ys!p)
  by (simp add: list-all2-conv-all-nth)

lemma list-all2-nthD2:
  \([\text{list-all2 } P xs ys; p < \text{size } ys]\] \implies P (xs!p) (ys!p)
  by (frule list-all2-lengthD) (auto intro: list-all2-nthD)

lemma list-all2-map1:
  \(\text{list-all2 } P (\text{map } f as) bs = \text{list-all2 } (\lambda x y. P (f x) y) as bs\)
  by (simp add: list-all2-conv-all-nth)

lemma list-all2-map2:
  \(\text{list-all2 } P as (\text{map } f bs) = \text{list-all2 } (\lambda x y. P x (f y)) as bs\)
  by (auto simp add: list-all2-conv-all-nth)

lemma list-all2-refl [intro?]::
  \(\forall x. P x x \implies \text{list-all2 } P xs xs\)
  by (simp add: list-all2-conv-all-nth)

lemma list-all2-update-cong:
  \(\forall i. P (xs y) \implies \text{list-all2 } P (xs[i:=x]) (ys[i:=y])\)
  by (cases \(i < \text{size } ys\)) (auto simp add: list-all2-conv-all-nth nth-list-update)

lemma list-all2-takeI [simp,intro?]::
  \(\text{list-all2 } P xs ys \implies \text{list-all2 } P (\text{take } n xs) (\text{take } n ys)\)
  proof (induct \(xs\) arbitrary: \(n\), \(ys\))
    case (Cons \(x\) \(xs\))
    then show ?case
      by (cases \(n\)) (auto simp: list-all2-Cons1)
  qed auto

lemma list-all2-dropI [simp,intro?]::
  \(\text{list-all2 } P xs ys \implies \text{list-all2 } P (\text{drop } n xs) (\text{drop } n ys)\)
  proof (induct \(xs\) arbitrary: \(n\), \(ys\))
    case (Cons \(x\) \(xs\))

then show \( ?\text{case} \)
by (cases \( n \)) (auto simp: list-all2-Cons1)
qed auto

lemma list-all2-mono [intro?]:
list-all2 \( P \) \( xs \) \( ys \) \( \Longrightarrow \) \( (\forall \) \( xs \) \( ys \). \( P \) \( xs \) \( ys \) \( \Longrightarrow \) \( Q \) \( xs \) \( ys \)) \( \Longrightarrow \) list-all2 \( Q \) \( xs \) \( ys \)
by (rule list.rel-mono-strong)

lemma list-all2-eq:
\( xs = ys \) \( \iff \) list-all2 \( (=) \) \( xs \) \( ys \)
by (induct \( xs \) \( ys \) rule: list-induct2′) auto

lemma list-eq-iff-zip-eq:
\( xs = ys \) \( \iff \) length \( xs \) = length \( ys \) \( \land \) \( (\forall \) \( (x, y) \in \) set \( \) (zip \( xs \) \( ys \))\( . \) \( x = y \))
by (auto simp add: set-zip list-all2-eq list-all2-conv-all-nth cong: conj-cong)

lemma list-all2-same:
list-all2 \( P \) \( xs \) \( xs \) \( \iff \) \( (\forall \) \( x \in \) set \( xs \). \( P \) \( x \) \( x \))
by (auto simp add: list-all2-conv-all-nth set-conv-nth)

lemma zip-assoc:
zip \( xs \) \( (\) zip \( ys \) \( zs \) \( )\) = map \( \lambda \) \( ((x, y), z) \). \( (x, y, z) \)) \( (\) zip \( ys \) \( zs \) \( )\) \( xs \)
by (rule list-all2-all-nthI[where \( P\)\( =\)\( (=)\), unfolded list.eq]) simp-all

lemma zip-commute:
zip \( xs \) \( ys \) = map \( \lambda \) \( (x, y) \). \( (y, x) \)) \( (\) zip \( ys \) \( xs \) \( )\)
by (rule list-all2-all-nthI[where \( P\)\( =\)\( (=)\), unfolded list.eq]) simp-all

lemma zip-left-commute:
zip \( xs \) \( (\) zip \( ys \) \( zs \) \( )\) = map \( \lambda \) \( ((y, (x, z)) , (x, y, z)) \). \( (zip \) \( ys \) \( zip \) \( zs \) \( )\) \( xs \)
by (rule list-all2-all-nthI[where \( P\)\( =\)\( (=)\), unfolded list.eq]) simp-all

lemma zip-replicate2:
zip \( xs \) \( \) (replicate \( n \) \( y \)) \( = \) map \( \lambda x \) \( (x, y) \)) \( (\) take \( n \) \( xs \) \( )\)
by (subst zip-commute)(simp add: zip-replicate1)

66.1.18 List.product and product-lists

lemma product-concat-map:
List.product \( xs \) \( ys \) = concat \( (\) map \( \lambda x \. \) map \( \lambda y \. \) \( (x, y) \)) \( ys \) \( )\) \( xs \)
by (induction \( xs \)) (simp)+

lemma set-product[simp]: set \( (\) List.product \( xs \) \( ys \) \( )\) = set \( xs \) \( \times \) set \( ys \)
by (induct \( xs \)) auto

lemma length-product \[ simp \]:
length \( (\) List.product \( xs \) \( ys \) \( )\) = length \( xs \) \( \times \) length \( ys \)
by (induct \( xs \)) simp-all

lemma product-nth:
assumes \( n < \) length \( xs \) \( \times \) length \( ys \)
shows List.product \( xs \) \( ys \) \( ! \) \( n \) = \( (xs \) \( ! \) \( (n \) div \( length \) \( ys \)), \( ys \) \( ! \) \( (n \) mod \( length \) \( ys \)) \)
using assms proof (induct xs arbitrary: n)
case Nil then show ?case by simp
next
case (Cons x xs n)
then have length ys > 0 by auto
with Cons show ?case
  by (auto simp add: nth-append not-less le-mod-geq le-div-geq)
qed

lemma in-set-product-lists-length:
xs ∈ set (product-lists xss) ⇒ length xs = length xss
by (induct xss arbitrary: xs) auto

lemma product-lists-set:
set (product-lists xss) = \{xs. list-all2 (λx ys. x ∈ set ys) xs xss\} (is ?L = Collect ?R)
proof (intro equalityI subsetI, unfold mem-Collect-eq)
fix xs assume xs ∈ ?L
then have length xs = length xss by (rule in-set-product-lists-length)
from this \{xs ∈ ?L\} show ?R xs by (induct xs xss rule: list-induct2) auto
next
fix xs assume ?R xs
then show xs ∈ ?L by induct auto
qed

66.1.19 fold with natural argument order

lemma fold-simps [code]: — eta-expanded variant for generated code — enables
tail-recursion optimisation in Scala
fold f [] s = s
fold f (x # xs) s = fold f xs (f x s)
by simp-all

lemma fold-remove1-split:
[ \[ \forall x y. x ∈ set xs ⇒ y ∈ set xs ⇒ f x ∘ f y = f y ∘ f x; 
x ∈ set xs \] ]
⇒ fold f xs = fold f (remove1 x xs) ∘ f x
by (induct xs) (auto simp add: comp-assoc)

lemma fold-cong [fundef-cong]:
a = b ⇒ xs = ys ⇒ (∀x. x ∈ set xs ⇒ f x = g x)
⇒ fold f xs a = fold g ys b
by (induct ys arbitrary: a b xs) simp-all

lemma fold-id: (∀x. x ∈ set xs ⇒ f x = id) ⇒ fold f xs = id
by (induct xs) simp-all

lemma fold-commute:
(∀x. x ∈ set xs ⇒ h ∘ g x = f x ∘ h) ⇒ h ∘ fold g xs = fold f xs ∘ h
THEORY "List"

by (induct xs) (simp-all add: fun-eq-iff)

lemma fold-commute-apply:
  assumes \( \forall x. x \in \text{set} \ xs \implies h \circ g x = f x \circ h \)
  shows \( h \ (\text{fold} \ g \ xs \ s) = \text{fold} \ f \ xs \ (h \ s) \)
proof –
  from assms have \( h \circ \text{fold} \ g \ xs = \text{fold} \ f \ xs \circ h \) by (rule fold-commute)
  then show \( \ ?thesis \) by (simp add: fun-eq-iff)
qed

lemma fold-invariant:
  \[ \[ \forall x. x \in \text{set} \ xs \implies Q x; \ P s; \ \forall x s. Q x \implies P s \implies P \ (f x \ s) \ \] \]
  \implies P \ (\text{fold} \ f \ xs \ s)
by (induct xs arbitrary: s) simp-all

lemma fold-append [simp]: \( \text{fold} \ f \ (\text{xs} @ \text{ys}) = \text{fold} \ f \ ys \circ \text{fold} \ f \ xs \)
by (induct xs) simp-all

lemma fold-map [code-unfold]: \( \text{fold} \ g \ (\text{map} \ f \ xs) = \text{fold} \ (g \circ f) \ xs \)
by (induct xs) simp-all

lemma fold-filter:
  \( \text{fold} \ f \ (\text{filter} \ P \ xs) = \text{fold} \ (\lambda x. \text{if} \ P \ x \ \text{then} \ f \ x \ \text{else} \ \text{id}) \ xs \)
by (induct xs) simp-all

lemma fold-rev:
  \( (\forall x y. x \in \text{set} \ xs \implies y \in \text{set} \ xs \implies f y \circ f x = f x \circ f y) \)
  \implies \text{fold} \ f \ (\text{rev} \ xs) = \text{fold} \ f \ xs
by (induct xs) (simp-all add: fold-commute-apply fun-eq-iff)

lemma fold-Cons-rev: \( \text{fold} \ \text{Cons} \ xs = \text{append} \ (\text{rev} \ xs) \)
by (induct xs) simp-all

lemma rev-conv-fold [code]: \( \text{rev} \ xs = \text{fold} \ \text{Cons} \ xs \ [] \)
by (simp add: fold-Cons-rev)

lemma fold-append-concat-rev: \( \text{fold} \ \text{append} \ xss = \text{append} \ (\text{concat} \ (\text{rev} \ xss)) \)
by (induct xss) simp-all

Finite-Set.fold and fold

lemma (in comp-fun-commute-on) fold-set-fold-remdups:
  assumes \( \text{set} \ xs \subseteq S \)
  shows \( \text{Finite-Set.fold} \ f \ y \ (\text{set} \ xs) = \text{fold} \ f \ (\text{remdups} \ xs) \ y \)
by (rule sym, use assms in (induct xs arbitrary: y))
  (simp-all add: insert-absorb fold-fun-left-comm)

lemma (in comp-fun-idem-on) fold-set-fold:
  assumes \( \text{set} \ xs \subseteq S \)
  shows \( \text{Finite-Set.fold} \ f \ y \ (\text{set} \ xs) = \text{fold} \ f \ xs \ y \)
by (rule sym, use assms in (induct xs arbitrary: y)) (simp-all add: fold-fun-left-comm)

lemma union-set-fold [code]: set xs ∪ A = fold Set.insert xs A
proof –
  interpret comp-fun-idem Set.insert
  by (fact comp-fun-idem-insert)
  show ?thesis by (simp add: union-fold-insert fold-set-fold)
qed

lemma union-coset-filter [code]:
  List.coset xs ∪ A = List.coset (List.filter (λx. x ∉ A) xs)
by auto

lemma minus-set-fold [code]: A − set xs = fold Set.remove xs A
proof –
  interpret comp-fun-idem Set.remove
  by (fact comp-fun-idem-remove)
  show ?thesis
    by (simp add: minus-fold-remove [of - A] fold-set-fold)
qed

lemma minus-coset-filter [code]:
  A − List.coset xs = set (List.filter (λx. x ∈ A) xs)
by auto

lemma inter-set-filter [code]:
  A ∩ set xs = set (List.filter (λx. x ∈ A) xs)
by auto

lemma inter-coset-fold [code]:
  A ∩ List.coset xs = fold Set.remove xs A
by (simp add: Diff-eq [symmetric] minus-set-fold)

lemma (in semilattice-set) set-eq-fold [code]:
  F (set (x # xs)) = fold f xs x
proof –
  interpret comp-fun-idem f
  by standard (simp-all add: fun-eq-iff left-commute)
  show ?thesis by (simp add: eq-fold fold-set-fold)
qed

lemma (in complete-lattice) Inf-set-fold:
  Inf (set xs) = fold inf xs top
proof –
  interpret comp-fun-idem inf :: 'a ⇒ 'a ⇒ 'a
  by (fact comp-fun-idem-inf)
  show ?thesis by (simp add: Inf-fold-inf fold-set-fold inf-commute)
qed
THEORY “List”

declare Inf-set-fold [where 'a = 'a set, code]

lemma (in complete-lattice) Sup-set-fold:
  Sup (set xs) = fold sup xs bot
proof –
  interpret comp-fun-idem sup :: 'a ⇒ 'a ⇒ 'a
  by (fact comp-fun-idem-sup)
  show ?thesis by (simp add: Sup-fold-sup fold-set-fold sup-commute)
qed

declare Sup-set-fold [where 'a = 'a set, code]

lemma (in complete-lattice) INF-set-fold:
  ⨍ (f ' set xs) = fold (inf ◦ f) xs top
using Inf-set-fold [of map f xs] by (simp add: fold-map)

lemma (in complete-lattice) SUP-set-fold:
  ⨆ (f ' set xs) = fold (sup ◦ f) xs bot
using Sup-set-fold [of map f xs] by (simp add: fold-map)

66.1.20 Fold variants: foldr and foldl

Correspondence

lemma foldr-conv-fold [code-abbrev]: foldr f xs = fold f (rev xs)
by (induct xs) simp-all

lemma foldl-conv-fold: foldl f s xs = fold (λx s. f s x) xs s
by (induct xs arbitrary: s) simp-all

lemma foldr-cong-conv-fold: — The “Third Duality Theorem” in Bird & Wadler:
  foldr f xs a = foldl (λx y. f y x) a (rev xs)
by (simp add: foldr-cong-conv-fold foldl-cong-conv-fold)

lemma foldl-cong-conv-fold:
  foldl f a xs = foldr (λx y. f y x) (rev xs) a
by (simp add: foldr-cong-conv-fold foldl-cong-conv-fold)

lemma foldr-fold:
  (∀x y. x ∈ set xs ⇒ y ∈ set xs ⇒ f y ◦ f x = f x ◦ f y)
  ⇒ foldr f xs = fold f xs
unfolding foldr-cong-conv-fold by (rule fold-rev)

lemma foldr-cong [fundef-cong]:
  a = b ⇒ l = k ⇒ (∀a x. x ∈ set l ⇒ f x a = g x a) ⇒ foldr f l a = foldr g k b
  by (auto simp add: foldr-cong fold-cong-trivial)

lemma foldl-cong [fundef-cong]:
  a = b ⇒ l = k ⇒ (∀a x. x ∈ set l ⇒ f a x = g a x) ⇒ foldl f a l = foldl g
b k
by (auto simp add: foldl-conv-fold intro: fold-cong)

lemma foldr-append [simp]: foldr f (xs @ ys) a = foldr f xs (foldr f ys a)
by (simp add: foldr-conv-fold)

lemma foldl-append [simp]: foldl f a (xs @ ys) = foldl f (foldl f a xs) ys
by (simp add: foldl-conv-fold)

lemma foldr-map [code-unfold]: foldr g (map f xs) a = foldr (g ◦ f) xs a
by (simp add: foldr-conv-fold fold-map rev-map)

lemma foldr-filter:
foldr f (filter P xs) = foldr (λx. if P x then f x else id) xs
by (simp add: foldr-conv-fold rev-filter fold-filter)

lemma foldl-map [code-unfold]:
foldl g a (map f xs) = foldl (λa x. g a (f x)) a xs
by (simp add: foldl-conv-fold fold-map comp-def)

lemma concat-conv-foldr [code]:
c = foldr append c
by (simp add: fold-append-concat-rev foldr-conv-fold)

66.1.21 upt

lemma upt-rec [code]: [i..<] = (if i<j then i#[Suc i..<] else [])
― simp does not terminate!
by (induct j) auto

lemmas upt-rec-numeral[simp] = upt-rec[of numeral m numeral n] for m n

lemma upt-conv-Nil [simp]: j ≤ i ⇒ [i..<] = []
by (subst upt-rec) simp

lemma upt-eq-Nil-conv [simp]: ([i..<] = []) = (j = 0 ∨ j ≤ i)
by (induct j) simp-all

lemma upt-eq-Cons-conv:
([i..<] = x#xs) = (i < j ∧ i = x ∧ [i+1..<] = xs)
proof (induct j arbitrary: x xs)
case (Suc j)
then show ?case
by (simp add: upt-rec)
qed simp

lemma upt-Suc-append: i ≤ j ⇒ [i..<](Suc j) = [i..<]@[j]
― Only needed if upt-Suc is deleted from the simpset.
by simp
lemma upt-conv-Cons: \( i < j \implies [i..<j] = i \# [\text{Suc } i..<j] \)
by (simp add: upt-rec)

lemma upt-conv-Cons-Cons: — no precondition
\( m \# n \# ns = [m..<q] \iff n \# ns = [\text{Suc } m..<q] \)
proof (cases \( m < q \))
case False then show \( \text{thesis} \) by simp
next
case True then show \( \text{thesis} \) by (simp add: upt-conv-Cons)
qed

lemma upt-add-eq-append: \( i < j \implies [i..<j] + k = [i..<j] @ [j..<j+k] \)
— LOOPS as a simprule, since \( j \leq j \).
by (induct \( k \)) auto

lemma length-upt[simp]: \( \text{length } [i..<j] = j - i \)
by (induct \( j \)) (auto simp add: Suc-diff-le)

lemma nth-upt[simp]: \( i + k < j \implies [i..<j] ! k = i + k \)
by (induct \( j \)) (auto simp add: less-Suc-eq nth-append split: nat-diff-split)

lemma hd-upt[simp]: \( i < j \implies \text{hd } [i..<j] = i \)
by (simp add: upt-conv-Cons)

lemma tl-upt[simp]: \( \text{tl } [m..<n] = [\text{Suc } m..<n] \)
by (simp add: upt-rec)

lemma last-upt[simp]: \( i < j \implies \text{last } [i..<j] = j - 1 \)
by (cases \( j \)) (auto simp: upt-Suc-append)

lemma take-upt[simp]: \( i + m \leq n \implies \text{take } m [i..<n] = [i..<i+m] \)
proof (induct \( m \) arbitrary: \( i \))
case (Suc \( m \))
then show \( \text{thesis} \) by (subst take-Suc-cons-app-nth) auto
qed simp

lemma drop-upt[simp]: \( \text{drop } m [i..<j] = [i+m..<j] \)
by (induct \( j \)) auto

lemma map-Suc-upt: \( \text{map } \text{Suc } [m..<n] = [\text{Suc } m..<\text{Suc } n] \)
by (induct \( n \)) auto

lemma map-add-upt: \( \text{map } (\lambda i. i + n) [0..<m] = [n..<m + n] \)
by (induct \( m \)) simp-all

lemma nth-map-upt: \( i < n-m \implies (\text{map } f [m..<n]) ! i = f(m+i) \)
proof (induct \( n m \) arbitrary: \( i \) rule: diff-induct)
case (3 x y)
then show ?case
  by (metis add.commute length-upt less-diff-conv nth-map nth-upt)
qed auto

lemma map-decr-upt: map (λn. n − Suc 0) [Suc m..<Suc n] = [m..<n]
  by (induct n) simp-all

lemma map-upt-Suc: map f [0..< Suc n] = f 0 # map (λi. f (Suc i)) [0..< n]
  by (induct n arbitrary: f) auto

lemma nth-take-lemma:
  k ≤ length xs ⇒ k ≤ length ys ⇒
  (∀i. i < k ⇒ xs!i = ys!i) ⇒ take k xs = take k ys
proof (induct k arbitrary: xs ys)
case (Suc k)
then show ?case
  by (simp add: less-Suc-eq-0-disj)
by (simp add: Suc.prems(3) take-Suc-conv-app-nth)
qed simp

lemma map-nth:
  map (λi. xs ! i) [0..<length xs] = xs
  by (rule nth-equalityI, auto)

lemma nth-equalityI:
  [length xs = length ys; ∀i. i < length xs ⇒ xs!i = ys!i] ⇒ xs = ys
  by (frule nth-take-lemma [OF le-refl eq-imp-le], simp-all)

lemma list-all2-antisym:
  [ (∀x y. [P x y; Q y x] ⇒ x = y); list-all2 P xs ys; list-all2 Q ys xs ]
  ⇒⇒ xs = ys
  by (simp add: list-all2-conv-all-nth nth-equalityI)

lemma take-equalityI: (∀i. take i xs = take i ys) ⇒ xs = ys
— The famous take-lemma.
  by (metis length-take min.commute order-refl take-all)

lemma take-Cons':
  take n (x # xs) = (if n = 0 then [] else x # take (n − 1) xs)
  by (cases n) simp-all

lemma drop-Cons':
  drop n (x # xs) = (if n = 0 then x # xs else drop (n − 1) xs)
  by (cases n) simp-all

lemma nth-Cons': (x # xs)!n = (if n = 0 then x else xs!(n − 1))
  by (cases n) simp-all
lemma \textit{take-Cons-numeral} [simp]:
\begin{align*}
take \ (\text{numeral} \ v) \ (x \ # \ xs) &= x \ # \ take \ (\text{numeral} \ v - 1) \ xs \\
\text{by} \ (\text{simp add: take-Cons}) \end{align*}

lemma \textit{drop-Cons-numeral} [simp]:
\begin{align*}
drop \ (\text{numeral} \ v) \ (x \ # \ xs) &= drop \ (\text{numeral} \ v - 1) \ xs \\
\text{by} \ (\text{simp add: drop-Cons}) \end{align*}

lemma \textit{nth-Cons-numeral} [simp]:
\begin{align*}
(x \ # \ xs) ! \ \text{numeral} \ v &= xs ! \ (\text{numeral} \ v - 1) \\
\text{by} \ (\text{simp add: nth-Cons}) \end{align*}

lemma \textit{map-upt-eqI}:
\begin{align*}
\langle \map f \ [m..<n] = xs \rangle \ &\text{if} \ \langle \text{length xs} = n - m \rangle \\
\ &\text{then} \ \langle \forall i. \ i < \text{length xs} \implies xs ! i = f (m + i) \rangle \\
\text{proof} \ (\text{rule nth-equalityI}) \\
\text{from} \ \langle \text{length xs} = n - m \rangle \ &\text{show} \ \langle \text{length} \ (\map f \ [m..<n]) = \text{length} \ xs \rangle \\
\ &\text{by} \ \text{simp} \\
\end{align*}

next
\textbf{fix} \ i
\textbf{assume} \ \langle i < \text{length} \ (\map f \ [m..<n]) \rangle
\textbf{then have} \ \langle i < n - m \rangle
\textbf{by} \ \text{simp}
\textbf{with} \ \text{that} \ \textbf{have} \ \langle xs ! i = f (m + i) \rangle
\textbf{by} \ \text{simp}
\textbf{with} \ \langle i < n - m \rangle \ &\text{show} \ \langle \map f \ [m..<n] ! i = xs ! i \rangle \\
\ &\text{by} \ \text{simp} \\
\text{qed}

\textbf{66.1.22} \ \textit{upto: interval-list on int}

\textbf{function} \ \textit{upto} :: \ int \ ⇒ \ int ⇒ int list \ ((\text{\texttt{[\ldots\ldots]}})) \ \textbf{where}
\textit{upto} \ i \ j = (if \ i \leq \ j \ \text{then} \ i \ # \ [i+1..j] \ \text{else} \ [])
\textbf{by} \ \text{auto}

\textbf{termination}
\textbf{by}(\text{relation measure}(\% (i::int,j). \ \text{nat}(j - i + 1)))) \ \text{auto}

\textbf{declare} \ \textit{upto.simps}[simp del]

\textbf{lemmas} \ \textit{upto-rec-numeral} [simp] =
\begin{align*}
\textit{upto.simps}[\text{of numeral} \ m \ \text{numeral} \ n] \\
\textit{upto.simps}[\text{of numeral} \ m - \ \text{numeral} \ n] \\
\textit{upto.simps}[\text{of} - \ \text{numeral} \ m \ \text{numeral} \ n] \\
\textit{upto.simps}[\text{of} - \ \text{numeral} \ m - \ \text{numeral} \ n] \ \textbf{for} \ m \ n
\end{align*}

\textbf{lemma} \ \textit{upto-empty}[simp]: \ \langle j < i \ \implies \ [i..j] = [] \rangle
\textbf{by}(\text{simp add: upto.simps})

\textbf{lemma} \ \textit{upto-single}[simp]: \ \langle i..i \rangle = \ [i]
by (simp add: upto.simps)

**lemma** upto-Nil[simp]: \([i..j] = [] \iff j < i\)
by (simp add: upto.simps)

**lemmas** upto-Nil2[simp] = upto-Nil[THEN eq_iff_swap]

**lemma** upto-rec1: \(i \leq j \implies [i..j] = \{i\}\#[i+1..j]\)
by (simp add: upto.simps)

**lemma** upto-rec2: \(i \leq j \implies [i..j] = [i..j-1]@\{j\}\)
proof (induct (nat (j - i)) arbitrary: \(i\ j\))
  case 0 thus ?case by (simp add: upto-rec1 of \(i\ j\))
next
case (Suc \(n\))
hence \(n = \text{nat} (j - (i + 1))\) \(i < j\) by linarith+
  from this(2) Suc.hyps(1) [OF this(1)] Suc(2,3) upto-rec1 show ?case by simp
qed

**lemma** length-upto[simp]: \(\text{length} [i..j] = \text{nat} (j - i + 1)\)
by (induction \(i\ j\) rule: upto.induct)

**lemma** set-upto[simp]: \(\text{set}[i..j] = \{i..j\}\)
proof (induct \(i\ j\) rule: upto.induct)
  case (1 \(i\ j\))
  from this show ?case unfolding upto.simps[of \(i\ j\)] by auto
qed

**lemma** nth-upto[simp]: \(i + \text{int} k \leq j \implies [i..j] ! k = i + \text{int} k\)
proof (induction \(i\ j\) arbitrary: \(k\) rule: upto.induct)
  case (1 \(i\ j\))
  then show ?case by (auto simp add: upto-rec1 of \(i\ j\) nth-Cons')
qed

**lemma** upto-split1: \(i \leq j \implies j \leq k \implies [i..k] = [i..j-1]@[j..k]\)
proof (induction \(j\) rule: int_ge_induct)
  case base thus ?case by (simp add: upto-rec1)
next
case step thus ?case using upto-rec1 upto-rec2 by simp
qed

**lemma** upto-split2: \(i \leq j \implies j \leq k \implies [i..k] = [i..j] @ [j+1..k]\)
using upto-rec1 upto-rec2 upto-split1 by auto

**lemma** upto-split3: \([i \leq j; j \leq k] \implies [i..k] = [i..j-1] @ j # [j+1..k]\)
using upto-rec1 upto-split1 by auto

Tail recursive version for code generation:

definition upto-aux :: int ⇒ int ⇒ int list ⇒ int list
where
upto-aux i j js = [i..j] @ js

lemma upto-aux-rec [code]:
upto-aux i j js = (if j < i then js else upto-aux i (j - 1) (j#js))
by (simp add: upto-aux-def upto-rec2)

lemma upto-code[code]: [i..j] = upto-aux i j []
by (simp add: upto-aux-def)

66.1.23 successively

lemma successively-Cons:
successively P (x # xs) ‒→ xs = [] ∨ P x (hd xs) ∧ successively P xs
by (cases xs) auto

lemma successively-cong [cong]:
assumes \( x \in \text{set} \ xs \implies y \in \text{set} \ xs \implies P x y \implies Q x y xs = ys \)
schools successively P xs ‒→ successively Q ys
unfolding assms(2) [symmetric] using assms(1)
by (induction xs) (auto simp: successively-Cons)

lemma successively-append-iff:
successively P (xs @ ys) ‒→
successively P xs ∧ successively P ys ∧
(xs = [] ∨ ys = [] ∨ P (last xs) (hd ys))
by (induction xs) (auto simp: successively-Cons)

lemma successively-if-sorted-wrt:
sorted-wrt P xs ⇒ successively P xs
by (induction xs rule: induct-list012) auto

lemma successively-if-sorted-wrt-strong:
assumes \( \forall x y z. x \in \text{set} \ xs \implies y \in \text{set} \ xs \implies z \in \text{set} \ xs \implies P x y \implies P y z \implies P x z \)
schools successively P xs ‒→ sorted-wrt P xs
proof
assume successively P xs
from this and assm show sorted-wrt P xs
proof (induction xs rule: induct-list012)
case (3 x y xs)
from 3.prems have P x y
by auto
have IH; sorted-wrt P (y # xs)
using 3.prems
by (intro 3.IH(2) list.set-intros(2))(simp blast intro: list.set-intros(2))

have \( P \ x \ z \) if asm: \( z \in \textit{set} \ xs \) for \( z \)

proof
  from IH and asm have \( P \ y \ z \)
  by auto
  with \( P \ x \ y \) show \( P \ x \ z \)
  using 3.prems asm by auto
qed

with IH and \( P \ x \ y \) show ?case by auto
qed auto

qed (use successively-if-sorted-wrt in blast)

lemma successively-conv-sorted-wrt:
assumes transp P
shows successively P xs \iff sorted-wrt P xs
using assms unfolding transp-def
by (intro successively-iff-sorted-wrt-strong) blast

lemma successively-rev [simp]: successively P (rev xs) \iff successively (\( \lambda x \ y. P \ y \ x \)) xs
by (induction xs rule: remdups-adj.induct)
(auto simp: successively-Cons)

lemma successively-map: successively P (map f xs) \iff successively (\( \lambda x \ y. P \ (f \ y) \)) xs
by (induction xs rule: induct-list012) auto

lemma successively-mono:
assumes successively P xs
assumes \( \forall x \ y. x \in \textit{set} \ xs \implies y \in \textit{set} \ xs \implies P \ x \ y \implies Q \ x \ y \)
shows successively Q xs
using assms by (induction Q xs rule: successively.induct) auto

lemma successively-altdef:
successively = (\( \lambda P. \ \textit{rec-list} \ True \ (\lambda x \ y. \ \textit{case} \ xs \ b. \ \textit{case} \ xs \ of \ [] \Rightarrow True \ | \ y \# \ - \Rightarrow P \ x \ y \ ∧ \ b) \) xs)
proof (intro ext)
fix P and xs :: 'a list
show successively P xs = rec-list True (\( \lambda x \ y. \ \textit{case} \ xs \ b. \ \textit{case} \ xs \ of \ [] \Rightarrow True \ | \ y \# \ - \Rightarrow P \ x \ y \ ∧ \ b) \) xs
  by (induction xs) (auto simp: successively-Cons split: list.splits)
qed

66.1.24 distinct and remdups and remdups-adj

lemma distinct-tl: distinct xs \implies distinct (tl xs)
by (cases xs) simp-all

lemma distinct-append [simp]:
distinct \((xs @ ys)\) = \((\text{distinct } xs \land \text{distinct } ys \land \text{set } xs \cap \text{set } ys = \{\})\)
by (induct xs) auto

lemma distinct-rev[simp]: \(\text{distinct} \{(rev \xs)\} = \text{distinct } \xs\)
by(induct xs) auto

lemma set-remdups [simp]: \(\text{set} \ \text{remdups } \xs = \text{set } \xs\)
by (induct xs) (auto simp add: insert-absorb)

lemma distinct-remdups [iff]: \(\text{distinct} \ \text{remdups } \xs\)
by (induct xs) auto

lemma distinct-remdups-id: \(\text{distinct } \xs \Rightarrow \text{remdups } \xs = \xs\)
by (induct xs) (auto)

lemma remdups-id-iff-distinct [simp]: \(\text{remdups } \xs = \ys \iff \text{distinct } \xs\)
by (metis distinct-remdups finite-list set-remdups)

lemma length-remdups-leq [iff]: \(\text{length} \ \text{remdups } \xs \leq \text{length } \xs\)
by (induct xs) (auto)

lemma length-remdups-eq [iff]: \(\text{length} \ \text{remdups } \xs = \text{length } \xs\)
by (induct xs)
proof (induct xs)
  case (Cons a xs)
  then show ?case
  by simp (metis Suc-n-not-le-n impossible-Cons length-remdups-leq)
qed auto

lemma remdups-filter: \(\text{remdups} \ \text{filter } P \xs = \text{filter } P \ \text{remdups } \xs\)
by (induct xs) (auto)

lemma distinct-map:
  \(\text{distinct} \ (\text{map } f \xs) = \ (\text{distinct } \xs \land \text{inj-on } f \ (\text{set } \xs))\)
by (induct xs) (auto)

lemma distinct-map-filter:
  \(\text{distinct} \ (\text{map } f \xs) = \Rightarrow \text{distinct} \ (\text{map } f \ (\text{filter } P \xs))\)
by (induct xs) auto

lemma distinct-filter [simp]: \(\text{distinct } \xs = \Rightarrow \text{distinct } (\text{filter } P \xs)\)
by (induct xs) auto

lemma distinct-upt[simp]: distinct[i..<j]
by (induct j) auto

lemma distinct-upto[simp]: distinct[i..<j]
proof (induction i j rule: upto.induct)
  case (1 i j)
  then show ?case
    by (simp add: upto.simps [of i])
qed

lemma distinct-take[simp]: distinct xs =⇒ distinct (take i xs)
proof (induct xs arbitrary: i)
  case (Cons a xs)
  then show ?case
    by (metis Cons.prems append-take-drop-id distinct-append)
qed auto

lemma distinct-drop[simp]: distinct xs =⇒ distinct (drop i xs)
proof (induct xs arbitrary: i)
  case (Cons a xs)
  then show ?case
    by (metis Cons.prems append-take-drop-id distinct-append)
qed auto

lemma distinct-list-update:
  assumes d: distinct xs and a: a ∉ set xs − {xs!i}
  shows distinct (xs[i:=a])
proof (cases i < length xs)
  case True
  with a have anot: a ∉ set (take i xs @ xs ! i ≠ drop (Suc i) xs) − {xs!i}
    by simp (metis in-set-dropD in-set-takeD)
  show ?thesis
  proof (cases a = xs!i)
    case True
    with d show ?thesis
      by auto
  next
    case False
    have set (take i xs) ∩ set (drop (Suc i) xs) = {} 
      by (metis True d disjoint-insert(1) distinct-append id-take-nth-drop list.set(2))
    then show ?thesis
      using d False anot ⟨i < length xs⟩ by (simp add: upd-conv-take-nth-drop)
  qed
  next
    case False with d show ?thesis by auto
  qed
lemma distinct-concat:
\[
\begin{align*}
\text{distinct } xs; \\
\quad \forall ys. \; ys \in \text{set } xs \implies \text{distinct } ys; \\
\quad \forall ys. \; [ \; ys \in \text{set } xs \; ; \; zs \in \text{set } xs \; ; \; ys \neq zs \; ] \implies \text{set } ys \cap \text{set } zs = \{ \} \\
\implies \text{distinct } (\text{concat } xs)
\end{align*}
\]
by (induct xs) auto

An iff-version of \(\forall \; \text{distinct } ?xs; \; \forall ys. \; ys \in \text{set } ?xs \implies \text{distinct } ys; \; \forall ys, zs. \; [ \; ys \in \text{set } ?xs \; ; \; zs \in \text{set } ?xs \; ; \; ys \neq zs \; ] \implies \text{set } ys \cap \text{set } zs = \{ \} \implies \text{distinct } (\text{concat } ?xs)\) is available further down as \text{distinct-concat-iff}.

It is best to avoid the following indexed version of distinct, but sometimes it is useful.

lemma distinct-conv-nth: \(\text{distinct } xs = (\forall i < \text{size } xs. \; \forall j < \text{size } xs. \; i \neq j \implies xs!i \neq xs!j)\)
proof (induct xs)
case (Cons x xs)
show ?case
apply (auto simp add: Cons nth-Cons less-Suc-eq-le split: nat.split-asm)
apply (metis Suc-le-eq
less-nat-zero-code)
done
qed auto

lemma nth-eq-iff-index-eq:
\[
\begin{align*}
\text{distinct } xs; \; i < \text{length } xs; \; j < \text{length } xs \implies (xs!i = xs!j) = (i = j)
\end{align*}
\]
by (auto simp: distinct-conv-nth)

lemma distinct-Ex1:
\[
\text{distinct } xs \implies \exists! i. \; i < \text{length } xs \land xs!i = x
\]
by (auto simp: distinct-conv-nth)

lemma inj-on-nth: \(\forall i < \text{length } xs \implies \text{inj-on} \; (\text{nth } xs) \; I\)
by (rule inj-onI) (simp add: nth-eq-iff-index-eq)

lemma bij-betw-nth: 
assumes \(\text{distinct } xs \; A = \{..<\text{length } xs\} \; B = \text{set } xs\)
shows \(\text{bij-betw} \; (\text{(!}) \; xs) \; A \; B\)
using assms unfolding bij-betw-def
by (auto intro!: inj-on-nth simp: set-conv-nth)

lemma set-update-distinct: 
\[
\begin{align*}
\text{distinct } xs; \; n < \text{length } xs \implies \\
\text{set}(xs[n := x]) = \text{insert } x \; (\text{set } xs - \{xs!n\})
\end{align*}
\]
by (auto simp: set-eq-iff in-set-conv-nth nth-list-update nth-eq-iff-index-eq)

lemma distinct-swap: 
\[
\begin{align*}
\text{distinct}\; [\; i < \text{size } xs; \; j < \text{size } xs]\implies \\
\text{distinct}(xs[i := xs!j, \; j := xs!i]) = \text{distinct } xs
\end{align*}
\]
apply (simp add: distinct-conv-nth nth-list-update)
apply (safe; metis)
done

lemma set-swap[simp]:
[ i < size xs; j < size xs ] \implies set(xs[i := xs!j, j := xs!i]) = set xs
by(simp add: set-conv-nth nth-list-update) metis

lemma distinct-card: distinct xs \implies card(set xs) = size xs
by (induct xs) auto

lemma card-distinct: card(set xs) = size xs \implies distinct xs
proof (induct xs)
case (Cons x xs)
show ?case
proof (cases x \in set xs)
case False with Cons show ?thesis by simp
next
case True with Cons.prems
have card(set xs) = Suc(length xs)
  by (simp add: card-insert-if split: if-split-asm)
moreover have card(set xs) \leq length xs by (rule card-length)
ultimately have False by simp
thus ?thesis ..
qed
qed simp

lemma distinct-length-filter: distinct xs \implies length(filter P xs) = card({x. P x} Int set xs)
by (induct xs) (auto)

lemma not-distinct-decomp: \neg distinct ws \implies \exists xs ys zs y. ws = xs@[y]@ys@[y]@zs
proof (induct n == length ws arbitrary:ws)
case (Suc n ws)
then show ?case
using length-Suc-conv[of ws n]
apply (auto simp: eq-commute)
apply (metis append-Nil in-set-conv-decomp-first)
by (metis append-Cons)
qed simp

lemma not-distinct-conv-prefix:
defines dec as xs y ys \equiv y \in set xs \land distinct xs \land as = xs \oplus y \# ys
shows \neg distinct as \iff \exists xs y ys. dec as xs y ys) (is ?L = ?R)
proof
assume ?L then show ?R
proof (induct length as arbitrary: as rule: less-induct)
case less
obtain xs ys zs y where decomp: as = (xs \oplus y \# ys) \oplus y \# zs
  using not-distinct-decomp[OF less.prems] by auto
show \(?case\)
proof (cases distinct (xs @ y # ys))
case True
  with decomp have \(\text{dec as (xs @ y # ys) y zs}\) by \((\text{simp add: dec-def})\)
  then show \(?\text{thesis}\) by blast
next
case False
  with less decomp obtain xs’ y’ ys’ where \(\text{dec (xs @ y # ys) xs’ y’ ys’}\)
    by atomize-elim auto
  with decomp have \(\text{dec as xs’ y’ (ys’ @ y # zs)}\) by \((\text{simp add: dec-def})\)
  then show \(?\text{thesis}\) by blast
qed
qed (auto simp: dec-def)

lemma distinct-product:
distinct xs \(\implies\) distinct ys \(\implies\) distinct \((\text{List.product} \, \text{xs} \, \text{ys})\)
by (induct xs) (auto intro: \(\text{inj-onI}\) simp add: distinct-map)

lemma distinct-product-lists:
assumes \(\forall \\text{xs} \in \text{set} \, \text{xss}.\) distinct xs
shows distinct \((\text{product-lists} \, \text{xss})\)
using assms proof (induction \text{xs})
case (Cons \text{xs} \text{xs}) note \(* = \text{this}\)
then show \(?\text{case}\)
proof (cases \text{product-lists} \text{xss})
case Nil then show \(?\text{thesis}\) by (induct \text{xs}) simp-all
next
case (Cons \text{ps} \text{pss}) with \(* \) show \(?\text{thesis}\)
  by (auto intro: \(\text{inj-onI}\) distinct-concat simp add: distinct-map)
qed
qed simp

lemma length-remdups-concat:
length \((\text{remdups} \, \text{concat} \, \text{xss})\) = card \(\bigcup \text{xs} \in \text{set} \, \text{xss}.\) \text{set} \text{xs} 
by (simp add: distinct-card [symmetric])

lemma remdups-append2:
\text{remdups} \((\text{xs} \@ \text{remdups} \, \text{ys})\) = \text{remdups} \((\text{xs} \@ \text{ys})\)
by (induction \text{xs}) auto

lemma length-remdups-card-conv: length(remdups \text{xs}) = card(set \text{xs})
proof
  have \text{xs}: \text{concat}[\text{xs}] = \text{xs} by simp
  from length-remdups-concat[of [\text{xs}]] show \(?\text{thesis}\) unfolding \text{xs} by simp
qed

lemma remdups-remdups: \text{remdups} \((\text{remdups} \, \text{xs})\) = \text{remdups} \text{xs}
by (induct \text{xs}) simp-all
lemma distinct-butlast:
  assumes distinct xs
  shows  distinct (butlast xs)
proof (cases xs = [])
  case False
    from \<xs \neq []\> obtain ys y where xs = ys @ [y] by (cases xs rule: rev-cases)
  auto
with \<distinct xs\> show ?thesis by simp
qed (auto)

lemma remdups-map-remdups:
  remdups (map f (remdups xs)) = remdups (map f xs)
by (induct xs) simp-all

lemma distinct-zipI1:
  assumes distinct xs
  shows  distinct (zip xs ys)
proof (rule zip-obtain-same-length)
  fix xs' :: 'a list and ys' :: 'b list and n
  assume length xs' = length ys'
  assume xs' = take n xs
  with assms have distinct xs' by simp
  with \<length xs' = length ys'\> show distinct (zip xs' ys')
    by (induct xs' ys' rule: list-induct2) (auto elim: in-set-zipE)
qed

lemma distinct-zipI2:
  assumes distinct ys
  shows  distinct (zip xs ys)
proof (rule zip-obtain-same-length)
  fix xs' :: 'b list and ys' :: 'a list and n
  assume length xs' = length ys'
  assume ys' = take n ys
  with assms have distinct ys' by simp
  with \<length xs' = length ys'\> show distinct (zip xs' ys')
    by (induct xs' ys' rule: list-induct2) (auto elim: in-set-zipE)
qed

lemma set-take-disj-set-drop-if-distinct:
  distinct vs =\<\Rightarrow\> i \leq j =\<\Rightarrow\> set (take i vs) \cap set (drop j vs) = {}
by (auto simp: in-set-conv-nth distinct-conv-nth)

lemma distinct-singleton: distinct [x] by simp

lemma distinct-length-2-or-more:
  distinct (a # b # xs) =\<\iff\> (a \neq b \land distinct (a # xs) \land distinct (b # xs))
by force

lemma remdups-adj-altdef: (remdups-adj xs = ys) \iff
(\exists f:\nat \Rightarrow \nat. \ mono f \wedge f \cdot \{0 ..< \size xs\} = \{0 ..< \size ys\}
\wedge (\forall i < \size xs. xsl i = ysl(f i))
\wedge (\forall i. i + 1 < \size xs \rightarrow (xsl i = xsl(i+1) \leftrightarrow f i = f(i+1))))
(\exists f. \ ?p f xs ys)

proof
  assume ?L
  then show \exists f. \ ?p f xs ys
    proof (induct xs arbitrary: ys rule: remdups-adj.induct)
      case (1 ys)
      thus \ ?case by (intro exI[of - id]) (auto simp: mono-def)
    next
      case (2 x ys)
      thus \ ?case by (intro exI[of - id]) (auto simp: mono-def)
    next
      case (3 x1 x2 xs ys)
      let ?xs = x1 \# x2 \# xs
      let ?cond = x1 = x2
      define zs where zs = remdups-adj (x2 \# xs)
      from \(3(1=2))[of zs]
      obtain f where \ ?p f (x2 \# xs) zs unfolding zs-def by (cases ?cond) auto
      then have \(f 0 = 0)
      by (intro mono-image-least[where f=f]) blast
      from p have mono: mono f and f-xs-zs: \ f \cdot \{0..<\size (x2 \# xs)\} = \{0..<\size zs\}
       by auto
      have ys: ys = (if x1 = x2 then zs else x1 \# zs)
      unfolding \(3(\exists)[symmetric] zs-def by auto
      have zs0: zs \# 0 = x2 unfolding zs-def by (induct xs) auto
      have zsne: zs \# 0 \# \ by (induct xs) auto
      let \ ?Succ = \if ?cond then id else Suc\n      let \ ?x1 = \if ?cond then id else x1\n      let \ ?f = \lambda i. \ if i = 0 then 0 else ?Succ (f (i - 1))
      have ys: ys = ?x1 zs unfolding ys by (cases ?cond, auto)
      have mono: mono \ ?f using \mono f; unfolding mono-def by auto
      show \ ?case unfolding ys
      proof (intro exI[of - \ f] conjI allI impI)
        show mono \ ?f by fact
      next
      fix i assume i: i < \size ?xs
      with p show \ ?xs ! i = ?x1 zs ! (i? i) using zs0 by auto
    next
      fix i assume i: i + 1 < \size ?xs
      with p show \ ?xs ! i = ?x1 zs ! (i + 1) = (\ff i = \ff (i + 1))
        by (cases i) (auto simp: f0)
    next
      have id: \{0 ..< \size (?x1 zs)\} = insert 0 (?Succ \{0 ..< \size zs\})
        using zsne by (cases ?cond, auto)
{ fix i assume i < Suc (length xs) 
  hence Suc i ∈ {0 ..< Suc (Suc (length xs))} ∩ Collect ((<) 0) by auto 
  from imageI[OF this, of λi. Suc (f i)] 
  have Suc (f i) ∈ {λi. Suc (f (i − Suc 0))) ∈ {0 ..< Suc (Suc (length xs))} ∩ Collect ((<) 0) by auto 
  }
  then show Suc (f i) ∈ {0 ..< length xs} = {0 ..< length x1 zs} 
  unfolding id f-xz-zs[symmetric] by auto 
qed 
qed

next 
assume ∃ f. ?p f xs ys 
then show ?L 
proof (induct xs arbitrary: ys rule: remdups-adj.induct) 
  case 1 then show ?case by auto 
next 
  case (2 x) then obtain f where f-img: f ' {0 ..< size x} = {0 ..< size ys} 
  and f-nth: ∀i. i < size x ⇒ [x]i = ys!f i 
  unfolding id f-xz-zs[symmetric] by auto 
  have length ys = card (f ' {0 ..< size x}) 
  using f-img by auto 
  then show ∗: length ys = 1 by auto 
  then have f 0 = 0 using f-img by auto 
  with ∗ show ?case using f-nth by (cases ys) auto 
next 
  case (3 x1 x2 xs) 
  from 3.prems obtain f where f-mono: mono f 
  and f-img: f ' {0 ..<length x1 # x2 # xs} = {0 ..<length ys} 
  and f-nth: 
  ∀i. i < length (x1 # x2 # xs) ⇒ (x1 # x2 # xs) ! i = ys !f i 
  ∀i. i + 1 < length (x1 # x2 # xs) ⇒ 
  ((x1 # x2 # xs) ! i = (x1 # x2 # xs) ! (i + 1)) = (f i = f (i + 1)) 
  unfolding id f-xz-zs[symmetric] by auto 
  show ?case 
  proof cases 
  assume x1 = x2 
  let ?f' = f ◦ Suc 
  have remdups-adj (x1 # xs) = ys 
  proof (intro 3.hyps exI conjI implI allI) 
    show mono ?f' 
    using f-mono by (simp add: mono-iff-le-Suc) 
  next 
    have ?f' ∈ {0 ..< length (x1 # xs)} = f ' {Suc 0 ..< length (x1 # x2 # xs)} 
    using less-Suc-eq-0-disj by auto
also have \( \ldots = f \cdot \{0 \ldots<\text{length } (x1 \# x2 \# xs)\} \)

proof –
  have \( f 0 = f (\text{Suc } 0) \) using \( \langle x1 = x2 \rangle \) \text{-nth[of } 0] \) by simp
  then show \( \neg \text{thesis} \)
    using \( \text{less-Suc-eq-0-disj} \) by auto

qed

also have \( \ldots = \{0 \ldots<\text{length } ys\} \) by fact

finally show \( \neg f' \cdot \{0 \ldots<\text{length } (x1 \# x2 \# xs)\} = \{0 \ldots<\text{length } ys\} \).

qed (insert \( \text{f-nth[of } \text{Suc } i \text{ for } i] \), auto simp: \( \langle x1 = x2 \rangle \))

then show \( \neg \text{thesis} \) using \( \langle x1 = x2 \rangle \) by simp

next

assume \( x1 \neq x2 \)

have two: \( \text{Suc } (\text{Suc } 0) \leq \text{length } ys \)

proof –
  have \( 2 = \text{card } \{f 0, f 1\} \) using \( \langle x1 \neq x2 \rangle \) \text{-nth[of } 0] \) by auto
  also have \( \ldots \leq \text{card } \{f' \cdot \{0 \ldots<\text{length } (x1 \# x2 \# xs)\}\} \)
    by (rule \( \text{card-mono} \) auto)

finally show \( \neg \text{thesis} \) using \( \text{f-img} \) by simp

qed

have \( f 0 = 0 \) using \( f\text{-mono } f\text{-img} \) by (rule \( \text{mono-image-least} \) simp)

have \( f (\text{Suc } 0) = \text{Suc } 0 \)

proof (rule \( \text{ccontr} \))
  assume \( f (\text{Suc } 0) \neq \text{Suc } 0 \)

then have \( \text{Suc } 0 < f (\text{Suc } 0) \) using \( \text{f-nth[of } 0] \) \( \langle x1 \neq x2 \rangle \). \( f 0 = 0 \).

by auto

then have \( \forall i. \text{Suc } 0 < f (\text{Suc } i) \)
  using \( f\text{-mono} \)
  by (meson \( \text{Suc-le-mono } \text{le0} \) \( \text{less-le-trans } \text{monoD} \))

then have \( \text{Suc } 0 \notin f' \cdot \{0 \ldots<\text{length } (x1 \# x2 \# xs)\} \) by auto

then show \( \neg \text{False} \) using \( \text{f-img two} \) by auto

qed

obtain \( ys' \) where \( ys = x1 \# x2 \# ys' \)
  using \( \text{two f-nth[of } 0] \) \( \text{f-nth[of } 1] \)
  by (auto simp: \( \text{Suc-le-length-ifff } \langle f 0 = 0 \rangle \). \( f (\text{Suc } 0) = \text{Suc } 0 \).)

have \( \text{Suc0-le-f-Suc}: \text{Suc } 0 \leq f (\text{Suc } i) \) for \( i \)
  by (metis \( \text{Suc-le-mono} \) \( f (\text{Suc } 0) = \text{Suc } 0 \) \( f\text{-mono } \text{le0 } \text{mono-def} \))

define \( f' \) where \( f' x = f (\text{Suc } x) - 1 \) for \( x \)

have \( f\text{-Suc}: f (\text{Suc } i) = \text{Suc } (f' i) \) for \( i \)
  using \( \text{Suc0-le-f-Suc} [i] \) by (auto simp: \( f'\text{-def} \))

have \( \text{remdups-adj } (x2 \# xs) = (x2 \# ys') \)

proof (intro 3.hyps exI conjI impI allI)
lemma hd-remdups-adj[simp]: hd (remdups-adj xs) = hd xs
  by (induction xs rule: remdups-adj.induct) simp-all

lemma remdups-adj-Cons: remdups-adj (x # xs) =
  (case remdups-adj xs of [] ⇒ [x] | y # xs ⇒ if x = y then y # xs else x # y # xs)
  by (induct xs arbitrary: x) (auto split: list.splits)

lemma remdups-adj-append-two:
  remdups-adj (xs @ [x,y]) = remdups-adj (xs @ [x]) @ (if x = y then [] else [y])
  by (induct xs rule: remdups-adj.induct, simp-all)

lemma remdups-adj-adjacent:
  Suc i < length (remdups-adj xs) ⇒ remdups-adj xs ! i ≠ remdups-adj xs ! Suc i
  proof (induction xs arbitrary: i rule: remdups-adj.induct)
  case (3 x y xs i)
  thus ?case by (cases i, cases x = y) (simp, auto simp: hd-cone-nth[ symmetric])
  qed simp-all

lemma remdups-adj-rev[simp]: remdups-adj (rev xs) = rev (remdups-adj xs)
  by (induct xs rule: remdups-adj.induct, simp-all add: remdups-adj-append-two)

lemma remdups-adj-length[simp]: length (remdups-adj xs) ≤ length xs
  by (induct xs rule: remdups-adj.induct, auto)

lemma remdups-adj-length-ge1[simp]: xs ≠ [] ⇒ length (remdups-adj xs) ≥ Suc 0
  by (induct xs rule: remdups-adj.induct, simp-all)
lemma remdups-adj-Nil-iff [simp]: remdups-adj xs = [] ←→ xs = []
  by (induct xs rule: remdups-adj.induct, simp-all)

lemma remdups-adj-set [simp]: set (remdups-adj xs) = set xs
  by (induct xs rule: remdups-adj.induct, simp-all)

lemma last-remdups-adj [simp]: last (remdups-adj xs) = last xs
  by (induction xs rule: remdups-adj.induct) auto

lemma remdups-adj-Cons-alt [simp]: x # tl (remdups-adj (x # xs)) = remdups-adj (x # xs)
  by (induct xs rule: remdups-adj.induct, auto)

lemma remdups-adj-distinct: distinct xs ⇒ remdups-adj xs = xs
  by (induct xs rule: remdups-adj.induct, simp-all)

lemma remdups-adj-append: remdups-adj (xs1 @ x # xs2) = remdups-adj (xs1 @ [x]) @ tl (remdups-adj (x # xs2))
  by (induct xs1 rule: remdups-adj.induct, simp-all)

lemma remdups-adj-singleton: remdups-adj xs = [x] =⇒ xs = replicate (length xs) x
  by (induction n) (auto simp: remdups-adj-Cons)

lemma remdups-adj-map-injective:
  assumes inj f
  shows remdups-adj (map f xs) = map f (remdups-adj xs)
  by (induct xs rule: remdups-adj.induct) (auto simp add: injD[OF assms])

lemma remdups-adj-replicate:
  remdups-adj (replicate n x) = (if n = 0 then [] else [x])
  by (induction n) (auto simp: remdups-adj-Cons)

lemma remdups-upt [simp]: remdups [m..<n] = [m..<n]
  proof (cases m ≤ n)
    case False then show ?thesis by simp
  next
case True then obtain q where n = m + q
    by (auto simp add: le_iff_add)
  moreover have remdups [m..<m + q] = [m..<m + q]
    by (induct q) simp-all
  ultimately show ?thesis by simp
next qed

lemma successively-remdups-adjI:
  successively P xs ⇒ successively P (remdups-adj xs)
  by (induction xs rule: remdups-adj.induct) (auto simp: successively-Cons)
lemma successively-remdups-adj-iff:
\((\forall x. x \in \text{set} \; \text{xs} \Rightarrow \text{P} \; x \; x) \Rightarrow \text{successively} \; \text{P} \; (\text{remdups-adj} \; \text{xs}) \iff \text{successively} \; \text{P} \; \text{xs}\)
by (induction \; \text{xs} \; \text{rule: remdups-adj.induct})(auto \; \text{simp: successively-Cons})

lemma successively-conv-nth:
\(\text{successively} \; \text{P} \; \text{xs} \iff (\forall i. \; \text{Suc} \; i < \text{length} \; \text{xs} \Rightarrow \text{P} \; (\text{xs} ! i) \; (\text{xs} ! \; \text{Suc} \; i))\)
by (induction \; \text{P} \; \text{xs} \; \text{rule: successively.induct})

lemma distinct-adj-conv-nth:
\(\text{distinct-adj} \; \text{xs} \iff (\forall i. \; \text{Suc} \; i < \text{length} \; \text{xs} \Rightarrow \text{xs} ! i \neq \text{xs} ! \; \text{Suc} \; i)\)

lemma distinct-adj-nth:
\(\text{distinct-adj} \; \text{xs} \Rightarrow \text{Suc} \; i < \text{length} \; \text{xs} \Rightarrow \text{xs} ! i \neq \text{xs} ! \; \text{Suc} \; i\)

lemma remdups-adj-Cons':
\(\text{remdups-adj} \; (x \# \text{xs}) = x \# \text{remdups-adj} \; (\text{dropWhile} \; (\lambda y. \; y = x) \; \text{xs})\)
by (induction \; \text{xs})

lemma remdups-adj-singleton-iff:
\(\text{length} \; (\text{remdups-adj} \; \text{xs}) = \text{Suc} \; 0 \iff \text{xs} \neq [] \land \text{xs} = \text{replicate} \; (\text{length} \; \text{xs}) \; (\text{hd} \; \text{xs})\)

lemma remdups-adj-append-dropWhile:
\(\text{remdups-adj} \; (\text{xs} @ y \# \text{ys}) = \text{remdups-adj} \; (\text{dropWhile} \; (\lambda x. \; x = \text{hd} \; \text{ys}) \; (\text{tl} \; \text{ys}))\)
by (subst remdups-adj-append) (simp add: tl-remdups-adj)

lemma remdups-adj-append':

assumes $xs = [] \lor ys = [] \lor \text{last } xs \neq \text{hd } ys$

shows $\text{remdups-adj } (xs @ ys) = \text{remdups-adj } xs \circ \text{remdups-adj } ys$

proof –

have $\negthesis$ if $[\text{simp}]: xs \neq [] \land ys \neq [] \land \text{last } xs \neq \text{hd } ys$

proof –

obtain $x xs'$ where $xs = xs' @ [x]$

by (cases $xs$ rule: rev-cases) auto

have $\text{remdups-adj } (xs' @ x \neq ys) = \text{remdups-adj } (xs' @ [x]) @ \text{remdups-adj } ys$

using $\text{last } xs \neq \text{hd } ys$: unfolding $xs$

by (metis (full-types) dropWhile-eq-self-iff last-snoc remdups-adj-append-dropWhile)

thus $\negthesis$ by (simp add: $xs$)

qed

thus $\negthesis$ using $\text{assms}$

by (cases $xs = []$; cases $ys = []$) auto

qed

lemma $\text{remdups-adj-append}'': xs \neq []$

$\implies \text{remdups-adj } (xs @ ys) = \text{remdups-adj } xs \circ \text{remdups-adj } (\text{dropWhile } (\lambda y. y = \text{last } xs) ys)$

by (induction $xs$ rule: remdups-adj.induct) (auto simp: remdups-adj-Cons)

lemma $\text{remdups-filter-last}$:

$
\text{last } [x \leftarrow \text{remdups } xs, P x] = \text{last } [x \leftarrow xs, P x]
$

by (induction $xs$, auto simp: filter-empty-cone)

lemma $\text{remdups-append}$:

$
\text{set } xs \subseteq \text{set } ys \implies \text{remdups } (xs @ ys) = \text{remdups } ys
$

by (induction $xs$, simp-all)

lemma $\text{remdups-concat}$:

$
\text{remdups } (\text{concat } (\text{remdups } xs)) = \text{remdups } (\text{concat } xs)
$

proof (induction $xs$)

case Nil

then show $\negcase$ by simp

next

case (Cons $a$ $xs$)

show $\negcase$

proof (cases $a \in \text{set } xs$)

  case True

  then have $\text{remdups } (\text{concat } xs) = \text{remdups } (a \circ \text{concat } xs)$

  by (metis remdups-append concat.simps(2) insert-absorb set-simps(2) set-append set-concat sup-ge1)

  then show $\negthesis$

  by (simp add: Cons True)

next

  case False

  then show $\negthesis$

  by (metis Cons remdups-append2 concat.simps(2) remdups.simps(2))

qed
distinct-adj

lemma distinct-adj-Cons [simp]: distinct-adj (x # y # xs) ←→ x ≠ y ∧ distinct-adj (y # xs)
  by (auto simp: distinct-adj-def)

lemma distinct-adj-Cons: distinct-adj (x # xs) ←→ xs = [] ∨ x ≠ hd xs ∧ distinct-adj xs
  by (cases xs) auto

lemma distinct-adj-ConsD: distinct-adj (x # xs) =⇒ distinct-adj xs
  by (cases xs) auto

lemma distinct-adj-remdups-adj [simp]: distinct-adj (remdups-adj xs)
  by (induction xs rule: remdups-adj.induct) (auto simp: distinct-adj-Cons)

lemma distinct-adj-altdef: distinct-adj xs ←→ remdups-adj xs = xs
  proof
    assume remdups-adj xs = xs
    with distinct-adj-remdups-adj[of xs] show distinct-adj xs
      by simp
  next
    assume distinct-adj xs
    thus remdups-adj xs = xs
      by (induction xs rule: induct-list012) auto
  qed

lemma distinct-adj-mapI: distinct-adj xs =⇒ inj-on f (set xs) =⇒ distinct-adj (map f xs)
  unfolding distinct-adj-def successively-map
  by (erule successively-mono) (auto simp: inj-on-def)

lemma distinct-adj-mapD: distinct-adj (map f xs) =⇒ distinct-adj xs
unfolding distinct-adj-def successively-map by (erule successively-mono) auto

lemma distinct-adj-map-iff: inj-on f (set xs) \(\implies\) distinct-adj (map f xs) \(\iff\) distinct-adj xs
using distinct-adj-mapD distinct-adj-mapI by blast

lemma distinct-adj-conv-length-remdups-adj:
distinct-adj xs \(\iff\) length (remdups-adj xs) = length xs
proof (induction xs rule: remdups-adj.induct)
case (3 x y xs)
thus \(?case\)
  using remdups-adj-length[of y # xs] by auto
qed auto

66.2.1 insert

lemma in-set-insert [simp]:
x \in set xs \(\implies\) List.insert x xs = xs
by (simp add: List.insert-def)

lemma not-in-set-insert [simp]:
x \notin set xs \(\implies\) List.insert x xs = x # xs
by (simp add: List.insert-def)

lemma insert-Nil [simp]: List.insert x [] = [x]
by simp

lemma set-insert [simp]: set (List.insert x xs) = insert x (set xs)
by (auto simp add: List.insert-def)

lemma distinct-insert [simp]: distinct (List.insert x xs) = distinct xs
by (simp add: List.insert-def)

lemma insert-remdups:
  List.insert x (remdups xs) = remdups (List.insert x xs)
by (simp add: List.insert-def)

66.2.2 List.union

This is all one should need to know about union:

lemma set-union[simp]: set (List.union xs ys) = set xs \cup set ys
unfolding List.union-def
by(induct xs arbitrary: ys) simp-all

lemma distinct-union[simp]: distinct(List.union xs ys) = distinct ys
unfolding List.union-def
by(induct xs arbitrary: ys) simp-all
66.2.3 \textbf{find}

\textbf{lemma} find-None-iff: $\text{List.find } P \, \text{xs} = \texttt{None} \iff \neg (\exists x. x \in \text{set } \text{xs} \land P \, x)$
\textbf{proof} (induction $\text{xs}$)
\hspace{1em} case Nil thus \texttt{?case} by simp
\hspace{1em} next
\hspace{1em} case ($\text{Cons } x \, \text{xs}$) thus \texttt{?case} by (fastforce split: if-splits)
\hspace{1em} qed

\textbf{lemmas} find-None-iff2 = find-None-iff[THEN eq-iff-swap]

\textbf{lemma} find-Some-iff:
\hspace{1em} $\text{List.find } P \, \text{xs} = \texttt{Some } x \iff (\exists i < \text{length } \text{xs}. P \, (\text{xs} ! i) \land x = \text{xs} ! i \land (\forall j < i. \neg P \, (\text{xs} ! j)))$
\textbf{proof} (induction $\text{xs}$)
\hspace{1em} case Nil thus \texttt{?case} by simp
\hspace{1em} next
\hspace{1em} case ($\text{Cons } x \, \text{xs}$) thus \texttt{?case}
\hspace{2em} apply (auto simp: nth-Cons' split: if-splits)
\hspace{2em} using diff-Suc-1[unfolded One-nat-def] less-Suc-eq-0-disj by fastforce
\hspace{1em} qed

\textbf{lemmas} find-Some-iff2 = find-Some-iff[THEN eq-iff-swap]

\textbf{lemma} find-cong[fundef-cong]:
\hspace{1em} assumes $\text{xs} = \text{ys}$ and $\land x. x \in \text{set } \text{ys} \Rightarrow P \, x = Q \, x$
\hspace{1em} shows $\text{List.find } P \, \text{xs} = \text{List.find } Q \, \text{ys}$
\textbf{proof} (cases $\text{List.find } P \, \text{xs}$)
\hspace{1em} case None thus \texttt{?thesis} by (metis find-None-iff assms)
\hspace{1em} next
\hspace{1em} case ($\text{Some } x$)
\hspace{2em} hence $\text{List.find } Q \, \text{ys} = \texttt{Some } x$ using assms
\hspace{2em} by (auto simp add: find-Some-iff)
\hspace{1em} thus \texttt{?thesis} using Some by auto
\hspace{1em} qed

\textbf{lemma} find-dropWhile:
\hspace{1em} $\text{List.find } P \, \text{xs} = (\text{case } \text{dropWhile } (\text{Not } \circ P) \, \text{xs}
\hspace{1em} \mid \emptyset \Rightarrow \texttt{None}
\hspace{1em} \mid x \neq - \Rightarrow \text{Some } x)$
\hspace{1em} by (induct $\text{xs}$) simp-all

66.2.4 \textbf{count-list}

This library is intentionally minimal. See the remark about multisets at the point above where count-list is defined.

\textbf{lemma} count-list-append[simp]: $\text{count-list } (\text{xs} @ \text{ys}) \, x = \text{count-list } \text{xs} \, x + \text{count-list } \text{ys} \, x$
\hspace{1em} by (induction $\text{xs}$) simp-all
lemma count-list-0-iff: count-list xs x = 0 ←→ x ∉ set xs
by (induction xs) simp-all

lemma count-notin[simp]: x ∉ set xs ⇒ count-list xs x = 0
by(simp add: count-list-0-iff)

lemma count-le-length: count-list xs x ≤ length xs
by (induction xs) auto

lemma count-list-map-ge: count-list xs x ≤ count-list (map f xs) (f x)
by (induction xs) auto

lemma count-list-inj-map: inj-on f (set xs); x ∈ set xs ⇒ count-list (map f xs) (f x) = count-list xs x
by (induction xs) simp, fastforce

lemma count-list-rev[simp]: count-list (rev xs) x = count-list xs x
by (induction xs) auto

lemma sum-count-set: set xs ⊆ X ⇒ finite X ⇒ sum (count-list xs) X = length xs
proof (induction xs arbitrary: X)
case (Cons x xs)
then show ?case using sum.remove[of X x count-list xs]
by (auto simp: sum.If-cases simp flip: diff-eq)
qed simp

66.2.5 List.extract

lemma extract-None-iff: List.extract P xs = None ←→ ¬ (∃ x∈set xs. P x)
by(auto simp: extract-def dropWhile-eq-Cons-conv split: list.splits)
   (metis in-set-conv-decomp)

lemma extract-SomeE:
  List.extract P xs = Some (ys, y, zs) ⇒
xz = ys @ y # zs ∧ P y ∧ ¬ (∃ y ∈ set ys. P y)
by(auto simp: extract-def dropWhile-eq-Cons-conv split: list.splits)

lemma extract-Some-iff:
  List.extract P xs = Some (ys, y, zs) ←→
xz = ys @ y # zs ∧ P y ∧ ¬ (∃ y ∈ set ys. P y)
by(auto simp: extract-def dropWhile-eq-Cons-conv dest: set-takeWhileD split: list.splits)

lemma extract-Nil-code[code]: List.extract P [] = None
by(simp add: extract-def)

lemma extract-Cons-code[code]:
List.extract \( P \ (x \neq xs) = \) (if \( P \ x \) then Some ([], x, xs) else
(case List.extract \( P \) xs of
  None \Rightarrow \) None 
  Some (ys, y, zs) \Rightarrow Some (x\#ys, y, zs))
by (auto simp add: extract-def comp-def split: list.splits
  (metis dropWhile-eq-Nil-conv list.distinct(1))

66.2.6 remove1

lemma remove1-append:
remove1 x (xs @ ys) =
(if \( x \in \) set \( xs \) then remove1 x \( xs \) @ ys else \( xs \) @ remove1 x \( ys \))
by (induct \( xs \)) auto

lemma remove1-commute: remove1 x (remove1 y \( zs \)) = remove1 y (remove1 x \( zs \))
by (induct \( zs \)) auto

lemma in-set-remove1[simp]:
\( a \neq b \Rightarrow a \in \) set(remove1 b \( xs \)) = (\( a \in \) set \( xs \))
by (induct \( xs \)) auto

lemma set-remove1-subset: set(remove1 x \( xs \)) \subseteq set \( xs \)
by (induct \( xs \)) auto

lemma set-remove1-eq [simp]: distinct \( xs \) \Rightarrow set(remove1 x \( xs \)) = set \( xs \) \- {\( x \)}
by (induct \( xs \)) auto

lemma length-remove1:
distinct \( xs \) \Rightarrow length(remove1 x \( xs \)) = (if \( x \in \) set \( xs \) then length \( xs \) \- 1 else length \( xs \))
by (induct \( xs \)) (auto dest!: length-pos-if-in-set)

lemma remove1-filter-not[simp]:
\( \neg P \ x \Rightarrow \) remove1 x (filter \( P \) \( xs \)) = filter \( P \) \( xs \)
by (induct \( xs \)) auto

lemma filter-remove1:
filter \( Q \) (remove1 x \( xs \)) = remove1 x (filter \( Q \) \( xs \))
by (induct \( xs \)) auto

lemma notin-set-remove1[simp]: \( x \notin \) set \( xs \) \Rightarrow \( x \notin \) set(remove1 y \( xs \))
by (insert set-remove1-subset) fast

lemma distinct-remove1[simp]: distinct \( xs \) \Rightarrow distinct(remove1 x \( xs \))
by (induct \( xs \)) simp-all

lemma remove1-remdups:
distinct \( xs \) \Rightarrow remove1 x (remdups \( xs \)) = remdups(remove1 x \( xs \))
by (induct \( xs \)) simp-all
lemma remove1-idem: $x \notin \text{set } xs \implies \text{remove1 } x \ xs = xs$
  by (induct xs) simp-all

lemma remove1-split:
  $a \in \text{set } xs \implies \text{remove1 } a \ xs = ys \iff (\exists \ l s \ r s. \ xs = l s @ a @ r s \land a \notin \text{set } l s \land \ ys = l s @ r s)$
  by (metis remove1.simps(2) remove1-append split-list-first)

66.2.7  removeAll

lemma removeAll-filter-not-eq:
  $\text{removeAll } x = \text{filter } (\lambda y. \ x \neq y)$
proof
  fix $xs$
  show $\text{removeAll } x \ xs = \text{filter } (\lambda y. \ x \neq y) \ xs$
    by (induct xs) auto
qed

lemma removeAll-append[simp]:
  $\text{removeAll } x \ (xs @ ys) = \text{removeAll } x \ xs @ \text{removeAll } x \ ys$
by (induct xs) auto

lemma set-removeAll[simp]: $\text{set}(\text{removeAll } x \ xs) = \text{set } xs - \{x\}$
by (induct xs) auto

lemma removeAll-id[simp]: $x \notin \text{set } xs \implies \text{removeAll } x \ xs = xs$
by (induct xs) auto

lemma removeAll-filter-not[simp]:
  $\neg P x \implies \text{removeAll } x \ (\text{filter } P \ xs) = \text{filter } P \ xs$
by (induct xs) auto

lemma distinct-removeAll:
  $\text{distinct } xs \implies \text{distinct } (\text{removeAll } x \ xs)$
by simp add: removeAll-filter-not-eq

lemma distinct-remove1-removeAll:
  $\text{distinct } xs \implies \text{remove1 } x \ xs = \text{removeAll } x \ xs$
by (induct xs) simp-all

lemma map-removeAll-inj-on: inj-on f (insert x (set xs)) $\implies$
  map f (removeAll x xs) = removeAll (f x) (map f xs)
by (induct xs) (simp-all add: inj-on-def)

lemma map-removeAll-inj: inj f $\implies$
  map f (removeAll x xs) = removeAll (f x) (map f xs)
by (rule map-removeAll-inj-on, erule subset-inj-on, rule subset-UNIV)
lemma \textit{length-removeAll-less-eq} \ [simp]:
\begin{align*}
\text{length} \ (\text{removeAll} \ x \ xs) \leq \text{length} \ xs
\end{align*}
by (simp add: removeAll-filter-not-eq)

lemma \textit{length-removeAll-less} \ [termination-simp]:
\begin{align*}
x \in \text{set} \ xs \implies \text{length} \ (\text{removeAll} \ x \ xs) < \text{length} \ xs
\end{align*}
by (auto dest: length-filter-less simp add: removeAll-filter-not-eq)

lemma \textit{distinct-concat-iff}: \text{distinct} \ (\text{concat} \ xs) \iff
\begin{align*}
(\forall \ y s. \ y \in \text{set} \ xs \implies \text{distinct} \ y s) \land
(\forall \ y s z. \ y \in \text{set} \ xs \land z \in \text{set} \ xs \land y \neq z \implies \text{set} \ y s \cap \text{set} zs = \{\})
\end{align*}
apply (induct \ xs)
apply (simp-all, safe, auto)
by (metis Int-iff UN-I empty-iff equals0I set-empty)

66.2.8 \ replicate

lemma \textit{length-replicate} \ [simp]: \text{length} \ (\text{replicate} \ n \ x) = n
by (induct \ n)

lemma \textit{replicate-eqI}:
\begin{align*}
\text{assumes} \ \text{length} \ xs = n \text{ and } \forall y. \ y \in \text{set} \ xs \implies y = x
\text{shows} \ xs = \text{replicate} \ n \ x
\end{align*}
using assms
proof (induct \ xs arbitrary: \ n)
\begin{align*}
\text{case Nil then show \ ?case by simp}
\end{align*}
next
\begin{align*}
\text{case (Cons \ x \ xs) then show \ ?case by (cases \ n) \ simp-all}
\end{align*}
qed

lemma \textit{Ex-list-of-length}: \exists \ xs. \text{length} \ xs = n
by (rule exI [of - replicate \ n \ undefined]) simp

lemma \textit{map-replicate} \ [simp]: \text{map} \ f \ (\text{replicate} \ n \ x) = \text{replicate} \ n \ (f x)
by (induct \ n)

lemma \textit{map-replicate-const}:
\begin{align*}
\text{map} \ (\lambda \ x. \ k) \ lst = \text{replicate} \ (\text{length} \ lst) \ k
\end{align*}
by (induct \ lst)

lemma \textit{replicate-app-Cons-same}:
\begin{align*}
(\text{replicate} \ n \ x) @ (x \ # \ xs) = x \ # \ \text{replicate} \ n \ x \ @ \ xs
\end{align*}
by (induct \ n)

lemma \textit{rev-replicate} \ [simp]: \text{rev} \ (\text{replicate} \ n \ x) = \text{replicate} \ n \ x
by (induct \ n) (auto simp: replicate-app-Cons-same)
lemma replicate-add: replicate \((n + m)\) \(x\) = replicate \(n\) \(x\) @ replicate \(m\) \(x\)
by (induct \(n\)) auto

Courtesy of Matthias Daum:

lemma append-replicate-commute:
  replicate \(n\) \(x\) @ replicate \(k\) \(x\) = replicate \(k\) \(x\) @ replicate \(n\) \(x\)
by (metis add.commute replicate-add)

Courtesy of Andreas Lochbihler:

lemma filter-replicate:
  filter \(P\) (replicate \(n\) \(x\)) = (if \(P\) \(x\) then replicate \(n\) \(x\) else [])
by (induct \(n\)) auto

lemma hd-replicate [simp]: \(n \neq 0 \implies\) hd (replicate \(n\) \(x\)) = \(x\)
by (induct \(n\)) auto

lemma tl-replicate [simp]: tl (replicate \(n\) \(x\)) = replicate \((n - 1)\) \(x\)
by (induct \(n\)) auto

lemma last-replicate [simp]: \(n \neq 0 \implies\) last (replicate \(n\) \(x\)) = \(x\)
by (atomize (full), (induct \(n\)) auto)

lemma nth-replicate [simp]: \(i < n \implies\) (replicate \(n\) \(x\))!\(i\) = \(x\)
by (induct \(n\) arbitrary: \(i\))(auto simp: nth-Cons split: nat.split)

Courtesy of Matthias Daum (2 lemmas):

lemma take-replicate[simp]: take \(i\) (replicate \(k\) \(x\)) = replicate \((\min i k)\) \(x\)
proof (cases \(k \leq i\))
  case True
  then show \(?thesis\)
    by (simp add: min-def)
next
  case False
  then have replicate \(k\) \(x\) = replicate \(i\) \(x\) @ replicate \((k - i)\) \(x\)
    by (simp add: replicate-add [symmetric])
  then show \(?thesis\)
    by (simp add: min-def)
qed

lemma drop-replicate[simp]: drop \(i\) (replicate \(k\) \(x\)) = replicate \((k - i)\) \(x\)
proof (induct \(k\) arbitrary: \(i\))
  case (Suc \(k\))
  then show \(?case\)
    by (simp add: drop-Cons')
qed simp

lemma set-replicate-Suc: set (replicate (Suc \(n\)) \(x\)) = \{\(x\)\}
by (induct \(n\)) auto
lemma set-replicate [simp]: $n \neq 0 \Rightarrow \text{set} \ (\text{replicate} \ n \ x) = \{x\}$
by (fast dest!: not0-implies-Suc intro!: set-replicate-Suc)

lemma set-replicate-conv-if: $\text{set} \ (\text{replicate} \ n \ x) = (\text{if} \ n = 0 \ \text{then} \ \{\} \ \text{else} \ \{x\})$
by auto

lemma in-set-replicate [simp]: $(x \in \text{set} \ (\text{replicate} \ n \ y)) = (x = y \land n \neq 0)$
by (simp add: set-replicate-conv-if)

lemma card-set-1-iff-replicate:
\[ \text{card} \ (\text{set} \ xs) = \text{Suc} \ 0 \iff xs \neq [] \land (\exists x. xs = \text{replicate} \ (\text{length} \ xs) \ x) \]
by (metis card-1-singleton-iff empty-iff insert-iff replicate-eqI set-empty2 set-replicate)

lemma Ball-set-replicate [simp]:
\( (\forall x \in \text{set} \ (\text{replicate} \ n \ a). P x) = (P a \lor n=0) \)
by (simp add: set-replicate-conv-if)

lemma Bex-set-replicate [simp]:
\( (\exists x \in \text{set} \ (\text{replicate} \ n \ a). P x) = (P a \land n \neq 0) \)
by (simp add: set-replicate-conv-if)

lemma replicate-append-same:
\( \text{replicate} \ i \ x \ # \ [x] = x \neq \text{replicate} \ i \ x \)
by (induct i) simp-all

lemma map-replicate-trivial:
\( \text{map} \ (\lambda i. x) [0..<i] = \text{replicate} \ i \ x \)
by (induct i) (simp-all add: replicate-append-same)

lemma concat-replicate-trivial [simp]:
\( \text{concat} \ (\text{replicate} \ i \ []) = [] \)
by (induct i) (auto simp add: map-replicate-const)

lemma replicate-empty [simp]: $(\text{replicate} \ n \ x = []) \iff n=0$
by (induct n) auto

lemmas empty-replicate [simp] = replicate-empty[THEN eq_iff_swap]

lemma replicate-eq-replicate [simp]:
\( (\text{replicate} \ m \ x = \text{replicate} \ n \ y) \iff (m=n \land (m \neq 0 \Rightarrow x=y)) \)
proof (induct m arbitrary: n)
  case (Suc m n)
then show ?case
  by (induct n) auto
qed simp

lemma takeWhile-replicate [simp]:
takeWhile \ P \ (\text{replicate} \ n \ x) = (\text{if} \ P \ x \ \text{then} \ \text{replicate} \ n \ x \ \text{else} \ [])
using `takeWhile-eq-Nil-iff` by `fastforce`

**lemma** `dropWhile-replicate[simp]`:

\[
dropWhile P \ (\text{replicate} \ n \ x) = (\text{if} \ P \ x \ \text{then} \ [] \ \text{else} \ \text{replicate} \ n \ x)
\]

using `dropWhile-eq-self-iff` by `fastforce`

**lemma** `replicate-length-filter`:

\[
\text{replicate} \ (\text{length} \ \text{(filter} \ (\lambda y. \ x = y) \ xs)) \ x = \text{filter} \ (\lambda y. \ x = y) \ xs
\]

by `(induct xs)` auto

**lemma** `comm-append-are-replicate`:

\[
xs \ @ \ ys = ys \ @ \ xs \implies \exists \ m \ n \ zs. \ \text{concat} \ (\text{replicate} \ m \ zs) = xs \land \ \text{concat} \ (\text{replicate} \ n \ zs) = ys
\]

proof `(induction length (xs @ ys) + length xs arbitrary; xs ys rule: less-induct)`

case less

consider (1) `length ys < length xs` | (2) `xs = []` | (3) `length xs \leq length ys \land xs \neq []`

by `linarith`
then show ?case

proof `(cases)`

case 1

using `less.hyps[OF - less.prems[symmetric]] nat-add-left-cancel-less` by `auto`

next
case 2

then have `concat (\text{replicate} \ 0 \ ys) = xs \land \ \text{concat} (\text{replicate} \ 1 \ ys) = ys`

by `simp`
then show ?thesis

by `blast`

next
case 3

then have `length xs \leq length ys` and `xs \neq []`

by `blast`

from `\{length xs \leq length ys\}` and `\{xs @ ys = ys @ xs\}`

obtain `ws` where `ys = xs @ ws`

by `(auto simp: append-eq-append-conv2)`

from `this` and `\{xs \neq []\}`

have `length ws < length ys`

by `simp`

from `\{xs @ ys = ys @ xs\}` [unfolded `ys = xs @ ws`]

have `xs @ ws = ws @ xs`

by `simp`

from `less.hyps[OF - this]` `\{length ws < length ys\}`

obtain `m n' zs` where `concat (\text{replicate} \ m \ zs) = xs` and `concat (\text{replicate} \ n' \ zs) = ws`

by `auto`
then have `concat (\text{replicate} \ (m+n') \ zs) = ys`

using `\{ys = xs @ ws\}`

by `(simp add: replicate-add)`
then show \textit{thesis}
  using \texttt{(concat (replicate m zs) = xs)} \textbf{by blast}
qed

lemma \textit{comm-append-is-replicate}:
  fixes \texttt{xs ys :: 'a list}
  assumes \texttt{xs \neq [] \hspace{1em} ys \neq []}
  assumes \texttt{xs \@ ys = ys \@ xs}
  shows \texttt{\exists n zs. n > 1 \land concat (replicate n zs) = xs \@ ys}
proof –
  obtain \texttt{m n zs where concat (replicate m zs) = xs}
  and \texttt{concat (replicate n zs) = ys}
  using \textit{comm-append-are-replicate}[OF assms] \textbf{by blast}
  then have \texttt{m + n > 1 \hspace{1em} and concat (replicate (m+n) zs) = xs \@ ys}
  using \texttt{(xs \neq []) \hspace{1em} and (ys \neq [])}
  by (auto simp: replicate-add)
  then show \textit{thesis} \textbf{by blast}
qed

lemma \textit{Cons-replicate-eq}:
  \texttt{x \# xs = replicate n y \rightleftharpoons x = y \land n > 0 \land xs = replicate (n-1) x}
by (induct n) auto

lemma \textit{replicate-length-same}:
  \texttt{(\forall y \in set xs. y = x) \implies replicate (length xs) x = xs}
by (induct xs) simp-all

lemma \textit{foldr-replicate} [simp]:
  \texttt{foldr f (replicate n x) = f x ^^ n}
by (induct n) (simp-all)

lemma \textit{fold-replicate} [simp]:
  \texttt{fold f (replicate n x) = f x ^^ n}
by (subst foldr-fold [symmetric]) simp-all

66.2.9 \textit{enumerate}

lemma \textit{enumerate-simps} [simp, code]:
  \texttt{enumerate n [] = []}
  \texttt{enumerate n (x \# xs) = (n, x) \# enumerate (Suc n) xs}
by (simp-all add: enumerate-eq-zip upt-rec)

lemma \textit{length-enumerate} [simp]:
  \texttt{length (enumerate n xs) = length xs}
by (simp add: enumerate-eq-zip)

lemma \textit{map-fst-enumerate} [simp]:
  \texttt{map fst (enumerate n xs) = [n..<n + length xs]}
THEORY "List"

by (simp add: enumerate-eq-zip)

lemma map-snd-enumerate [simp]:
  map snd (enumerate n xs) = xs
by (simp add: enumerate-eq-zip)

lemma in-set-enumerate-eq:
  p ∈ set (enumerate n xs) ⟷ n ≤ fst p ∧ fst p < length xs + n ∧ nth xs (fst p - n) = snd p
proof -
  { fix m
    assume n ≤ m
    moreover assume m < length xs + n
    ultimately have [n..<n + length xs] ! (m - n) = m ∧
      xs ! (m - n) = xs ! (m - n) ∧ m - n < length xs by auto
    then have ∃ q. [n..<n + length xs] ! q = m ∧
      xs ! q = xs ! (m - n) ∧ q < length xs ..
  } then show ?thesis by (cases p) (auto simp add: enumerate-eq-zip in-set-zip)
qed

lemma nth-enumerate-eq: m < length xs ⇒ enumerate n xs ! m = (n + m, xs ! m)
by (simp add: enumerate-eq-zip)

lemma enumerate-replicate-eq:
  enumerate n (replicate m a) = map (λ q. (q, a)) [n..<n + m]
by (rule pair-list-eqI)
  (simp-all add: enumerate-eq-zip comp-def map-replicate-const)

lemma enumerate-Suc-eq:
  enumerate (Suc n) xs = map (apfst Suc) (enumerate n xs)
by (rule pair-list-eqI)
  (simp-all add: not-le, simp del: map-map add: map-Suc-upt map-map [symmetric])

lemma distinct-enumerate [simp]:
  distinct (enumerate n xs)
by (simp add: enumerate-eq-zip distinct-zipI1)

lemma enumerate-append-eq:
  enumerate n (xs @ ys) = enumerate n xs @ enumerate (n + length xs) ys
by (simp add: enumerate-eq-zip add.assoc zip-append2)

lemma enumerate-map-upt:
  enumerate n (map f [n..<m]) = map (λ k. f (Suc k)) [n..<m]
by (cases n ≤ m) (simp-all add: zip-map2 zip-same-conv-map enumerate-eq-zip)

66.2.10 rotate1 and rotate

lemma rotate0 [simp]: rotate 0 = id
by (simp add: rotate-def)

lemma rotate-Suc[simp]: rotate (Suc n) xs = rotate1 (rotate n xs)
by (simp add: rotate-def)

lemma rotate-add:
rotate (m+n) = rotate m o rotate n
by (simp add: rotate-def funpow-add)

lemma rotate-rotate: rotate m (rotate n xs) = rotate (m+n) xs
by (simp add: rotate-def)

lemma rotate1-map: rotate1 (map f xs) = map f (rotate1 xs)
by (cases xs) simp-all

lemma rotate1-rotate-swap: rotate1 (rotate n xs) = rotate n (rotate1 xs)
by (simp add: rotate-def funpow-swap1)

lemma rotate1-length01[simp]: length xs ≤ 1 ⇒ rotate1 xs = xs
by (cases xs) simp-all

lemma rotate-length01[simp]: length xs ≤ 1 ⇒ rotate n xs = xs
by (induct n) (simp-all add: rotate-def)

lemma rotate1-hd-tl: xs ≠ [] ⇒ rotate1 xs = tl xs @ [hd xs]
by (cases xs) simp-all

lemma rotate-drop-take:
rotate n xs = drop (n mod length xs) xs @ take (n mod length xs) xs
proof (induct n)
case (Suc n)
show ?case
proof (cases xs = [])
case False
then show ?thesis
proof (cases n mod length xs = 0)
case True
then show ?thesis
by (auto simp add: mod-Suc False Suc.hyps drop-Suc rotate1-hd-tl take-Suc Suc-length-conv)
next
case False
with ⟨xs ≠ []⟩ Suc
show ?thesis
by (simp add: rotate-def mod-Suc rotate1-hd-tl drop-Suc symmetric drop-tl symmetric
  take-hd-drop linorder-not-le)
qed
qed simp
qed simp
lemma rotate-conv-mod: rotate n xs = rotate (n mod length xs) xs
  by (simp add: rotate-drop-take)

lemma rotate-id[simp]: n mod length xs = 0 ⟹ rotate n xs = xs
  by (simp add: rotate-drop-take)

lemma length-rotate1[simp]: length(rotate1 xs) = length xs
  by (cases xs) simp-all

lemma length-rotate[simp]: length(rotate n xs) = length xs
  by (induct n arbitrary: xs) (simp-all add: rotate-def)

lemma distinct1-rotate[simp]: distinct(rotate1 xs) = distinct xs
  by (cases xs) auto

lemma distinct-rotate[simp]: distinct(rotate n xs) = distinct xs
  by (induct n) (simp-all add: rotate-def)

lemma rotate-map: rotate n (map f xs) = map f (rotate n xs)
  by (simp add: rotate-drop-take take-map drop-map)

lemma set-rotate1[simp]: set(rotate1 xs) = set xs
  by (cases xs) auto

lemma set-rotate[simp]: set(rotate n xs) = set xs
  by (induct n) (simp-all add: rotate-def)

lemma rotate1-replicate[simp]: rotate1 (replicate n a) = replicate n a
  by (cases n) (simp-all add: replicate-append-same)

lemma rotate1-is-Nil-conv[simp]: (rotate1 xs = []) = (xs = [])
  by (cases xs) auto

lemma rotate-is-Nil-conv[simp]: (rotate n xs = []) = (xs = [])
  by (induct n) (simp-all add: rotate-def)

lemma rotate-rev:
  rotate n (rev xs) = rev(rotate (length xs - (n mod length xs)) xs)
proof (cases length xs = 0 ∨ n mod length xs = 0)
  case False
  then show ?thesis
    by (simp add: rotate-drop-take rev-drop rev-take)
qed force

lemma hd-rotate-conv-nth:
  assumes xs ≠ [] shows hd(rotate n xs) = xs!(n mod length xs)
proof
  have n mod length xs < length xs
using assms by simp
then show ?thesis  
  by (metis drop-eq-nil hd-append2 hd-drop-conv-nth leD rotate-drop-take)
qed

lemma rotate-append: rotate (length l) (l @ q) = q @ l  
  by (induct l arbitrary: q) (auto simp add: rotate1-rotate-swap)

lemma nth-rotate:  
  rotate (length xs) (xs ! (m + n) mod length xs) if n < length xs
  using that apply (auto simp add: rotate-drop-take nth-append not-less less-diff-conv
  ac-simps dest: le-Suc-ex)
  apply (metis add.commute mod-add-right-eq mod-less)
  apply (metis (no-types, lifting) Nat.diff-diff-right add.commute add-diff-cancel-right'
  diff-le-self dual-order.strict-trans2 length-greater-0-conv less-nat-zero-code list.size(3)
  mod-add-right-eq mod-add-self2 mod-le-divisor mod-less)
  done

lemma nth-rotate1:  
  rotate1 xs ! n = xs ! (Suc n mod length xs) if n < length xs
  using that nth-rotate [of n xs 1] by simp

lemma inj-rotate1: inj rotate1  
proof
  fix xs ys :: 'a list show rotate1 xs = rotate1 ys =⇒ xs = ys
  by (cases xs; cases ys; simp)
qed

lemma surj-rotate1: surj rotate1  
proof (safe, simp-all)
  fix xs :: 'a list show xs ∈ range rotate1
  proof (cases xs rule: rev-exhaust)
    case Nil
    hence xs = rotate1 [] by auto
    thus ?thesis by fast
  next
    case (snoc as a)
    hence xs = rotate1 (a#as) by force
    thus ?thesis by fast
  qed

lemma bij-rotate1: bij (rotate1 :: 'a list ⇒ 'a list)  
using bijI inj-rotate1 surj-rotate1 by blast

lemma rotate1-fixpoint-card: rotate1 xs = xs =⇒ xs = [] ∨ card(set xs) = 1  
66.2.11  \textit{nths} — a generalization of (!) to sets

\textbf{lemma} \textit{nths-empty} [simp]: \textit{nths} \{\} = []
  by (auto simp add: \textit{nths-def})

\textbf{lemma} \textit{nths-nil} [simp]: \textit{nths} [] \ A = []
  by (auto simp add: \textit{nths-def})

\textbf{lemma} \textit{nths-all} \:\forall i < \text{length} \ xs. i \in I \implies \textit{nths} \ xs \ I = \xs
  apply (simp add: \textit{nths-def})
  apply (subst filter-True)
  apply (auto simp: in-set-zip subset_iff)
  done

\textbf{lemma} \textit{length-nths}:
  \text{length} (\textit{nths} \ xs \ I) = \text{card} \{i. i < \text{length} \ xs \land i \in I\}
  by (simp add: \textit{nths-def} length-filter-conv-card cong: conj-cong)

\textbf{lemma} \textit{nths-shift-lemma-Suc}:
  \text{map} \ \textbf{fst} (\text{filter} (\lambda p. \ P(Suc(\text{snd} \ p))) (\text{zip} \ xs \ is)) = \\
  \text{map} \ \textbf{fst} (\text{filter} (\lambda p. \ P(\text{snd} \ p)) (\text{zip} \ xs \ (\text{map} \ \textbf{Suc} \ is)))
  proof (induct \ xs \ arbitrary: is)
    case (Cons \ x \ xs \ is)
    then show ?case
      by (cases is) (auto simp add: Cons.hyps)
  qed simp

\textbf{lemma} \textit{nths-shift-lemma}:
  \text{map} \ \textbf{fst} (\text{filter} (\lambda p. \ \textbf{snd} \ p \in A) (\text{zip} \ xs \ [i..<i + \text{length} \ xs])) = \\
  \text{map} \ \textbf{fst} (\text{filter} (\lambda p. \ \text{snd} \ p + i \in A) (\text{zip} \ xs \ [0..<\text{length} \ xs]))
  by (induct \ xs \ rule: \text{rev-induct}) (simp-all add: \textit{add-commute})

\textbf{lemma} \textit{nths-append}:
  \textit{nths} (l \ @ \ l') \ A = \textit{nths} \ l \ A \ @ \textit{nths} \ l' \ \{j. j + \text{length} \ l \in A\}
  unfolding \textit{nths-def}
  proof (induct \ l' \ rule: \text{rev-induct})
    case (snoc \ x \ xs)
    then show ?case
      by (simp add: \textit{upt-add-eq-append[of 0]} \textit{nths-shift-lemma} add\textit{.commute})
  qed auto

\textbf{lemma} \textit{nths-Cons}:
  \textit{nths} (x \ # \ l) \ A = (if \theta \in A \ then \ [x] \ else [] \) \ @ \textit{nths} \ l \ \{j. \ Suc \ j \in A\}
  proof (induct \ l \ rule: \text{rev-induct})
    case (snoc \ x \ xs)
    then show ?case
      by (simp flip: \textit{append-Cons} add: \textit{nths-append})
  qed (auto simp: \textit{nths-def})

\textbf{lemma} \textit{nths-map}:
  \textit{nths} (\textit{map} \ f \ \xs) \ I = \textit{map} \ f (\textit{nths} \ xs \ I)
THEORY "List"

by (induction xs arbitrary: I) (simp-all add: nths-Cons)

lemma set-nths: set(nths xs I) = \{xs!i\mid i<size xs \land i \in I\}
  by (induct xs arbitrary: I) (auto simp: nths-Cons nth-Cons split:nat.split dest!: gr0-implies-Suc)

lemma set-nths-subset: set(nths xs I) \subseteq set xs
  by (auto simp add: set-nths)

lemma notin-set-nthsI [simp]: x \notin set xs \implies x \notin set(nths xs I)
  by (auto simp add: set-nths)

lemma in-set-nthsD: x \in set(nths xs I) \implies x \in set xs
  by (auto simp add: set-nths)

lemma nths-singleton [simp]: nths [x] A = (if 0 \in A then [x] else [])
  by (simp add: nths-Cons)

lemma distinct-nthsI [simp]: distinct xs = \implies distinct (nths xs I)
  by (induction xs arbitrary: I) (auto simp add: nths-Cons)

lemma nths-upt-eq-take [simp]: nths l {..<n} = take n l
  by (induct l rule: rev-induct) (simp-all split:nat-diff-split add: nths-append)

lemma nths-nths: nths(nths xs A) B = nths xs \{i \in A. \exists j \in B. card\{i' \in A. i'<i\} = j\}
  by (induction xs arbitrary: A B) (auto simp add: nths-Cons card-less-Suc card-less-Suc2)

lemma drop-eq-nths: drop n xs = nths xs \{i. i \geq n\}
  by (induct xs arbitrary: n) (auto simp add: nths-Cons simp flip: atLeastLessThan-iff
    intro: arg-cong2[where f=nths, OF refl])

lemma nths-drop: nths (drop n xs) I = nths xs ((+ n) \cdot I)
  by (force simp: drop-eq-nths nths-nths simp flip: atLeastLessThan-iff
    intro: arg-cong2[where f=nths, OF refl])

lemma filter-eq-nths: filter P xs = nths xs \{i. i<length xs \land P(xs!i)\}
  by (induction xs) (auto simp: nths-Cons)

lemma filter-in-nths:
  distinct xs \implies filter (%x. x \in set (nths xs s)) xs = nths xs s
proof (induct xs arbitrary: s)
  case Nil thus \case by simp
next
  case (Cons a xs)
  then have \forall x. x \in set xs \implies x \neq a \by auto
  with Cons show \case by (simp add: nths-Cons cong:filter-cong)
qed
66.2.12 subseqs and List.n-lists

**lemma** length-subseqs: length (subseqs xs) = 2 ^ length xs
  by (induct xs) (simp-all add: Let-def)

**lemma** subseqs-powset: set ' set (subseqs xs) = Pow (set xs)
  proof
    have aux: \( \forall x A. \) set ' Cons x ' A = insert x ' set ' A
      by (auto simp add: image-def)
    have set (map set (subseqs xs)) = Pow (set xs)
      by (induct xs) (simp-all add: aux Let-def Pow-insert Un-commute comp-def del: map-map)
    then show ?thesis by simp
  qed

**lemma** distinct-set-subseqs:
  assumes distinct xs
  shows distinct (map set (subseqs xs))
  by (simp add: assms card-Pow card-distinct distinct-card length-subseqs subseqs-powset)

**lemma** n-lists-Nil [simp]: List.n-lists n [] = (if n = 0 then [[]] else [])
  by (induct n) simp-all

**lemma** length-n-lists-elem: ys \in set (List.n-lists n xs) \implies length ys = n
  by (induct n arbitrary: ys) auto

**lemma** set-n-lists: set (List.n-lists n xs) = \{ys. length ys = n \& set ys \subseteq set xs\}
  proof (rule set-eqI)
    fix ys :: 'a list
    show ys \in set (List.n-lists n xs) \iff ys \in \{ys. length ys = n \& set ys \subseteq set xs\}
      proof
        have ys \in set (List.n-lists n xs) \implies length ys = n
          by (induct n arbitrary: ys) auto
        moreover have \( \forall x. \) ys \in set (List.n-lists n xs) \implies x \in set ys \implies x \in set xs
          by (induct n arbitrary: ys) auto
        moreover have set ys \subseteq set xs \implies ys \in set (List.n-lists (length ys) xs)
          by (induct ys) auto
        ultimately show ?thesis by auto
      qed
  qed

**lemma** subseqs-refl: xs \in set (subseqs xs)
  by (induct xs) (simp-all add: Let-def)

**lemma** subset-subseqs: X \subseteq set xs \implies X \in set ' set (subseqs xs)
  unfolding subseqs-powset by simp

**lemma** Cons-in-subseqsD: y \# ys \in set (subseqs xs) \implies ys \in set (subseqs xs)
  by (induct xs) (auto simp: Let-def)
lemma subseqs-distinctD: \[ ys \in \text{set} (\text{subseqs} \; xs); \; \text{distinct} \; xs \] \implies \text{distinct} \; ys

proof (induct xs arbitrary: ys)
  case (Cons x xs ys)
  then show ?case by (auto simp: Let-def) (metis Pow-iff contra-subsetD image-eqI subseqs-powset)
qed simp

66.2.13 splice

lemma splice-Nil2 [simp]: \( \text{splice} \; xs \; [] = \; xs \)
  by (cases xs) simp-all

lemma length-splice [simp]: \( \text{length} \; (\text{splice} \; xs \; ys) = \text{length} \; xs + \text{length} \; ys \)
  by (induct xs ys rule: splice.induct) auto

lemma split-Nil-iff [simp]: \( \text{splice} \; xs \; ys = \; [] \iff xs = \; [] \land ys = \; [] \)
  by (induct xs ys rule: splice.induct) auto

lemma splice-replicate [simp]: \( \text{splice} \; (\text{replicate} \; m \; x) \; (\text{replicate} \; n \; x) = \text{replicate} \; (m+n) \; x \)

proof (induction replicate replicate arbitrary: m n rule: splice.induct)
  case (2 x xs)
  then show ?case by (auto simp add: Cons-replicate-eq dest: gr0-implies-Suc)
qed auto

66.2.14 shuffles

lemma shuffles-commutes: \( \text{shuffles} \; xs \; ys = \text{shuffles} \; ys \; xs \)
by (induction xs ys rule: shuffles.induct) (simp-all add: Un-commute)

lemma Nil-in-shuffles [simp]: \( [] \in \text{shuffles} \; xs \; ys \iff xs = \; [] \land ys = \; [] \)
  by (induct xs ys rule: shuffles.induct) auto

lemma shufflesE:
  \( zs \in \text{shuffles} \; xs \; ys \implies \)
  \( (zs = zs \implies ys = [] \implies P) \implies \)
  \( (zs = ys \implies xs = [] \implies P) \implies \)
  \( (\forall xs' \; zs'. \; xs = x \# xs' \implies zs = z \# zs' \implies x = z \implies zs' \in \text{shuffles} \; xs') \)
  \( ys \implies P) \implies \)
  \( (\forall y \; ys' \; zs'. \; ys = y \# ys' \implies zs = z \# zs' \implies y = z \implies zs' \in \text{shuffles} \; xs \)
  \( ys' \implies P) \implies P \)
  by (induct xs ys rule: shuffles.induct) auto

lemma Cons-in-shuffles-iff:
  \( z \# zs \in \text{shuffles} \; xs \; ys \iff \)
  \( (xs \neq [] \land \text{hd} \; xs = z \land zs \in \text{shuffles} \; (\text{tl} \; xs) \; ys \lor \)
  \( ys \neq [] \land \text{hd} \; ys = z \land zs \in \text{shuffles} \; (\text{tl} \; ys) \)
  by (induct xs ys rule: shuffles.induct) auto
lemma splice-in-shuffles [simp, intro]: splice xs ys ∈ shuffles xs ys
    by (induction xs ys rule: splice.induct) (simp-all add: Cons-in-shuffles-iff shuffles-commutes)

lemma Nil-in-shufflesI: xs = [] ⇒ ys = [] ⇒ [] ∈ shuffles xs ys
    by simp

lemma Cons-in-shuffles-leftI: zs ∈ shuffles xs ys ⇒ z # zs ∈ shuffles (z # xs) ys
    by (cases ys) auto

lemma Cons-in-shuffles-rightI: zs ∈ shuffles xs ys ⇒ z # zs ∈ shuffles xs (z # ys)
    by (cases xs) auto

lemma finite-shuffles [simp, intro]: finite (shuffles xs ys)
    by (induction xs ys rule: shuffles.induct) simp-all

lemma length-shuffles: zs ∈ shuffles xs ys ⇒ length zs = length xs + length ys
    by (induction xs ys arbitrary: zs rule: shuffles.induct) auto

lemma set-shuffles: zs ∈ shuffles xs ys ⇒ set zs = set xs ∪ set ys
    by (induction xs ys arbitrary: zs rule: shuffles.induct) auto

lemma distinct-disjoint-shuffles:
    assumes distinct xs distinct ys set xs ∩ set ys = {0} zs ∈ shuffles xs ys
    shows distinct zs
    using assms
    proof (induction xs ys arbitrary: zs rule: shuffles.induct)
        case (3 x xs y ys)
        show ?case
        proof (cases zs)
            case (Cons z zs')
            with 3.prems and 3.IH[of zs'] show ?thesis by (force dest: set-shuffles)
        qed
        qed simp-all
    qed simp-all

lemma Cons-shuffles-subset1: (#) x ' shuffles xs ys ⊆ shuffles (x # x) ys
    by (cases ys) auto

lemma Cons-shuffles-subset2: (#) y ' shuffles xs ys ⊆ shuffles xs (y # ys)
    by (cases xs) auto

lemma filter-shuffles:
    filter P ' shuffles xs ys = shuffles (filter P xs) (filter P ys)
    proof
        have *: filter P ' (##) x ' A = (if P x then (##) x ' filter P ' A else filter P ' A) for x A
by (auto simp: image-image)
show ?thesis
by (induction xs ys rule: shuffles.induct)
  (simp-all split: if-splits add: image-Un * Un-absorb1 Un-absorb2
   Cons-shuffles-subset1 Cons-shuffles-subset2)
qed

lemma filter-shuffles-disjoint1:
  assumes set xs ⊓ set ys = {} zs ∈ shuffles xs ys
  shows filter (λx. x ∈ set xs) zs = xs (is filter ?P = -)
  and filter (λx. x ∉ set xs) zs = ys (is filter ?Q = -)
  using assms
proof –
  from assms have filter ?P zs ∈ filter ?P ' shuffles xs ys
    by blast
  also have filter ?P ' shuffles xs ys = shuffles (filter ?P xs) (filter ?P ys)
    by (rule filter-shuffles)
  also have filter ?P xs = xs by (rule filter-True) simp-all
  also have filter ?P ys = [] by (rule filter-False) (insert assms(1), auto)
  also have shuffles xs [] = [xs] by simp
  finally show filter ?P zs = xs by simp
next
  from assms have filter ?Q zs ∈ filter ?Q ' shuffles xs ys by blast
  also have filter ?Q ' shuffles xs ys = shuffles (filter ?Q xs) (filter ?Q ys)
    by (rule filter-shuffles)
  also have filter ?Q ys = ys by (rule filter-True) (insert assms(1), auto)
  also have filter ?Q xs = [] by (rule filter-False) (insert assms(1), auto)
  also have shuffles [] ys = [ys] by simp
  finally show filter ?Q zs = ys by simp
qed

lemma filter-shuffles-disjoint2:
  assumes set xs ⊓ set ys = {} zs ∈ shuffles xs ys
  shows filter (λx. x ∈ set ys) zs = ys
  using filter-shuffles-disjoint1[of ys xs zs] assms
  by (simp-all add: shuffles-commutes Int-commute)

lemma partition-in-shuffles:
  xs ∈ shuffles (filter P xs) (filter (λx. ¬P x) xs)
proof (induction xs)
  case (Cons x xs)
  show ?case
  proof (cases P x)
    case True
    hence x # xs ∈ (#) x ' shuffles (filter P xs) (filter (λx. ¬P x) xs)
      by (intro imageI Cons.IH)
    also have ... ⊆ shuffles (filter P (x # xs)) (filter (λx. ¬P x) (x # xs))
      by (simp add: True Cons-shuffles-subset1)
    finally show ?thesis .
  next
case False
hence $x \not\in \text{shuffles} (\text{filter } P \, x \, \text{ss}) \text{ (filter } (\lambda x. \neg P \, x) \, x\text{)}$
by (intro imageI Cons.IH)
also have $\ldots \subseteq \text{shuffles} (\text{filter } P (x \# \text{ss})) \text{ (filter } (\lambda x. \neg P \, x) (x \# \text{ss})\text{)}$
by (simp add: False Cons-shuffles-subset2)
finally show ?thesis.
qed

lemma inv-image-partition:
assumes $\forall x. x \in \text{set } x \Rightarrow P \, x$
$\forall y. y \in \text{set } y \Rightarrow \neg P \, y$
shows $\text{partition } P \sim \{ (x, y) \} = \text{shuffles } x \, y$
proof (intro equalityI subsetI)
fix $zs$
assume $zs \in \text{shuffles } x \, y$
hence $\text{set } zs = \text{set } x \cup \text{set } y$
by (rule set-shuffles)
from assms have $\text{filter } P \, zs = \text{filter } (\lambda x. x \in \text{set } x) \, zs$
$\text{filter } (\lambda x. \neg P \, x) \, zs = \text{filter } (\lambda x. x \in \text{set } y) \, zs$
by (intro filter-cong refl; force)+
moreover from assms have $\text{set } x \cap \text{set } y = \{ \}$
by auto
ultimately show $zs \in \text{partition } P \sim \{ (x, y) \}$
using zs
by (simp add: o-def filter-shuffles-disjoint1 filter-shuffles-disjoint2)
next
fix $zs$
assume $zs \in \text{partition } P \sim \{ (x, y) \}$
thus $zs \in \text{shuffles } x \, y$
using partition-in-shuffles[of zs]
by (auto simp: o-def)
qed

66.2.15 Transpose

function transpose where
$\text{transpose } [] = []$
$\text{transpose } ([\ ] \# \text{ss}) = \text{transpose } \text{ss} \ |
\text{transpose } ((x\#\text{ss}) \# \text{ss}) = (x \# [h. (h\#t) \leftarrow \text{ss}]) \# \text{transpose } (x \# [t. (h\#t) \leftarrow \text{ss}])$
by pat-completeness auto

lemma transpose-aux-filter-head:
$\text{concat } (\text{map } (\text{case-list } [] (\lambda h. [h])) \, \text{ss}) = \text{map } (\lambda x. \text{hd } x) \, (\text{filter } (\lambda y. y \not\in [] \, \text{ss}) \, x)$
by (induct xss) (auto split: list.split)

lemma transpose-aux-filter-tail:
$\text{concat } (\text{map } (\text{case-list } [] (\lambda h. [t])) \, \text{ss}) = \text{map } (\lambda x. \text{tl } x) \, (\text{filter } (\lambda y. y \not\in [] \, \text{ss}) \, x)$
by (induct xss) (auto split: list.split)

lemma transpose-aux-max:
$\text{max } (\text{Suc } (\text{length } \text{ss})) \, (\text{foldr } (\lambda x. \text{max } (\text{length } \text{ss}) \, \text{ss}) \, 0) = \text{Suc } (\text{max } (\text{length } \text{ss}) \, (\text{foldr } (\lambda x. \text{max } (\text{length } x \sim \text{Suc } 0) \, (\text{filter } (\lambda y. y \not\in [] \, \text{ss}) \, 0))))$
THEORY "List"

(is max - ?foldB = Suc (max - ?foldA))

proof (cases (filter (λys. ys ≠ []) xss) = [])

  case True
  hence foldr (λxs. max (length xs)) xss 0 = 0
  proof (induct xss)
    case (Cons x xs)
    then have x = [] by (cases x) auto
    with Cons show ?case by auto
    qed simp
  thus ?thesis using True by simp

  next
  case False

  have foldA: ?foldA = foldr (λx. max (length x)) (filter (λys. ys ≠ []) xss) 0 - 1
  by (induct zss) auto

  have foldB: ?foldB = foldr (λx. max (length x)) (filter (λys. ys ≠ []) xss) 0
  by (induct zss) auto

  have 0 < ?foldB
  proof -
    from False
    obtain z zs where zs: (filter (λys. ys ≠ []) xss) = z#zs by (auto simp: neq-Nil-conv)
    hence z ∈ set (filter (λys. ys ≠ []) xss) by auto
    hence z ≠ [] by auto
    thus ?thesis unfolding foldB zs
    by (auto simp: max-def intro: less-le-trans)
    qed
  thus ?thesis unfolding foldA foldB max-Suc-Suc[symmetric] by simp
  qed

termination transpose
by (relation measure (λxs. foldr (λxs. max (length xs)) xs 0 + length xs))
  (auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max
   less-Suc-eq-le)

lemma transpose-empty: (transpose xs = []) ↔ (∀x ∈ set xs. x = [])
by (induct rule: transpose.induct) simp-all

lemma length-transpose:
  fixes xs :: 'a list list
  shows length (transpose xs) = foldr (λxs. max (length xs)) xs 0
  by (induct rule: transpose.induct)
    (auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max
     max-Suc-Suc[symmetric] simp del: max-Suc-Suc)
lemma nth-transpose:
fixes xs :: 'a list list
assumes i < length (transpose xs)
shows transpose xs ! i = map (λxs. xs ! i) (filter (λys. i < length ys) xs)
using assms proof (induct arbitrary: i rule: transpose.induct)
case (3 x xs xss)
define XS where XS = (x # xs) # xss
hence [simp]: XS ≠ [] by auto
thus ?case proof (cases i)
case 0
thus ?thesis by (simp add: transpose-aux-filter-head hd-conv-nth)
next
case (Suc j)
have ∗: \(\forall xss. xss ≠ [] \Rightarrow \exists xs. xs # map tl xss = map tl \((x#xs)#xss)\) by simp
have ∗∗: \(\forall xss. xss ≠ [] \Rightarrow \exists xs. \map{filter} {(λys. ys ≠ [])} xss = \map{filter} {(λys. ys ≠ [])} \((x#xs)#xss)\) by simp
{ fix xs :: ('a list) have Suc j < length xs ∧ j < length xs − Suc 0 by (cases xs) simp-all
  } note ∗∗∗ = this
have j-less: j < length (transpose (xs # concat (\map{map f} \((\case-list [] (λh t. [t]))\)) xss)))
  using 3.prems by (simp add: transpose-aux-filter-tail length-transpose Suc)
show ?thesis unfolding transpose.simps < i = Suc j; nth-Cons-Suc 3.hyps[OF j-less]
  apply (auto simp: transpose-aux-filter-tail filter-map comp-def length-transpose
  ∗ ∗ ∗ XS-def[ symmetric ])
  by (simp add: nth-tl)
qed
qed simp-all

lemma transpose-map-map:
transpose (map (map f) xs) = map (map f) (transpose xs)
proof (rule nth-equalityI)
  have [simp]: length (transpose (map (map f) xs)) = length (transpose xs)
  by (simp add: length-transpose foldr-map comp-def)
  show length (transpose (map (map f) xs)) = length (map (map f) (transpose xs))
  by simp

  fix i assume i < length (transpose (map (map f) xs))
  thus transpose (map (map f) xs) ! i = map (map f) (transpose xs) ! i
  by (simp add: nth-transpose filter-map comp-def)
qed

66.2.16 min and arg-min

lemma min-list-Min: xs ≠ [] ⟹ min-list xs = Min (set xs)
by (induction xs rule: induct-list012) (auto)

lemma f-arg-min-list-f: \( \text{xs} \neq [] \implies f \text{ (arg-min-list f xs)} = \text{Min} \ (f \ ('(\text{set} \ \text{xs}))) \)
by (induction f xs rule: arg-min-list.induct) (auto simp: min_def intro: antisym)

lemma arg-min-list-in: \( \text{xs} \neq [] \implies \text{arg-min-list f xs} \in \text{set} \ \text{xs} \)
by (induction xs rule: induct-list012) (auto simp: Let-def)

66.2.17 (In)finiteness

lemma finite-list-length: finite \{\text{xs}: ('a::finite) list. length \text{xs} = \text{n}\}
proof (induction \text{n})
  case (Suc \text{n})
  have \{\text{xs}: 'a list. length \text{xs} = \text{Suc} \text{n}\} = (\bigcup \text{x}. (\#) \text{x} : \{\text{xs. length} \text{xs} = \text{n}\})
  by (auto simp: length-Suc-conv)
  then show \?case using Suc by simp
qed simp

lemma finite-maxlen:
  finite \{\text{M: ('a list set)} \implies \exists \text{n. } \forall \text{s} \in \text{M. size} \text{s} < \text{n}\}
proof (induct rule: finite.induct)
  case emptyI
  show \?case by simp
next
  case (insertI \text{M} \text{xs})
  then obtain \text{n} where \(\forall \text{s} \in \text{M. length} \text{s} < \text{n}\) by blast
  hence \(\forall \text{s} \in \text{insert} \text{xs} \text{M. size} \text{s} < \text{max} \text{n} \ (\text{size} \text{xs}) + 1\) by auto
  thus \?case ..
qed

lemma lists-length-Suc-eq:
\{\text{xs. set} \text{xs} \subseteq \text{A} \land length \text{xs} = \text{Suc} \text{n}\} =
(\lambda (\text{xs, n}). \text{n#xs} \ ' \ (\{\text{xs. set} \text{xs} \subseteq \text{A} \land length} \text{xs} = \text{n}\} \times \text{A})
by (auto simp: length-Suc-conv)

lemma assumes finite \text{A}
shows finite-lists-length-eq: finite \{\text{xs. set} \text{xs} \subseteq \text{A} \land length} \text{xs} = \text{n}\}
and card-lists-length-eq: card \{\text{xs. set} \text{xs} \subseteq \text{A} \land length} \text{xs} = \text{n}\} = (\text{card} \text{A})^\text{n}
using finite \text{A}
by (induct \text{n})
(auto simp: card-image inj-split-Cons lists-length-Suc-eq cong: conj-cong)

lemma finite-lists-length-le:
assumes finite \text{A}
shows finite \{\text{xs. set} \text{xs} \subseteq \text{A} \land length} \text{xs} \leq \text{n}\}
(is finite \?S)
proof -
  have \?S = (\bigcup n\in\{0..\text{n}\}. \{\text{xs. set} \text{xs} \subseteq \text{A} \land length} \text{xs} = \text{n}\}) by auto
  thus \?thesis by (auto intro: finite-lists-length-eq[OF finite \text{A}] simp only:)
qed
lemma card-lists-length-le:
  assumes finite A shows card \{xs. set xs \subseteq A \land length xs \leq n\} = (\sum i \leq n. \text{card } A^i)
proof
  have (\sum i \leq n. \text{card } A^i) = \text{card } (\bigcup i \leq n. \{xs. set xs \subseteq A \land length xs = i\})
    using finite A
    by (subst card-UN-disjoint)
    (auto simp add: card-lists-length-eq finite-lists-length-eq)
  also have (\bigcup i \leq n. \{xs. set xs \subseteq A \land length xs = i\}) = \{xs. set xs \subseteq A \land length xs \leq n\}
    by auto
  finally show \?thesis by simp
qed

lemma finite-subset-distinct:
  assumes finite A shows finite \{xs. set xs \subseteq A \land distinct xs\} (is finite \?S)
proof (rule finite-subset)
  from assms show \?S \subseteq \{xs. set xs \subseteq A \land length xs \leq \text{card } A\}
    by (clarsimp)
  from assms show finite ...
    by (rule finite-lists-length-le)
qed

lemma card-lists-distinct-length-eq:
  assumes finite A k \leq \text{card } A
  shows card \{xs. length xs = k \land distinct xs \land set xs \subseteq A\} = \prod \{\text{card } A - k + 1 \ldots \text{card } A\}
using assms
proof (induct k)
  case 0
  then have \{xs. length xs = 0 \land distinct xs \land set xs \subseteq A\} = \{[]\}
    by auto
  then show \?case by simp
next
  case (Suc k)
  let \?k-list = \lambda xs. length xs = k \land distinct xs \land set xs \subseteq A
  have inj-Cons: \bigwedge A. inj-on (\lambda(xs, n). n \# xs) A
    by (rule inj-onI)
proof
  from Suc have k \leq \text{card } A by simp
  moreover note \langle finite A \rangle
  moreover have finite \{xs. \?k-list k xs\}
    by (rule finite-subset) (use finite-lists-length-eq[OF \langle finite A \rangle, of k] in auto)
  moreover have \bigwedge i. i \neq j \longrightarrow \{i\} \times (A - set i) \cap \{j\} \times (A - set j) = \{
    by auto
  moreover have \bigwedge i. i \in \{xs. \?k-list k xs\} \Longrightarrow \text{card } (A - set i) = \text{card } A - k
    by (simp add: card-Diff-subset distinct-card)
  moreover have \{xs. \?k-list (Suc k) xs\} =
    \langle\lambda(xs, n). n \# xs\rangle \cup ((\lambda(xs. \{xs\} \times (A - set xs)) \cup \{xs. \?k-list k xs\})
    by (auto simp: length-Suc-conv)
moreover have Suc (card A - Suc k) = card A - k using Suc.prems by simp
then have (card A - k) * \prod{k} (Suc (card A - k) .. card A) = \prod{k} (Suc (card A - Suc k) .. card A).
  by (subst prod.insert[symmetric]) (simp add: atLeastAtMost-insertL)+
ultimately show ?thesis
  by (simp add: card-image inj-Cons card-UN-disjoint Suc.hyps algebra-simps)
qed

lemma card-lists-distinct-length-eq':
  assumes k < card A
  shows card {xs. length xs = k ∧ distinct xs ∧ set xs ⊆ A} = \prod{k+1} (Suc (card A - k) .. card A)
proof -
  from k < card A have finite A and k ≤ card A using card.infinite by force+
  from this show ?thesis by (rule card-lists-distinct-length-eq)
qed

lemma infinite-UNIV-listI: ¬ finite(UNIV::'a list set)
  by (metis UNIV-I finite-maxlen length-replicate less-irrefl)

lemma same-length-different:
  assumes xs ≠ ys and length xs = length ys
  shows ∃ pre x xs' y ys'. x≠y ∧ xs = pre @ [x] @ xs' ∧ ys = pre @ [y] @ ys'
using assms
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case by auto
next
case (Cons x xs)
  then obtain z zs where ys = Cons z zs
    by (metis length-Suc-conv)
  show ?case
  proof (cases x=z)
    case True
    then have xs ≠ zs length xs = length zs
      using Cons.prems ys by auto
    then obtain pre u xs' v ys' where u≠v and xs: xs = pre @ [u] @ xs' and zs:
      zs = pre @ [v] @ ys'
      using Cons.IH by meson
    then have x ≠ xs = (z≠pre) @ [u] @ xs' ∧ ys = (z≠pre) @ [v] @ ys'
      by (simp add: True ys)
    with u≠v show ?thesis
      by blast
  next
case False
    then have x ≠ xs = [] @ [x] @ xs ∧ ys = [] @ [z] @ zs
      by (simp add: ys)
    then show ?thesis
      using False by blast
66.3 Sorting

66.3.1 sorted-wrt

Sometimes the second equation in the definition of \textit{sorted-wrt} is too aggressive because it relates each list element to \textit{all} its successors. Then this equation should be removed and \textit{sorted-wrt2-simps} should be added instead.

\textbf{lemma} \textit{sorted-wrt1}: \textit{sorted-wrt} \( P \left[ x \right] = True \)
\textbf{by} (simp)

\textbf{lemma} \textit{sorted-wrt2}: \textit{transp} \( P \rightarrow\) \textit{sorted-wrt} \( P (x \ # y \ # zs) = (P x y \land \textit{sorted-wrt} P (y \ # zs)) \)
\textbf{proof} (induction \( zs \) arbitrary; \( x y \))
\textbf{case} (Cons \( z zs \))
\textbf{then show} ?case
\textbf{by} simp (meson transpD)+
\textbf{qed} auto

\textbf{lemmas} \textit{sorted-wrt2-simps} = \textit{sorted-wrt1} \textit{sorted-wrt2}

\textbf{lemma} \textit{sorted-wrt-true} [simp]:
\textit{sorted-wrt} \( (\lambda - . \ True) \) \( xs \)
\textbf{by} (induction \( xs \)) simp-all

\textbf{lemma} \textit{sorted-wrt-append}:
\textit{sorted-wrt} \( P (xs @ ys) \rightarrow\) \textit{sorted-wrt} \( P \) \( xs \land \textit{sorted-wrt} \( P \) \( ys \land (\forall x \in \textit{set} \) \( xs \). \( \forall y \in \textit{set} \) \( ys \). \( P \) \( x \) \( y \)) \)
\textbf{by} (induction \( xs \)) auto

\textbf{lemma} \textit{sorted-wrt-map}:
\textit{sorted-wrt} \( R (\textit{map} f \) \( xs \)) = \textit{sorted-wrt} \( (\lambda x y. R (f x) (f y)) \) \( xs \)
\textbf{by} (induction \( xs \)) simp-all

\textbf{lemma} \textbf{assumes} \textit{sorted-wrt f xs}
\textbf{shows} \textit{sorted-wrt-take}: \textit{sorted-wrt} \( f \) \( \text{take} n \) \( xs \)
\textbf{and} \textit{sorted-wrt-drop}: \textit{sorted-wrt} \( f \) \( \text{drop} n \) \( xs \)
\textbf{proof} –
\textbf{from} \textbf{assms} have \textit{sorted-wrt f} \( \text{take} n \) \( xs \) \( @ \) \( \text{drop} n \) \( xs \) \textbf{by} simp
\textbf{thus} \textit{sorted-wrt f} \( \text{take} n \) \( xs \) \textbf{and} \textit{sorted-wrt f} \( \text{drop} n \) \( xs \)
\textbf{unfolding} \textit{sorted-wrt-append} \textbf{by} simp-all
\textbf{qed}

\textbf{lemma} \textit{sorted-wrt-filter}:
\textit{sorted-wrt} \( f \) \( xs \rightarrow\) \textit{sorted-wrt} \( f \) \( \text{filter} P \) \( xs \)
\textbf{by} (induction \( xs \)) auto
lemma sorted-wrt-rev:
  sorted-wrt \( P \) (rev xs) = sorted-wrt (\( \lambda x y. P y x \) ) xs
by (induction xs) (auto simp add: sorted-wrt-append)

lemma sorted-wrt-mono-rel:
  \( (\forall x y. [ x \in \text{set} \ xs; y \in \text{set} \ xs; P x y ] \implies Q x y) \implies \text{sorted-wrt} \ P \ xs \implies \text{sorted-wrt} \ Q \ xs \)
by (induction xs) (auto)

lemma sorted-wrt01: length xs \( \leq 1 \implies \text{sorted-wrt} \ P \ xs \)
by (auto simp: le-Suc-eq length-Suc-conv)

lemma sorted-wrt-iff-nth-less:
  \( \text{sorted-wrt} \ P \ xs = (\forall i j. i < j \implies j < \text{length} \ xs \implies P (xs ! i) (xs ! j)) \)
by (induction xs) (auto simp add: \( \cdot \) in-set-conv-nth Ball-def nth-Cons split: nat.split)

lemma sorted-wrt-iff-nth-Suc-transp: assumes \( \text{transp} \ P \) shows \( \text{sorted-wrt} \ P \ xs \iff (\forall i. \text{Suc} i < \text{length} \ xs \implies P (xs!i) (xs! (\text{Suc} i))) \) (is \( ?L = ?R \))
proof
  assume \( ?L \)
  thus \( ?R \)
  by (simp add: sorted-wrt-iff-nth-less)
next
  assume \( ?R \)
  have \( i < j \implies j < \text{length} \ xs \implies P (xs ! i) (xs ! j) \) for \( i j \)
  by (induct i j rule: less-Suc-induct) (simp add: \( ?R \), meson assms \( \text{transpE} \) transp-on-less)
  thus \( ?L \)
  by (simp add: sorted-wrt-iff-nth-less)
qed

lemma sorted-wrt-upt[simp]: \( \text{sorted-wrt} \ (<) \ [m..<n] \)
by (induction n) (auto simp: sorted-wrt-append)

lemma sorted-wrt-upto[simp]: \( \text{sorted-wrt} \ (<) \ [i..j] \)
proof (induct i j rule: upto.induct)
  case (1 i j)
  from this show \( ?case \)
  unfolding upto.simps[of i j] by auto
qed

Each element is greater or equal to its index:

lemma sorted-wrt-less-idx:
sorted-wrt \( (<) \) \( ns \mapsto i < \text{length}\ ns \mapsto i \leq \text{ns}\!i \)

**proof** (induction \( ns \) arbitrary; \( i \) rule: rev-induct)
- **case** Nil
  - **thus** \( ?\text{case} \) by simp

**next**
- **case** snoc
  - **thus** \( ?\text{case} \)
    - by (simp add: nth-append sorted-wrt-append)
    - (metis less-antisym not-less nth-mem)

qed

### 66.3.2 sorted

**context** linorder

**begin**

Sometimes the second equation in the definition of \( \text{sorted} \) is too aggressive because it relates each list element to all its successors. Then this equation should be removed and \( \text{sorted2-simps} \) should be added instead. Executable code is one such use case.

**lemma** sorted0: \( \text{sorted} \ [\ ] = \text{True} \)
  - by simp

**lemma** sorted1: \( \text{sorted} \ [x] = \text{True} \)
  - by simp

**lemma** sorted2: \( \text{sorted} \ (x \# y \# zs) = (x \leq y \land \text{sorted} \ (y \# zs)) \)
  - by auto

**lemmas** sorted2-simps = sorted1 sorted2

**lemma** sorted-append:
  \( \text{sorted} \ (xs \@ ys) = (\text{sorted} \ xs \land \text{sorted} \ ys \land (\forall x \in \text{set} \ xs. \forall y \in \text{set} \ ys. x \leq y)) \)
  - by (simp add: sorted-wrt-append)

**lemma** sorted-map:
  \( \text{sorted} \ \text{(map} \ f \ \text{xs}) = \text{sorted-wrt} \ \text{(\lambda} x \ y. \ f \ x \leq f \ y) \ \text{xs} \)
  - by (simp add: sorted-wrt-map)

**lemma** sorted01: \( \text{length} \ xs \leq 1 \implies \text{sorted} \ xs \)
  - by (simp add: sorted-wrt01)

**lemma** sorted-tl:
  \( \text{sorted} \ xs \implies \text{sorted} \ \text{(tl} \ \text{xs}) \)
  - by (cases \ xs) (simp-all)

**lemma** sorted-iff-nth-mono-less:
  \( \text{sorted} \ xs = (\forall i \ j. \ i < j \rightarrow j < \text{length} \ xs \rightarrow xs \! i \leq xs \! j) \)
  - by (simp add: sorted-wrt-iff-nth-less)
lemma sorted-iff-nth-mono:
\[\text{sorted } \text{xs} \Leftrightarrow (\forall i \ j. \ i \leq j \rightarrow j < \text{length xs} \rightarrow \text{xs}!i \leq \text{xs}!j)\]
by (auto simp: sorted-iff-nth-mono-less nat-less-le)

lemma sorted-nth-mono:
\[\text{sorted } \text{xs} \Rightarrow i \leq j \Rightarrow j < \text{length xs} \Rightarrow \text{xs}!i \leq \text{xs}!j\]
by (auto simp: sorted-iff-nth-mono)

lemma sorted-iff-nth-Suc:
\[\text{sorted } \text{xs} \Leftrightarrow (\forall i. \ Suc \ i < \text{length xs} \rightarrow \text{xs}!i \leq \text{xs}!(Suc \ i))\]
by (simp add: sorted-wrt-iff-nth-Suc-transp)

lemma sorted-rev-nth-mono:
\[\text{sorted } (\text{rev } \text{xs}) \Rightarrow i \leq j \Rightarrow j < \text{length xs} \Rightarrow \text{xs}!j \leq \text{xs}!i\]
by (metis local.\text{order-class}.antisym-cone1 sorted-wrt-rev)

lemma sorted-rev-iff-nth-mono:
\[\text{sorted } (\text{rev } \text{xs}) \leftrightarrow (\forall i \ j. \ i \leq j \rightarrow j < \text{length xs} \rightarrow \text{xs}!j \leq \text{xs}!i)\]
proof -
  assume ?L thus ?R by (blast intro: sorted-rev-nth-mono)
next
  assume ?R
  have \text{rev xs}!k \leq \text{rev xs}!l if \text{asms}: \ k \leq l \ l < \text{length}(\text{rev xs}) for \ k \ l
  proof -
    have \text{length xs} \ l \leq \text{length xs} \text{Suc} \ k \leq \text{length xs} \text{Suc} \ k \leq \text{length xs}
    using \text{asms} by auto
    thus \text{rev xs}!k \leq \text{rev xs}!l
    by (simp add: \text{rev-nth})
  qed
  thus ?L by (simp add: sorted-iff-nth-mono)
qed

lemma sorted-rev-iff-nth-Suc:
\[\text{sorted } (\text{rev } \text{xs}) \leftrightarrow (\forall i. \ Suc \ i < \text{length xs} \rightarrow \text{xs}!(Suc \ i) \leq \text{xs}!i)\]
proof -
  interpret dual: linorder (\lambda x y. \ y \leq x) (\lambda x y. \ y < x)
  using dual-linorder .
  show ?thesis
    using dual-linorder dual.sorted-iff-nth-Suc dual.sorted-iff-nth-mono
    unfolding sorted-rev-iff-nth-mono by simp
qed

lemma sorted-map-remove1:
\[\text{sorted } (\text{map } f \text{ xs}) \Rightarrow \text{sorted } (\text{map } f (\text{remove1 } x \text{ xs}))\]
by (induct xs) (auto)

lemma sorted-remove1: \text{sorted } \text{xs} \Rightarrow \text{sorted } (\text{remove1 } a \text{ xs)}
using sorted-map-remove1 [of λx. x] by simp

lemma sorted-butlast:
  assumes sorted xs
  shows sorted (butlast xs)
  by (simp add: assms butlast-conv-take sorted-wrt-take)

lemma sorted-replicate [simp]: sorted(replicate n x)
by (induction n) (auto)

lemma sorted-remdups[simp]:
  sorted xs ⇒ sorted (remdups xs)
by (induct xs) (auto)

lemma sorted-remdups-adj[simp]:
  sorted xs ⇒ sorted (remdups-adj xs)
by (induct xs rule: remdups-adj.induct, simp-all split: if-split-asm)

lemma sorted-nths:
  sorted xs ⇒ sorted (nths xs I)
by (induction xs arbitrary: I) (auto simp: nths-Cons)

lemma sorted-distinct-set-unique:
  assumes sorted xs distinct xs sorted ys distinct ys set xs = set ys
  shows xs = ys
proof
  from assms have 1: length xs = length ys by (auto dest!: distinct-card)
  from assms show ?thesis
  proof (induct rule: list-induct2[OF 1])
    case 1 show ?case by simp
  next
    case (2 x xs y ys)
    then show ?case
      by (cases [x = y]) (auto simp add: insert-eq-iff)
  qed
qed

lemma map-sorted-distinct-set-unique:
  assumes inj-on f (set xs ∪ set ys)
  assumes sorted (map f xs) distinct (map f xs)
  sorted (map f ys) distinct (map f ys)
  assumes set xs = set ys
  shows xs = ys
using assms map-inj-on sorted-distinct-set-unique by fastforce

lemma sorted-dropWhile: sorted xs ⇒ sorted (dropWhile P xs)
by (auto dest: sorted-wrt-drop simp add: dropWhile-equiv)

lemma sorted-takeWhile: sorted xs ⇒ sorted (takeWhile P xs)
by (subst takeWhile-equiv) (auto dest: sorted-wrt-take)
lemma sorted-filter:
  \[ \text{sorted} \left( \text{map} \ f \ \text{xs} \right) \implies \text{sorted} \left( \text{map} \ f \left( \text{filter} \ P \ \text{xs} \right) \right) \]
  by (induct \text{xs}) simp-all

lemma foldr-max-sorted:
  assumes \[ \text{sorted} \left( \text{rev} \ \text{xs} \right) \]
  shows \[ \text{foldr} \ \max \ \text{xs} \ y = \begin{cases} y & \text{if } \text{xs} = [] \text{ then } y \text{ else } \max \left( \text{xs} \setminus \{0\} \right) \ y \\ \text{otherwise} \end{cases} \]
  using assms proof (induct \text{xs})
  case (Cons \ x \ \text{xs})
  then have \[ \text{sorted} \left( \text{rev} \ \text{xs} \right) \]
  by (auto)
  with Cons show ?case by (cases \text{xs}) (auto simp add: sorted-append max-def)
  qed simp

lemma filter-equals-takeWhile-sorted-rev:
  assumes \[ \text{sorted} \]: \[ \text{sorted} \left( \text{rev} \left( \text{map} \ f \ \text{xs} \right) \right) \]
  shows \[ \text{filter} \left( \lambda \ x. \ t < f \ x \right) \ \text{xs} = \text{takeWhile} \left( \lambda \ x. \ t < f \ x \right) \ \text{xs} \]
  proof (rule takeWhile-eq-filter [symmetric])
    let \[ ?dW = \text{dropWhile} \ ?P \ \text{xs} \]
    fix \ x \ assume \[ \ x \in \text{set} \ ?dW \]
    then obtain \[ i \] \ where \[ i < \text{length} \ ?dW \text{ and } \text{nth-i}: \ x = ?dW ! i \]
      unfolding in-set-conv-nth by auto
    hence \[ \text{length} \ ?tW + i < \text{length} \ ?tW @ ?dW \]
      unfolding length-append by simp
    hence \[ \text{length} \ ?W + i < \text{length} \ ?tW \]
      unfolding length-append by simp
    have \[ \text{length} \ ?tW @ \text{map} \ f \ ?dW \leq \text{length} \ ?tW + i \leq \text{length} \ ?tW \]
      unfolding sorted-rev-nth-mono[OF sorted - i', of length \ ?tW]
      unfolding map-append[symmetric] by simp
    hence \[ f \ x \leq f \ (\text{map} \ f \ ?dW ! 0) \]
      unfolding nth-append-length-plus nth-i
      using \ i \ preorder-class.le-less-trans[OF le0 i] \ by simp
    also have ... \leq \ t \]
      by (metis hd-conv-nth hd-dropWhile length-greater-0-conv length-pos-if-in-set local.leq x)
    finally show \[ \neg \ t < f \ x \]
      by simp
  qed

lemma sorted-map-same:
  \[ \text{sorted} \left( \text{map} \ f \ \left( \text{filter} \left( \lambda x. f \ x = g \ \text{xs} \right) \ \text{xs} \right) \right) \]
  proof (induct \text{xs} arbitrary: g)
    case Nil then show ?case by simp
  next
    case (Cons \ x \ \text{xs})
    then have \[ \text{sorted} \left( \text{map} \ f \ \left( \text{filter} \left( \lambda y. f \ y = (\lambda x. f \ x) \ \text{xs} \right) \ \text{xs} \right) \right) \]
      moreover from Cons have \[ \text{sorted} \left( \text{map} \ f \ \left( \text{filter} \left( \lambda y. f \ y = (g \circ \text{Cons} \ x) \ \text{xs} \right) \right) \right) \]
ultimately show ?case by simp-all

end

lemma sorted-upto[simp]: sorted [m..n] 
  by(simp add: sorted-wrt-mono-rel[OF sorted-wrt-upto])

66.3.3 Sorting functions

Currently it is not shown that sort returns a permutation of its input because
the nicest proof is via multisets, which are not part of Main. Alternatively
one could define a function that counts the number of occurrences of an
element in a list and use that instead of multisets to state the correctness
property.

coloring theory

lemma set-insort-key:
  set (insort-key f x xs) = insert x (set xs)
by (induct xs) auto

lemma length-insort [simp]:
  length (insort-key f x xs) = Suc (length xs)
by (induct xs) simp-all

lemma insort-key-left-comm:
  assumes f x ≠ f y
  shows insort-key f y (insort-key f x xs) = insort-key f x (insort-key f y xs)
by (induct xs) (auto simp add: assms dest: order.antisym)

lemma insort-left-comm:
  insort x (insort y xs) = insort y (insort x xs)
by (cases x = y) (auto intro: insort-key-left-comm)

lemma comp-fun-commute-insort: comp-fun-commute insort
proof
  qed (simp add: insort-left-comm fun-eq-iff)

lemma sort-key-simps [simp]:

xs)) .
ultimately show ?case by simp-all

end
sort-key $f \mathcal{L} = \mathcal{L}$
\[\text{sort-key } f (x \# \mathcal{L} x) = \text{insort-key } f x \text{ (sort-key } f \mathcal{L})\]
by (simp-all add: sort-key-def)

lemma sort-key-conv-fold:
assumes inj-on $f$ (set $\mathcal{L} x$)
shows sort-key $f \mathcal{L}$ x = fold (insort-key $f$) $\mathcal{L}$ x
proof –
  have fold (insort-key $f$) (rev $\mathcal{L} x$) = fold (insort-key $f$) $\mathcal{L}$ x  
  proof (rule fold-rev, rule ext)
    fix $\mathcal{L} x y$
    assume $x \in$ set $\mathcal{L} x$ $y \in$ set $\mathcal{L} x$
    with assms have *: $f y = f x \implies y = x$ by (auto dest: inj-onD)
    have **: $x = y \iff y = x$ by auto
    show (insort-key $f$ $y \circ$ insort-key $f$ $x$) $\mathcal{L} x$ =  (insort-key $f$ $x \circ$ insort-key $f$ $y$) $\mathcal{L} x$
      by (induct $\mathcal{L} x$) (auto intro: * simp add: **)
  qed
  then show ?thesis by (simp add: sort-key-def foldr-conv-fold)
  qed

lemma sort-conv-fold:
  sort $\mathcal{L} x$ = fold insort $\mathcal{L} x$
by (rule sort-key-conv-fold, simp)

lemma length-sort[simp]: length (sort-key $f \mathcal{L} x$) = length $\mathcal{L} x$
by (induct $\mathcal{L} x$, auto)

lemma set-sort[simp]: set(sort-key $f \mathcal{L} x$) = set $\mathcal{L} x$
by (induct $\mathcal{L} x$, simp-all add: set-insort-key)

lemma distinct-insort: distinct (insort-key $f \mathcal{L} x$ $\mathcal{L} x$) = ($x \notin$ set $\mathcal{L} x$ $\land$ distinct $\mathcal{L} x$)
by(induct $\mathcal{L} x$)(auto simp: set-insort-key)

lemma distinct-insort-key:
  distinct (map $f$ (insort-key $f \mathcal{L} x$ $\mathcal{L} x$)) = ($f x \notin$ $f \cdot$ set $\mathcal{L} x$ $\land$ (distinct (map $f$ $\mathcal{L} x$)))
by (induct $\mathcal{L} x$)(auto simp: set-insort-key)

lemma distinct-sort[simp]: distinct (sort-key $f \mathcal{L} x$) = distinct $\mathcal{L} x$
by (induct $\mathcal{L} x$)(simp-all add: distinct-insort)

lemma sorted-insort-key: sorted (map $f$ (insort-key $f \mathcal{L} x$ $\mathcal{L} x$)) = sorted (map $f$ $\mathcal{L} x$
by (induct $\mathcal{L} x$)(auto simp: set-insort-key)

lemma sorted-insort: sorted (insort $x \mathcal{L} x$) = sorted $\mathcal{L} x$
using sorted-insort-key [where $f$=$\lambda x$. $x$] by simp

theorem sorted-sort-key [simp]: sorted (map $f$ (sort-key $f \mathcal{L} x$))
by (induct $\mathcal{L} x$)(auto simp:sorted-insort-key)
theorem sorted-sort [simp]: sorted (sort xs)
using sorted-sort-key [where f=\lambda x. x] by simp

lemma insort-not-Nil [simp]:
  insort-key f a xs \neq []
by (induction xs) simp-all

lemma sort-key-id-if-sorted: sorted (map f xs) \implies sort-key f xs = xs
by (induction xs) (auto simp add: insort-is-Cons)

Subsumed by sorted (map ?f ?xs) \implies sort-key ?f ?xs = ?xs but easier to find:

lemma sorted-sort-id: sorted xs \implies sort xs = xs
by (simp add: sort-key-id-if-sorted)

lemma insort-key-remove1:
  assumes a \in set xs and sorted (map f xs) and hd (filter (\lambda x. f a = f x) xs) = a
  shows insort-key f a (remove1 a xs) = xs
using assms proof (induct xs)
  case (Cons x xs)
  then show ?case
  proof (cases x = a)
    case False
    then have f x \neq f a using Cons.prems by auto
    then have f x < f a using Cons.prems by auto
    with f x \neq f a show ?thesis using Cons by (auto simp: insort-is-Cons)
  qed (auto simp: insort-is-Cons)
  qed simp

lemma insort-remove1:
  assumes a \in set xs and sorted xs
  shows insort a (remove1 a xs) = xs
proof (rule insort-key-remove1)
  define n where n = length (filter (\=) a) xs - 1
  from \langle a \in set xs \rangle show a \in set xs .
  from \langle sorted xs \rangle show sorted (map (\lambda x. x) xs) by simp
  from \langle a \in set xs \rangle have a \in set (filter (\=) a) xs by auto
  then have set (filter (\=) a) xs \neq {} by auto
  then have filter (\=) a xs \neq [] by (auto simp only: set-empty)
  then have length (filter (\=) a) xs > 0 by simp
  then have n: Suc n = length (filter (\=) a) xs by (simp add: n-def)
  moreover have replicate (Suc n) a = a \# replicate n a
  by simp
  ultimately show hd (filter (\=) a) xs = a by (simp add: replicate-length-filter)
  qed
lemma finite-sorted-distinct-unique:
assumes finite A shows \( \exists!xs. \text{set } xs = A \land \text{sorted } xs \land \text{distinct } xs \)
proof -
  obtain xs where distinct xs A = set xs
    using finite-distinct-list [OF assms] by metis
  then show \( \text{thesis} \)
    by (rule-tac a=sort xs in ex1I) (auto simp: sorted-distinct-set-unique)
qed

lemma insort-insert-key-triv:
  \( f \ x \in f \ ' \ \text{set } xs \implies \text{insort-insert-key } f \ x \ x s = x s \)
by (simp add: insort-insert-key-def)

lemma insort-insert-triv:
  \( x \in \text{set } xs \implies \text{insort-insert } x \ x s = x s \)
using insort-insert-key-triv [of \( \lambda \) x. x] by simp

lemma insort-insert-insort-key:
  \( f \ x \notin f \ ' \ \text{set } xs \implies \text{insort-insert-key } f \ x \ x s = \text{insort-key } f \ x \ x s \)
by (simp add: insort-insert-key-def)

lemma insort-insert-insort:
  \( x \notin \text{set } xs \implies \text{insort-insert } x \ x s = \text{insort } x \ x s \)
using insort-insert-insort-key [of \( \lambda \) x. x] by simp

lemma set-insort-insert:
set (\text{insort-insert } x \ x s) = insert x (\text{set } x s)
by (auto simp add: insort-insert-key-def set-insort-key)

lemma distinct-insort-insert:
assumes distinct xs
shows distinct (\text{insort-insert-key } f \ x \ x s)
using assms by (induct xs) (auto simp add: insort-insert-key-def set-insort-key)

lemma sorted-insort-insert-key:
assumes sorted (map f xs)
shows sorted (map f (\text{insort-insert-key } f \ x \ x s))
using assms by (simp add: insort-insert-key-def sorted-insort-key)

lemma sorted-insort-insert:
assumes sorted xs
shows sorted (\text{insort-insert } x \ x s)
using assms sorted-insort-insert-key [of \( \lambda \) x. x] by simp

lemma filter-insort-triv:
\( \neg \ P \ x \implies \text{filter } P \ (\text{insort-key } f \ x \ x s) = \text{filter } P \ x s \)
by (induct xs) simp-all
lemma filter-insort:
  \[\text{sorted} \ (\text{map} \ f \ \text{xs}) \implies P \ x \implies \text{filter} \ P \ (\text{insort-key} \ f \ \text{xs}) = \text{insort-key} \ f \ x \ (\text{filter} \ P \ \text{xs})\]
  by (induct \ \text{xs}) (auto, subst \ insort-is-Cons, auto)

lemma filter-sort:
  \[\text{filter} \ P \ (\text{sort-key} \ f \ \text{xs}) = \text{sort-key} \ f \ (\text{filter} \ P \ \text{xs})\]
  by (induct \ \text{xs}) (simp-all add: filter-insort-triv filter-insort)

lemma remove1-insort-key [simp]:
  \[\text{remove1} \ x \ (\text{insort-key} \ f \ x \ \text{xs}) = \text{xs}\]
  by (induct \ \text{xs}) simp-all

end

lemma sort-upt [simp]: \[\text{sort} \ [m..<n] = [m..<n]\]
  by (rule sort-key-id-if-sorted) simp

lemma sort-upto [simp]: \[\text{sort} \ [i..j] = [i..j]\]
  by (rule sort-key-id-if-sorted) simp

lemma sorted-find-Min:
  \[\text{sorted} \ \text{xs} = \implies \exists \ x \in \text{set} \ \text{xs}. \ P \ x = \implies \text{List.} \text{find} \ P \ \text{xs} = \text{Some} \ (\text{Min} \ \{x \in \text{set} \ \text{xs}. \ P \ x\})\]
  proof (induct \ \text{xs})
    case Nil then show \ ?case by simp
  next
    case (Cons \ x \ \text{xs}) show \ ?case proof (cases \ P \ x)
      case True
      with Cons show \ ?thesis by (auto intro: Min-eqI [symmetric])
    next
    case False then have \ {y. \ (y = x \lor y \in \text{set} \ \text{xs}) \land P \ y\} = \{y \in \text{set} \ \text{xs}. \ P \ y\}
        by auto
      with Cons False show \ ?thesis by (simp-all)
  qed

lemma sorted-enumerate [simp]: \[\text{sorted} \ (\text{map} \ \text{fst} \ (\text{enumerate} \ n \ \text{xs}))\]
  by (simp add: enumerate-eq-zip)

lemma sorted-insort-is-snoc: \[\text{sorted} \ \text{xs} \implies \forall \ x \in \text{set} \ \text{xs}. \ a \geq x \implies \text{insort} \ a \ \text{xs} = \text{xs} @ [a]\]
  by (induct \ \text{xs}) (auto dest!: insort-is-Cons)

Stability of sort-key:

lemma sort-key-stable: \[\text{filter} \ (\lambda y. \ f \ y = k) \ (\text{sort-key} \ f \ \text{xs}) = \text{filter} \ (\lambda y. \ f \ y = k) \ \text{xs}\]
  by (induction \ \text{xs}) (auto simp: filter-insort-is-Cons filter-insort-triv)

corollary stable-sort-key-sorted-key: \[\text{stable-sort-key} \text{sort-key}\]
  by (simp add: stable-sort-key-def sort-key-stable)
lemma sort-key-const: sort-key (λx. c) xs = xs
by (metis (mono-tags) filter-True sort-key-stable)

66.3.4 transpose on sorted lists

lemma sorted-transpose[simp]: sorted (rev (map length (transpose xs)))
by (auto simp: sorted-iff-nth-mono rev-nth nth-transpose
    length-filter-conv-card intro: card-mono)

lemma transpose-max-length:
foldr (λxs. max (length xs)) (transpose xs) 0 = length (filter (λx. x ≠ []) xs)
(is ?L = ?R)
proof (cases transpose xs = [])
case False
have ?L = foldr max (map length (transpose xs)) 0
  by (simp add: foldr-map comp-def)
also have ... = length (transpose xs ! 0)
  using False sorted-transpose by (simp add: foldr-max-sorted)
finally show ?thesis
  using False by (simp add: nth-transpose)
next
case True
hence filter (λx. x ≠ []) xs = []
  by (auto intro!: filter-False simp: transpose-empty)
thus ?thesis by (simp add: transpose-empty True)
qed

lemma length-transpose-sorted:
fixes xs :: 'a list list
assumes sorted: sorted (rev (map length xs))
shows length (transpose xs) = (if xs = [] then 0 else length (xs ! 0))
proof (cases xs = [])
case False
thus ?thesis
  using foldr-max-sorted[OF sorted] False
unfolding length-transpose foldr-map comp-def
  by simp
qed simp

lemma nth-nth-transpose-sorted[simp]:
fixes xs :: 'a list list
assumes sorted: sorted (rev (map length xs))
and i: i < length (transpose xs)
and j: j < length (filter (λys. i < length ys) xs)
shows transpose xs ! i ! j = xs ! j ! i
using j filter-equals-takeWhile-sorted-rev[OF sorted, of i]
  nth-transpose[OF i] nth-map[OF j]
by (simp add: takeWhile-nth)
lemma transpose-column-length:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs)) and i < length xs
  shows length (filter (λys. i < length ys) (transpose xs)) = length (xs ! i)
proof -
  have xs ≠ [] using i < length xs by auto
  note filter-equals-takeWhile-sorted-rev[OF sorted, simp]
  { fix j assume j ≤ i
    note sorted-rev-nth-mono[OF sorted, of j i, simplified, OF this i < length xs]
  } note sortedE = this[consumes 1]

  have {j. j < length (transpose xs) ∧ i < length (transpose xs ! j)}
    = {..< length (xs ! i)}
proof safe
  fix j
  assume j < length (transpose xs) and i < length (transpose xs ! j)
  with this(2) nth-transpose[OF this(1)]
  have i < length (takeWhile (λys. j < length ys) xs) by simp
  from nth-mem[OF this] takeWhile-nth[OF this]
  show j < length (xs ! i) by (auto dest: set-takeWhileD)
next
  fix j assume j < length (xs ! i)
  thus i < length (transpose xs)
    using foldr-max-sorted[OF sorted] ‹xs ≠ []›, sortedE[OF le0]
    by (auto simp: length-transpose comp_def foldr-map)

  have Suc i ≤ length (takeWhile (λys. j < length ys) xs)
    using i < length xs ‹j < length (xs ! i)› less-Suc-eq-le
    by (auto intro: length-takeWhile-less-P-nth dest!: sortedE)
  with nth-transpose[OF ‹j < length (transpose xs)›]
  show i < length (transpose xs ! j) by simp
qed
thus ?thesis by (simp add: length-filter-cone-card)
qed

lemma transpose-column:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs)) and i < length xs
  shows map (λys. ys ! i) (filter (λys. i < length ys) (transpose xs))
    = xs ! i (is ?R = -)
proof (rule nth-equalityI)
  show length: length ?R = length (xs ! i)
    using transpose-column-length[OF assms] by simp

  fix j assume j: j < length ?R
  note * = less-le-trans[OF this, unfolded length-map, OF length-filter-le]
  from j have j-less: j < length (xs ! i) using length by simp
  have i-less-tW: Suc i ≤ length (takeWhile (λys. Suc j ≤ length ys) xs)
proof (rule length-takeWhile-less-P-nth)
  show Suc i \leq length xs using (i < length xs) by simp
  fix k assume k < Suc i
  hence k \leq i by auto
  with sorted-rev-nth-mono[OF sorted this] (i < length xs)
  have length (xs ! i) \leq length (xs ! k) by simp
  thus Suc j \leq length (xs ! k) using j-less by simp
qed

have i-less-filter: i < length (filter (\ys. j < length ys) xs)
  unfolding filter-equals-takeWhile-sorted-rev
  has j-less using j-less-tW by (simp-all add: Suc-le-eq)
from j show ?R

lemma transpose-transpose:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  shows transpose (transpose xs) = takeWhile (\xs x \noteq \[]) xs (is ?L = ?R)
proof −
  have len: length ?L = length ?R
    unfolding length-transpose transpose-max-length
    using filter-equals-takeWhile-sorted-rev[OF sorted, of 0]
    by simp
  { fix i assume i < length ?R
    with less-le-trans[OF - length-takeWhile-le[of - xs]]
    have i < length xs by simp
    } note * = this
  show ?thesis
    by (rule nth-equalityI)
    (simp-all add: takeWhile-nth nth-nth-transpose-sorted[OF sorted * i-less-filter])
qed

theorem transpose-rectangle:
  assumes xs = [] \implies n = 0
  assumes rect: \exists i. i < length xs \implies length (xs ! i) = n
  shows transpose xs = map (\lambda i. map (\lambda j. xs ! j ! i) [0..<length xs]) [0..<n]
  (is ?trans = ?map)
proof (rule nth-equalityI)
  have sorted (rev (map length xs))
    by (auto simp: rev-nth rect sorted-iff-nth-mono)
  from foldr-max-sorted[OF this] asms
  show len: length ?trans = length ?map
    by (simp-all add: length-transpose foldr-map comp-def)
moreover
  { fix i assume i < n hence filter (\ys. i < length ys) xs = xs
    using rect by (auto simp: in-set-conv-nth intro: filter-True) }
ultimately show \( \forall i. i < \text{length (transpose } xs) \Rightarrow ?\text{trans } i = ?\text{map } i \)

by (auto simp: nth-transpose intro: nth-equalityI)

qed

66.3.5 \texttt{sorted-key-list-of-set}

This function maps (finite) linearly ordered sets to sorted lists. The linear order is obtained by a key function that maps the elements of the set to a type that is linearly ordered. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via \texttt{set}).

Note: this is a generalisation of the older \texttt{sorted-list-of-set} that is obtained by setting the key function to the identity. Consequently, new theorems should be added to the locale below. They should also be aliased to more convenient names for use with \texttt{sorted-list-of-set} as seen further below.

definition (in \textit{linorder}) \texttt{sorted-key-list-of-set} :: (\'b \Rightarrow \'a) \Rightarrow \'b set \Rightarrow \'b list
where \texttt{sorted-key-list-of-set} f \equiv \texttt{folding-on.} F (\texttt{insort-key } f) []

locale \texttt{folding-insort-key} = lo?: \textit{linorder less-eq} :: \'a \Rightarrow \'a \Rightarrow bool less
for \texttt{less-eq} (\texttt{infix} \leq 50) \texttt{and} \texttt{less} (\texttt{infix} \prec 50) +
fixes \( S \)
fixes \( f :: \'b \Rightarrow \'a \)
assumes \texttt{inj-on}: \texttt{inj-on } f S

begin

lemma \texttt{insort-key-commute}:
\( x \in S \Rightarrow y \in S \Rightarrow \texttt{insort-key } f y \circ \texttt{insort-key } f x = \texttt{insort-key } f x \circ \texttt{insort-key } f y \)

proof (rule ext, goal-cases)
  case (1 \( xs \))
  with \texttt{inj-on} show ?case by (induction \( xs \)) (auto simp: \texttt{inj-onD})

qed

sublocale \texttt{fold-insort-key} = \texttt{folding-on } S \texttt{insort-key } f []

rewrites \texttt{folding-on.} F (\texttt{insort-key } f) [] \equiv \texttt{sorted-key-list-of-set } f

proof –
  show \texttt{folding-on } S (\texttt{insort-key } f)
  by standard (simp add: \texttt{insort-key-commute})

qed (simp add: \texttt{sorted-key-list-of-set-def})

lemma \texttt{idem-if-sorted-distinct}:
assumes \( \texttt{set } xs \subseteq S \texttt{ and} \texttt{sorted } (\texttt{map } f \texttt{ } xs) \texttt{ distinct } xs \)
shows \( \texttt{sorted-key-list-of-set } f \texttt{ } (\texttt{set } xs) = \texttt{xs} \)

proof (cases \( S = [] \))
  case \texttt{True}
  then show ?thesis using \( \texttt{set } xs \subseteq S \) by \texttt{auto}

next
  case \texttt{False}
with assms show ?thesis
proof (induction xs)
  case (Cons a xs)
    with Cons show ?case by (cases xs) auto
qed simp
qed

lemma sorted-key-list-of-set-empty:
  sorted-key-list-of-set f {} = []
  by (fact fold-insort-key.empty)

lemma sorted-key-list-of-set-insert:
  assumes insert x A ⊆ S and finite A x /∈ A
  shows sorted-key-list-of-set f (insert x A) = insort-key f x (sorted-key-list-of-set f A)
  using assms by (fact fold-insort-key.insert)

lemma sorted-key-list-of-set-insert-remove [simp]:
  assumes insert x A ⊆ S and finite A
  shows sorted-key-list-of-set f (insert x A) = insort-key f x (sorted-key-list-of-set f (A − {x}))
  using assms by (fact fold-insort-key.insert-remove)

lemma sorted-key-list-of-set-eq-Nil-iff [simp]:
  assumes A ⊆ S and finite A
  shows sorted-key-list-of-set f A = {} ←→ A = {}
  using assms by (auto simp fold-insort-key.remove)

lemma set-sorted-key-list-of-set [simp]:
  assumes A ⊆ S and finite A
  shows set (sorted-key-list-of-set f A) = A
  using assms(2,1)
  by (induct A rule: finite-induct) (simp-all add: set-insort-key)

lemma sorted-sorted-key-list-of-set [simp]:
  assumes A ⊆ S
  shows sorted (map f (sorted-key-list-of-set f A))
  proof (cases finite A)
    case True thus ?thesis using A ⊆ S
    by (induction A) (simp-all add: sorted-insort-key)
  next
    case False thus ?thesis by simp
  qed

lemma distinct-if-distinct-map: distinct (map f xs) ⇒ distinct xs
  using inj-on by (simp add: distinct-map)

lemma distinct-sorted-key-list-of-set [simp]:
  assumes A ⊆ S
THEORY “List”

shows distinct (map f (sorted-key-list-of-set f A))

proof (cases finite A)
  case True thus thesis using ‹A ⊆ S› inj-on
    by (induction A) (force simp: distinct-insert-key dest: inj-onD)+
  next
  case False thus thesis by simp
qed

lemma length-sorted-key-list-of-set [simp]:
  assumes A ⊆ S
  shows length (sorted-key-list-of-set f A) = card A
proof (cases finite A)
  case True
  with assms inj-on show thesis
    using distinct-card [symmetric, OF distinct-sorted-key-list-of-set]
    by (auto simp: subset-inj-on intro: card-image)
next
  case False thus thesis by simp
qed auto

lemmas sorted-key-list-of-set = set-sorted-key-list-of-set sorted-sorted-key-list-of-set distinct-sorted-key-list-of-set

lemma sorted-key-list-of-set-remove:
  assumes insert x A ⊆ S and finite A
  shows sorted-key-list-of-set f (A − {x}) = remove1 x (sorted-key-list-of-set f A)
proof (cases x ∈ A)
  case False with assms have x /∈ set (sorted-key-list-of-set f A) by simp
  next
  case True then obtain B where A: A = insert x B by (rule Set.set-insert)
  next
  case False thus thesis by simp
qed

lemma strict-sorted-key-list-of-set [simp]:
  A ⊆ S ⇒ sorted-wrt (≺) (map f (sorted-key-list-of-set f A))
by (cases finite A) (auto simp: strict-sorted-iff subset-inj-on[OF inj-on])

lemma finite-set-strict-sorted:
  assumes A ⊆ S and finite A
  obtains l where sorted-wrt (≺) (map f l) set l = A length l = card A
using assms
by (meson length-sorted-key-list-of-set set-sorted-key-list-of-set strict-sorted-key-list-of-set)

lemma (in linorder) strict-sorted-equal:
  assumes sorted-wrt (≺) xs
    and sorted-wrt (≺) ys
    and set ys = set xs
  shows ys = xs
using assms
proof (induction xs arbitrary: ys)
THEORY “List”

case (Cons x xs)
  show ?case
  proof (cases ys)
    case Nil
    then show ?thesis
      using Cons.prems by auto
  next
    case (Cons y ys)
    then have xs = ys' by (metis Cons.prems list.inject sorted-distinct-set-unique strict-sorted-iff)
    moreover have x = y
    using Cons.prems \( \langle xs = ys' \rangle \) local.Cons by fastforce
    ultimately show ?thesis
      using local.Cons by blast
  qed
qed auto

lemma (in linorder) strict-sorted-equal-Uniq: \( \exists \leq_1 xs. \) sorted-wrt \( (\prec) \) xs \& set xs = A
  by (simp add: Uniq-def strict-sorted-equal)

lemma sorted-key-list-of-set-inject:
  assumes A \( \subseteq S \) B \( \subseteq S \)
  assumes sorted-key-list-of-set f A = sorted-key-list-of-set f B finite A finite B
  shows A = B
  using assms set-sorted-key-list-of-set by metis

lemma sorted-key-list-of-set-unique:
  assumes A \( \subseteq S \) and finite A
  shows sorted-wrt \( (\prec) \) (map f l) \& set l = A \& length l = card A
  \( \longleftrightarrow \) sorted-key-list-of-set f A = l
  using assms
  by (auto simp: strict-sorted-iff card-distinct idem-if-sorted-distinct)

end

context linorder
begin

definition sorted-list-of-set \( \equiv \) sorted-key-list-of-set \( (\lambda x::'a. \ x) \)
We abuse the rewrites functionality of locales to remove trivial assumptions that result from instantiating the key function to the identity.

sublocale sorted-list-of-set: folding-insort-key \( (\leq) \) UNIV \( (\lambda x. \ x) \)
rewrites sorted-key-list-of-set \( (\lambda x. x) \) = sorted-list-of-set
  and \( \forall xs. \) map \( (\lambda x. x) \) xs \( \equiv \) xs
  and \( \forall X. \) \( X \subseteq \) UNIV \( \equiv \) True
  and \( \forall x. \) \( x \in \) UNIV \( \equiv \) True
  and \( \forall P. \) \( \text{True} \Rightarrow P \) \( \equiv \) Trueprop P
and \( \land P \land Q \).  

\[
\begin{align*}
\text{PROP } P \land \text{PROP } Q & \equiv \text{PROP } P \land \text{PROP } Q \\
\text{proof} & - \\
\text{show} & \text{folding-insort-key} (\leq) (\prec) \text{UNIV} (\lambda x. \ x) \\
& \text{by standard simp} \\
\text{qed} & (\text{simp-all add: sorted-list-of-set-def})
\end{align*}
\]

Alias theorems for backwards compatibility and ease of use.

**lemmas** sorted-list-of-set = sorted-list-of-set.sorted-key-list-of-set and  

sorted-list-of-set-empty = sorted-list-of-set.sorted-key-list-of-set-empty and  

sorted-list-of-set-insert = sorted-list-of-set.sorted-key-list-of-set-insert and  

sorted-list-of-set-insert-remove = sorted-list-of-set.sorted-key-list-of-set-insert-remove and  

sorted-list-of-set-eq-Nil-iff = sorted-list-of-set.sorted-key-list-of-set-eq-Nil-iff and  

set-sorted-list-of-set = sorted-list-of-set.sorted-key-list-of-set and  

sorted-sorted-list-of-set = sorted-list-of-set.sorted-sorted-key-list-of-set and  

distinct-sorted-list-of-set = sorted-list-of-set.distinct-sorted-key-list-of-set and  

length-sorted-list-of-set = sorted-list-of-set.length-sorted-key-list-of-set and  

sorted-list-of-set-remove = sorted-list-of-set.sorted-key-list-of-set-remove and  

strict-sorted-list-of-set = sorted-list-of-set.strict-sorted-key-list-of-set and  

sorted-list-of-set-inject = sorted-list-of-set.sorted-key-list-of-set-inject and  

finite-set-strict-sorted = sorted-list-of-set.sorted-key-list-of-set-unique and  

sorted-list-of-set-sort-remdups [code]:  

sorted-list-of-set (set xs) = sort (remdups xs)  

**proof** -  

**interpret** comp-fun-commute insort by (fact comp-fun-commute-insort)  

**show** ?thesis  

**by** (simp add: sorted-list-of-set.fold-insort-key.eq-fold sort-conv-fold fold-set-fold-remdups)  

**qed**

end

**lemma** sorted-list-of-set-sort-range [simp]:  

sorted-list-of-set \{m..<n\} = \{m..<n\}  

**by** (rule sorted-distinct-set-unique) simp-all

**lemma** sorted-list-of-set-lessThan-Suc [simp]:  

sorted-list-of-set \{..<\text{Suc } k\} = sorted-list-of-set \{..<k\} @ \{k\}  

**using** le0 lessThan-atLeast0 sorted-list-of-set-range upt-Suc-append by presburger

**lemma** sorted-list-of-set-atMost-Suc [simp]:  

sorted-list-of-set \{..\text{Suc } k\} = sorted-list-of-set \{..k\} @ \{\text{Suc } k\}  

**using** lessThan-Suc-atMost sorted-list-of-set-lessThan-Suc by fastforce

**lemma** sorted-list-of-set-nonempty:
assumes finite A A ≠ {}  
shows sorted-list-of-set A = Min A # sorted-list-of-set (A − {Min A})  
using assms  
by (auto simp: less-le simp flip: sorted-list-of-set.sorted-key-list-of-set-unique intro: Min-in)

lemma sorted-list-of-set-greaterThanLessThan:  
assumes Suc i < j  
shows sorted-list-of-set {i..<j} = Suc i # sorted-list-of-set {Suc i..<j}

proof –  
have {i..<j} = insert (Suc i) {Suc i..<j}  
using assms by auto  
then show ?thesis  
by (metis assms atLeastSucLessThan-greaterThanLessThan sorted-list-of-set-range upt-conv-Cons)
qued

lemma sorted-list-of-set-greaterThanAtMost:  
assumes Suc i ≤ j  
shows sorted-list-of-set {i..<j} = Suc i # sorted-list-of-set {Suc i..<j}

using sorted-list-of-set-greaterThanLessThan [of i Suc j]  
by (metis assms greaterThanAtMost-def greaterThanLessThan-eq le-imp-less-Suc lessThan-Suc-atMost)

lemma nth-sorted-list-of-set-greaterThanLessThan:  
n < j − Suc i ⇒ sorted-list-of-set {i..<j} ! n = Suc (i+n)

by (induction n arbitrary: i) (auto simp: sorted-list-of-set-greaterThanLessThan)

lemma nth-sorted-list-of-set-greaterThanAtMost:  
n < j − i ⇒ sorted-list-of-set {i..<j} ! n = Suc (i+n)

using nth-sorted-list-of-set-greaterThanLessThan [of n Suc j i]

by (simp add: greaterThanAtMost-def greaterThanLessThan-eq lessThan-Suc-atMost)

66.3.6 lists: the list-forming operator over sets

inductive-set
lists :: 'a set ⇒ 'a list set  
for A :: 'a set

where  
Nil [intro!, simp]: [] ∈ lists A  
| Cons [intro!, simp]: [a ∈ A; l ∈ lists A] ⇒ a#l ∈ lists A

inductive-cases listsE [elim!]: x#l ∈ lists A
inductive-cases listspE [elim!]: listsp A (x # l)

inductive-simps listsp-simps[code]:
listsp A []
listsp A (x # xs)
lemma listsp-mono [mono]: \( A \leq B \implies \text{listsp } A \leq \text{listsp } B \)
by (rule predicate1I, erule listsp.induct, blast+)

lemmas lists-mono = listsp-mono [to-set]

lemma listsp-infI:
  assumes l: listsp A l shows listsp B l \implies listsp (inf A B) l using l
by blast+ 

lemmas listsp-IntI = listsp-infI [to-set]

lemma listsp-inf-eq [simp]: \( \text{listsp } (\inf A B) = \inf (\text{listsp } A) (\text{listsp } B) \)
proof (rule mono-inf [where f=listsp, THEN order-antisym])
  show mono listsp by (simp add: mono-def listsp-mono)
  show inf (listsp A) (listsp B) \leq listsp (inf A B) by (blast intro!: listsp-infI)
qed

lemmas listsp-conj-eq [simp] = listsp-inf-eq [simplified inf-fun-def inf-bool-def]

lemmas lists-Int-eq [simp] = listsp-inf-eq [to-set]

lemma Cons-in-lists-iff [simp]: \( x \# xs \in \text{lists } A \iff x \in A \land xs \in \text{lists } A \)
by auto

lemma append-in-listsp-conv [iff]:
  \( \text{listsp } A (xs @ ys) = (\text{listsp } A xs \land \text{listsp } A ys) \)
by (induct xs) auto 


lemma in-listsp-conv-set: \( \text{listsp } A xs = (\forall x \in \text{set } xs. A x) \)
— eliminate listsp in favour of set
by (induct xs) auto


lemma in-listspD [dest!]: \( \text{listsp } A xs \implies \forall x \in \text{set } xs. A x \)
by (rule in-lists-conv-set [THEN iffD1])

lemmas in-listsD [dest!] = in-listspD [to-set]

lemma in-listspI [intro!]: \( \forall x \in \text{set } xs. A x \implies \text{listsp } A xs \)
by (rule in-listsp-conv-set [THEN iffD2])

lemmas in-listsI [intro!] = in-listspI [to-set]

lemma lists-eq-set: \( \text{lists } A = \{ xs. \text{ set } xs \leq A \} \)
by auto
lemma lists-empty [simp]: lists {} = [[]]
by auto

lemma lists-UNIV [simp]: lists UNIV = UNIV
by auto

lemma lists-image: lists (f' A) = map f' lists A
proof
  { fix xs have ∀ x ∈ set xs. x ∈ f' A ⇒ xs ∈ map f' lists A
    by (induct xs) (auto simp del: list.map simp add: list.map[symmetric] intro!: imageI) }
  then show ?thesis by auto
qed

66.3.7 Inductive definition for membership

inductive ListMem :: 'a ⇒ 'a list ⇒ bool
where
  elem: ListMem x (x # xs)
| insert: ListMem x xs ⇒ ListMem x (y # xs)

lemma ListMem-iff: (ListMem x xs) = (x ∈ set xs)
proof
  show ListMem x xs ⇒ x ∈ set xs
    by (induct set: ListMem) auto
  show x ∈ set xs ⇒ ListMem x xs
    by (induct xs) (auto intro: ListMem.intros)
qed

66.3.8 Lists as Cartesian products

set-Cons A Xs: the set of lists with head drawn from A and tail drawn from Xs.

definition set-Cons :: 'a set ⇒ 'a list ⇒ 'a list set
where
  set-Cons A XS = { z. ∃ xs. z = x # xs ∧ x ∈ A ∧ xs ∈ XS }

lemma set-Cons-sing-Nil [simp]: set-Cons A {} = (%x. [x])'A
by (auto simp add: set-Cons-def)
Yields the set of lists, all of the same length as the argument and with elements drawn from the corresponding element of the argument.

primrec listset :: 'a set list ⇒ 'a list set
where
  listset [] = {[]} |
  listset (A # As) = set-Cons A (listset As)
66.4 Relations on Lists

66.4.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These ordering are not used in dictionaries.

primrec — The lexicographic ordering for lists of the specified length

\texttt{lexn} :: ('a × 'a) set ⇒ nat ⇒ ('a list × 'a list) set where

\texttt{lexn} r 0 = {}
\texttt{lexn} r (Suc n) =
(map-prod (%(x, xs). x#xs) (%(x, xs). x#xs) ' (r <*lex*> lexn r n)) Int
\{(xs, ys). length xs = Suc n ∧ length ys = Suc n\}

definition \texttt{lex} :: ('a × 'a) set ⇒ ('a list × 'a list) set where
\texttt{lex} r = (\bigcup n. lexn r n) — Holds only between lists of the same length

definition \texttt{lenlex} :: ('a × 'a) set ⇒ ('a list × 'a list) set where
\texttt{lenlex} r = inv-image (less-than <*lex*> lex r) (λxs. (length xs, xs))
— Compares lists by their length and then lexicographically

lemma \texttt{wf-lexn}: assumes \texttt{wf r} shows \texttt{wf (lexn r n)}
proof (induct n)
case (Suc n)
  have \texttt{inj}: inj (λ(x, xs). x ≠ xs)
  using \texttt{assms} by (auto simp: inj-on-def)
  have \texttt{wf}: \texttt{wf (map-prod (λ(x, xs). x ≠ xs) (λ(x, xs). x ≠ xs) '< (r <*lex*> lexn r n)'})
  by (simp add: Suc.\texttt{hyps} \texttt{assms} \texttt{wf-lex-prod} \texttt{wf-map-prod-image} \texttt{OF - inj})
  then show ?case
  by (rule \texttt{wf-subset}) auto
qed auto

lemma \texttt{lexn-length}:
(xs, ys) ∈ lexn r n ⇒ length xs = n ∧ length ys = n
by (induct n arbitrary: xs ys) auto

lemma \texttt{wf-lex} [intro!]:
assumes \texttt{wf r} shows \texttt{wf (lex r)}
unfolding \texttt{lex-def}
proof (rule \texttt{wf-UN})
  show \texttt{wf (lexn r i)} for i
    by (simp add: \texttt{assms} \texttt{wf-lexn})
  show \texttt{∀ i j. lexn r i ≠ lexn r j ⇒ Domain (lexn r i) ∩ Range (lexn r j) = {}}
    by (metis DomainE Int-emptyI RangeE lexn-length)
qed

lemma \texttt{lexn-conv}:
lexn r n =
\{(xs,ys). length xs = n ∧ length ys = n ∧
(∃xys x y xs' ys', xs= xys @ x#xs' ∧ ys= xys @ y # ys' ∧ (x, y) ∈ r)}

proof (induction n)
case (Suc n)
then show ?case
  apply (simp add: image-Collect lex-prod-def, safe, blast)
  apply (rule-tac x = ab # xys in exI, simp)
  apply (case-tac xys; force)
done
qed auto

By Mathias Fleury:

proposition lexn-transI:
  assumes trans r shows trans (lexn r n)
unfolding trans-def
proof (intro allI impI)
  fix as bs cs
assume asbs: (as, bs) ∈ lexn r n and bscs: (bs, cs) ∈ lexn r n
obtain abs a b as' bs' where
  n: length as = n and length bs = n and
  as: as = abs @ a # as' and
  bs: bs = abs @ b # bs' and
  abr: (a, b) ∈ r
  using asbs unfolding lexn-conv by blast
obtain bcs b' c' cs' bs' where
  n': length cs = n and length bs = n and
  bs': bs = bcs @ b' # bs' and
  cs: cs = bcs @ c' # cs' and
  b'c'r: (b', c') ∈ r
  using bscs unfolding lexn-conv by blast
consider (le) length bcs < length abs
  | (eq) length bcs = length abs
  | (ge) length bcs > length abs by linarith
thus (as, cs) ∈ lexn r n
proof cases
  let ?k = length bcs
  case le
  hence as ! ?k = bs ! ?k unfolding as bs by (simp add: nth-append)
  hence (as ! ?k, cs ! ?k) ∈ r using b'c'r unfolding bs' cs by auto
  moreover have length bcs < length as using le unfolding as by simp
  from id-take-nth-drop[OF this]
  have as = take ?k as @ as ! ?k # drop (Suc ?k) as .
  moreover have length bcs < length cs unfolding cs by simp
  from id-take-nth-drop[OF this]
  have cs = take ?k cs @ cs ! ?k # drop (Suc ?k) cs .
  moreover have take ?k as = take ?k cs
    using le arg-cong[OF bs, of take (length bcs)]
  unfolding cs as bs' by auto
ultimately show \(?thesis\) using \(n\ n'\ unfolding\ lexn-conv\ by\ auto\)

next
let \(?k = length\ abs\)
case \(\ge\)
hence \(bs\ !\ ?k = cs\ !\ ?k\ unfolding\ bs\ cs\ by\ (simp\ add: n-th-append)\)
hence \((as\ !\ ?k,\ cs\ !\ ?k)\in r\ using\ abr\ unfolding\ as\ bs\ by\ auto\)
moreover
have \(length\ abs < length\ as\ using\ ge\ unfolding\ as\ by\ simp\)
from \(id-take-nth-drop[OF\ this]\)
have \(as = take\ ?k\ as\ @\ as\ !\ ?k\ #\ drop\ (Suc\ ?k)\ as\).
moreover have \(length\ abs < length\ cs\ using\ n\ n'\ unfolding\ as\ by\ simp\)
from \(id-take-nth-drop[OF\ this]\)
have \(cs = take\ ?k\ cs\ @\ cs\ !\ ?k\ #\ drop\ (Suc\ ?k)\ cs\).
moreover have \(take\ ?k\ as = take\ ?k\ cs\ using\ ge\ arg-cong[OF\ bs',\ of\ take\ (length\ abs)]\)
unfolding\ cs\ as\ bs\ by\ auto
ultimately show \(?thesis\) using \(n\ n'\ unfolding\ lexn-conv\ by\ auto\)

next
let \(?k = length\ abs\)
case \(\eq\)
hence \(*\colon\ abs = bcs\ b = b'\ using\ bs\ bs'\ by\ auto\)
hence \((a,\ c')\in r\ using\ abr\ b'c'r\ assms\ unfolding\ trans-def\ by\ blast\)
with \(*\) show \(?thesis\) using \(n\ n'\ unfolding\ lexn-conv\ as\ bs\ cs\ by\ auto\)
qed


corollary\ lex-transI:
  assumes\ trans\ r\ shows\ trans\ (lex\ r)
  using\ lexn-transI\ [OF\ assms]\ by\ (clarsimp\ simp\ add: lex-def\ trans-def)\ (metis\ lexn-length)

lemma\ lex-conv:
  \(lex\ r =\)
  \(\{(xs,ys).\ length\ xs = length\ ys\ \&\\)
  \((\exists\ y\ y'\ ys',\ xs = \xs\ @\ x\ #\ xs'\ \&\ xs = \ys\ @\ y\ #\ ys'\ \&\ (x,\ y)\in r)\}\)
by\ (force\ simp\ add: lex-def\ lexn-conv)

lemma\ wf-lenlex\ [intro!]:\ wf\ r\ \\Longrightarrow\ \(wf\ (lenlex\ r)\)
by\ (unfold\ lenlex-def)\ blast

lemma\ lenlex-conv:
  \(lenlex\ r = \{(xs,ys).\ length\ xs < length\ ys\ \\vee\)
  \(\\neg\ length\ xs = length\ ys\ \&\ (xs,\ ys)\in lex\ r\}\)
by\ (auto\ simp\ add: lex-def\ Id-on-def\ lex-prod-def\ inv-image-def)

lemma\ total-lenlex:
  assumes\ total\ r
  shows\ total\ (lenlex\ r)
proof
  have \((xs, ys) \in \text{lex} r (\text{length} \; xs) \lor (ys, xs) \in \text{lex} r (\text{length} \; xs)\)
  if \(xs \neq ys\) and \(\text{len: length} \; xs = \text{length} \; ys\) for \(xs\) and \(ys\)

proof
  obtain \(pre \; x \; xs' \; y \; ys'\) where \(x \neq y\) and \(xs = pre @ [x] @ xs'\) and \(ys = pre @ [y] @ ys'\)
    by (meson \text{len} (xs \neq ys) \text{same-length-different})

then consider \((x, y) \in r \mid (y, x) \in r\)
  by (meson \text{UNIV-I assms total-on-def})

then show \(\text{thesis}\)
  by cases (use \text{len} in \((\text{force simp add: lexn-conv xs ys})+)\)

qed

then show \(\text{thesis}\)
  by (fastforce simp \text{lenlex-def total-on-def lex-def})

qed

lemma \text{lenlex-transI} \{\text{intro}\}: trans \; r \implies \text{trans} \; (\text{lenlex} \; r)

unfolding \text{lenlex-def}
  by (meson \text{lex-transI trans-inv-image trans-less-than trans-lex-prod})

lemma \text{Nil-notin-lex} \{\text{iff}\}: ([], ys) \notin \text{lex} \; r
  by (simp add: \text{lex-conv})

lemma \text{Nil2-notin-lex} \{\text{iff}\}: (xs, []) \notin \text{lex} \; r
  by (simp add: \text{lex-conv})

lemma \text{Cons-in-lex} \{\text{simp}\}:
  \((x \# xs, y \# ys) \in \text{lex} \; r \iff (x, y) \in r \land \text{length} \; xs = \text{length} \; ys \lor x = y \land (xs, ys) \in \text{lex} \; r\)
  (is \(\text{?lhs} = ?rhs\))

proof
  assume \(\text{?lhs}\) then show \(\text{?rhs}\)
    by (simp add: \text{lex-conv}) (metis \text{hd-append list.sel(1) list.sel(3) tl-append2})

next
  assume \(\text{?rhs}\) then show \(\text{?lhs}\)
    by (simp add: \text{lex-conv}) (blast intro: Cons-eq-appendI)

qed

lemma \text{Nil-lenlex-iff1} \{\text{simp}\}: ([], ns) \in \text{lenlex} \; r \iff ns \neq []

and \text{Nil-lenlex-iff2} \{\text{simp}\}: (ns, []) \notin \text{lenlex} \; r

by (auto simp: \text{lenlex-def})

lemma \text{Cons-lenlex-iff}:
  \(((m \# ms, n \# ns) \in \text{lenlex} \; r) \iff length \; ms < length \; ns \lor length \; ms = length \; ns \land (m, n) \in r \lor (m = n \land (ms, ns) \in \text{lenlex} \; r)\)

by (auto simp: \text{lenlex-def})
lemma lenlex-irrefl: \((\forall x. (x,x) \notin r) \implies (xs,xs) \notin \text{lenlex } r\)
by (induction xs) (auto simp add: Cons-lenlex-iff)

lemma lenlex-trans:
\[
\begin{array}{l}
((x,y) \in \text{lenlex } r; (y,z) \in \text{lenlex } r; \text{trans } r) \implies (x,z) \in \text{lenlex } r
\end{array}
\]
by (meson lenlex-transI transD)

lemma lenlex-length: \((ms, ns) \in \text{lenlex } r \implies \text{length } ms \leq \text{length } ns\)
by (auto simp: lenlex-def)

lemma lex-append-rightI:
\[(xs, ys) \in \text{lex } r \implies \text{length } vs = \text{length } us \implies (xs @ us, ys @ vs) \in \text{lex } r\]
by (fastforce simp: lex-def lexn-conv)

lemma lex-append-leftI:
\[(ys, zs) \in \text{lex } r \implies (xs @ ys, xs @ zs) \in \text{lex } r\]
by (induct xs) auto

lemma lex-append-leftD:
\[\forall x. (x,x) \notin r \implies (xs @ ys, xs @ zs) \in \text{lex } r \implies (ys, zs) \in \text{lex } r\]
by (induct xs) auto

lemma lex-append-left-iff:
\[\forall x. (x,x) \notin r \implies (xs @ ys, xs @ zs) \in \text{lex } r \iff (ys, zs) \in \text{lex } r\]
by (metis lex-append-leftD lex-append-leftI)

lemma lex-take-index:
assumes \((xs, ys) \in \text{lex } r\)
obtains \(i\) where \(i < \text{length } xs\) and \(i < \text{length } ys\) and take \(i\) \(xs = \) take \(i\) \(ys\)
and \((xs ! i, ys ! i) \in r\)
proof –
obtain \(n\) us \(x\) \(xs'\) \(y\) \(ys'\) where \((xs, ys) \in \text{lexn } r\) \(n\) and \(\text{length } xs = n\) and \(\text{length } ys = n\)
and \(xs = us \oplus x \# xs'\) and \(ys = us \oplus y \# ys'\)
and \((x, y) \in r\)
using assms by (fastforce simp: lex-def lexn-conv)
then show \(?\)thesis by (intro that [of \(\text{length } us\)]) auto
qed

lemma irrefl-lex: \(\text{irrefl } r \implies \text{irrefl } (\text{lex } r)\)
by (meson irrefl-def lex-take-index)

lemma lexl-not-refl [simp]: \(\text{irrefl } r \implies (x,x) \notin \text{lex } r\)
by (meson irrefl-def lex-take-index)

66.4.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a" < "ab" < "b". This ordering does not preserve well-foundedness. Author: N. Voelker, March 2005.

definition lexord :: \('a \times 'a\) set \(\Rightarrow\) \('a list \times 'a list\) set where
\text{THEORY} \quad \text{“List”} \\

\text{lexord } r = \{(x,y). \exists \ a \ v. \ y = x \# a \# v \lor \}
\quad (\exists \ u \ b \ v. \ (a,b) \in r \land x = u \# (a \# v) \land y = u \# (b \# w))\}

\text{lemma lexord-Nil-left[simp]}: \ ([],y) \in \text{lexord } r = (\exists \ a. \ y = a \# x)
\text{by} \ (\text{unfold lexord-def, induct-tac } y, \text{auto})

\text{lemma lexord-Nil-right[simp]}: \ (x,[]) \notin \text{lexord } r
\text{by} \ (\text{unfold lexord-def, induct-tac } x, \text{auto})

\text{lemma lexord-cons-cons[simp]}:
\quad (a \# x, b \# y) \in \text{lexord } r \leftrightarrow (a,b) \in r \lor (a = b \land (x,y) \in \text{lexord } r) \quad (\text{is ?lhs = ?rhs})
\text{proof}
\quad \begin{array}{l}
\hspace{1em} \text{assume ?lhs} \\
\hspace{1em} \text{then show ?rhs}
\quad \begin{array}{l}
\hspace{1em} \text{apply (simp add: lexord-def)} \\
\hspace{1em} \text{apply (metis hd-append list.sel(1) list.sel(3) tl-append2)}
\end{array}
\end{array}
\text{done}
\text{qed} \ (\text{auto simp add: lexord-def; (blast | meson Cons-eq-appendI)})

\text{lemmas lexord-simps = lexord-Nil-left lexord-Nil-right lexord-cons-cons}

\text{lemma lexord-same-pref-iff}:
\quad (xs @ ys, xs @ zs) \in \text{lexord } r \leftrightarrow (\exists x \in \text{set } xs. \ (x,x) \in r) \lor (ys, zs) \in \text{lexord } r
\text{by (induction } xs) \text{ auto}

\text{lemma lexord-same-pref-if-irrefl[simp]}:
\quad \text{irrefl } r \implies (xs @ ys, xs @ zs) \in \text{lexord } r \leftrightarrow (ys, zs) \in \text{lexord } r
\text{by (simp add: irrefl-def lexord-same-pref-iff)}

\text{lemma lexord-append-rightI}: \exists b \ z. \ y = b \# z \implies (x, x @ y) \in \text{lexord } r
\text{by (metis append-Nil2 lexord-Nil-left lexord-same-pref-iff)}

\text{lemma lexord-append-left-rightI}:
\quad (a,b) \in r \implies (u \# a \# x, u \# b \# y) \in \text{lexord } r
\text{by (simp add: lexord-same-pref-iff)}

\text{lemma lexord-append-leftI}:
\quad (u,v) \in \text{lexord } r \implies (x @ u, x @ v) \in \text{lexord } r
\text{by (simp add: lexord-same-pref-iff)}

\text{lemma lexord-append-leftD}:
\quad [(x @ u, x @ v) \in \text{lexord } r; (\forall a. \ (a,a) \notin r)] \implies (u,v) \in \text{lexord } r
\text{by (simp add: lexord-same-pref-iff)}

\text{lemma lexord-take-index-conv}:
\quad ((x,y) \in \text{lexord } r) =
\quad ((\text{length } x < \text{length } y \land \text{take (length } x) \ y = x) \lor
\quad (\exists i. \ i < \text{min(length } x)(\text{length } y) \land \text{take } i \ x = \text{take } i \ y \land (x[i],y[i]) \in r))
\text{proof
have \((\exists \ a \ v. \ y = x \circ a \ # v) = (\text{length } x < \text{length } y \land \text{take } (\text{length } x) \ y = x)\) by (metis Cons-nth-drop-Suc append-eq-conv-conj drop-all list.simps(3) not-le)

moreover have \((\exists \ u \ a \ b. \ ((a, b) \in r \land (\exists v. \ x = u @ a \ # v) \land (\exists w. \ y = u @ b \ # w))) = (\exists i < \text{length } x, \ i < \text{length } y \land \text{take } i x = \text{take } i y \land (x!i, y!i) \in r)\)

apply safe
using less_iff_Suc_add apply auto[1]
by (metis id-take-nth-drop)
ultimately show \(?thesis\)
by (auto simp: lexord_def Let_def)
qed

— lexicord is extension of partial ordering List.lex

lemma lexord-lex: \((x, y) \in \text{lex } r = ((x,y) \in \text{lexord } r \land \text{length } x = \text{length } y)\)

proof (induction x arbitrary: y)
  case (Cons a x y)
  then show \(?thesis\)
  by (cases y) (force+)
qed auto

lemma lexord-sufI:
  assumes \((u, w) \in \text{lexord } r \land \text{length } w \leq \text{length } u\)
shows \((\circ w@v, w@z) \in \text{lexord } r\)

proof -
  from leD[OF assms(2)] assms(1)[unfolded lexord-take-index-conv[of u w r] min-absorb2[OF assms(2)]]
  obtain i where take i u = take i w and \((u, i, w@i) \in r\) and \(i < \text{length } w\)
  by blast
  hence \((u@v)!i, (w@z)!i) \in r\)
  unfolding nth-append using less_le_trans[OF \(i < \text{length } w\) assms(2)] \((u!i, w@i)\)
  by presburger
  moreover have \(i < \text{min } (\text{length } (u@v)) (\text{length } (w@z))\)
  using assms(2) \(i < \text{length } w\) by simp
  moreover have \(\text{take } i (u@v) = \text{take } i (w@z)\)
  using assms(2) \(i < \text{length } w\) \((\text{take } i u = \text{take } i w)\) by simp
  ultimately show \(?thesis\)
  using lexord-take-index-conv by blast
qed

lemma lexord-sufE:
  assumes \((xs@zs, y@qs) \in \text{lexord } r \land xs \neq ys \land \text{length } xs = \text{length } ys \land \text{length } zs = \text{length } qs\)
shows \((xs, ys) \in \text{lexord } r\)

proof -
  obtain i where \(i < \text{length } (xs@zs)\) and \(i < \text{length } (ys@qs)\) and \(\text{take } i (xs@zs) = \text{take } i (ys@qs)\)
  and \((xs@zs)!i, (ys@qs)!i) \in r\)
  using assms(1) lex-take-index[unfolded lexord-lex[of \(xs \circ zs \circ ys \circ qs \circ r\)]
  length-append[of \(xs \circ zs\), unfolded assms(3,4), folded length-append[of \(ys \ circ qs\)]]
by blast
have length (take i xs) = length (take i ys)
  by (simp add: assms(3))
have \( i < \text{length} \, xs \)
  using assms(2,3) le-less-linear take-all[of xs i] take-all[of ys i]
  \langle\text{take } i \, (xs \circ ys) = \text{take } i \, (ys \circ qs)\rangle\, \text{append-eq-append-conv take-append}
  by metis
hence \((xs \, i, \, ys \, i) \in \, r\)
  using \((\text{take } i \, (xs \circ zs) \, i, \, (ys \circ qs) \, i) \in \, r\)\, \text{assms(3)}\, \text{by simp add: nth-append}
moreover have \(\text{take } i \, xs = \text{take } i \, ys\)
  using assms(3) \langle\text{take } i \, (xs \circ zs) = \text{take } i \, (ys \circ qs)\rangle\, \text{by auto}
ultimately show \(?\text{thesis}\)
  unfolding lexord\,take\,index\,conv using \(i < \text{length } xs\)\, \text{assms(3)}\, \text{by fastforce}
qed

lemma lexord-irreflexive: \(\forall \, x. \, (x,x) \notin r \implies (xs,ys) \notin \text{lexord } r\)
by (induct xs) auto

By René Thiemann:

lemma lexord-partial-trans:
  \((\forall \, x \, y \, z. \, x \in \, set \, xs \implies (x,y) \in \, r \implies (y,z) \in \, r \implies (x,z) \in \, r)\)
  \implies (xs,ys) \in \text{lexord } r \implies (ys,zs) \in \text{lexord } r \implies (xs,zs) \in \text{lexord } r
proof (induct xs arbitrary: ys zs)
case Nil
from Nil(3) show \(?\text{case unfolding lexord-def by (cases zs, auto)}\)
next
case (Cons x xs yys zzs)
from Cons(3) obtain y ys where yys = y # ys unfolding lexord-def
  by (cases yys, auto)
note Cons = Cons[unfolded yys]
from Cons(3) have one: \((x,y) \in \, r \lor x = y \land (xs,ys) \in \text{lexord } r\)\, \text{by auto}
from Cons(4) obtain z zs where zzs = z # zs unfolding lexord-def
  by (cases zzs, auto)
note Cons = Cons[unfolded zzs]
from Cons(4) have two: \((y,z) \in \, r \lor y = z \land (ys,zs) \in \text{lexord } r\)\, \text{by auto}

\{ assume \((xs,ys) \in \text{lexord } r\)\, and \((ys,zs) \in \text{lexord } r\)
  from Cons(1)[OF - this] Cons(2)
  have \((xs,zs) \in \text{lexord } r\)\, \text{by auto}
\} note ind1 = this

\{ assume \((x,y) \in \, r\)\, and \((y,z) \in \, r\)
  from Cons(2)[OF - this] have \((x,z) \in \, r\)\, \text{by auto}
\} note ind2 = this
from one two ind1 ind2
have \((x,z) \in \, r \lor x = z \land (xs,zs) \in \text{lexord } r\)\, \text{by blast}
thus \(?\text{case unfolding zzs by auto}\)
qed
lemma lexord-trans:
\[ \begin{array}{c}
(x, y) \in \text{lexord } r \land (y, z) \in \text{lexord } r \land \text{trans } r \Rightarrow (x, z) \in \text{lexord } r
\end{array} \]
by (auto simp: trans_def intro: lexord-partial-trans)

lemma lexord-transI: trans r \Rightarrow trans (lexord r)
by (meson lexord-trans transI)

lemma total-lexord: total r \Rightarrow total (lexord r)
unfolding total-on-def

proof clarsimp
fix x y
assume \( \forall x y. x \neq y \rightarrow (x, y) \in r \lor (y, x) \in r \)
and \((x :: 'a list) \neq y\)
and \((y, x) \notin \text{lexord } r\)
then
show \((x, y) \in \text{lexord } r\)
proof (induction x arbitrary: y)
  case Nil
  then show ?case by (metis lexord-Nil-left list.exhaust)
next
  case (Cons a x y)
  then obtain z zs where ys: \(ys = z \# zs\)
  by (cases ys) auto
  with assms Cons show ?case by (auto dest: asymD)
qed

corollary lexord-linear: \(\forall a b. (a, b) \in r \lor a = b \lor (b, a) \in r \Rightarrow (x, y) \in \text{lexord } r \lor x = y \lor (y, x) \in \text{lexord } r\)
using total-lexord by (metis UNIV-I total-on-def)

lemma lexord-irrefl:
irrefl R \Rightarrow irrefl (lexord R)
by (simp add: irrefl_def lexord-irreflexive)

lemma lexord-asym:
assumes asym R
shows asym (lexord R)
proof
fix xs ys
assume \((xs, ys) \in \text{lexord } R\)
then show \((ys, xs) \notin \text{lexord } R\)
proof (induct xs arbitrary: ys)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then obtain z zs where ys: \(ys = z \# zs\)
  by (cases ys) auto
  with assms Cons show ?case by (auto dest: asymD)
qed
qed

lemma lexord-asymmetric:
assumes asym R
assumes hyp: (a, b) ∈ lexord R
shows (b, a) /∈ lexord R
proof
  from ‹asym R› have asym (lexord R) by (rule lexord-asym)
  then show ?thesis by (auto simp: hyp dest: asymD)
qed

lemma asym-lex: asym R =⇒ asym (lex R)
by (meson asymI asymD irrefl-lex lexord-asym lexord-lex)

lemma asym-lenlex: asym R =⇒ asym (lenlex R)
by (simp add: lenlex-def asym-inv-image asym-less-than asym-lex asym-lex-prod)

lemma lenlex-append1: [simp]:
assumes irrefl R
shows (us @ xs, us @ ys) ∈ lenlex R ←→ (xs, ys) ∈ lenlex R
proof (induction us)
  case Nil
  then show ?case
    by (simp add: lenlex-def eq)
next
  case (Cons u us)
  with assms show ?case
    by (fastforce simp add: lenlex-def eq)
qed

lemma lenlex-append2: [simp]:
assumes irrefl R
shows (us @ xs, us @ ys) ∈ lenlex R ←→ (xs, ys) ∈ lenlex R
proof (induction us)
  case Nil
  then show ?case
    by (simp add: lenlex-def)
next
  case (Cons u us)
  with assms show ?case
    by (auto simp: lenlex-def irrefl-def)
qed

Predicate version of lexicographic order integrated with Isabelle’s order type classes. Author: Andreas Lochbihler

context ord
begin
inductive lexordp :: 'a list ⇒ 'a list ⇒ bool where
  Nil: lexordp [] (y # ys)
  Cons: x < y ⇒ lexordp (x # xs) (y # ys)
  Cons-eq:
  [ ¬ x < y; ¬ y < x; lexordp zs ys ] ⇒ lexordp (x # xs) (y # ys)
end

lemma lexordp-simps [simp, code]:
  lexordp [] ys = (ys ≠ [])
  lexordp zs [] = False
  lexordp (x # xs) (y # ys) ⨾ x < y ∨ ¬ y < x ∧ lexordp xs ys
by(subst lexordp.simps, fastforce simp add: neq-Nil-conv)+

inductive lexordp-eq :: 'a list ⇒ 'a list ⇒ bool where
  Nil: lexordp-eq [] ys
  Cons: x < y ⇒ lexordp-eq (x # xs) (y # ys)
  Cons-eq: [ ¬ x < y; ¬ y < x; lexordp-eq xs ys ] ⇒ lexordp-eq (x # xs) (y # ys)

lemma lexordp-eq-simps [simp, code]:
  lexordp-eq [] ys = True
  lexordp-eq zs [] ⨾ zs = []
  lexordp-eq (x # xs) [] = False
  lexordp-eq (x # xs) (y # ys) ⨾ x < y ∨ ¬ y < x ∧ lexordp-eq xs ys
by(subst lexordp-eq.simps, fastforce)+

lemma lexordp-append-rightI: ys ≠ Nil ⇒ lexordp xs (xs @ ys)
by(induct xs)(auto simp add: neq-Nil-conv)

lemma lexordp-append-left-rightI: x < y ⇒ lexordp (us @ x # xs) (us @ y # ys)
by(induct us) auto

lemma lexordp-eq-refl: lexordp-eq xs xs
by(induct xs) simp-all

lemma lexordp-append-leftI: lexordp us vs ⇒ lexordp (xs @ us) (xs @ vs)
by(induct xs) auto

lemma lexordp-append-leftD: [ lexordp (xs @ us) (xs @ vs); ∀ a. ¬ a < a ] ⇒
lexordp us vs
by(induct xs) auto

lemma lexordp-irreflexive:
  assumes irrefl: ∀ x. ¬ x < x
theory "List"

shows \( \neg \text{lexordp } xs \, xs \)
proof
  assume \( \text{lexordp } xs \, xs \)
  thus \( False \) by (induct \( xs \, ys \equiv xs \)) simp-all: irrefl
qed

lemma lexordp-into-lexordp-eq:
  \( \text{lexordp } xs \, ys \Rightarrow \text{lexordp-eq } xs \, ys \)
by (induction rule: lexordp.induct) simp-all

lemma lexordp-eq-pref: lexordp-eq \( u \) (\( u @ v \))
by (metis append-Nil2 lexordp-append-right1 lexordp-eq-refl lexordp-into-lexordp-eq)
end

context order
begin
lemma lexordp-antisym:
  assumes \( \text{lexordp } xs \, ys \) \( \text{lexordp } ys \, xs \)
shows \( False \)
using assms by induct auto

lemma lexordp-irreflexive': \( \neg \text{lexordp } xs \, xs \)
by (rule lexordp-irreflexive) simp
end

context linorder begin
lemma lexordp-cases [consumes 1, case-names Nil Cons Cons-eq, cases pred: lexordp]:
  assumes lexordp \( xs \, ys \)
  obtains (Nil) \( y \, ys' \) where \( xs = [] \, ys = y \# \, ys' \)
  \| (Cons) \( x \, xs' \, y \, ys' \) where \( xs = x \# \, xs' \, ys = y \# \, ys' \, x < y \)
  \| (Cons-eq) \( x \, xs' \, ys' \) where \( xs = x \# \, xs' \, ys = x \# \, ys' \) lexordp \( xs' \, ys' \)
using assms by cases (fastforce simp add: not-less-iff-gr-or-eq+)

lemma lexordp-induct [consumes 1, case-names Nil Cons Cons-eq, induct pred: lexordp]:
  assumes major: lexordp \( xs \, ys \)
  and Nil: \( \forall y \, ys. \, P [] (y \# \, ys) \)
  and Cons: \( \forall x \, xs \, y \, ys. \, x < y \Rightarrow P (x \# \, xs) (y \# \, ys) \)
  and Cons-eq: \( \forall x \, xs \, ys. \, [ \text{lexordp } xs \, ys; \, P \, xs \, ys \] \Rightarrow P (x \# \, xs) (x \# \, ys) \)
  shows \( P \, xs \, ys \)
using major by induct (simp-all add: Nil Cons not-less-iff-gr-or-eq Cons-eq)
lemma lexordp-iff:
lexordp xs ys ←→ (∃x vs. ys = xs @ x # vs) ∨ (∃a b vs us. a < b ∧ xs = us @ a # vs ∧ ys = us @ b # ws)
(is ?lhs = ?rhs)
proof
assume ?lhs thus ?rhs
proof
induct
case Cons-eq thus ?case by simp (metis append.simps(2))
qed
del: disjCI intro: exI[where x=[]])+
next
assume ?rhs thus ?lhs
by(auto intro: lexordp-append-leftI[where us=[], simplified] lexordp-append-leftI)
qed

lemma lexordp-conv-lexord:
lexordp xs ys ←→ (xs, ys) ∈ lexord {(x, y). x < y}
by(simp add: lexordp-iff lexord-def)

lemma lexordp-eq-antisym:
assumes lexordp-eq xs ys lexordp-eq ys xs
shows xs = ys
using assms by induct simp-all

lemma lexordp-eq-trans:
assumes lexordp-eq xs ys lexordp-eq ys zs
shows lexordp-eq xs zs
using assms
by (induct arbitrary: zs) (case-tac zs; auto)+

lemma lexordp-trans:
assumes lexordp xs ys lexordp ys zs
shows lexordp xs zs
using assms
by (induct arbitrary: zs) (case-tac zs; auto)+

lemma lexordp-linear: lexordp xs ys ∨ xs = ys ∨ lexordp ys xs
by(induct xs arbitrary: ys; case_tac ys; fastforce)

lemma lexordp-conv-lexordp-eq: lexordp xs ys ←→ lexordp-eq xs ys ∧ ¬ lexordp-eq
xs ys
(is ?lhs ←→ ?rhs)
proof
assume ?lhs
hence ¬ lexordp-eq ys xs by induct simp-all
with (?lhs) show ?rhs by (simp add: lexordp-into-lexordp-eq)
next
assume ?rhs
hence lexordp-eq xs ys ¬ lexordp-eq ys xs by simp-all
thus ?lhs by induct simp-all
qed

lemma lexordp-eq-conv-lexord: lexordp-eq xs ys ←→ xs = ys ∨ lexordp xs ys
by(auto simp add: lexordp-conv-lexordp-eq lexordp-eq-refl dest: lexordp-eq-antisym)

lemma lexordp-eq-linear: lexordp-eq xs ys ∨ lexordp-eq ys xs
by (induct xs arbitrary: ys) (case-tac ys; auto)+

lemma lexordp-eq-refl: refl (x, y) ∈ lexordp-eq xs ys

end

66.4.3 Lexicographic combination of measure functions

These are useful for termination proofs

definition measures fs = inv-image (lex less-than) (%a. map (%f. f a) fs)

lemma wf-measures[simp]: wf (measures fs)
unfolding measures-def by blast

lemma in-measures[simp]:
  (x, y) ∈ measures [] = False
  (x, y) ∈ measures (f # fs)
  = (f x < f y ∨ (f x = f y ∧ (x, y) ∈ measures fs))
unfolding measures-def by auto

lemma measures-less: f x < f y ⇒ (x, y) ∈ measures (f#fs)
by simp

lemma measures-leqseq: f x ≤ f y ⇒ (x, y) ∈ measures fs ⇒ (x, y) ∈ measures (f#fs)
by auto

66.4.4 Lifting Relations to Lists: one element

definition listrel1 :: ('a × 'a) set ⇒ ('a list × 'a list) set where
listrel1 r = \{ (xs,ys).
  ∃ us z z’ vs. xs = us @ z # vs ∧ (z,z’) ∈ r ∧ ys = us @ z’ # vs \}

lemma listrel1I:
  (x, y) ∈ r;  xs = us @ x # vs;  ys = us @ y # vs ⊢ (xs,ys) ∈ listrel1 r
unfolding listrel1-def by auto
lemma listrel1E:
\[ \begin{array}{l}
\exists x y u v. ((x, y) \in r; \ x = u \# x \# v; \ y = u \# y \# v) \implies P
\end{array} \]
unfolding listrel1-def by auto

lemma not-Nil-listrel1 [iff]: ([], xs) \notin listrel1 r
unfolding listrel1-def by blast

lemma not-listrel1-Nil [iff]: (xs, []) \notin listrel1 r
unfolding listrel1-def by blast

lemma Cons-listrel1-Cons [iff]:
\[ ((x \# xs, y \# ys) \in listrel1 r) \leftarrow (x, y) \in r \land xs = ys \lor x = y \land (xs, ys) \in listrel1 r \]
by (simp add: listrel1-def Cons-eq-append-cone) (blast)

lemma listrel1I1: (x, y) \in r \implies (x \# xs, y \# ys) \in listrel1 r
by fast

lemma listrel1I2: (xs, ys) \in listrel1 r \implies (x \# xs, x \# ys) \in listrel1 r
by fast

lemma append-listrel1I:
\[ ((xs, ys) \in listrel1 r \land us = vs \lor xs = ys \land (us, vs) \in listrel1 r) \implies (xs @ us, ys @ vs) \in listrel1 r \]
unfolding listrel1-def by auto (blast intro: append-eq-appendI)+

lemma Cons-listrel1E1[elim!]:
assumes (x \# xs, y \# ys) \in listrel1 r
and \( \forall y. \ y = y \# xs \implies (x, y) \in r \implies R \)
and \( \forall zs. \ zs = x \# zs \implies (xz, zs) \in listrel1 r \implies R \)
shows R
using assms by (cases ys) blast+

lemma Cons-listrel1E2[elim!]:
assumes (xs, y \# ys) \in listrel1 r
and \( \forall x. \ x = x \# ys \implies (x, y) \in r \implies R \)
and \( \forall zs. \ zs = y \# zs \implies (zs, ys) \in listrel1 r \implies R \)
shows R
using assms by (cases xs) blast+

lemma snoc-listrel1-snoc-iff:
\[ ((xs @ [x], ys @ [y]) \in listrel1 r) \leftarrow (xs, ys) \in listrel1 r \land x = y \lor xs = ys \land (x, y) \in r \leftarrow ?L \leftarrow ?R \]
proof
assume ?L thus ?R
by (fastforce simp: listrel1-def snoc-iff-butlast butlast-append)
next
  assume \( ?R \) then show \( ?L \) unfolding listrel1-def by force
qed

lemma listrel1-eq-len: \( (xs,ys) \in \text{listrel1 } r \implies \text{length } xs = \text{length } ys \)
unfolding listrel1-def by auto

lemma listrel1-mono:
  \( r \subseteq s \implies \text{listrel1 } r \subseteq \text{listrel1 } s \)
unfolding listrel1-def by blast

lemma listrel1-converse: \( \text{listrel1 } (r^{-1}) = (\text{listrel1 } r)^{-1} \)
unfolding listrel1-def by blast

lemma in-listrel1-converse:
  \( (x,y) \in \text{listrel1 } (r^{-1}) \iff (x,y) \in (\text{listrel1 } r)^{-1} \)
unfolding listrel1-def by blast

lemma listrel1-iff-update:
  \( (xs,ys) \in (\text{listrel1 } r) \iff (\exists \, n \, (xs ! n, y) \in r \land n < \text{length } xs \land ys = xs[n:=y]) \) (is \( ?L \iff ?R \))
proof
  assume \( ?L \)
  then obtain \( x \, y \, u \, v \) where
    \( xs = u @ x \# v \) \( ys = u @ y \# v \)
    \((x,y) \in r\)
    unfolding listrel1-def by auto
  then have \( ys = xs[length u := y] \) and \( \text{length } u < \text{length } xs \)
  and \( (xs ! length u, y) \in r \) by auto
  then show \( ?R \) by auto
next
  assume \( ?R \)
  then obtain \( x \, y \, n \) where
    \( (xs!n, y) \in r \land n < \text{size } xs \)
    \( ys = xs[n:=y] \)
    \( x = xs!n \)
  by auto
  then obtain \( u \, v \) where
    \( xs = u @ x \# v \)
    \( ys = u @ y \# v \)
  and \( (x,y) \in r \)
  by (auto intro: upd-conv-take-nth-drop id-take-nth-drop)
  then show \( ?L \) by (auto simp: listrel1-def)
qed

Accessible part and wellfoundedness:

lemma Cons-acc-listrel1I [intro]:
  \( x \in \text{Wellfounded.acc } r \implies xs \in \text{Wellfounded.acc } \) (listrel1 \( r \) \( \implies x \# xs \) \in \text{Wellfounded.acc } (\text{listrel1 } r)\)
proof (induction arbitrary: \( xs \) set: \text{Wellfounded.acc})
  case outer: \( (1 \, u) \)
  show \( ?case \)
  proof (induct \( xs \) rule: acc-induct)
    case \( 1 \)
    show \( xs \in \text{Wellfounded.acc } (\text{listrel1 } r) \)
by (simp add: outer.prems)
qed (metis (no-types, lifting) Cons-listrel1E2 acc.simps outer.IH)
qed

lemma lists-accD: xs ∈ lists (Wellfounded.acc) r =⇒ xs ∈ Wellfounded.acc (listrel1 r)
proof (induct set: lists)
case Nil
then show ?case
  by (meson acc.intro not-listrel1-Nil)
next
case (Cons a l)
then show ?case
  by blast
qed

lemma lists-accI: xs ∈ Wellfounded.acc (listrel1 r) =⇒ xs ∈ lists (Wellfounded.acc) r
proof (induction set: Wellfounded.acc)
case (1 x)
then have ∃ u v. ⟨u ∈ set x; (v, u) ∈ r⟩ =⇒ v ∈ Wellfounded.acc r
  by (metis in-lists-conv-set in-set-conv-decomp listrel1I)
then show ?case
  by (meson acc.intro in-listsI)
qed

lemma wf-listrel1-iff[simp]: wf(listrel1 r) = wf r
by (auto simp: wf-acc-iff
    intro: lists-accD lists-accI[THEN Cons-in-lists-iff[THEN iffD1, THEN conjunct1]]
)

66.4.5 Lifting Relations to Lists: all elements

inductive-set
listrel :: ('a × 'b) set ⇒ ('a list × 'b list) set
for r :: ('a × 'b) set
where
  Nil: ⟨[],[]⟩ ∈ listrel r
| Cons: ⟨(x,y) ∈ r; (xs,ys) ∈ listrel r⟩ =⇒ (x#xs, y#ys) ∈ listrel r

inductive-cases listrel-Nil1 [elim!]: ⟨[],xs⟩ ∈ listrel r
inductive-cases listrel-Nil2 [elim!]: ⟨xs,[]⟩ ∈ listrel r
inductive-cases listrel-Cons1 [elim!]: ⟨y#ys,xs⟩ ∈ listrel r
inductive-cases listrel-Cons2 [elim!]: ⟨xs,y#ys⟩ ∈ listrel r

lemma listrel-eq-len: ⟨xs, ys⟩ ∈ listrel r =⇒ length xs = length ys
by (induct rule: listrel.induct) auto
lemma listrel-iff-zip [code-unfold]: \((xs, ys) \in \text{listrel } r \iff \) 

\[
\text{length } xs = \text{length } ys \land (\forall (x, y) \in \text{set(zip } xs \ ys). \ (x, y) \in r) \ (\text{is } ?L \iff ?R) 
\]

proof
assume ?L thus ?R by induct (auto intro: listrel-eq-len)
next
assume ?R thus ?L
apply (clarify)
by (induct rule: list-induct2) (auto intro: listrel.intros)
qed

lemma listrel-iff-nth: \((xs, ys) \in \text{listrel } r \iff \) 

\[
\text{length } xs = \text{length } ys \land (\forall n < \text{length } xs. \ (xs!n, ys!n) \in r) \ (\text{is } ?L \iff ?R) 
\]

by (auto simp add: all-set-conv-all-nth listrel-iff-zip)

lemma listrel-mono: \(r \subseteq s \implies \text{listrel } r \subseteq \text{listrel } s\)

by (meson listrel-iff-nth subrelI subset-eq)

lemma listrel-subset:
assumes \(r \subseteq A \times A\) shows \(\text{listrel } r \subseteq \text{lists } A \times \text{lists } A\)

proof clarify
show \(a \in \text{lists } A \land b \in \text{lists } A\) if \((a, b) \in \text{listrel } r\) for \(a\) \(b\)
using that assms unfolding refl-on-def
by (induction \(l\), auto intro: listrel.intros)
then show ?thesis
by (meson assms listrel-subset refl-on-def)
qed

lemma listrel-refl-on:
assumes refl-on \(A\) \(r\) shows refl-on \((\text{lists } A)\) \((\text{listrel } r)\)

proof
have \(l \in \text{lists } A \implies (l, l) \in \text{listrel } r\) for \(l\)
using assms unfolding refl-on-def
by (induction \(l\), auto intro: listrel.intros)
then show ?thesis
by (meson assms listrel-subset refl-on-def)
qed

lemma listrel-sym: \(\text{sym } r \implies \text{sym } (\text{listrel } r)\)
by (simp add: listrel-iff-nth sym-def)

lemma listrel-trans:
assumes \(\text{trans } r\) shows trans \((\text{listrel } r)\)

proof
have \((x, z) \in \text{listrel } r\) if \((x, y) \in \text{listrel } r\) \((y, z) \in \text{listrel } r\) for \(x\) \(y\) \(z\)
using that
proof induction
case \((\text{Cons } x y xs ys)\)
then show ?case
by clarsimp (metis assms listrel.Cons listrel-iff-nth transD)
qed auto
then show ?thesis
by blast
theorem equiv-listrel: equiv A r ⟷ equiv (lists A) (listrel r)
  by (simp add: equiv_def listrel-refl-on listrel-sym listrel-trans)

lemma listrel-rtrancl-refl[iff]: (xs, xs) ∈ listrel(r^*)
  using listrel-refl-on[of UNIV, OF refl-rtrancl]
  by (auto simp add: refl-on_def)

lemma listrel-rtrancl-trans:
[[xs,ys) ∈ listrel(r^*); (ys, zs) ∈ listrel(r^*)]] ⟹ (xs, zs) ∈ listrel(r^*)
by (metis listrel1I rtrancl-rtrancl-into-rtrancl)

lemma listrel-nil: listrel r "{[]} = {[[]}"
by (blast intro: listrel.intros)

Relating listrel1, listrel and closures:

lemma listrel1-rtrancl-subset-rtrancl-listrel1: listrel1(r^*) ⊆ (listrel1 r)^*
proof (rule subrelI)
  fix xs ys assume 1: (xs, ys) ∈ listrel1(r^*)
  { fix x y us vs
    have (x, y) ∈ r^* ⟹ (us @ x # vs, us @ y # vs) ∈ (listrel1 r)^*
    proof (induct rule: rtrancl.induct)
      case rtrancl-refl show ?case by simp
      next
      case rtrancl-into-rtrancl thus ?case
      by (metis listrel1I rtrancl-rtrancl-into-rtrancl)
    qed }
  thus (xs, ys) ∈ (listrel1 r)^* using 1 by (blast elim: listrel1E)
qed

lemma rtrancl-listrel1-eq-len: (x, y) ∈ (listrel1 r)^* ⟹ length x = length y
by (induct rule: rtrancl.induct) (auto intro: listrel1-eq-len)

lemma rtrancl-listrel1-cons1:
(ys, zs) ∈ (listrel1 r)^* ⟹ (x#xs, x#ys) ∈ (listrel1 r)^*
proof (induction rule: rtrancl.induct)
  case rtrancl-into-rtrancl a b c
  then show ?case
  by (metis listrel1I rtrancl-rtrancl-into-rtrancl)
qed auto

lemma rtrancl-listrel1-cons2:
(x, y) ∈ r^* ⟹ (xs, ys) ∈ (listrel1 r)^* ⟹ (x # xs, y # ys) ∈ (listrel1 r)^*
by (meson in_mono listrel1I listrel1-rtrancl-subset-rtrancl-listrel1 rtrancl-listrel1-cons1 rtrancl-listrel1-cons2)
lemma listrel1-subset-listrel:
\[ r \subseteq r' \implies \text{refl } r' \implies \text{listrel1 } r \subseteq \text{listrel } (r') \]
by (auto elim: listrel1E simp add: listrel_iff_zip set-zip refl-on-def)

lemma listrel-reflcl-if-listrel1:
\[(xs, ys) \in \text{listrel1 } r \implies (xs, ys) \in \text{listrel } (r^*)\]
by (erule listrel1E) (auto simp add: listrel_iff_zip set-zip)

lemma listrel-rtrancl-eq-rtrancl-listrel1: \(\text{listrel } (r^*) = (\text{listrel1 } r)^*\)
proof
\{ fix \(x\) \(y\) assume \((x, y) \in \text{listrel } (r^*)\)
  then have \((x, y) \in (\text{listrel1 } r)^*\)
  by induct (auto intro: rtrancl-listrel1-ConsI2) \}
then show \(\text{listrel } (r^*) \subseteq (\text{listrel1 } r)^*\)
  by (rule subrelI)
next
show \(\text{listrel } (r^*) \supseteq (\text{listrel1 } r)^*\)
proof (rule subrelI)
fix \(xs\) \(ys\) assume \((xs, ys) \in (\text{listrel1 } r)^*\)
then show \((xs, ys) \in \text{listrel } (r^*)\)
proof induct
  case base show \(?case\) by (auto simp add: listrel_iff_zip set-zip)
next
  case (step ys zs)
  thus \(?case\) by (metis listrel-reflcl-if-listrel1 listrel-rtrancl-trans)
qed
qed

lemma rtrancl-listrel1-if-listrel:
\[(xs, ys) \in \text{listrel } r \implies (xs, ys) \in (\text{listrel1 } r)^*\]
by (metis listrel-rtrancl-eq-rtrancl-listrel1 subsetD [OF listrel_mono] r-into-rtrancl subsetI)

lemma listrel-subset-rtrancl-listrel1: \(\text{listrel } r \subseteq (\text{listrel1 } r)^*\)
by (fast intro: rtrancl-listrel1-if-listrel)

66.5 Size function

lemma [measure-function]: \(\text{is-measure } f \implies \text{is-measure } (\text{size-list } f)\)
by (rule is-measure-trivial)

lemma [measure-function]: \(\text{is-measure } f \implies \text{is-measure } (\text{size-option } f)\)
by (rule is-measure-trivial)

lemma size-list-estimation[termination-simp]:
\(x \in \text{set } xs \implies y < f x \implies y < \text{size-list } f xs\)
by (induct xs) auto

**lemma** size-list-estimation[termination-simp]:
\[ x \in \text{set } xs \implies y \leq f x \implies y \leq \text{size-list } f xs \]
by (induct xs) auto

**lemma** size-list-map[simp]: \[ \text{size-list } f (\text{map } g xs) = \text{size-list } (f \circ g) xs \]
by (induct xs) auto

**lemma** size-list-append[simp]: \[ \text{size-list } f (xs @ ys) = \text{size-list } f xs + \text{size-list } f ys \]
by (induct xs, auto)

**lemma** size-list-pointwise[termination-simp]:
\[ (\forall x. x \in \text{set } xs \implies f x \leq g x) \implies \text{size-list } f xs \leq \text{size-list } g xs \]
by (induct xs) force+

### 66.6 Monad operation

**definition** bind :: \('a list \Rightarrow (\'a \Rightarrow \'b list) \Rightarrow \'b list\) where
\[ \text{bind } xs f = \text{concat } (\text{map } f xs) \]

hide-const (open) bind

**lemma** bind-simps [simp]:
\[ \text{List.bind } [] f = [] \]
\[ \text{List.bind } (x \# xs) f = f x @ \text{List.bind } xs f \]
by (simp-all add: bind-def)

**lemma** list-bind-cong [fundef-cong]:
\[ \text{assumes } xs = ys \quad (\forall x. x \in \text{set } xs \implies f x = g x) \]
\[ \text{shows } \quad \text{List.bind } xs f = \text{List.bind } ys g \]
proof –
from assms(2) have \( \text{List.bind } xs f = \text{List.bind } xs g \)
by (induction xs) simp-all
with assms(1) show ?thesis by simp
qed

**lemma** set-list-bind: \( \text{set } (\text{List.bind } xs f) = (\bigcup x \in \text{set } xs. \text{set } (f x)) \)
by (induction xs) simp-all

### 66.7 Code generation

Optional tail recursive version of \( \text{map} \). Can avoid stack overflow in some target languages.

**fun** map-tailrec-rev :: \('a \Rightarrow \'b) \Rightarrow \(\'a \Rightarrow \'b \Rightarrow \)\'b list\Rightarrow \'b list\ where
\[ \text{map-tailrec-rev } f [] bs = bs \]
\[ \text{map-tailrec-rev } f (a\#as) bs = \text{map-tailrec-rev } f as (f a \# bs) \]

**lemma** map-tailrec-rev:
map-tailrec-rev f as bs = rev(map f as) @ bs
by(induction as arbitrary: bs) simp-all

definition map-tailrec :: ('a => 'b) => 'a list => 'b list where
map-tailrec f as = rev (map-tailrec-rev f as [])

Code equation:

lemma map-eq-map-tailrec: map = map-tailrec
by(simp add: fun-eq-iff map-tailrec-def map-tailrec-rev)

66.7.1 Counterparts for set-related operations

definition member :: 'a list => 'a => bool where
[code-abbrev]: member xs x <-> x ∈ set xs

Use member only for generating executable code. Otherwise use x ∈ set xs
instead — it is much easier to reason about.

lemma member-rec [code]:
member (x # xs) y <-> x = y ∨ member xs y
member [] y <-> False
by (auto simp add: member-def)

lemma in-set-member :
x ∈ set xs <-> member xs x
by (simp add: member-def)

lemmas list-all-iff [code-abbrev] = fun-cong[OF list.pred-set]

definition list-ex :: ('a => bool) => 'a list => bool where
list-ex-iff [code-abbrev]: list-ex P xs <-> Bex (set xs) P

definition list-ex1 :: ('a => bool) => 'a list => bool where
list-ex1-iff [code-abbrev]: list-ex1 P xs <-> (∃! x. x ∈ set xs ∧ P x)

Usually you should prefer ∀ x∈set xs, ∃x∈set xs and ∃! x. x∈set xs ∧ - over
list-all, list-ex and list-ex1 in specifications.

lemma list-all-simps [code]:
list-all P (x # xs) <-> P x ∧ list-all P xs
list-all P [] <-> True
by (simp-all add: list-all-iff)

lemma list-ex-simps [simp, code]:
list-ex P (x # xs) <-> P x ∨ list-ex P xs
list-ex P [] <-> False
by (simp-all add: list-ex-iff)

lemma list-ex1-simps [simp, code]:
list-ex1 P [] = False
list-ex1 \( P \ (x \neq xs) = \begin{cases} \text{if } P \ x \ \text{then } \text{list-all} \ (\lambda y. \neg P \ y \lor x = y) \ xs \ \text{else} \ \text{list-ex1} \ P \ xs \end{cases} \)
by (auto simp add: list-ex1-iff list-all-iff)

lemma Ball-set-list-all:
\( \text{Ball} \ (\text{set} \ xs) \ P \leftrightarrow \text{list-all} \ P \ xs \)
by (simp add: list-all-iff)

lemma Bex-set-list-ex:
\( \text{Bex} \ (\text{set} \ xs) \ P \leftrightarrow \text{list-ex} \ P \ xs \)
by (simp add: list-ex-iff)

lemma list-all-append [simp]:
\( \text{list-all} \ P \ (xs @ ys) \leftrightarrow \text{list-all} \ P \ xs \land \text{list-all} \ P \ ys \)
by (auto simp add: list-all-iff)

lemma list-ex-append [simp]:
\( \text{list-ex} \ P \ (xs @ ys) \leftrightarrow \text{list-ex} \ P \ xs \lor \text{list-ex} \ P \ ys \)
by (auto simp add: list-ex-iff)

lemma list-all-rev [simp]:
\( \text{list-all} \ P \ (\text{rev} \ xs) \leftrightarrow \text{list-all} \ P \ xs \)
by (simp add: list-all-iff)

lemma list-ex-rev [simp]:
\( \text{list-ex} \ P \ (\text{rev} \ xs) \leftrightarrow \text{list-ex} \ P \ xs \)
by (simp add: list-ex-iff)

lemma list-all-length:
\( \text{list-all} \ P \ xs \leftrightarrow (\forall n < \text{length} \ xs. \ P \ (xs \ ! n)) \)
by (auto simp add: list-all-iff set-conv-nth)

lemma list-ex-length:
\( \text{list-ex} \ P \ xs \leftrightarrow (\exists n < \text{length} \ xs. \ P \ (xs \ ! n)) \)
by (auto simp add: list-ex-iff set-conv-nth)

lemmas list-all-cong [fundef-cong] = list-pred-cong

lemma list-ex-cong [fundef-cong]:
\( xs = ys \Longrightarrow (\forall x. \ x \in \text{set} \ ys \Longrightarrow f \ x = g \ x) \Longrightarrow \text{list-ex} \ f \ xs = \text{list-ex} \ g \ ys \)
by (simp add: list-ex-iff)

definition can-select :: \('a \Rightarrow \text{bool}\) \Rightarrow 'a set \Rightarrow \text{bool} where
[code-abbrev]: can-select \( P \ A = (\exists x \in A. \ P \ x) \)

lemma can-select-set-list-ex1 [code]:
\( \text{can-select} \ P \ (\text{set} \ A) = \text{list-ex1} \ P \ A \)
by (simp add: list-ex1-iff can-select-def)

Executable checks for relations on sets
**THEORY “List”**

**definition** listrel1p :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool where
listrel1p r xs ys = ((xs, ys) ∈ listrel1 {(x, y). r x y})

**lemma** [code-unfold]:
(xs, ys) ∈ listrel1 r = listrel1p (λx y. (x, y) ∈ r) xs ys

**unfolding** listrel1p-def by auto

**definition**
lexordp :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool where
lexordp r xs ys = ((xs, ys) ∈ lexord {(x, y). r x y})

**lemma** [code-unfold]:
(xs, ys) ∈ lexord r = lexordp (λx y. (x, y) ∈ r) xs ys

**unfolding** lexordp-def by auto

Bounded quantification and summation over nats.

**lemma** atMost-upto [code-unfold]:
{..n} = set [0..<Suc n]
by auto

**lemma** atLeast-upt [code-unfold]:
{..<n} = set [0..<n]
by auto

**lemma** greaterThanLessThan-upt [code-unfold]:
{n..<m} = set [Suc n..<m]
by auto

**lemmas** atLeastLessThan-upt [code-unfold] = set-upt [symmetric]

**lemma** greaterThanAtMost-upt [code-unfold]:
{n..<m} = set [Suc n..<Suc m]
by auto

**lemma** atLeastAtMost-upt [code-unfold]:
{n..m} = set [n..<Suc m]
THEORY "List"

by auto

lemma all-nat-less-eq [code-unfold]:
(∀ m<n::nat. P m) ↔ (∀ m ∈ {0..<n}. P m)
by auto

lemma ex-nat-less-eq [code-unfold]:
(∃ m<n::nat. P m) ↔ (∃ m ∈ {0..<n}. P m)
by auto

lemma all-nat-less [code-unfold]:
(∀ m≤n::nat. P m) ↔ (∀ m ∈ {0..n}. P m)
by auto

lemma ex-nat-less [code-unfold]:
(∃ m≤n::nat. P m) ↔ (∃ m ∈ {0..n}. P m)
by auto

Bounded LEAST operator:
definition Bleast S P = (LEAST x. x ∈ S ∧ P x)
definition abort-Bleast S P = (LEAST x. x ∈ S ∧ P x)
declare [[code abort: abort-Bleast]]

lemma Bleast-code [code]:
Bleast (set xs) P = (case filter P (sort xs) of
x#xs ⇒ x |
[] ⇒ abort-Bleast (set xs) P)
proof (cases filter P (sort xs))
case Nil thus ?thesis by (simp add: Bleast-def abort-Bleast-def)
next
case (Cons x ys)
have (LEAST x. x ∈ set xs ∧ P x) = x
proof (rule Least-equality)
show x ∈ set xs ∧ P x
  by (metis Cons Cons-eq-filter-iff in-set-conv-decomp set-sort)
next
fix y assume y ∈ set xs ∧ P y
hence y ∈ set (filter P xs) by auto
thus x ≤ y
  by (metis Cons eq-iff filter-sort set-ConsD set-sort sorted-wrt.simps(2) sorted-sort)
qed
thus ?thesis using Cons by (simp add: Bleast-def)
qed

declare Bleast-def[symmetric, code-unfold]

Summation over ints.
lemma greaterThanLessThan-upto [code-unfold]:
\{i ..< j :: int\} = set [i+1..j - 1]
by auto

lemma atLeastLessThan-upto [code-unfold]:
\{i ..< j :: int\} = set [i..j - 1]
by auto

lemma greaterThanAtMost-upto [code-unfold]:
\{i ..< j :: int\} = set [i+1..j]
by auto

lemmas atLeastAtMost-upto [code-unfold] = set-upto [symmetric]

66.7.2 Optimizing by rewriting

definition null :: 'a list ⇒ bool where
\[\text{code-abbrev}: \text{null \ } xs \longleftrightarrow \text{xs} = []\]

Efficient emptiness check is implemented by null.

lemma null-rec [code]:
null (x # xs) ←→ False
null [] ←→ True
by (simp-all add: null-def)

lemma eq-Nil-null:
xs = [] ←→ null xs
by (simp add: null-def)

lemma equal-Nil-null [code-unfold]:
HOL.equal xs [] ←→ null xs
HOL.equal [] = null
by (auto simp add: equal null-def)

definition maps :: ('a ⇒ 'b list) ⇒ 'a list ⇒ 'b list where
\[\text{code-abbrev}: \text{maps \ } f \ \text{xs} = \text{concat (map} \ f \ \text{xs)}\]

definition map-filter :: ('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list where
\[\text{code-post}: \text{map-filter} \ f \ \text{xs} = \text{map (the o f) (filter (λx. \ f \ \ x \neq \ None) \ xs)}\]

Operations maps and map-filter avoid intermediate lists on execution – do not use for proving.

lemma maps-simps [code]:
maps f (x # xs) = f x @ maps f xs
maps f [] = []
by (simp-all add: maps-def)

lemma map-filter-simps [code]:
map-filter \( f \) (\( x \neq xs \)) = (case \( f x \) of None \( \Rightarrow \) map-filter \( f \) \( xs \) | Some \( y \) \( \Rightarrow y \neq \) map-filter \( f \) \( xs \))

\( \text{map-filter} \) \( [] \) = \( [] \)

by (simp-all add: map-filter-def split: option.split)

**lemma** concat-map-maps:
concat (map \( f \) \( xs \)) = maps \( f \) \( xs \)
by (simp add: maps-def)

**lemma** map-filter-map-filter [code-unfold]:
\( \text{map} \) \( f \) \( \text{filter} \) \( P \) \( xs \) = \( \text{map-filter} \) (\( \lambda \) \( x \). if \( P x \) then Some \( f x \) else None) \( xs \)
by (simp add: map-filter-def)

Optimized code for \( \forall i \in \{a..b::\text{int}\} \) and \( \forall n: \{a..<b::\text{nat}\} \) and similarly for \( \exists \).

**definition** all-interval-nat :: (nat \( \Rightarrow \) bool) \( \Rightarrow \) nat \( \Rightarrow \) nat \( \Rightarrow \) bool where
\( \text{all-interval-nat} \) \( P \) \( i \) \( j \) \( \iff \) (\( \forall n \in \{i..<j\} \). \( P n \))

**lemma** [code]:
\( \text{all-interval-nat} \) \( P \) \( i \) \( j \) \( \iff \) \( i \geq j \lor P i \land \text{all-interval-nat} \) \( P \) \( \text{Suc} i \) \( j \)

**proof** –
\( \text{have} \) *: \( \forall n. P i \Rightarrow \forall n \in \{\text{Suc} i..<j\}. P n \Rightarrow i \leq n \Rightarrow n < j \Rightarrow P n \)
using \( \text{le-less-Suc-eq} \) by \( \text{fastforce} \)
show \( ?\text{thesis} \) by (auto simp add: all-interval-nat-def intro: *)

qed

**lemma** list-all-iff-all-interval-nat [code-unfold]:
\( \text{list-all} \) \( P \) \( \{i..<j\} \) \( \iff \) \( \text{all-interval-nat} \) \( P \) \( i \) \( j \)
by (simp add: list-all-iff all-interval-nat-def)

**lemma** list-ex-iff-not-all-inverval-nat [code-unfold]:
\( \text{list-ex} \) \( P \) \( \{i..<j\} \) \( \iff \) \( \neg \text{all-interval-nat} \) (\( \text{Not} \circ P \)) \( i \) \( j \)
by (simp add: list-ex-iff all-interval-nat-def)

**definition** all-interval-int :: (int \( \Rightarrow \) bool) \( \Rightarrow \) int \( \Rightarrow \) int \( \Rightarrow \) bool where
\( \text{all-interval-int} \) \( P \) \( i \) \( j \) \( \iff \) (\( \forall k \in \{i..j\} \). \( P k \))

**lemma** [code]:
\( \text{all-interval-int} \) \( P \) \( i \) \( j \) \( \iff \) \( i > j \lor P i \land \text{all-interval-int} \) \( P \) \( i + 1 \) \( j \)

**proof** –
\( \text{have} \) *: \( \forall k. P i \Rightarrow \forall k \in \{i+1..<j\}. P k \Rightarrow i \leq k \Rightarrow k \leq j \Rightarrow P k \)
by (smt (verit, best) atLeastAtMost-iff)
show \( ?\text{thesis} \) by (auto simp add: all-interval-int-def intro: *)

qed

**lemma** list-all-iff-all-interval-int [code-unfold]:
\( \text{list-all} \) \( P \) \( \{i..<j\} \) \( \iff \) \( \text{all-interval-int} \) \( P \) \( i \) \( j \)
by (simp add: list-all-iff all-interval-int-def)

**lemma** list-ex-iff-not-all-inverval-int [code-unfold]:
list-ex $P$ [$i..j$] $\iff$ $\neg$ (all-interval-int ($\neg \circ P$) $i$ $j$)  
by (simp add: list-ex-iff all-interval-int-def)

optimized code (tail-recursive) for $\text{length}$

definition $\text{gen-length} :: \text{nat} \Rightarrow 'a \text{ list} \Rightarrow \text{nat}$  
where $\text{gen-length} n \; \text{xs} = n + \text{length} \; \text{xs}$

lemma $\text{gen-length-code}$ [code]:  
\begin{align*}
\text{gen-length} n \; [] &= n \\
\text{gen-length} n \; (x \# \text{xs}) &= \text{gen-length} (\text{Suc} \; n) \; \text{xs}
\end{align*}
by (simp-all add: gen-length-def)

declare list.size($3-4$)[code del]

lemma $\text{length-code}$ [code]: $\text{length} = \text{gen-length} \; 0$
by (simp add: gen-length-def fun-eq-iff)

hide-const (open) member null maps map-filter all-interval-nat all-interval-int

66.7.3 Pretty lists

ML

(* Code generation for list literals. *)

signature LIST-CODE =
  sig
    val add-literal-list: string -> theory -> theory
  end;

structure List-Code : LIST-CODE =
  struct

open Basic-Code-Thingol;

fun implode-list $t =$ 
  let
    fun dest-cons (IConst { sym = Code-Symbol.Constant $\text{const-name}$ Cons, ... }) $\# \; t1 \; \# \; t2$  = SOME (t1, t2)
      | dest-cons _ = NONE;
    val (ts, t') = Code-Thingol.unfoldr dest-cons $t$;
    in case $t'$ of $\text{IConst}$ { sym = Code-Symbol.Constant $\text{const-name}$ Nil, ... } => SOME $\text{ts}$
        | _ => NONE
    end;

fun print-list (target-fxy, target-cons) pr fxy $t1 \; t2 =$ 
  Code-Printer.brackify-infix (target-fxy, Code-Printer.R) fxy (
fun add-literal-list target =
  let
  fun pretty literals pr - vars fxy [(t1, -), (t2, -)] =
    case Option.map (cons t1) (implode-list t2)
    of SOME ts =>
      Code-Printer.literal-list literals (map (pr vars Code-Printer.NOBR) ts)
    | NONE =>
      print-list (Code-Printer.infix-cons literals) (pr vars) fxy t1 t2;
  in
  Code-Target.set-printings (Code-Symbol.Constant (const-name: Cons),
  [(target, SOME (Code-Printer.complex-const-syntax (2, pretty))))])
  end
end;

> code-printing

  type-constructor list -> (SML) - list
  and (OCaml) - list
  and (Haskell) ![(-)]
  and (Scala) List[-]
  | constant Nil ->
    (SML) []
  and (OCaml) []
  and (Haskell) []
  and (Scala) !Nil
  | class-instance list :: equal ->
    (Haskell) =
  | constant HOL.equal :: 'a list ⇒ 'a list ⇒ bool →
    (Haskell) infix 4 ==

setup (fold (List-Code.add-literal-list) [SML, OCaml, Haskell, Scala]);

code-reserved SML list

code-reserved OCaml list

66.7.4 Use convenient predefined operations

code-printing
  constant (@) →
THEORY "List"

(SML) infixr 7 @
and (OCaml) infixr 6 @
and (Haskell) infixr 5 ++
and (Scala) infixl 7 ++
| constant map ↦
  (Haskell) map
| constant filter ↦
  (Haskell) filter
| constant concat ↦
  (Haskell) concat
| constant List.maps ↦
  (Haskell) concatMap
| constant rev ↦
  (Haskell) reverse
| constant zip ↦
  (Haskell) zip
| constant List.null ↦
  (Haskell) null
| constant takeWhile ↦
  (Haskell) takeWhile
| constant dropWhile ↦
  (Haskell) dropWhile
| constant list-all ↦
  (Haskell) all
| constant list-ex ↦
  (Haskell) any

66.7.5 Implementation of sets by lists

lemma is-empty-set [code]:
Set.is-empty (set xs) ←→ List.null xs
by (simp add: Set.is-empty-def null-def)

lemma empty-set [code]:
{} = set []
by simp

lemma UNIV-coset [code]:
UNIV = List.coset []
by simp

lemma compl-set [code]:
− set xs = List.coset xs
by simp

lemma compl-coset [code]:
− List.coset xs = set xs
by simp
THEORY “List”

lemma [code]:
\[
x \in \text{set } xs \iff \text{List.member } xs \ x
\]
\[
x \in \text{List.coset } xs \iff \neg \text{List.member } xs \ x
\]
by (simp-all add: member-def)

lemma insert-code [code]:
\[
\text{insert } x (\text{set } xs) = \text{set } (\text{List.insert } x xs)
\]
\[
\text{insert } x (\text{List.coset } xs) = \text{List.coset } (\text{removeAll } x xs)
\]
by simp-all

lemma remove-code [code]:
\[
\text{Set.remove } x (\text{set } xs) = \text{set } (\text{removeAll } x xs)
\]
\[
\text{Set.remove } x (\text{List.coset } ys) = \text{List.coset } (\text{List.insert } x xs)
\]
by (simp-all add: remove-def Compl-insert)

lemma filter-set [code]:
\[
\text{Set.filter } P (\text{set } xs) = \text{set } (\text{filter } P xs)
\]
by auto

lemma image-set [code]:
\[
\text{image } f (\text{set } xs) = \text{set } (\text{map } f xs)
\]
by simp

lemma subset-code [code]:
\[
\text{set } xs \subseteq B \iff (\forall x \in \text{set } xs. \ x \in B)
\]
\[
A \subseteq \text{List.coset } ys \iff (\forall y \in \text{set } ys. \ y \notin A)
\]
\[
\text{List.coset } [] \subseteq \text{set } [] \iff \text{False}
\]
by auto

A frequent case – avoid intermediate sets

lemma [code-unfold]:
\[
\text{set } xs \subseteq \text{set } ys \iff \text{list-all } (\forall x. \ x \in \text{set } ys) \ xs
\]
by (auto simp: list-all-iff)

lemma Ball-set [code]:
\[
\text{Ball } (\text{set } xs) \ P \iff \text{list-all } P \ xs
\]
by (simp add: list-all-iff)

lemma Bex-set [code]:
\[
\text{Bex } (\text{set } xs) \ P \iff \text{list-ex } P \ xs
\]
by (simp add: list-ex-iff)

lemma card-set [code]:
\[
\text{card } (\text{set } xs) = \text{length } (\text{remdups } xs)
\]
by (simp add: length-remdups-card-conv)

lemma the-elem-set [code]:
\[
\text{the-elem } (\text{set } [x]) = x
\]
by simp
lemma Pow-set [code]:
  \[ \text{Pow} \left( \text{set } [] \right) = \{ \{ \} \} \]
  \[ \text{Pow} \left( \text{set } (x \# \text{xs}) \right) = (\text{let } A = \text{Pow} \left( \text{set } \text{xs} \right) \text{ in } A \cup \text{insert } x \cdot A) \]
by (simp-all add: Pow-insert Let-def)

definition map-project :: ('a ⇒ 'b option) ⇒ 'a set ⇒ 'b set
where
  \[ \text{map-project } f A = \{ b. \exists a \in A. f a = \text{Some } b \} \]

lemma [code]:
  \[ \text{map-project } f \left( \text{set } \text{xs} \right) = \text{set} \left( \text{List.map-filter } f \text{ xs} \right) \]
by (auto simp add: map-project-def map-filter-def image-def)

hide-const (open) map-project

Operations on relations

lemma product-code [code]:
  \[ \text{Product-Type.product } \left( \text{set } \text{xs} \right) \left( \text{set } \text{ys} \right) = \text{set } [(x, y). x \leftarrow \text{xs}, y \leftarrow \text{ys}] \]
by (auto simp add: Product-Type.product-def)

lemma Id-on-set [code]:
  \[ \text{Id-on } \left( \text{set } \text{xs} \right) = \text{set } [(x, x). x \leftarrow \text{xs}] \]
by (auto simp add: Id-on-def)

lemma [code]:
  \[ R \circ S = \text{List.map-project } (\lambda (x, y). \text{if } x \in S \text{ then } \text{Some } y \text{ else } \text{None}) \text{ R} \]
unfolding map-project-def by (auto split: prod.split if-split-asm)

lemma trancl-set-ntrancl [code]:
  \[ \text{trancl } \left( \text{set } \text{xs} \right) = \text{ntrancl } (\text{card } \left( \text{set } \text{xs} \right) - 1) \left( \text{set } \text{xs} \right) \]
by (simp add: finite-trancl-ntranl)

lemma set-relcomp [code]:
  \[ \text{set } \text{xys} \circ \text{set } \text{yzs} = \text{set } \left[ ((\text{fst } \text{xy}, \text{snd } \text{yz}), \text{xy} \leftarrow \text{xys}, \text{yz} \leftarrow \text{yzs}, \text{snd } \text{xy} = \text{fst } \text{yz}) \right] \]
by auto (auto simp add: Bex-def image-def)

lemma wf-set:
  \[ \text{wf } \left( \text{set } \text{xs} \right) = \text{acyclic } \left( \text{set } \text{xs} \right) \]
by (simp add: wf-iff-acyclic-if-finite)

lemma wf-code-set[code]: \[ \text{wf-code } \left( \text{set } \text{xs} \right) = \text{acyclic } \left( \text{set } \text{xs} \right) \]
unfolding wf-code-def using wf-set.

66.8 Setup for Lifting/Transfer

66.8.1 Transfer rules for the Transfer package

context includes lifting-syntax
begin
THEORY “List”

lemma tl-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A) tl tl
  unfolding tl-def[abs-def] by transfer-prover

lemma butlast-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A) butlast butlast
  by (rule rel-funI, rule list-all2-induct, auto)

lemma map-rec: map f xs = rec-list Nil (%x - y. Cons (f x) y) xs
  by (induct xs) auto

lemma append-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A ===> list-all2 A) append append
  unfolding List.append-def by transfer-prover

lemma rev-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A) rev rev
  unfolding List.rev-def by transfer-prover

lemma filter-transfer [transfer-rule]:
  ((A ===> (=)) ===> list-all2 A ===> list-all2 A) filter filter
  unfolding List.filter-def by transfer-prover

lemma fold-transfer [transfer-rule]:
  ((A ===> B ===> B) ===> list-all2 A ===> B ===> B) fold fold
  unfolding List.fold-def by transfer-prover

lemma foldr-transfer [transfer-rule]:
  ((A ===> B ===> B) ===> list-all2 A ===> B ===> B) foldr foldr
  unfolding List.foldr-def by transfer-prover

lemma foldl-transfer [transfer-rule]:
  ((B ===> A ===> B) ===> B ===> list-all2 A ===> B) foldl foldl
  unfolding List.foldl-def by transfer-prover

lemma concat-transfer [transfer-rule]:
  (list-all2 (list-all2 A) ===> list-all2 A) concat concat
  unfolding List.concat-def by transfer-prover

lemma drop-transfer [transfer-rule]:
  ((=) ===> list-all2 A ===> list-all2 A) drop drop
  unfolding List.drop-def by transfer-prover

lemma take-transfer [transfer-rule]:
  ((=) ===> list-all2 A ===> list-all2 A) take take
  unfolding List.take-def by transfer-prover

lemma list-update-transfer [transfer-rule]:
  (list-all2 A ===> (=) ===> A ===> list-all2 A) list-update list-update
unfolding list-update-def by transfer-prover

lemma takeWhile-transfer [transfer-rule]:
  \((A \rightarrow (\_)) \rightarrow list-all2 A \rightarrow list-all2 A)\) takeWhile takeWhile
unfolding takeWhile-def by transfer-prover

lemma dropWhile-transfer [transfer-rule]:
  \((A \rightarrow (\_)) \rightarrow list-all2 A \rightarrow list-all2 A)\) dropWhile dropWhile
unfolding dropWhile-def by transfer-prover

lemma zip-transfer [transfer-rule]:
  \((list-all2 A \rightarrow list-all2 B \rightarrow list-all2 (rel-prod A B))\) zip zip
unfolding zip-def by transfer-prover

lemma product-transfer [transfer-rule]:
  \((list-all2 A \rightarrow list-all2 B \rightarrow list-all2 (rel-prod A B))\) List.product List.product
unfolding List.product-def by transfer-prover

lemma product-lists-transfer [transfer-rule]:
  \((list-all2 (list-all2 A) \rightarrow list-all2 (list-all2 A))\) product-lists product-lists
unfolding product-lists-def by transfer-prover

lemma insert-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows \((A \rightarrow list-all2 A \rightarrow list-all2 A)\) List.insert List.insert
unfolding List.insert-def [abs-def] by transfer-prover

lemma find-transfer [transfer-rule]:
  \((A \rightarrow (\_)) \rightarrow list-all2 A \rightarrow rel-option A)\) List.find List.find
unfolding List.find-def by transfer-prover

lemma those-transfer [transfer-rule]:
  \((list-all2 (rel-option P) \rightarrow rel-option (list-all2 P))\) those those
unfolding List.those-def by transfer-prover

lemma remove1-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows \((A \rightarrow list-all2 A \rightarrow list-all2 A)\) remove1 remove1
unfolding remove1-def by transfer-prover

lemma removeAll-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows \((A \rightarrow list-all2 A \rightarrow list-all2 A)\) removeAll removeAll
unfolding removeAll-def by transfer-prover

lemma successively-transfer [transfer-rule]:
  \((A \rightarrow (\_)) \rightarrow list-all2 A \rightarrow (\_))\) successively successively
unfolding successively-altdef by transfer-prover
lemma distinct-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> (=)) distinct distinct
  unfolding distinct-def by transfer-prover

lemma distinct-adj-transfer [transfer-rule]:
  assumes bi-unique A
  shows (list-all2 A ===> (=)) distinct-adj distinct-adj
  unfolding rel-fun-def
proof (intro allI impI)
  fix xs ys assume list-all2 A xs ys
  thus distinct-adj xs <-> distinct-adj ys
proof (induction rule: list-all2-induct)
  case (Cons x xs y ys)
  show ?case
    by (metis Cons assms bi-unique-def distinct-adj-Cons list.rel-set)
qed auto
qed

lemma remdups-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 A) remdups remdups
  unfolding remdups-def by transfer-prover

lemma remdups-adj-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 A) remdups-adj remdups-adj
  proof (rule rel-funI, erule list-all2-induct)
  qed (auto simp: remdups-adj-Cons assms[unfolded bi-unique-def] split: list.splits)

lemma replicate-transfer [transfer-rule]:
  ((=) ===> A ===> list-all2 A) replicate replicate
  unfolding replicate-def by transfer-prover

lemma length-transfer [transfer-rule]:
  (list-all2 A ===> (=)) length length
  unfolding size-list-overloaded-def size-list-def by transfer-prover

lemma rotate1-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A) rotate1 rotate1
  unfolding rotate1-def by transfer-prover

lemma rotate-transfer [transfer-rule]:
  ((=) ===> list-all2 A ===> list-all2 A) rotate rotate
  unfolding rotate-def [abs-def] by transfer-prover

lemma nths-transfer [transfer-rule]:
  (list-all2 A ===> rel-set (=) ===> list-all2 A) nths nths
  unfolding nths-def [abs-def] by transfer-prover
lemma subseqs-transfer [transfer-rule]:
\[(\text{list-all2 } A \Longrightarrow \text{list-all2 } (\text{list-all2 } A)) \text{ subseqs subseqs}\]
unfolding subseqs-def [abs-def] by transfer-prover

lemma partition-transfer [transfer-rule]:
\[(\text{list-all2 } A \Longrightarrow \text{rel-prod } (\text{list-all2 } A) (\text{list-all2 } A)) \text{ partition partition}\]
unfolding partition-def by transfer-prover

lemma lists-transfer [transfer-rule]:
\[(\text{rel-set } A \Longrightarrow \text{rel-set } (\text{list-all2 } A)) \text{ lists lists}\]
proof (rule rel-funI, rule rel-setI)
show \[\forall l \in \text{lists } X; \text{rel-set } A X Y \Longrightarrow \exists y \in \text{lists } Y. \text{list-all2 } A l y \text{ for } X Y l\]
proof (induction \ l rule: lists.induct)
case (Cons a l)
  then show ?case
  by (simp only: rel-set-def list-all2-Cons1, metis lists.Cons)
qed auto

show \[\forall l \in \text{lists } Y; \text{rel-set } A X Y \Longrightarrow \exists x \in \text{lists } X. \text{list-all2 } A x l \text{ for } X Y l\]
proof (induction \ l rule: lists.induct)
case (Cons a l)
  then show ?case
  by (simp only: rel-set-def list-all2-Cons2, metis lists.Cons)
qed auto

lemma set-Cons-transfer [transfer-rule]:
\[(\text{rel-set } A \Longrightarrow \text{rel-set } (\text{list-all2 } A) \Longrightarrow \text{rel-set } (\text{list-all2 } A)) \text{ set-Cons set-Cons}\]
unfolding rel-fun-def rel-set-def set-Cons-def
by (fastforce simp add: list-all2-Cons1 list-all2-Cons2)

lemma listset-transfer [transfer-rule]:
\[(\text{list-all2 } (\text{rel-set } A) \Longrightarrow \text{rel-set } (\text{list-all2 } A)) \text{ listset listset}\]
unfolding listset-def by transfer-prover

lemma null-transfer [transfer-rule]:
\[(\text{list-all2 } (\text{rel-set } A) \Longrightarrow \text{rel-set } (\text{list-all2 } A)) \text{ listset listset}\]
unfolding rel-fun-def List.null-def by auto

lemma list-all-transfer [transfer-rule]:
\[(\text{list-all2 } A \Longrightarrow \text{list-all2 } A \Longrightarrow \text{list-all2 } A) \text{ list-all list-all}\]
unfolding list-all_iff [abs-def] by transfer-prover

lemma list-ex-transfer [transfer-rule]:
\[(\text{list-all2 } A \Longrightarrow \text{list-all2 } A \Longrightarrow \text{list-ex list-ex}\]
unfolding list-ex_iff [abs-def] by transfer-prover
lemma splice-transfer [transfer-rule]:
(list-all2 A ===> list-all2 A ===> list-all2 A) splice splice
apply (rule rel-funI, erule list-all2-induct, simp add: rel-fun-def, simp)
apply (rule rel-funI)
apply (erule_tac xs=x in list-all2-induct, simp, simp add: rel-fun-def)
done

lemma shuffles-transfer [transfer-rule]:
(list-all2 A ===> list-all2 A ===> rel-set (list-all2 A)) shuffles shuffles
proof (intro rel-funI, goal_cases)
case (1 xs xs' ys ys')
thus ?case
proof (induction xs ys arbitrary: xs' ys' rule: shuffles.induct)
case (3 x xs y ys xs' ys')
  from 3.prems obtain x' xs'' where xs': xs' = x' # xs'' by (cases xs') auto
  from 3.prems obtain y' ys'' where ys': ys' = y' # ys'' by (cases ys') auto
have [transfer-rule]: A x' A y' list-all2 A xs xs'' list-all2 A ys ys''
  using 3.prems by (simp-all add: xs' ys')
have [transfer-rule]: rel-set (list-all2 A) (shuffles xs (y # ys)) (shuffles xs'' ys'')
and [transfer-rule]: rel-set (list-all2 A) (shuffles (x # xs) ys) (shuffles x' ys'')
  using 3.prems by (auto intro: 3.IH simp: xs' ys')
have rel-set (list-all2 A) ((#) x ' shuffles xs (y # ys) ∪ (#) y ' shuffles (x # xs) ys)
  ((#) x' ' shuffles xs'' ys' ∪ (#) y' ' shuffles x' ys'') by transfer-prover
thus ?case by (simp add: xs' ys')
qed (auto simp: rel-set-def)
qed

lemma rtrancl-parametric [transfer-rule]:
assumes [transfer-rule]: bi-unique A bi-total A
shows (rel-set (rel-prod A A) ===> rel-set (rel-prod A A)) rtrancl rtrancl
unfolding rtrancl-def by transfer-prover

lemma monotone-parametric [transfer-rule]:
assumes [transfer-rule]: bi-total A
shows ((A ===> A ===> (=)) ===> (B ===> B ===> (=)) ===> (A ===> B) ===> (=)) monotone monotone
unfolding monotone-def[abs-def] by transfer-prover

lemma fun-ord-parametric [transfer-rule]:
assumes [transfer-rule]: bi-total C
shows ((A ===> B ===> (=)) ===> (C ===> A) ===> (C ===> B) ===> (=)) fun-ord fun-ord
unfolding fun-ord-def[abs-def] by transfer-prover

lemma fun-lub-parametric [transfer-rule]:
assumes [transfer-rule]: bi-total A bi-unique A
shows ((rel-set A ===> B) ===> rel-set (C ===> A) ===> C ===> B)
67 Sum and product over lists

theory Groups-List
imports List
begin

locale monoid-list = monoid
begin

definition $F$ :: 'a list ⇒ 'a
where
eq-foldr [code]: $F$ xs = foldr $f$ xs 1

lemma Nil [simp]:
$F$ [] = 1
by (simp add: eq-foldr)

lemma Cons [simp]:
$F$ ($x$ # $xs$) = $x$ * $F$ $xs$
by (simp add: eq-foldr)

lemma append [simp]:
$F$ ($xs$ @ $ys$) = $F$ $xs$ * $F$ $ys$
by (induct $xs$) (simp-all add: assoc)

end

locale comm-monoid-list = comm-monoid + monoid-list
begin

lemma rev [simp]:
$F$ (rev $xs$) = $F$ $xs$
by (simp add: eq-foldr foldr-fold fold-rev fun-eq-iff assoc left-commute)

end

locale comm-monoid-list-set = list: comm-monoid-list + set: comm-monoid-set
begin

lemma distinct-set-conv-list:
distinct $xs$ ⇒ set.$F$ ($g$ (set $xs$)) = list.$F$ ($map$ $g$ $xs$)
by (induct $xs$) simp-all
lemma set-conv-list[code]:
set.F g (set xs) = list.F (map g (remdups xs))
by (simp add: distinct-set-conv-list[symmetric])

lemma list-conv-set-nth:
list.F xs = set.F (λi. xs ! i) {0..<length xs}
proof –
  have xs = map (λi. xs ! i) [0..<length xs]
    by (simp add: map-nth)
  also have list.F .. = set.F (λi. xs ! i) {0..<length xs}
    by (subst distinct-set-conv-list[symmetric]) auto
  finally show ?thesis .
qed

end

67.1 List summation
context monoid-add
begin

sublocale sum-list: monoid-list plus 0
defines
  sum-list = sum-list.F ..

end

context comm-monoid-add
begin

sublocale sum-list: comm-monoid-list plus 0
rewrites
  monoid-list.F plus 0 = sum-list
proof –
  show comm-monoid-list plus 0 ..
  then interpret sum-list: comm-monoid-list plus 0 .
  from sum-list-def show monoid-list.F plus 0 = sum-list by simp
qed

sublocale sum: comm-monoid-list-set plus 0
rewrites
  monoid-list.F plus 0 = sum-list
  and comm-monoid-set.F plus 0 = sum
proof –
  show comm-monoid-list-set plus 0 ..
  then interpret sum: comm-monoid-list-set plus 0 .
  from sum-list-def show monoid-list.F plus 0 = sum-list by simp
  from sum-def show comm-monoid-set.F plus 0 = sum by (auto intro: sym)
Some syntactic sugar for summing a function over a list:

\[
\text{syntax (ASCII)} \\
\text{-sum-list :: pttrn => 'a list => 'b => 'b } ((\exists\sum \rightarrow \rightarrow \cdot) [0, 51, 10] 10)
\]

\[
\text{translations} \quad \text{Beware of argument permutation!} \\
\sum x \leftarrow xs. b == \text{CONST sum-list}\ (\text{CONST map (\lambda x. b)} \ xs)
\]

context
  includes lifting-syntax

begin

lemma sum-list-transfer [transfer-rule]:
  (list-all2 A ===> A) sum-list sum-list
  if [transfer-rule]: A 0 0 (A ===> A ===> A) (+) (+)
  unfolding sum-list.eq-foldr [abs-def]
  by transfer-prover

end

TODO duplicates

lemmas sum-list-simps = sum-list.Nil sum-list.Cons
lemmas sum-list-append = sum-list.append
lemmas sum-list-rev = sum-list.rev

lemma (in monoid-add) fold-plus-sum-list-rev:
  fold plus xs x = plus (sum-list (rev xs))
proof
  fix x
  have fold plus xs x = sum-list (rev xs @ [x])
    by (simp add: foldr-cone-fold sum-list.eq-foldr)
  also have \ldots = sum-list (rev xs) + x
    by simp
  finally show fold plus xs x = sum-list (rev xs) + x
    .
  qed

lemma (in comm-monoid-add) sum-list-map-remove1:
  \( x \in \text{set xs} \implies \text{sum-list (map f xs)} = f x + \text{sum-list (map f (remove1 x xs))} \)
by (induct xs) (auto simp add: ac-simps)

lemma (in monoid-add) size-list-cone-sum-list:
  size-list f xs = sum-list (map f xs) + size xs
by (induct xs) auto
lemma (in monoid-add) length-concat:
  length (concat xss) = sum-list (map length xss)
by (induct xss) simp-all

lemma (in monoid-add) length-product-lists:
  length (product-lists xss) = foldr (∗) (map length xss) 1
proof (induct xss)
  case (Cons xs xss) then show ?case by (induct xs) (auto simp: length-concat o-def)
qed simp

lemma (in monoid-add) sum-list-map-filter:
  assumes ⋀ x. x ∈ set xs =⇒ ¬ P x =⇒ f x = 0
  shows sum-list (map f (filter P xs)) = sum-list (map f xs)
  using assms by (induct xs) auto

lemma sum-list-filter-le-nat:
  fixes f :: 'a ⇒ nat
  shows sum-list (map f (filter P xs)) ≤ sum-list (map f xs)
by (induction xs; simp)

lemma (in comm-monoid-add) distinct-sum-list-conv-Sum:
  distinct xs =⇒ sum-list xs = Sum (set xs)
by (induct xs) simp-all

lemma sum-list-upt[simp]:
  m ≤ n =⇒ sum-list [m..<n] = ∑ {m..<n}
by (simp add: distinct-sum-list-conv-Sum)

context ordered-comm-monoid-add
begin
lemma sum-list-nonneg: (⋀ x. x ∈ set xs =⇒ 0 ≤ x) =⇒ 0 ≤ sum-list xs
by (induction xs) auto
lemma sum-list-nonpos: (⋀ x. x ∈ set xs =⇒ x ≤ 0) =⇒ sum-list xs ≤ 0
by (induction xs) (auto simp: add-nonpos-nonpos)
lemma sum-list-nonneg-eq-0-iff:
  (⋀ x. x ∈ set xs =⇒ 0 ≤ x) =⇒ sum-list xs = 0 =⇒ (⋀ x ∈ set xs. x = 0)
by (induction xs) (simp-all add: add-nonneg-eq-0-iff sum-list-nonneg)
end

context canonically-ordered-monoid-add
begin
lemma sum-list-eq-0-iff [simp]:
  sum-list ns = 0 =⇒ (∀ n ∈ set ns. n = 0)
by (simp add: sum-list-nonneg-eq-0-iff)

lemma member-le-sum-list:
x ∈ set xs ⇒ x ≤ sum-list xs
by (induction xs) (auto simp: add-increasing add-increasing2)

lemma elem-le-sum-list:
k < size ns ⇒ ns ! k ≤ sum-list (ns)
by (rule member-le-sum-list) simp

end

lemma (in ordered-cancel-comm-monoid-diff) sum-list-update:
k < size xs ⇒ sum-list (xs[k := x]) = sum-list xs + x - ns ! k
apply (induction xs arbitrary: k)
apply (auto simp: add-ac split: nat.split)
apply (drule elem-le-sum-list)
by (simp add: local.add-diff-assoc local.add-increasing)

lemma (in monoid-add) sum-list-triv:
(∑ x←xs. r) = of-nat (length xs) * r
by (induct xs) (simp-all add: distrib-right)

lemma (in monoid-add) sum-list-0 [simp]:
(∑ x←xs. 0) = 0
by (induct xs) (simp-all add: distrib-right)

For non-Abelian groups xs needs to be reversed on one side:

lemma (in ab-group-add) uminus-sum-list-map:
− sum-list (map f xs) = sum-list (map (uminus o f) xs)
by (induct xs) simp-all

lemma (in comm-monoid-add) sum-list-addf:
(∑ x←xs. f x + g x) = sum-list (map f xs) + sum-list (map g xs)
by (induct xs) (simp-all add: algebra-simps)

lemma (in ab-group-add) sum-list-subtractf:
(∑ x←xs. f x - g x) = sum-list (map f xs) - sum-list (map g xs)
by (induct xs) (simp-all add: algebra-simps)

lemma (in semiring-0) sum-list-const-mult:
(∑ x←xs. c * f x) = c * (∑ x←xs. f x)
by (induct xs) (simp-all add: algebra-simps)

lemma (in semiring-0) sum-list-mult-const:
(∑ x←xs. f x * c) = (∑ x←xs. f x) * c
by (induct xs) (simp-all add: algebra-simps)

lemma (in ordered-ab-group-add-ords) sum-list-ords:
THEORY “Groups-List”

\[
|\text{sum-list } xs| \leq \text{sum-list (map abs } xs) \\
\text{by (induct } xs) \ (\text{simp-all add: order-trans [OF abs-triangle-ineq]})
\]

lemma \text{sum-list-mono}: 
fixes \( f, g :: 'a \Rightarrow 'b :: \{ \text{monoid-add, ordered-ab-semigroup-add} \} \)
shows \( (\forall x. x \in \text{set } xs \Rightarrow f x \leq g x) \Rightarrow (\sum x \leftarrow xs. f x) \leq (\sum x \leftarrow xs. g x) \) 
by (induct \( xs \)) \ (\text{simp add: add-mono})

lemma \text{sum-list-strict-mono}: 
fixes \( f, g :: 'a \Rightarrow 'b :: \{ \text{monoid-add, strict-ordered-ab-semigroup-add} \} \)
shows \( [\text{length } xs \neq []; \forall x. x \in \text{set } xs \Rightarrow f x < g x ] \Rightarrow \text{sum-list (map } f \) \) \( xs) < \text{sum-list (map } g \) \( xs) \) 
proof (induction \( xs \)) 
\begin{itemize}
  \item case \text{Nil} \ thus \ ?case \ by \ simp
  \item case \( C \) \( :: (\text{Cons } - \) \( \) \( xs) \)
    \begin{itemize}
      \item show \ ?case 
        \begin{itemize}
          \item case \text{Nil} thus \ ?thesis using \( C \).prems \ by \ simp
        \end{itemize}
      \item case \text{Cons} thus \ ?thesis using \( C \) \( \text{by} (\text{simp add: add-strict-mono}) \)
    \end{itemize}
next
\end{itemize}
qed

A much more general version of this monotonicity lemma can be formulated 
with multisets and the multiset order

lemma \text{sum-list-strict-mono2}: 
fixes \( xs :: 'a :: \{ \text{monoid-add, ordered-comm-monoid-add list} \} \)
shows \( [\text{length } xs = \text{length } ys; \forall i. i < \text{length } xs \Rightarrow zs!i \leq ys!i ] \Rightarrow \text{sum-list (map } f \) \( xs) < \text{sum-list (map } g \) \( xs) \) 
apply (\text{induction } xs \ ys \ rule: list-induct2) 
by (auto \text{simp: nth-Cons' less-Suc-eq-0-disj imp-ex add-mono})

lemma \text{distinct } xs \Rightarrow \text{sum-list (map } f \) \( xs) = \text{sum } f \ (\text{set } xs) \) 
by (induct \( xs \)) \ (\text{simp-all})

lemma \text{interv-sum-list-conv-sum-set-nat}: 
\( \text{sum-list (map } f \ [m..<n]) = \text{sum } f \ (\text{set } [m..<n]) \) 
by (simp add: \text{sum-list-distinct-conv-sum-set})

lemma \text{interv-sum-list-conv-sum-set-int}: 
\( \text{sum-list (map } f \ [k..l]) = \text{sum } f \ (\text{set } [k..l]) \) 
by (simp add: \text{sum-list-distinct-conv-sum-set})

General equivalence between \text{sum-list} and \text{sum}

lemma \text{sum-list-sum-nth}: 
\( \text{sum-list } xs = (\sum i = 0 ..< \text{length } xs. xs!i) \) 
\text{using} \text{interv-sum-list-conv-sum-set-nat \ [of (\!) } \( xs \ 0 \text{ length } xs) \ by (\text{simp add: map-nth}) \)
**lemma** sum-list-map-eq-sum-count:

\[ \text{sum-list (map } f \text{ xs}) = \text{sum } (\lambda x. \text{count-list xs } x * f x) \text{ (set xs)} \]

**proof** (induction xs)

**case** (Cons x xs)

**show** ?case (is \(?l = \?r\))

**proof** cases

**assume** \(x \in \text{set xs}\)

**have** \(\?l = f x + (\sum x \in \text{set xs}. \text{count-list xs } x * f x)\) by (simp add: Cons.IH)

**also have** \(\text{set xs} = \text{insert } x \text{ (set xs} - \{x\})\) using \(x \in \text{set xs}\) by blast

**also have** \(f x + (\sum x \in \text{insert } x \text{ (set xs} - \{x\}). \text{count-list xs } x * f x) = ?r\)

by (simp add: sum.insert-remove eq-commute)

**finally show** ?thesis.

**next**

**assume** \(x \notin \text{set xs}\)

**hence** \(\forall xa. xa \in \text{set xs} \Longrightarrow x \neq xa\) by blast

**thus** ?thesis by (simp add: Cons.IH \(x \notin \text{set xs}\))

**qed**

qed simp

**lemma** sum-list-map-eq-sum-count2:

assumes \(\text{set xs} \subseteq X \text{ finite } X\)

**shows** \(\text{sum-list (map } f \text{ xs}) = \text{sum } (\lambda x. \text{count-list xs } x * f x) \text{ X}\)

**proof**

**let** \(\?F = \lambda x. \text{count-list xs } x * f x\)

**have** \(\text{sum } \?F \text{ X} = \text{sum } \?F \text{ (set xs} \cup (X - \text{set xs}))\)

using Un.absorb1[OF assms(1)] by (simp)

**also have** \(\ldots = \text{sum } \?F \text{ (set xs)}\)

using assms(2)

by (simp add: sum.union-disjoint[OF - - Diff-disjoint del: Un-Diff-cancel])

**finally show** ?thesis by (simp add: sum-list-map-eq-sum-count)

**qed**

**lemma** sum-list-replicate:

\(\text{sum-list (replicate } n \text{ c}) = \text{of-nat } n * c\)

by (induction \(n\)) (auto simp add: distrib-right)

**lemma** sum-list-nonneg:

\((\forall x. x \in \text{set xs} \Longrightarrow (x :: 'a :: ordered-comm-monoid-add) \geq 0) \Longrightarrow \text{sum-list xs} \geq 0\)

by (induction xs) simp-all

**lemma** sum-list-Suc:

\(\text{sum-list (map } (\lambda x. \text{Suc } f x) \text{ xs}) = \text{sum-list } (\text{map } f \text{ xs} \text{) + length xs}\)

by (induction \(xs\); simp)

**lemma** (in monoid-add) sum-list-map-filter':

\(\text{sum-list (map } f \text{ (filter } P \text{ xs}) = \text{sum-list } (\text{map } (\lambda x. \text{if } P x \text{ then } f x \text{ else } 0) \text{) xs}\)

by (induction \(xs\)) simp-all
Summation of a strictly ascending sequence with length $n$ can be upper-bounded by summation over $\{0..<n\}$.

**lemma** sorted-wrt-less-sum-mono-lowerbound:

- **fixes** $f : \text{nat} \Rightarrow (\text{'}b:\text{ordered-comm-monoid-add})$
- **assumes** mono: $\forall x y. \ x \leq y \implies f x \leq f y$
- **shows** sorted-wrt $(\text{}<)\ ns \implies (\sum i \in \{0..<\text{length}\ ns\}. f i) \leq (\sum i \in \text{ns}. f i)$

**proof** (induction $\text{ns}$ rule: rev-induct)

- **case** Nil
  - then show $?case$ by simp

  **next**
  - **case** (snoc $n\ ns$)
    - have $\text{sum}\ f\ \{0..<\text{length}\ (\text{ns} @ [n])\}\ =\ \text{sum}\ f\ \{0..<\text{length}\ ns\} + f\ \text{length}\ ns$ by simp
    - also have $\text{sum}\ f\ \{0..<\text{length}\ ns\} \leq \text{sum-list}\ (\text{map}\ f\ \text{ns})$
      - using $\text{snoc}$ by (auto simp: sorted-wrt-append)
    - also have $\text{length}\ ns \leq n$
      - using $\text{sorted-wrt-less-idx}\ [\text{OF}\ \text{snoc.prems(1)},\ \text{of}\ \text{length}\ \text{ns}]$ by auto
    - finally have $\text{sum}\ f\ \{0..<\text{length}\ (\text{ns} @ [n])\}\ \leq\ \text{sum-list}\ (\text{map}\ f\ \text{ns}) + f\ n$
      - using $\text{mono}\ \text{add-mono}$ by blast
    - thus $?case$ by simp

**qed**

### 67.2 Horner sums

**context** comm-semiring-0

**begin**

**definition** horner-sum :: $\langle\text{'}b\Rightarrow\text{'}a\Rightarrow\text{'}b\text{ list}\Rightarrow\text{'}a\rangle$

- **where** horner-sum-foldr: $\langle\text{horner-sum}\ f\ a\ xs = \text{foldr}\ (\lambda x b. f x + a \ast b)\ xs\ \text{0}\rangle$

**lemma** horner-sum-simps [simp]:

- $\langle\text{horner-sum}\ f\ a\ []\ =\ \text{0}\rangle$
- $\langle\text{horner-sum}\ f\ a\ (x \#\ xs) = f\ x + a \ast \text{horner-sum}\ f\ a\ xs\rangle$
  - by (simp-all add: horner-sum-foldr)

**lemma** horner-sum-eq-sum-funpow:

- $\langle\text{horner-sum}\ f\ a\ xs = (\sum n = 0..<\text{length}\ xs. ((\ast)\ a \sim n)\ (f\ (\text{xs}!\ n)))\rangle$

**proof** (induction $\text{xs}$)

- **case** Nil
  - then show $?case$
    - by simp

  **next**
  - **case** (Cons $x\ xs$)
    - then show $?case$
      - by (simp add: sum.atLeast0-lessThan-Suc-shift sum-distrib-left det: sum.op-ivl-Suc)

**qed**
context

includes lifting-syntax

begin

lemma horner-sum-transfer [transfer-rule]:
\[(B \iff A) \iff \text{list-all2 } B \iff A) \text{ horner-sum horner-sum}\]
if [transfer-rule]: \((A \neq 0 \neq 0)\)
  and [transfer-rule]: \((A \iff A \iff A) (+) (+)\)
  and [transfer-rule]: \((A \iff A \iff A) (*)(*)\)
by (unfold horner-sum-foldr) transfer-prover

end

context comm-semiring-1

begin

lemma horner-sum-eq-sum:
\[\text{horner-sum } f \ a \ x s = (\sum \ n = 0..<\text{length } x s. \ f (x s ! n) * a ^ n)\]
proof –
  have \((*) a ^^ n = (*)(a ^ n)\) for \(n\)
    by (induction \(n\)) (simp-all add: ac-simps)
then show \(?\text{thesis}\)
  by (simp add: horner-sum-eq-sum-fanpow ac-simps)
qed

lemma horner-sum-append:
\[\text{horner-sum } f \ a (x s \ @ ys) = \text{horner-sum } f \ a \ x s + a ^ \text{length } x s * \text{horner-sum } f \ a \ ys\]
using sum.atLeastLessThan-shift-bounds [of \(- 0 \langle\text{length } x s\rangle \langle\text{length } ys\rangle\]
  atLeastLessThan-add-Un [of \(0 \langle\text{length } x s\rangle \langle\text{length } ys\rangle\]
by (simp add: horner-sum-eq-sum sum-distrib-left sum.union-disjoint ac-simps nth-append power-add)

end

context linordered-semidom

begin

lemma horner-sum-nonnegative:
\[0 \leq \text{horner-sum } \text{of-bool } 2 \ b s\]
by (induction \(b s\)) simp-all

end

context discrete-linordered-semidom

begin
lemma horner-sum-bound:
  \langle\text{horner-sum of-bool}\ 2\ bs\ <\ 2^{\ \text{length}\ bs}\rangle
proof (induction bs)
  case Nil
  then show ?case
  by simp
next
  case (Cons b bs)
  moreover define a where
    \langle a = 2^{\ \text{length}\ bs} - \text{horner-sum of-bool}\ 2\ bs\rangle
  ultimately have *: \langle 2^{\ \text{length}\ bs} = \text{horner-sum of-bool}\ 2\ bs + a\rangle
    by simp
  have \langle 0 < a\rangle
    using Cons \ast by simp
  moreover have \langle 1 \leq a\rangle
    using \langle 0 < a\rangle by (simp add: less-eq-iff-succ-less)
  ultimately have \langle 0 + 1 < a + a\rangle
    by (rule add-less-le-mono)
  then have \langle 1 < a \ast 2\rangle
    by (simp add: mult-2-right)
  with Cons show ?case
    by (simp add: \ast algebra-simps)
qed

lemma horner-sum-of-bool-2-less:
  \langle\text{horner-sum of-bool}\ 2\ bs\ <\ 2^{\ \text{length}\ bs}\rangle
by (fact horner-sum-bound)
end

context discrete-linordered-semidom
begin

lemma horner-sum-less-eq-iff-lexordp-eq:
  \langle\text{horner-sum of-bool}\ 2\ bs\ \leq\ \text{horner-sum of-bool}\ 2\ cs\ \iff\ \text{lexordp-eq}\ (\text{rev}\ bs)\ (\text{rev}\ cs)\rangle
if \langle\text{length}\ bs = \text{length}\ cs\rangle
proof -
  have \langle\text{horner-sum of-bool}\ 2\ (\text{rev}\ bs)\ \leq\ \text{horner-sum of-bool}\ 2\ (\text{rev}\ cs)\ \iff\ \text{lexordp-eq}\ bs\ cs\rangle
    if \langle\text{length}\ bs = \text{length}\ cs\rangle for bs cs
  using that proof (induction bs cs rule: list-induct2)
  case Nil
  then show ?case
    by simp
next
THEORY “Groups-List”

case (Cons b bs c cs)
with horner-sum-nonnegative [of (rev bs)] horner-sum-nonnegative [of (rev cs)]
  horner-sum-bound [of (rev bs)] horner-sum-bound [of (rev cs)]
show ?case
  by (auto simp add: horner-sum-append not-less Cons intro: add-strict-increasing2 add-increasing)
qed
from that this [of (rev bs) (rev cs)] show ?thesis
  by simp
qed

lemma horner-sum-less-iff-lexordp:
  (horner-sum of-bool 2 bs < horner-sum of-bool 2 cs) ←→ ord-class. lexordp (rev bs) (rev cs)
if (length bs = length cs)
proof –
  have (horner-sum of-bool 2 (rev bs) < horner-sum of-bool 2 (rev cs) ←→ ord-class. lexordp bs cs)
    if (length bs = length cs) for bs cs
using that proof (induction bs cs rule: list-induct2)
  case Nil
  then show ?case
    by simp
next
  case (Cons b bs c cs)
  with horner-sum-nonnegative [of (rev bs)] horner-sum-nonnegative [of (rev cs)]
    horner-sum-bound [of (rev bs)] horner-sum-bound [of (rev cs)]
  show ?case
    by (auto simp add: horner-sum-append not-less Cons intro: add-strict-increasing2 add-increasing)
qed
from that this [of (rev bs) (rev cs)] show ?thesis
  by simp
qed

end

67.3 Further facts about List.n-lists

lemma length-n-lists: length (List.n-lists n xs) = length xs ^ n
  by (induct n) (auto simp add: comp-def length-concat sum-list-triv)

lemma distinct-n-lists:
  assumes distinct xs
  shows distinct (List.n-lists n xs)
proof (rule card-distinct)
  from assms have card-length: card (set xs) = length xs by (rule distinct-card)
  have card (set (List.n-lists n xs)) = card (set xs) ^ n
  proof (induct n)
case 0 then show ?case by simp
next
case (Suc n)
moreover have card (⋃ ys ∈ set (List.n-lists n xs). (λ y. y # ys) ' set xs)
    = (∑ ys ∈ set (List.n-lists n xs). card ((λ y. y # ys) ' set xs))
    by (rule card-UN-disjoint) auto
moreover have ∩ ys. card ((λ y. y # ys) ' set xs) = card (set xs)
    by (rule card-image) (simp add: inj-on-def)
ultimately show ?case by auto
qed
also have . . . = length xs ^ n by (simp add: card-length)
finally show card (set (List.n-lists n xs)) = length (List.n-lists n xs)
    by (simp add: length-n-lists)
qed

67.4 Tools setup
lemmas sum-code = sum.set-conv-list

lemma sum-set-upto-conv-sum-list-int [code-unfold]:
  sum f (set [i..j::int]) = sum-list (map f [i..j])
  by (simp add: interv-sum-list-conv-sum-set-int)

lemma sum-set-upt-conv-sum-list-nat [code-unfold]:
  sum f (set [m..<n]) = sum-list (map f [m..<n])
  by (simp add: interv-sum-list-conv-sum-set-nat)

67.5 List product
context monoid-mult
begin

sublocale prod-list: monoid-list times 1
defines prod-list = prod-list.F ..

end

context comm-monoid-mult
begin

sublocale prod-list: comm-monoid-list times 1
rewrites monoid-list.F times 1 = prod-list
proof –
show comm-monoid-list times 1 ..
then interpret prod-list: comm-monoid-list times 1.
from prod-list-def show monoid-list.F times 1 = prod-list by simp
qed
Theorem "Bit-Operations"

locale prod
  assumes "monoid-list.F times 1 = prod-list
            and comm-monoid-set.F times 1 = prod

proof
  show comm-monoid-list-set-times 1 ..
  then interpret prod: comm-monoid-list-set-times 1 .
  from prod-list-def show monoid-list.F times 1 = prod-list by simp
  from prod-def show comm-monoid-set.F times 1 = prod by (auto intro: sym)
qed

end

Some syntactic sugar:

syntax (ASCII)
  _prod-list :: pttrn => 'a list => 'b => 'b ((3PROD _<-> _<-> _) [0, 51, 10] 10)
syntax
  _prod-list :: pttrn => 'a list => 'b => 'b ((3∏ _<-> _<-> _) [0, 51, 10] 10)
translations — Beware of argument permutation!
  ∏ x<-xs. b = CONST prod-list (CONST map (λx. b) xs)

context
  includes lifting-syntax
begin

lemma prod-list-transfer [transfer-rule]:
  (list-all2 A === A) prod-list prod-list
  if [transfer-rule]: A I I (A === A ===> A) (*) (*)
  unfolding prod-list.eq-foldr [abs-def]
  by transfer-prover

end

lemma prod-list-zero-iff:
  prod-list xs = 0 <-> (0 :: 'a :: {semiring-no-zero-divisors, semiring-1}) ∈ set xs
  by (induction xs) simp-all

end

68 Bit operations in suitable algebraic structures

theory Bit-Operations
  imports Presburger Groups-List
begin

68.1 Abstract bit structures

class semiring-bits = semiring-parity + semiring-modulo-trivial +
  assumes bit-induct [case-names stable rec]:


\langle \forall a. a \div 2 = a \implies P \ a \rangle \\
\implies \langle \forall a. b. P \ a \implies (\text{of-bool} \ b + 2 \ast a) \div 2 = a \implies P \ (\text{of-bool} \ b + 2 \ast a) \rangle \\
\implies P \ a.

assumes bits-mod-div-trivial [simp]: \langle a \mod b \div b = 0 \rangle \\
and half-div-exp-eq: \langle a \div 2 \div 2^\langle n \rangle = a \div 2^\langle \text{Suc} \ n \rangle \rangle \\
and even-double-div-exp-iff: \langle 2^\langle \text{Suc} \ n \rangle \not= 0 \implies \text{even} \ (2 \ast a \div 2^\langle \text{Suc} \ n \rangle) \rangle \\
\leftarrow even \ (a \div 2^\langle \text{Suc} \ n \rangle) \rangle \\
fixes bit :: \langle 'a \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle \\
assumes bit-iff-odd: \langle \text{bit} \ a \ n \leftarrowarrow \text{odd} \ (a \div 2^\langle n \rangle) \rangle \\
begin

Having \text{bit} as definitional class operation takes into account that specific instances can be implemented differently wrt. code generation.

lemma half-1 [simp]:
\langle 1 \div 2 = 0 \rangle \\
using even-half-succ-eq [of 0] by simp

lemma div-exp-eq-funpow-half:
\langle a \div 2^\langle \text{Suc} \ n \rangle = ((\lambda a. a \div 2)^\langle n \rangle) \ a \rangle \\
proof –
have \langle ((\lambda a. a \div 2)^\langle n \rangle) \ a \rangle \\
by (induction n) \\
(simp-all del: funpow.simps power.simps add: power-0 funpow-Suc-right half-div-exp-eq) \\
then show \langle \text{thesis} \rangle \\
by simp
qed

lemma div-exp-eq:
\langle a \div 2^\langle m \rangle \div 2^\langle n \rangle = a \div 2^\langle m + n \rangle \rangle \\
by (simp add: div-exp-eq-funpow-half Groups.add.commute [of m] funpow-add)

lemma bit-0:
\langle \text{bit} \ a \ 0 \leftarrowarrow \text{odd} \ a \rangle \\
by (simp add: bit-iff-odd)

lemma bit-Suc:
\langle \text{bit} \ a \ (\text{Suc} \ n) \leftarrowarrow \text{bit} \ (a \div 2) \ n \rangle \\
using div-exp-eq [of a 1 n] by (simp add: bit-iff-odd)

lemma bit-rec:
\langle \text{bit} \ a \ n \leftarrowarrow \ (\text{if} \ n = 0 \ \text{then} \ \text{odd} \ a \ \text{else} \ \text{bit} \ (a \div 2) \ (n - 1)) \rangle \\
by (cases n) (simp-all del: bit-Suc bit-0)

context
fixes a 
assumes stable: \langle a \div 2 = a \rangle \\
begin

lemma bits-stable-imp-add-self:
\[
\langle a + a \mod 2 = 0 \rangle
\]

proof –

have \( \langle a \div 2 \ast 2 + a \mod 2 = a \rangle \)
  by (fact div-mul-mod-eq)
then have \( \langle a \ast 2 + a \mod 2 = a \rangle \)
  by (simp add: stable)
then show \( \langle \text{thesis} \rangle \)
  by (simp add: mult-2-right ac-simps)
qed

lemma stable-imp-bit-iff-odd:
\( \langle \text{bit } a \ n \longleftrightarrow \text{odd } a \rangle \)
by (induction \( n \)) (simp-all add: stable bit-Suc bit-0)
end

lemma bit-iff-odd-imp-stable:
\( \langle a \div 2 = a \rangle \) if \( \langle \forall n. \text{bit } a \ n \longleftrightarrow \text{odd } a \rangle \)
using that proof (induction \( a \) rule: bit-induct)
case (stable \( a \))
then show \( \langle \text{case} \rangle \)
  by simp
next
case (rec \( a \) \( b \))
from rec.prems [of \( 1 \)] have [simp]: \( \langle b = \text{odd } a \rangle \)
  by (simp add: rec.hyps bit-Suc bit-0)
from rec.hyps have hyp: \( \langle \text{of-bool } (\text{odd } a) + 2 \ast a \rangle \div 2 = a \rangle \)
  by simp
have \( \langle \text{bit } a \ n \longleftrightarrow \text{odd } a \rangle \) for \( n \)
  using rec.prems [of \( \langle \text{Suc } n \rangle \)] by (simp add: hyp bit-Suc)
then have \( \langle a \div 2 = a \rangle \)
  by (rule rec.IH)
then have \( \langle \text{of-bool } (\text{odd } a) + 2 \ast a = 2 \ast (a \div 2) + \text{of-bool } (\text{odd } a) \rangle \)
  by (simp add: ac-simps)
also have \( \langle \ldots = a \rangle \)
  using mult-div-mod-eq [of \( 2 \) \( a \)]
  by (simp add: of-bool-odd-eq-mod-2)
finally show \( \langle \text{case} \rangle \)
  using \( \langle a \div 2 = a \rangle \) by (simp add: hyp)
qed

lemma even-succ-div-exp [simp]:
\( \langle (1 + a) \div 2 ^ n = a \div 2 ^ {\langle n > 0} \rangle \) if \( \langle \text{even } a \rangle \) and \( \langle n > 0 \rangle \)
proof (cases \( n \))
case \( 0 \)
with that show \( \langle \text{thesis} \rangle \)
  by simp
next
case (Suc \( n \))
with ‹even a› have ‹(1 + a) div 2 ^ Suc n = a div 2 ^ Suc n›
proof (induction n)
  case 0
  then show ‹case›
    by simp
next
  case (Suc n)
  then show ‹case›
    using div-exp-eq [of - 1 ‹Suc n›, symmetric]
    by simp
qed
with Suc show ‹thesis›
  by simp
qed

lemma even-succ-mod-exp [simp]:
  ‹(1 + a) mod 2 ^ n = 1 + (a mod 2 ^ n)› if ‹even a› and ‹n > 0›
using div-mult-mod-eq [of ‹1 + a› ‹2 ^ n›] div-mult-mod-eq [of ‹a› ‹2 ^ n›] that
by simp (metis (full-types) add.left-commute add-left-imp-eq)

named-theorems bit-simps (Simplification rules for const ‹bit›)

definition possible-bit :: ‹a itself ⇒ nat ⇒ bool›
  where ‹possible-bit TYPE('a) n ←→ 2 ^ n ≠ 0›
― This auxiliary avoids non-termination with extensionality.

lemma possible-bit-0 [simp]:
  ‹possible-bit TYPE('a) 0›
  by (simp add: possible-bit-def)

lemma fold-possible-bit:
  ‹2 ^ n = 0 ←→ ¬ possible-bit TYPE('a) n›
  by (simp add: possible-bit-def)

lemma bit-imp-possible-bit:
  ‹possible-bit TYPE('a) n› if ‹bit a n›
  by (rule ccontr) (use that in ‹auto simp add: bit-iff-odd possible-bit-def›)

lemma impossible-bit:
  ‹¬ bit a n› if ‹¬ possible-bit TYPE('a) n›
  using that by (blast dest: bit-imp-possible-bit)

lemma possible-bit-less-imp:
  ‹possible-bit TYPE('a) j› if ‹possible-bit TYPE('a) i› ‹j ≤ i›
  using power-add [of ‹2 ^ i • (i - j)›] that mult-not-zero [of ‹2 ^ j› ‹2 ^ (i - j)›] that
  by (simp add: possible-bit-def)

lemma possible-bit-min [simp]:
  ‹possible-bit TYPE('a) (min i j) ←→ possible-bit TYPE('a) i ∨ possible-bit
THEORY "Bit-Operations"

TYPE('a) j
  by (auto simp add: min-def elim: possible-bit-less-imp)

lemma bit-eqI:
  \(<a = b> \text{ if } \bigwedge n. \text{possible-bit TYPE('a) } n \implies \text{bit } a \ n \longleftrightarrow \text{bit } b \ n>\)
proof –
  have \(<\text{bit } a \ n \longleftrightarrow \text{bit } b \ n> \text{ for } n>\)
  proof (cases \(<\text{possible-bit TYPE('a) } n>\))
    case False
    then show \(<?\text{thesis}>\)
      by (simp add: impossible-bit)
  next
    case True
    then show \(<?\text{thesis}>\)
      by (rule that)
  qed
then show \(<?\text{thesis}>\)
proof (induction a arbitrary: b rule: bit-induct)
  case (stable a)
  from stable(2) [of 0] have **: \(<\text{even } b \longleftrightarrow \text{even } a>\)
    by (simp add: bit-0)
  have \(<b \text{ div } 2 = b>\)
  proof (rule bit-iff-odd-imp-stable)
    fix n
    from stable have *: \(<\text{bit } b \ n \longleftrightarrow \text{bit } a \ n>\)
      by simp
    also have \(<\text{bit } a \ n \longleftrightarrow \text{odd } a>\)
      using stable by (simp add: stable-imp-bit-iff-odd)
    finally show \(<\text{bit } b \ n \longleftrightarrow \text{odd } b>\)
      by (simp add: **) 
  qed
from ** have \(<a \text{ mod } 2 = b \text{ mod } 2>\)
  by (simp add: mod2-eq-if)
then have \(<a \text{ mod } 2 + (a + b) = b \text{ mod } 2 + (a + b)>\)
  by simp
then have \(<a + a \text{ mod } 2 + b = b + b \text{ mod } 2 + a>\)
  by (simp add: ac-simps)
with \(<a \text{ div } 2 = a>\) \(<b \text{ div } 2 = b>\)
  show \(<?\text{case}>\)
    by (simp add: bits-stable-imp-add-self)
next
  case (rec a p)
  from rec.prems [of 0] have [simp]: \(<p = \text{odd } b>\)
    by (simp add: bit-0)
  from rec.hyps have \(<\text{bit } a \ n \longleftrightarrow \text{bit } (b \text{ div } 2) \ n> \text{ for } n>\)
    using rec.prems [of \(<\text{Suc } n>\)] by (simp add: bit-Suc)
then have \(<a = b \text{ div } 2>\)
  by (rule rec.IH)
then have \(<2 * a = 2 * (b \text{ div } 2)>\)
  by simp
then have \(<b \text{ mod } 2 + 2 * a = b \text{ mod } 2 + 2 * (b \text{ div } 2)>\)
by simp
also have \( \ldots = b \)
by (fact mod-mult-div-eq)
finally show \(?case\)
by (auto simp add: mod2-eq-if)
qed
qed

lemma bit-eq-rec:
\( a = b \iff (\text{even } a \iff \text{even } b) \land a \div 2 = b \div 2) \) (is \(?P = ?Q\))
proof
assume \(?P\)
then show \(?Q\)
by simp
next
assume \(?Q\)
then have \(\text{even } a \iff \text{even } b\) and \(a \div 2 = b \div 2\)
by simp-all
show \(?P\)
proof (rule bit-eqI)
fix \(n\)
show \(\text{bit } a \ n \iff \text{bit } b \ n\)
proof (cases \(n\))
  case 0
  with \(\text{even } a \iff \text{even } b\) show \(?thesis\)
  by (simp add: bit-0)
next
  case (Suc \(n\))
  moreover from \(a \div 2 = b \div 2\) have \(\text{bit } (a \div 2) \ n = \text{bit } (b \div 2) \ n\)
  by simp
  ultimately show \(?thesis\)
  by (simp add: bit-Suc)
qed
qed

lemma bit-eq-iff:
\( a = b \iff (\forall n. \text{possible-bit } \text{TYPE}(\prime a) \ n \rightarrow \text{bit } a \ n \iff \text{bit } b \ n)\)
by (auto intro: bit-eqI simp add: possible-bit-def)

lemma bit-0-eq [simp]:
\(\text{bit } 0 = \bot\)
proof
  have \(\langle 0 \ \text{div } 2 \sim n = 0\rangle\) for \(n\)
  unfolding div-exp-eq-funpow-half by (induction \(n\)) simp-all
  then show \(?thesis\)
  by (simp add: fun-eq-iff bit-iff-odd)
qed
lemma bit-double-Suc-iff:
\begin{align*}
\langle \text{bit} (2 \ast a) \ (\text{Suc} \ n) \longleftrightarrow \text{possible-bit} \ TYPE\ ('a) \ (\text{Suc} \ n) \land \text{bit} \ a \ n \rangle \\
\text{using even-double-div-exp-iff [of n a]} \\
\text{by (cases \langle \text{possible-bit} \ TYPE\ ('a) \ (\text{Suc} \ n) \rangle)} \\
\text{(auto simp add: bit-iff-odd possible-bit-def)}
\end{align*}

lemma bit-double-iff [bit-simps]:
\begin{align*}
\langle \text{bit} (2 \ast a) \ n \longleftrightarrow \text{possible-bit} \ TYPE\ ('a) \ n \land n \neq 0 \land \text{bit} \ a \ (n - 1) \rangle \\
\text{by (cases n) (simp-all add: bit-0 bit-double-Suc-iff)}
\end{align*}

lemma even-bit-succ-iff:
\begin{align*}
\langle \text{bit} (1 + a) \ n \longleftrightarrow \text{bit} \ a \ n \lor n = 0 \rangle \ \text{if} \ \langle \text{even} \ a \rangle \\
\text{using that by (cases \langle n = 0 \rangle) (simp-all add: bit-iff-odd)}
\end{align*}

lemma odd-bit-iff-bit-pred:
\begin{align*}
\langle \text{bit} \ a \ n \longleftrightarrow \text{bit} (a - 1) \ n \lor n = 0 \rangle \ \text{if} \ \langle \text{odd} \ a \rangle \\
\text{proof} \\
\text{from} \ \langle \text{odd} \ a \rangle \ \text{obtain} \ b \ \text{where} \ \langle a = 2 \ast b + 1 \rangle \ .. \\
\text{moreover have} \ \langle \text{bit} (2 \ast b) \ n \lor n = 0 \longleftrightarrow \text{bit} (1 + 2 \ast b) \ n \rangle \\
\text{using even-bit-succ-iff by simp} \\
\text{ultimately show} \ \langle \text{thesis} \rangle \ \text{by (simp add: ac-simps)}
\end{align*}

qed

lemma bit-exp-iff [bit-simps]:
\begin{align*}
\langle \text{bit} (2 ^ m) \ n \longleftrightarrow \text{possible-bit} \ TYPE\ ('a) \ n \land n = m \rangle \\
\text{proof (cases \langle \text{possible-bit} \ TYPE\ ('a) \ n \rangle)} \\
\text{case False} \\
\text{then show} \ \langle \text{thesis} \rangle \\
\text{by (simp add: impossible-bit)}
\end{align*}

next
\begin{align*}
\text{case True} \\
\text{then show} \ \langle \text{thesis} \rangle \\
\text{proof (induction n arbitrary: m)} \\
\text{case 0} \\
\text{show} \ \langle \text{case} \rangle \\
\text{by (simp add: bit-0)}
\end{align*}

next
\begin{align*}
\text{case (Suc n)} \\
\text{then have} \ \langle \text{possible-bit} \ TYPE\ ('a) \ n \rangle \\
\text{by (simp add: possible-bit-less-imp)} \\
\text{show} \ \langle \text{case} \rangle \\
\text{proof (cases m)} \\
\text{case 0} \\
\text{then show} \ \langle \text{thesis} \rangle \\
\text{by (simp add: bit-Suc)}
\end{align*}

next
\begin{align*}
\text{case (Suc m)} \\
\text{with Suc.IH [of m] \langle \text{possible-bit} \ TYPE\ ('a) \ n \rangle \ \text{show} \ \langle \text{thesis} \rangle \\
\text{by (simp add: bit-double-Suc-iff)}
\end{align*}
lemma bit-1-iff [bit-simps]:
\[ \text{bit} \ 1 \ n \longleftrightarrow n = 0 \]
using bit-exp-iff [of 0 n] by auto

lemma bit-2-iff [bit-simps]:
\[ \text{bit} \ 2 \ n \longleftrightarrow \text{possible-bit} \ \text{TYPE}(a) \ 1 \land n = 1 \]
using bit-exp-iff [of 1 n] by auto

lemma bit-of-bool-iff [bit-simps]:
\[ \text{bit} \ (\text{of-bool} \ b) \ n \longleftrightarrow n = 0 \land b \]
by (simp add: bit-1-iff)

lemma bit-mod-2-iff [simp]:
\[ \text{bit} \ (a \ mod \ 2) \ n \longleftrightarrow n = 0 \land \text{odd} \ a \]
by (simp add: mod-2-eq-odd bit-simps)

end

lemma nat-bit-induct [case-names zero even odd]:
\[ P \ n \] if zero: \[ P \ 0 \]
and even: \[ \forall n. P \ n \longrightarrow n > 0 \longrightarrow P \ (2 \ast n) \]
and odd: \[ \forall n. P \ n \longrightarrow P \ (Suc \ (2 \ast n)) \]
proof (induction n rule: less-induct)
case (less n)
show \[ P \ n \]
proof (cases \[ n = 0 \])
case True with zero show \[ ?thesis \] by simp
next
case False
with less have hyp: \[ P \ (n \ div \ 2) \] by simp
show \[ ?thesis \]
proof (cases \[ \text{even} \ n \])
case True
then have \[ n \neq 1 \]
by auto
with \[ n \neq 0 \] have \[ n \ div \ 2 > 0 \]
by simp
with \[ \text{even} \ n \] hyp even [of \[ n \ div \ 2 \]] show \[ ?thesis \]
by simp
next
case False
with hyp odd [of \[ n \ div \ 2 \]] show \[ ?thesis \]
by simp
qed
qed
definition bit-nat :: (nat ⇒ nat ⇒ bool)
  where bit-nat m n ←→ odd (m div 2 ^ n)

instance proof show (P n) if stable: (∀ n. n div 2 = n ⇒ P n)
  and rec: (∀ n b. P n ⇒ (of-bool b + 2 * n) div 2 = n ⇒ P (of-bool b + 2 * n))
  for P and n :: nat
  proof (induction n rule: nat-bit-induct)
    case zero from stable [of 0] show ?case by simp
  next case (even n) with rec [of n False] show ?case by simp
  next case (odd n) with rec [of n True] show ?case by simp
  qed
  qed (auto simp add: div-mult2-eq bit-nat-def)
end

lemma possible-bit-nat [simp]:
  (possible-bit TYPE(nat) n)
  by (simp add: possible-bit-def)

lemma bit-Suc-0-iff [bit-simps]:
  (bit (Suc 0) n ←→ n = 0)
  using bit-1-iff [of n, where ?a = n] by simp

lemma not-bit-Suc-0-Suc [simp]:
  (¬ bit (Suc 0) (Suc n))
  by (simp add: bit-Suc)

lemma not-bit-Suc-0-numeral [simp]:
  (¬ bit (Suc 0) (numeral n))
  by (simp add: numeral-eq-Suc)

context semiring-bits
begin
lemma bit-of-nat-iff [bit-simps]:
\[ \text{bit (of-nat m) } n \iff \text{possible-bit TYPE(a) } n \land \text{bit m } n \]
proof (cases \(\text{possible-bit TYPE(a) } n\))
  case False
  then show \(\text{\textit{thesis}}\)
    by (simp add: impossible-bit)
next
  case True
  then have \(\text{bit (of-nat m) } n \iff \text{bit m } n\)
  proof (induction \(\text{m}\) arbitrary: \(\text{n}\) rule: nat-bit-induct)
    case zero
    then show \(\text{\textit{case}}\)
      by simp
  next
    case (even \(\text{m}\))
    then show \(\text{\textit{case}}\)
      by (cases \(\text{n}\))
      (auto simp add: bit-double-iff Bit-Operations.bit-double-iff possible-bit-def
      bit-0 dest: mult-not-zero)
  next
    case (odd \(\text{m}\))
    then show \(\text{\textit{case}}\)
      by (cases \(\text{n}\))
      (auto simp add: bit-double-iff even-bit-succ-iff possible-bit-def
      Bit-Operations.bit-Suc Bit-Operations.bit-0 dest: mult-not-zero)
  qed
with True show \(\text{\textit{thesis}}\)
  by simp
qed
end

lemma int-bit-induct [case-names zero minus even odd]:
\(\text{P k}\) if \(\text{zero-int}\): \(\text{P 0}\)
and \(\text{minus-int}\): \(\text{P (} - 1\text{)}\)
and \(\text{even-int}\): \(\forall k. \text{P } k \Rightarrow k \neq 0 \Rightarrow \text{P } (k \ast 2)\)
and \(\text{odd-int}\): \(\forall k. \text{P } k \Rightarrow k \neq -1 \Rightarrow \text{P } (1 + (k \ast 2))\), \text{ for k :: int}
proof (cases \(\text{k} \geq 0\))
case True
define \(\text{n where} \ (\text{n} = \text{nat } \text{k})\)
with True have \(\text{k = int } \text{n}\)
  by simp
then show \(\text{P } \text{k}\)
proof (induction \(\text{n}\) arbitrary: \(\text{k}\) rule: nat-bit-induct)
case zero
  then show \(\text{\textit{case}}\)
    by (simp add: zero-int)
next
case (even n)
  have \( P (\text{int } n \times 2) \)
    by (rule even-int) (use even in simp-all)
  with even show ?case
    by (simp add: ac-simps)
next
  case (odd n)
  have \( P (1 + (\text{int } n \times 2)) \)
    by (rule odd-int) (use odd in simp-all)
  with odd show ?case
    by (simp add: ac-simps)
qed

next
  case False
  define n where \( n = \text{nat} (- k - 1) \)
  with False have \( k = - \text{int } n - 1 \)
    by simp
  then show \( P k \)
    proof (induction n arbitrary: k rule: nat-bit-induct)
      case zero
      then show ?case
        by (simp add: minus-int)
    next
      case (even n)
      have \( P (1 + (- \text{int } (\text{Suc } n) \times 2)) \)
        by (rule odd-int) (use even in \( \langle \text{simp-all add: algebra-simps} \rangle \))
      also have \( \ldots = - \text{int } (2 \times n) - 1 \)
        by (simp add: algebra-simps)
      finally show ?case
        using even.prems by simp
    next
      case (odd n)
      have \( P (- \text{int } (\text{Suc } n) \times 2) \)
        by (rule even-int) (use odd in \( \langle \text{simp-all add: algebra-simps} \rangle \))
      also have \( \ldots = - \text{int } (\text{Suc } (2 \times n)) - 1 \)
        by (simp add: algebra-simps)
      finally show ?case
        using odd.prems by simp
    qed
  qed

instantiation int :: semiring-bits
begin

definition bit-int :: \( \langle \text{int } \Rightarrow \text{nat } \Rightarrow \text{bool} \rangle \)
  where \( \text{bit-int } k n \longleftrightarrow \text{odd } (k \text{ div } 2 ^ n) \)

instance
proof
show \( \langle P \rangle \) if \( \text{stable} \): \( \langle \forall k. k \div 2 = k \Rightarrow P k \rangle \)
and rec: \( \langle \forall k. P k \Rightarrow (\text{of-bool } b + 2 \ast k) \div 2 = k \Rightarrow P (\text{of-bool } b + 2 \ast k) \rangle \)
for \( P \) and \( k :: \text{int} \)
proof (induction \( k \) rule: \text{int-bit-induct})
case zero
from \( \text{stable} [\, \text{of } 0\,] \) show \( \text{?case} \)
by simp
next
case \( \text{minus} \)
from \( \text{stable} [\, \text{of } (-1)\,] \) show \( \text{?case} \)
by simp
next
case \( \text{even } k \)
with rec [\, \text{of } k \, \text{False}\,] show \( \text{?case} \)
by (simp add: \text{ac-simps})
next
case \( \text{odd } k \)
with rec [\, \text{of } k \, \text{True}\,] show \( \text{?case} \)
by (simp add: \text{ac-simps})
qed
qed (auto simp add: \text{zdiv-zmult2-eq bit-int-def})
end

lemma possible-bit-int [simp]:
\( \langle \text{possible-bit } \text{TYPE(int) } n \rangle \)
by (simp add: possible-bit-def)

lemma bit-nat-iff [bit-simps]:
\( \langle \text{bit } (\text{nat } k) \, n \iff k \geq 0 \land \text{bit } k \, n \rangle \)
proof (cases \( k \geq 0 \))
case True
moreover define \( m \) where \( \langle m = \text{nat } k \rangle \)
ultimately have \( \langle k = \text{int } m \rangle \)
by simp
then show \( \text{?thesis} \)
by (simp add: \text{bit-simps})
next
case False
then show \( \text{?thesis} \)
by simp
qed

68.2 Bit operations

class \text{semiring-bit-operations} = \text{semiring-bits} +
fixes and :: \( \langle \, \langle a \Rightarrow a \Rightarrow a \rangle \, \langle \text{infixr } \langle \text{AND}\rangle \, 64 \rangle \)
and or :: \( \langle a \Rightarrow a \Rightarrow a \rangle \, \langle \text{infixr } \langle \text{OR}\rangle \, 59 \rangle \)
and xor :: \texttt{'a ⇒ 'a ⇒ 'a} \ (\textit{infixr} \langle\textsc{XOR}\rangle \ 59)
and mask :: \texttt{'nat ⇒ 'a}
and set-bit :: \texttt{'nat ⇒ 'a ⇒ 'a}
and unset-bit :: \texttt{'nat ⇒ 'a ⇒ 'a}
and flip-bit :: \texttt{'nat ⇒ 'a ⇒ 'a}
and push-bit :: \texttt{'nat ⇒ 'a ⇒ 'a}
and drop-bit :: \texttt{'nat ⇒ 'a ⇒ 'a}
and take-bit :: \texttt{'nat ⇒ 'a ⇒ 'a}

assumes \textit{and-rec}:
\langle\texttt{a AND b} = \texttt{of-bool (odd} \texttt{a} \land \texttt{odd} \texttt{b} + 2 \ast ((\texttt{a div} \texttt{2}) \texttt{AND} (\texttt{b div} 2))\rangle
and \textit{or-rec}:
\langle\texttt{a OR b} = \texttt{of-bool (odd} \texttt{a} \lor \texttt{odd} \texttt{b} + 2 \ast ((\texttt{a div} \texttt{2}) \texttt{OR} (\texttt{b div} 2))\rangle
and \textit{xor-rec}:
\langle\texttt{a XOR b} = \texttt{of-bool (odd} \texttt{a} \not= \texttt{odd} \texttt{b} + 2 \ast ((\texttt{a div} \texttt{2}) \texttt{XOR} (\texttt{b div} 2))\rangle
and \textit{mask-eq-exp-minus-1}:
\langle\texttt{mask n} = \texttt{2} ^ \texttt{n} \not- \texttt{1}\rangle
and \textit{set-bit-eq-or}:
\langle\texttt{set-bit n a} = \texttt{a OR push-bit n 1}\rangle
and \textit{unset-bit-eq-or-xor}:
\langle\texttt{unset-bit n a} = (\texttt{a OR push-bit n 1}) \texttt{XOR push-bit n 1}\rangle
and \textit{flip-bit-eq-xor}:
\langle\texttt{flip-bit n a} = \texttt{a XOR push-bit n 1}\rangle
and \textit{push-bit-eq-mult}:
\langle\texttt{push-bit n a} = \texttt{a \ast 2} ^ \texttt{n}\rangle
and \textit{drop-bit-eq-div}:
\langle\texttt{drop-bit n a} = \texttt{a div} \texttt{2} ^ \texttt{n}\rangle
and \textit{take-bit-eq-mod}:
\langle\texttt{take-bit n a} = \texttt{a mod} \texttt{2} ^ \texttt{n}\rangle

begin

We want the bitwise operations to bind slightly weaker than + and -. Logically, \textit{push-bit}, \textit{drop-bit} and \textit{take-bit} are just aliases; having them as separate operations makes proofs easier, otherwise proof automation would fiddle with concrete expressions \langle\texttt{2} :: \texttt{'}a\rangle^n in a way obfuscating the basic algebraic relationships between those operations. For the sake of code generation operations are specified as definitional class operations, taking into account that specific instances of these can be implemented differently wrt. code generation.

lemma \textit{bit-iff-odd-drop-bit}:
\langle\texttt{bit a n} \iff\texttt{odd} \texttt{(drop-bit} \texttt{n} \texttt{a)}\rangle
by \texttt{(simp add: bit-iff-odd drop-bit-eq-div)}

lemma \textit{even-drop-bit-iff-not-bit}:
\langle\texttt{even} \texttt{(drop-bit} \texttt{n} \texttt{a)} \iff\texttt{~bit} \texttt{a} \texttt{n}\rangle
by \texttt{(simp add: bit-iff-odd-drop-bit)}

lemma \textit{bit-and-iff} \ [bit-simps]:
\langle\texttt{bit} \ (\texttt{a AND} \texttt{b}) \texttt{n} \iff\texttt{bit} \texttt{a} \texttt{n} \lor \texttt{bit} \texttt{b} \texttt{n}\rangle
proof (induction \texttt{n} arbitrary: \texttt{a} \texttt{b})
case \texttt{0}
show \texttt{?case}
by \texttt{(simp add: bit-0 and-rec [of \texttt{a} \texttt{b}] even-bit-succ-iff)}
next
case \texttt{(Suc} \texttt{n)}
from \texttt{Suc} \texttt{[of} \langle\texttt{a div} \texttt{2}\rangle \langle\texttt{b div} \texttt{2}\rangle\rangle
show ?case
  by (simp add: and-rec [of a b] bit-Suc)
    (auto simp flip: bit-Suc simp add: bit-double-iff dest: bit-imp-possible-bit)
qed

lemma bit-or-iff [bit-simps]:
  ⟨bit (a OR b) n ⟷ bit a n ∨ bit b n⟩
proof (induction n arbitrary: a b)
  case 0
    show ?case
      by (simp add: bit-0 or-rec [of a b] even-bit-succ-iff)
next
  case (Suc n)
  from Suc [of ⟨a div 2, b div 2⟩]
  show ?case
    by (simp add: or-rec [of a b] bit-Suc)
      (auto simp flip: bit-Suc simp add: bit-double-iff dest: bit-imp-possible-bit)
qed

lemma bit-xor-iff [bit-simps]:
  ⟨bit (a XOR b) n ⟷ bit a n ⊻ bit b n⟩
proof (induction n arbitrary: a b)
  case 0
    show ?case
      by (simp add: bit-0 xor-rec [of a b] even-bit-succ-iff)
next
  case (Suc n)
  from Suc [of ⟨a div 2, b div 2⟩]
  show ?case
    by (simp add: xor-rec [of a b] bit-Suc)
      (auto simp flip: bit-Suc simp add: bit-double-iff dest: bit-imp-possible-bit)
qed

sublocale and: semilattice ⟨(AND)⟩
  by standard (auto simp add: bit-eq-iff bit-and-iff)

sublocale or: semilattice-neutr ⟨(OR)⟩ 0
  by standard (auto simp add: bit-eq-iff bit-or-iff)

sublocale xor: comm-monoid ⟨(XOR)⟩ 0
  by standard (auto simp add: bit-eq-iff bit-xor-iff)

lemma even-and-iff:
  ⟨even (a AND b) ⟷ even a ∨ even b⟩
using bit-and-iff [of a b 0] by (auto simp add: bit-0)

lemma even-or-iff:
  ⟨even (a OR b) ⟷ even a ∧ even b⟩
using bit-or-iff [of a b 0] by (auto simp add: bit-0)
lemma even-xor-iff:
\[\text{even} (a \text{ XOR } b) \iff (\text{even } a \iff \text{even } b)\]
using bit-xor-iff \([a b 0]\) by (auto simp add: bit-0)

lemma zero-and-eq [simp]:
\[0 \text{ AND } a = 0\]
by (simp add: bit-eq-iff bit-and-iff)

lemma and-zero-eq [simp]:
\[a \text{ AND } 0 = 0\]
by (simp add: bit-eq-iff bit-and-iff)

lemma one-and-eq:
\[1 \text{ AND } a = a \mod 2\]
by (simp add: bit-eq-iff bit-and-iff) (auto simp add: bit-1 iff bit-0)

lemma and-one-eq:
\[a \text{ AND } 1 = a \mod 2\]
using one-and-eq \([a]\) by (simp add: ac-simps)

lemma one-or-eq:
\[1 \text{ OR } a = a + \text{of-bool } (\text{even } a)\]
by (simp add: bit-eq-iff bit-or-iff add.commute \([a - 1]\) even-bit-succ-iff)
(auto simp add: bit-1 iff bit-0)

lemma or-one-eq:
\[a \text{ OR } 1 = a + \text{of-bool } (\text{even } a)\]
using one-or-eq \([a]\) by (simp add: ac-simps)

lemma one-xor-eq:
\[1 \text{ XOR } a = a + \text{of-bool } (\text{even } a) - \text{of-bool } (\text{odd } a)\]
by (simp add: bit-eq-iff bit-xor-iff add.commute \([a - 1]\) even-bit-succ-iff)
(auto simp add: bit-1 iff odd-bit-succ-bit-0 elim: oddE)

lemma xor-one-eq:
\[a \text{ XOR } 1 = a + \text{of-bool } (\text{even } a) - \text{of-bool } (\text{odd } a)\]
using one-xor-eq \([a]\) by (simp add: ac-simps)

lemma xor-self-eq [simp]:
\[a \text{ XOR } a = 0\]
by (rule bit-eqI) (simp add: bit-simps)

lemma mask-0 [simp]:
\[\text{mask } 0 = 0\]
by (simp add: mask-eq-exp-minus-1)

lemma inc-mask-eq-exp:
\[\text{mask } n + 1 = 2 \sim n\]
proof (induction n)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  from Suc.IH [symmetric] have \(2 \sim Suc \cdot n = 2 \ast mask \cdot n + 2\)
    by (simp add: algebra-simps)
  also have \(\ldots = 2 \ast mask \cdot n + 1 + 1\)
    by (simp add: add.assoc)
  finally have \(*: 2 \sim Suc \cdot n = 2 \ast mask \cdot n + 1 + 1\).
  then show ?case
    by (simp add: mask-eq-exp-minus-1)
qed

lemma mask-Suc-double:
  \(\text{mask}(Suc \cdot n) = 1 \ OR 2 \ast mask \cdot n\)
proof –
  have \(\text{mask}(Suc \cdot n) + 1 = (\text{mask} \cdot n + 1) + (\text{mask} \cdot n + 1)\)
    by (simp add: inc-mask-eq-exp mult-2)
  also have \(\ldots = (1 \ OR 2 \ast mask \cdot n) + 1\)
    by (simp add: one-or-eq mult-2-right algebra-simps)
  finally show ?thesis
    by simp
qed

lemma bit-mask-iff [bit-simps]:
  \(\text{bit}(\text{mask} \cdot m \cdot n) \longleftrightarrow \text{possible-bit} \ TYPE\(\cdot\)a \cdot n \wedge n < m\)
proof (cases \(\text{possible-bit} \ TYPE\(\cdot\)a \cdot n\))
  case False
  then show ?thesis
    by (simp add: impossible-bit)
next
  case True
  then have \(\text{bit} \ (\text{mask} \cdot m \cdot n) \longleftrightarrow n < m\)
proof (induction \(m\) arbitrary: \(n\))
  case 0
  then show ?case
    by (simp add: bit-iff-odd)
next
  case (Suc \(m\))
  show ?case
proof (cases \(n\))
  case 0
  then show ?thesis
    by (simp add: bit-0 mask-Suc-double even-or-iff)
next
  case (Suc \(n\))
  with Suc.prems have \(\text{possible-bit} \ TYPE\(\cdot\)a \cdot n\)
using possible-bit-less-imp by auto
with Suc.IH [of n] have \( \langle \text{bit (mask m)} \ n \longleftrightarrow n < m \rangle \).
with Suc.prems show \( \text{thesis} \)
  by (simp add: Suc mask-Suc-double bit-simps)
qed
qed
with True show \( \text{thesis} \)
  by simp
qed

lemma even-mask-iff:
\( \langle \text{even (mask n)} \longleftrightarrow n = 0 \rangle \)
using bit-mask-iff [of n 0] by (auto simp add: bit-0)

lemma mask-Suc-0 [simp]:
\( \langle \text{mask (Suc 0)} = 1 \rangle \)
by (simp add: mask-Suc-double)

lemma mask-Suc-exp:
\( \langle \text{mask (Suc n)} = 2 ^ n \ OR \ mask n \rangle \)
by (auto simp add: bit-eq-iff bit-simps)

lemma mask-numeral:
\( \langle \text{mask (numeral n)} = 1 + 2 \ast \ mask (\text{pred-numeral n}) \rangle \)
by (simp add: numeral-eq-Suc mask-Suc-double ac-simps)

lemma push-bit-0-id [simp]:
\( \langle \text{push-bit 0 = id} \rangle \)
by (simp add: fun-eq-iff push-bit-eq-mult)

lemma push-bit-Suc [simp]:
\( \langle \text{push-bit (Suc n) a = push-bit n (a * 2)} \rangle \)
by (simp add: push-bit-eq-mult ac-simps)

lemma push-bit-double:
\( \langle \text{push-bit n (a * 2) = push-bit n a * 2} \rangle \)
by (simp add: push-bit-eq-mult ac-simps)

lemma bit-push-bit-iff [bit-simps]:
\( \langle \text{bit (push-bit m a) n} \longleftrightarrow m \leq n \ \&\ \ \text{possible-bit TYPE('a) n} \ \&\ \ \text{bit a} \ (n - m) \rangle \)
proof (induction n arbitrary: m)
case 0
  then show \( \text{?case} \)
    by (auto simp add: bit-0 push-bit-eq-mult)
next
case (Suc n)
  show \( \text{?case} \)
  proof (cases m)
    case 0
then show thesis
  by (auto simp add: bit-imp-possible-bit)
next
case (Suc m)
  with Suc.prems Suc.IH [of m] show thesis
  apply (simp add: push-bit-double)
  apply (simp add: bit-simps mult.commute [of - 2])
  apply (auto simp add: possible-bit-less-imp)
done
qed

lemma funpow-double-eq-push-bit:
  \(<\times 2 \sim n = \text{push-bit} \ n>\)
  by (induction n) (simp-all add: fun-eq-iff push-bit-double ac-simps)

lemma div-push-bit-of-1-eq-drop-bit:
  \(<a \div \text{push-bit} \ n 1 = \text{drop-bit} \ n \ a>\)
  by (simp add: push-bit-eq-mult drop-bit-eq-div)

lemma bits-ident:
  \(<\text{push-bit} \ n (\text{drop-bit} \ n \ a) + \text{take-bit} \ n \ a = a>\)
  using div-mult-mod-eq by (simp add: push-bit-eq-mult take-bit-eq-mod drop-bit-eq-div)

lemma push-bit-push-bit [simp]:
  \(<\text{push-bit} \ n (\text{push-bit} \ n \ a) = \text{push-bit} (m + n) \ a>\)
  by (simp add: push-bit-eq-mult power-add ac-simps)

lemma push-bit-of-0 [simp]:
  \(<\text{push-bit} \ n \ 0 = 0>\)
  by (simp add: push-bit-eq-mult)

lemma push-bit-of-1 [simp]:
  \(<\text{push-bit} \ n \ 1 = 2 \sim n>\)
  by (simp add: push-bit-eq-mult)

lemma push-bit-add:
  \(<\text{push-bit} \ n (a + b) = \text{push-bit} \ n a + \text{push-bit} \ n b>\)
  by (simp add: push-bit-eq-mult algebra-simps)

lemma push-bit-numeral [simp]:
  \(<\text{push-bit} (\text{numeral} \ l) (\text{numeral} \ k) = \text{push-bit} (\text{pred-numeral} \ l) (\text{numeral} (\text{Num.Bit0} \ k))>\)
  by (simp add: numeral-eq-Suc mult-2-right) (simp add: numeral-Bit0)

lemma bit-drop-bit-eq [bit-simps]:
  \(<\text{bit} (\text{drop-bit} \ n \ a) = \text{bit} a \circ (+) \ n>\)
  by rule (simp add: drop-bit-eq-div bit-iff-odd div-exp-eq)
lemma disjunctive-xor-eq-or:
\( a \ XOR b = a \ OR b \) if \( a \ AND b = 0 \)
using that by (auto simp add: bit-eq-iff bit-simps)

lemma disjunctive-add-eq-or:
\( a + b = a \ OR b \) if \( a \ AND b = 0 \)
proof (rule bit-eqI)
fix \( n \)
assume \( \langle \forall n. \neg \bit (a \ AND b) n \rangle \)
then have \( \langle \forall n. \neg \bit a n \lor \neg \bit b n \rangle \)
ultimately show \( \langle \forall n. \bit (a + b) n \iff \bit (a \ OR b) n \rangle \)
proof (induction \( n \) arbitrary: \( a \ b \))
case \( 0 \)
from \( 0 \) [of \( 0 \)] show ?case
by (auto simp add: even-or-iff bit-0)
next
case \( \ Suc \ n \)
from \( \ Suc \ . \ prems \) [of \( 0 \)] have even:
\( \langle even a \lor even b \rangle \)
by (auto simp add: bit-0)
have bit:
\( \langle \forall n. \neg \bit (a \ div \ 2) n \lor \neg \bit (b \ div \ 2) n \rangle \) for \( n \)
using \( \ Suc \ . \ prems \) [of \( \langle \Suc n \rangle \)] by (simp add: bit-Suc)
from \( \ Suc \ . \ prems \) have \( \langle \forall n. \neg \bit (a \ div \ 2) n \lor \neg \bit (b \ div \ 2) n \rangle \) \ Suc.IH [of \( \langle a \ div \ 2 \rangle \ \langle b \ div \ 2 \rangle \)]
have IH:
\( \langle \bit (a \ div \ 2 + b \ div \ 2) n \iff \bit (a \ div \ 2 \ OR \ b \ div \ 2) n \rangle \)
by (simp add: bit-Suc)
have \( \langle a + b = (a \ div \ 2 * 2 + a \ mod \ 2) + (b \ div \ 2 * 2 + b \ mod \ 2) \rangle \)
using \( \ Suc \ . \ prems \) [of \( \langle a \ 2 \rangle \ \langle b \ 2 \rangle \)] by simp
also have \( \ldots = \langle of\-\bit (odd a \lor odd b) + 2 * (a \ div \ 2 + b \ div \ 2) \rangle \)
using even by (auto simp add: algebra-simps mod2-eq-if)
finally have \( \langle \bit ((a + b) \ div \ 2) n \iff \bit (a \ div \ 2 + b \ div \ 2) n \rangle \)
using \( \langle possible\-\bit \ TYPE\('a\) \ (\Suc n) \rangle \) by simp (simp-all flip: bit-Suc add:
bit-double-iff possible-bit-def)
also have \( \ldots \iff \bit (a \ div \ 2 \ OR \ b \ div \ 2) n \)
by (rule IH)
finally show ?case
by (simp add: bit-simps flip: bit-Suc)
qed

lemma disjunctive-add-eq-xor:
\( a + b = a \ XOR b \) if \( a \ AND b = 0 \)
using that by (simp add: disjunctive-add-eq-or disjunctive-xor-eq-or)

lemma take-bit-0 [simp]:
take-bit 0 a = 0
by (simp add: take-bit-eq-mod)

lemma bit-take-bit-iff [bit-simps]:
\[ \text{bit (take-bit m a) n} \leftrightarrow n < m \land \text{bit a n} \]
proof -
  have \( \text{push-bit m (drop-bit m a) AND take-bit m a = 0} \) (is \( \text{?lhs = \_} \))
  proof (rule bit-eqI)
    fix n
    show \( \text{bit ?lhs n} \leftrightarrow \text{bit 0 n} \)
    proof (cases \( m \leq n \))
      case False
      then show \( \text{?thesis} \)
        by (simp add: bit-simps)
    next
      case True
      moreover define q where \( q = n - m \)
      ultimately have \( n = m + q \) by simp
      moreover have \( \neg \text{bit (take-bit m a) (m + q)} \)
        by (simp add: take-bit-eq-mod bit-iff-odd flip: div-exp-eq)
      ultimately show \( \text{?thesis} \)
        by (simp add: bit-simps)
    qed
    qed
    then have \( \text{push-bit m (drop-bit m a) XOR take-bit m a} = \text{push-bit m (drop-bit m a)} + \text{take-bit m a} \)
      by (simp add: disjunctive-add-eq-xor)
    also have \( \ldots = a \)
      by (simp add: bits-ident)
    finally have \( \text{bit (push-bit m (drop-bit m a) XOR take-bit m a) n} \leftrightarrow \text{bit a n} \)
      by simp
    also have \( \ldots \leftrightarrow (m \leq n \lor n < m) \land \text{bit a n} \)
      by auto
    also have \( \ldots \leftrightarrow m \leq n \land \text{bit a n} \land n < m \land \text{bit a n} \)
      by auto
    also have \( m \leq n \land \text{bit a n} \leftrightarrow \text{bit (push-bit m (drop-bit m a)) n} \)
      by (auto simp add: bit-simps bit-imp-possible-bit)
    finally show \( \text{?thesis} \)
      by (auto simp add: bit-simps)
  qed

lemma take-bit-Suc:
\( \text{take-bit (Suc n) a} = \text{take-bit n (a div 2) \ast 2 + a mod 2} \) (is \( \text{?lhs = \_} \))
proof (rule bit-eqI)
  fix m
  assume \( \text{possible-bit TYPE('a) m} \)
  then show \( \text{bit ?lhs m} \leftrightarrow \text{bit ?rhs m} \)
    apply (cases a rule: parity-cases; cases m)
    apply (simp-all add: bit-simps even-bit-succ-iff mult.commute [of - 2]

add commute [of - 1] flip: bit-Suc
apply (simp-all add: bit-0)
done
qed

lemma take-bit-rec:
\(\langle \text{take-bit } n \ a = (\text{if } n = 0 \ \text{then } 0 \ \text{else take-bit } (n - 1) \ (a \ \text{div} \ 2) \ * \ 2 + \ a \ \text{mod} \ 2) \rangle\)
by (cases \(n\)) (simp-all add: take-bit-Suc)

lemma take-bit-Suc-0 [simp]:
\(\langle \text{take-bit } (\text{Suc } 0) \ a = a \ \text{mod} \ 2 \rangle\)
by (simp add: take-bit-eq-mod)

lemma take-bit-of-0 [simp]:
\(\langle \text{take-bit } n \ 0 = 0 \rangle\)
by (rule bit-eqI) (simp add: bit-simps)

lemma take-bit-of-1 [simp]:
\(\langle \text{take-bit } n \ 1 = \text{of-bool} \ (n > 0) \rangle\)
by (cases \(n\)) (simp-all add: take-bit-Suc)

lemma drop-bit-of-0 [simp]:
\(\langle \text{drop-bit } n \ 0 = 0 \rangle\)
by (rule bit-eqI) (simp add: bit-simps)

lemma drop-bit-of-1 [simp]:
\(\langle \text{drop-bit } n \ 1 = \text{of-bool} \ (n = 0) \rangle\)
by (rule bit-eqI) (simp add: bit-simps ac-simps)

lemma drop-bit-0 [simp]:
\(\langle \text{drop-bit } 0 = \text{id} \rangle\)
by (simp add: fun-eq-iff drop-bit-eq-div)

lemma drop-bit-Suc:
\(\langle \text{drop-bit } (\text{Suc } n) \ a = \text{drop-bit } n \ (a \ \text{div} \ 2) \rangle\)
using div-exp-eq [of a 1] by (simp add: drop-bit-eq-div)

lemma drop-bit-rec:
\(\langle \text{drop-bit } n \ a = (\text{if } n = 0 \ \text{then } a \ \text{else drop-bit } (n - 1) \ (a \ \text{div} \ 2)) \rangle\)
by (cases \(n\)) (simp-all add: drop-bit-Suc)

lemma drop-bit-half:
\(\langle \text{drop-bit } n \ (a \ \text{div} \ 2) = \text{drop-bit } n \ a \ \text{div} \ 2 \rangle\)
by (induction \(n\) arbitrary: \(a\)) (simp-all add: drop-bit-Suc)

lemma drop-bit-of-bool [simp]:
\(\langle \text{drop-bit } n \ (\text{of-bool } b) = \text{of-bool} \ (n = 0 \ \land \ b) \rangle\)
by (cases \(n\)) simp-all
lemma even-take-bit-eq [simp]:
\( \text{even} \ (\text{take-bit} \ n \ a) \iff n = 0 \lor \text{even} \ a \)
by (simp add: take-bit-rec [of n a])

lemma take-bit-take-bit [simp]:
\( \text{take-bit} \ m \ (\text{take-bit} \ n \ a) = \text{take-bit} \ (\text{min} \ m \ n) \ a \)
by (rule bit-eqI) (simp add: bit-simps)

lemma drop-bit-drop-bit [simp]:
\( \text{drop-bit} \ m \ (\text{drop-bit} \ n \ a) = \text{drop-bit} \ (m + n) \ a \)
by (rule bit-eqI) (simp add: drop-bit-eq-div power-add div-exp-eq ac-simps)

lemma push-bit-take-bit:
\( \text{push-bit} \ m \ (\text{take-bit} \ n \ a) = \text{take-bit} \ (m + n) \ (\text{push-bit} \ m \ a) \)
by (rule bit-eqI) (auto simp add: bit-simps)

lemma take-bit-push-bit:
\( \text{take-bit} \ m \ (\text{push-bit} \ n \ a) = \text{push-bit} \ n \ (\text{take-bit} \ (m - n) \ a) \)
by (rule bit-eqI) (auto simp add: bit-simps)

lemma drop-bit-take-bit:
\( \text{drop-bit} \ m \ (\text{take-bit} \ n \ a) = \text{take-bit} \ (n - m) \ (\text{drop-bit} \ m \ a) \)
by (rule bit-eqI) (auto simp add: bit-simps)

lemma even-push-bit-iff [simp]:
\( \text{even} \ (\text{push-bit} \ n \ a) \iff n \neq 0 \lor \text{even} \ a \)
by (simp add: push-bit-eq-mult) auto

lemma exp-dvdE:
assumes \( 2 ^ n \ | a \)
obtains \( b \) where \( a = \text{push-bit} \ n \ b \)
proof -
from assms obtain \( b \) where \( a = 2 ^ n * b \) ..
then have \( a = \text{push-bit} \ n \ b \)
by (simp add: push-bit-eq-mult ac-simps)
with that show thesis .
qed

lemma take-bit-eq-0-iff:
\langle take-bit n a = 0 \iff \sim n \text{ dvd } a \rangle \ (\text{is } ?P \iff ?Q)
proof
assume ?P
then show ?Q
  by (simp add: take-bit-eq-mod mod-0-imp-dvd)
next
assume ?Q
then obtain b where \langle a = \text{push-bit } n \ b \rangle
  by (rule exp-dvdE)
then show ?P
  by (simp add: take-bit-push-bit)
qed

lemma take-bit-tightened:
\langle take-bit m a = take-bit m b \rangle \text{ if } \langle take-bit n a = take-bit n b \rangle \text{ and } \langle m \leq n \rangle
proof
from that have \langle take-bit m (take-bit n a) = take-bit m (take-bit n b) \rangle
by simp
then have \langle take-bit (\text{min } m n) a = take-bit (\text{min } m n) b \rangle
  by simp
with that show ?thesis
by (simp add: min-def)
qed

lemma take-bit-eq-self-iff-drop-bit-eq-0:
\langle take-bit n a = a \iff \text{drop-bit } n \ a = 0 \rangle \ (\text{is } ?P \iff ?Q)
proof
assume ?P
show ?Q
proof (rule bit-eql)
  fix m
  from \langle ?P \rangle have \langle a = \text{take-bit } n \ a \rangle ..
  also have \langle \sim \text{bit } (\text{take-bit } n \ a) \ (n + m) \rangle
    unfolding bit-simps
    by (simp add: bit-simps)
  finally show \langle \text{bit } (\text{drop-bit } n \ a) \ m \iff \text{bit } 0 \ m \rangle
    by (simp add: bit-simps)
qed
next
assume ?Q
show ?P
proof (rule bit-eql)
  fix m
  from \langle ?Q \rangle have \langle \sim \text{bit } (\text{drop-bit } n \ a) \ (m - n) \rangle
    by simp
then have \( \neg\) bit a (n + (m − n))
by (simp add: bit-simps)
then show \( \langle\) bit (take-bit n a) m \( \leftrightarrow\) bit a m
by (cases \( m < n\)) (auto simp add: bit-simps)
qed

lemma drop-bit-exp-eq:
\( \langle\) drop-bit m \( (2^\langle\) n \( \rangle) = \) of-bool \( (m \leq n \land possible-bit \ TYPE\langle a \rangle \ n) * 2^\langle n - m \rangle\)
by (auto simp add: bit-eq-iff bit-simps)

lemma take-bit-and [simp]:
\( \langle\) take-bit n (a AND b) = take-bit n a AND take-bit n b
by (auto simp add: bit-eq-iff bit-simps)

lemma take-bit-or [simp]:
\( \langle\) take-bit n (a OR b) = take-bit n a OR take-bit n b
by (auto simp add: bit-eq-iff bit-simps)

lemma take-bit-xor [simp]:
\( \langle\) take-bit n (a XOR b) = take-bit n a XOR take-bit n b
by (auto simp add: bit-eq-iff bit-simps)

lemma push-bit-and [simp]:
\( \langle\) push-bit n (a AND b) = push-bit n a AND push-bit n b
by (auto simp add: bit-eq-iff bit-simps)

lemma push-bit-or [simp]:
\( \langle\) push-bit n (a OR b) = push-bit n a OR push-bit n b
by (auto simp add: bit-eq-iff bit-simps)

lemma push-bit-xor [simp]:
\( \langle\) push-bit n (a XOR b) = push-bit n a XOR push-bit n b
by (auto simp add: bit-eq-iff bit-simps)

lemma drop-bit-and [simp]:
\( \langle\) drop-bit n (a AND b) = drop-bit n a AND drop-bit n b
by (auto simp add: bit-eq-iff bit-simps)

lemma drop-bit-or [simp]:
\( \langle\) drop-bit n (a OR b) = drop-bit n a OR drop-bit n b
by (auto simp add: bit-eq-iff bit-simps)

lemma drop-bit-xor [simp]:
\( \langle\) drop-bit n (a XOR b) = drop-bit n a XOR drop-bit n b
by (auto simp add: bit-eq-iff bit-simps)

lemma take-bit-of-mask [simp]:
\( \langle\) take-bit m (mask n) = mask \( \langle\) min m n \( \rangle\)
by (rule bit-eqI) (simp add: bit-simps)

lemma take-bit-eq-mask:
  ⟨take-bit n a = a AND mask n⟩
by (auto simp add: bit-eq-iff bit-simps)

lemma or-eq-0-iff:
  ⟨a OR b = 0 ⟷ a = 0 ∧ b = 0⟩
by (auto simp add: bit-eq-iff bit-or-iff)

lemma bit-iff-and-drop-bit-eq-1:
  ⟨bit a n ⟷ drop-bit n a AND 1 = 1⟩
by (simp add: bit-eq-iff bit-simps)

lemma bit-iff-and-push-bit-not-eq-0:
  ⟨bit a n ⟷ a AND push-bit n 1 ≠ 0⟩
by (cases possible-bit TYPE('a n)) (simp-all add: bit-eq-iff bit-simps)

lemma bit-set-bit-iff [bit-simps]:
  ⟨bit (set-bit m a) n ⟷ bit a n ∨ (m = n ∧ possible-bit TYPE('a n))⟩
by (auto simp add: set-bit-eq-or bit-or-iff bit-exp-iff)

lemma even-set-bit-iff:
  ⟨even (set-bit m a) ⟷ even a ∧ m ≠ 0⟩
using bit-set-bit-iff [of m a 0] by (auto simp add: bit-0)

lemma bit-unset-bit-iff [bit-simps]:
  ⟨bit (unset-bit m a) n ⟷ bit a n ∧ m ≠ n⟩
by (auto simp add: unset-bit-eq-or-xor bit-simps dest: bit-imp-possible-bit)

lemma even-unset-bit-iff:
  ⟨even (unset-bit m a) ⟷ even a ∨ m = 0⟩
using bit-unset-bit-iff [of m a 0] by (auto simp add: bit-0)

lemma bit-flip-bit-iff [bit-simps]:
  ⟨bit (flip-bit m a) n ⟷ (m = n ←→ ¬bit a n) ∧ possible-bit TYPE('a n)⟩
by (auto simp add: bit-eq-iff bit-simps flip-bit-eq-xor bit-imp-possible-bit)

lemma even-flip-bit-iff:
  ⟨even (flip-bit m a) ⟷ ¬(even a ←→ m = 0)⟩
using bit-flip-bit-iff [of m a 0] by (auto simp: possible-bit-def bit-0)

lemma and-exp-eq-0-iff-not-bit:
  ⟨a AND 2 ^ n = 0 ⟷ ¬bit a n⟩ (is ⟨?=P ⟷ ?Q⟩)
using bit-imp-possible-bit [of a n] by (auto simp add: bit-eq-iff bit-simps)

lemma bit-sum-mult-2-cases:
assumes a: \( \forall j. \neg \text{bit } a (\text{Suc } j) \)  
shows \( \text{bit } (a + 2 \times b) \ n = (\text{if } n = 0 \ \text{then odd } a \ \text{else bit } (2 \times b) \ n) \)  
proof –  
from a have \( n = 0 \) if \( \text{bit } a \ n \) for \( n \) using that  
by (cases \( n \)) simp-all  
then have \( a = 0 \lor a = 1 \),  
by (auto simp add: bit-eq-iff bit-1-iff)  
then show \( \exists \)thesis  
by (cases \( n \)) (auto simp add: bit-0 bit-double-iff even-bit-succ-iff)  
qed

lemma set-bit-0 \([\text{simp}]\):
\( \text{set-bit } 0 \ a = 1 + 2 \times (a \ \text{div } 2) \)
by (auto simp add: bit-eq-iff bit-simps even-bit-succ-iff simp flip: bit-Suc)

lemma set-bit-Suc:
\( \text{set-bit } (\text{Suc } n) \ a = a \ \text{mod } 2 + 2 \times \text{set-bit } n \ (a \ \text{div } 2) \)
by (auto simp add: bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 simp flip: bit-Suc elim: possible-bit-less-imp)

lemma unset-bit-0 \([\text{simp}]\):
\( \text{unset-bit } 0 \ a = 2 \times (a \ \text{div } 2) \)
by (auto simp add: bit-eq-iff bit-simps simp flip: bit-Suc)

lemma unset-bit-Suc:
\( \text{unset-bit } (\text{Suc } n) \ a = a \ \text{mod } 2 + 2 \times \text{unset-bit } n \ (a \ \text{div } 2) \)
by (auto simp add: bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 simp flip: bit-Suc)

lemma flip-bit-0 \([\text{simp}]\):
\( \text{flip-bit } 0 \ a = \text{of-bool } (\text{even } a) + 2 \times (a \ \text{div } 2) \)
by (auto simp add: bit-eq-iff bit-simps even-bit-succ-iff bit-0 simp flip: bit-Suc)

lemma flip-bit-Suc:
\( \text{flip-bit } (\text{Suc } n) \ a = a \ \text{mod } 2 + 2 \times \text{flip-bit } n \ (a \ \text{div } 2) \)
by (auto simp add: bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 simp flip: bit-Suc elim: possible-bit-less-imp)

lemma flip-bit-eq-if:
\( \text{flip-bit } n \ a = (\text{if bit } a \ n \ \text{then unset-bit else set-bit } n \ a) \)
by (rule bit-eqI) (auto simp add: bit-set-bit-iff bit-unset-bit-iff bit-flip-bit-iff)

lemma take-bit-set-bit-eq:
\( \text{take-bit } n \ (\text{set-bit } m \ a) = (\text{if } n \leq m \ \text{then take-bit } n \ a \ \text{else set-bit } m \ (\text{take-bit } n \ a)) \)
by (rule bit-eqI) (auto simp add: bit-take-bit-iff bit-set-bit-iff)

lemma take-bit-unset-bit-eq:
\( \text{take-bit } n \ (\text{unset-bit } m \ a) = (\text{if } n \leq m \ \text{then take-bit } n \ a \ \text{else unset-bit } m \ (\text{take-bit } n \ a)) \)
by (rule bit-eqI) (auto simp add: bit-take-bit-iff bit-unset-bit-iff)

**lemma** take-bit-flip-bit-eq:
\(<\text{take-bit} n \text{ (flip-bit} m \text{ a}) = (\text{if} n \leq m \text{ then} \text{take-bit} n \text{ a else} \text{flip-bit} m \text{ (take-bit} n \text{ a)})\>
by (rule bit-eqI) (auto simp add: bit-take-bit-iff bit-flip-bit-iff)

**lemma** bit-1-0 [simp]:
\(<\text{bit} 1 \text{ 0}\>
by (simp add: bit-0)

**lemma** not-bit-1-Suc [simp]:
\(<\neg \text{bit} 1 \text{ (Suc} n)\>
by (simp add: bit-Suc)

**lemma** push-bit-Suc-numeral [simp]:
\(<\text{push-bit} \text{ (Suc} n \text{) (numeral} k = \text{push-bit} n \text{ (numeral (Num.Bit0} k))}\>
by (simp add: numeral-eq-Suc mult-2-right) (simp add: numeral-Bit0)

**lemma** mask-eq-0-iff [simp]:
\(<\text{mask} n = 0 \leftrightarrow n = 0\>
by (cases n) (simp-all add: mask-Suc-double or-eq-0-iff)

**lemma** bit-horner-sum-bit-iff [bit-simps]:
\(<\text{bit} \text{(horner-sum} \text{ of-bool} 2 \text{ bs} \text{ n} \leftrightarrow \text{possible-bit} \text{ TYPE}(\text{'}a\text{)} \text{ n} \land \text{n} < \text{length} \text{ bs} \land \text{bs}!\text{n})\>
proof (induction bs arbitrary: n)
case Nil
then show \?case
by simp
next
case (Cons b bs)
show \?case
proof (cases n)
case 0
then show \?thesis
by (simp add: bit-0)
next
case (Suc m)
with bit-rec [of - n] Cons.prems Cons.IH [of m]
show \?thesis
by (simp add: bit-simps)
(auto simp add: possible-bit-less-imp bit-simps simp flip: bit-Suc)
qed

**lemma** horner-sum-bit-eq-take-bit:
\(<\text{horner-sum} \text{ of-bool} 2 \text{ (map} \text{ (bit} a) \text{ [0..<n}] = \text{take-bit} n \text{ a})\>
by (rule bit-eqI) (auto simp add: bit-simps)
THEORY “Bit-Operations”

lemma take-bit-horner-sum-bit-eq:
 \langle take-bit n (horner-sum of-bool 2 bs) = horner-sum of-bool 2 (take n bs) \rangle
 by (auto simp add: bit-eq-iff bit-take-bit-iff bit-horner-sum-bit-iff)

lemma take-bit-sum:
 \langle take-bit n a = (\sum k = 0..<n. push-bit k (of-bool (bit a k))) \rangle
 by (simp flip: horner-sum-bit-eq-take-bit add: horner-sum-eq-sum push-bit-eq-mult)

lemma set-bit-eq:
 \langle set-bit n a = a + of-bool (\neg bit a n) * 2 ^ n \rangle
 proof
 have \langle a AND of-bool (\neg bit a n) * 2 ^ n = 0 \rangle
 by (auto simp add: bit-eq-iff bit-simps)
 then show ?thesis
 by (auto simp add: bit-eq-iff bit-simps disjunctive-add-eq-or)
 qed

class ring-bit-operations = semiring-bit-operations + ring-parity +
 fixes not :: \langle \neg \rangle
 assumes not-eq-complement: \langle \neg a = - a - 1 \rangle
 begin

For the sake of code generation NOT is specified as definitional class operation. Note that NOT has no sensible definition for unlimited but only positive bit strings (type nat).

lemma bits-minus-1-mod-2-eq [simp]:
 \langle (- 1) mod 2 = 1 \rangle
 by (simp add: mod-2-eq-odd)

lemma minus-eq-not-plus-1:
 \langle a = NOT a + 1 \rangle
 using not-eq-complement [of a] by simp

lemma minus-eq-not-minus-1:
 \langle a = NOT (a - 1) \rangle
 using not-eq-complement [of \langle a - 1 \rangle] by simp (simp add: algebra-simps)

lemma not-rec:
 \langle NOT a = of-bool (even a) + 2 * NOT (a div 2) \rangle

lemma decr-eq-not-minus:
 \langle a - 1 = NOT (- a) \rangle
 using not-eq-complement [of \langle - a \rangle] by simp

lemma even-not-iff [simp]:
THEORY “Bit-Operations”

\(<\text{even } (\text{NOT } a) \iff \text{odd } a\>
\begin{align*}
\text{by (simp add: not-eq-complement)}
\end{align*}

\textbf{lemma bit-not-iff [bit-simps]:}
\text{\(<\text{bit } (\text{NOT } a) \iff \text{possible-bit } Type('a) n \land \neg \text{bit } a \ n\)>}
\textbf{proof (cases \text{\<\text{possible-bit Type('a) } n\>}'\text{\<\text{\>}})}
\text{\textbf{case False}}
\text{\quad then show \?thesis}
\begin{align*}
\text{by (auto dest: bit-imp-possible-bit)}
\end{align*}
\text{\textbf{next}}
\text{\textbf{case True}}
\text{\quad moreover have \(<\text{bit } (\text{NOT } a) \iff \neg \text{bit } a \ n\)>}
\text{\quad using \(<\text{possible-bit Type('a) } n\)>\textbf{ proof (induction } n \text{ arbitrary: } a)}
\text{\quad \textbf{case } 0}
\text{\quad \quad then show \?case}
\begin{align*}
\text{by (simp add: bit-0)}
\end{align*}
\text{\quad \textbf{next}}
\text{\quad \textbf{case } (Suc } n\text{)}
\text{\quad from Suc.prems Suc.IH [of \text{\<a div 2\>}'\text{\<\text{\>}}]}
\text{\quad \textbf{show } \?case}
\begin{align*}
\text{by (simp add: impossible-bit possible-bit-less-imp not-rec [of } a\text{] even-bit-succ-iff bit-double-iff flip: bit-Suc)}
\end{align*}
\textbf{qed}
\textbf{ultimately show \?thesis}
\begin{align*}
\text{by simp}
\end{align*}
\textbf{qed}

\textbf{lemma bit-not-exp-iff [bit-simps]:}
\text{\(<\text{bit } (\text{NOT } (2 ^ m)) \iff \text{possible-bit Type('a) } n \land n \neq m\)>}
\text{\begin{align*}
\text{by (auto simp add: bit-not-iff bit-exp-iff)}
\end{align*}}

\textbf{lemma bit-minus-iff [bit-simps]:}
\text{\(<\text{bit } (\text{a } \text{- } \text{a}) \iff \text{possible-bit Type('a) } n \land \neg \text{bit } (a - 1) \ n\)>}
\text{\begin{align*}
\text{by (simp add: minus-eq-not-minus-1 bit-not-iff)}
\end{align*}}

\textbf{lemma bit-minus-1-iff [simp]:}
\text{\(<\text{bit } (\text{a } \text{- } 1) \iff \text{possible-bit Type('a) } n\)>}
\text{\begin{align*}
\text{by (simp add: bit-minus-iff)}
\end{align*}}

\textbf{lemma bit-minus-exp-iff [bit-simps]:}
\text{\(<\text{bit } (\text{a } \text{- } (2 ^ m)) \iff \text{possible-bit Type('a) } n \land n \geq m\)>}
\text{\begin{align*}
\text{by (auto simp add: bit-simps simp flip: mask-eq-exp-minus-1)}
\end{align*}}

\textbf{lemma bit-minus-2-iff [simp]:}
\text{\(<\text{bit } (\text{a } \text{- } 2) \iff \text{possible-bit Type('a) } n \land n \geq 0\)>}
\text{\begin{align*}
\text{by (simp add: bit-minus-iff bit-1-iff)}
\end{align*}}

\textbf{lemma bit-decr-iff:}
\text{\(<\text{bit } (\text{a } \text{- } 1) \iff \text{possible-bit Type('a) } n \land \neg \text{bit } (\text{a } \text{- } 1) \ n\)>}
\begin{align*}
\end{align*}
by (simp add: decr-eq-not-minus bit-not-iff)

lemma bit-not-iff-eq:
\langle bit (NOT a) n \leftrightarrow 2 ^ n \neq 0 \land \neg bit a n \rangle
by (simp add: bit-simps possible-bit-def)

lemma not-one-eq [simp]:
\langle NOT 1 = - 2 \rangle
by (simp add: bit-eq-iff bit-not-iff) (simp add: bit-1-iff)

sublocale and: semilattice-neutr \langle (AND) \rangle (- 1)
by standard (rule bit-eqI, simp add: bit-and-iff)

sublocale bit: abstract-boolean-algebra \langle (AND) \rangle \langle (OR) \rangle NOT 0 (- 1)
by standard (auto simp add: bit-and-iff bit-or-iff bit-not-iff intro: bit-eqI)

sublocale bit: abstract-boolean-algebra-sym-diff \langle (AND) \rangle \langle (OR) \rangle NOT 0 (- 1)
\langle (XOR) \rangle
apply standard
apply (rule bit-eqI)
apply (auto simp add: bit-simps)
done

lemma and-eq-not-not-or:
\langle a AND b = NOT (NOT a OR NOT b) \rangle
by simp

lemma or-eq-not-not-and:
\langle a OR b = NOT (NOT a AND NOT b) \rangle
by simp

lemma not-add-distrib:
\langle NOT (a + b) = NOT a - b \rangle
by (simp add: not-eq-complement algebra-simps)

lemma not-diff-distrib:
\langle NOT (a - b) = NOT a + b \rangle
using not-add-distrib [of a (- b)] by simp

lemma and-eq-minus-1-iff:
\langle a AND b = - 1 \leftrightarrow a = - 1 \land b = - 1 \rangle
by (auto simp: bit-eq-iff bit-simps)

lemma disjunctive-and-not-eq-xor:
\langle a AND NOT b = a XOR b \rangle if \langle NOT a AND b = 0 \rangle
using that by (auto simp add: bit-eq-iff bit-simps)

lemma disjunctive-diff-eq-and-not:
\langle a - b = a AND NOT b \rangle if \langle NOT a AND b = 0 \rangle
proof  
from that have \( \text{NOT} \ a + b = \text{NOT} \ a \ OR \ b \)  
  by (rule disjunctive-add-eq-or)  
then have \( \text{NOT} \ (\text{NOT} \ a + b) = \text{NOT} \ (\text{NOT} \ a \ OR \ b) \)  
  by simp  
then show ?thesis  
  by (simp add: not-add-distrib)  
qed

lemma disjunctive-diff-eq-xor:  
\( \text{a AND NOT b} = \text{a XOR b} \) if \( \text{NOT a AND b} = 0 \)  
using that by (simp add: disjunctive-and-not-eq-xor disjunctive-diff-eq-and-not)

lemma push-bit-minus:  
\( \text{push-bit n (} - a \text{)} = - \text{push-bit n a} \)  
by (simp add: push-bit-eq-mult)

lemma take-bit-not-take-bit:  
\( \text{take-bit n (} \text{NOT (take-bit n a)} \text{)} = \text{take-bit n (} \text{NOT a} \text{)} \)  
by (auto simp add: bit-eq-iff bit-take-bit-iff bit-not-iff)

lemma take-bit-not-iff:  
\( \text{take-bit n (} \text{NOT a} \text{)} = \text{take-bit n (} \text{NOT b} \text{)} \iff \text{take-bit n a} = \text{take-bit n b} \)  
by (auto simp add: bit-eq-iff bit-sims)

proof  
have \( \text{NOT (} \text{mask n} \text{ AND take-bit n a} \equiv 0 \)  
  by (simp add: bit-eq-iff bit-mask-iff bit-take-bit-iff conj-commute)  
moreover have \( \text{take-bit n (} \text{NOT a} \text{)} = \text{mask n AND NOT (} \text{take-bit n a} \)  
  by (auto simp add: bit-eq-iff bit-sims)  
ultimately show ?thesis  
  by (simp add: disjunctive-diff-eq-and-not)  
qed

lemma mask-eq-take-bit-minus-one:  
\( \text{mask n = take-bit n (} - 1 \text{)} \)  
by (simp add: bit-eq-iff bit-mask-iff bit-take-bit-iff conj-commute)

lemma take-bit-minus-one-eq-mask [simp]:  
\( \text{take-bit n (} - 1 \text{)} = \text{mask n} \)  
by (simp add: mask-eq-take-bit-minus-one)

lemma minus-exp-eq-not-mask:  
\( - (2 \ ^{n}) = \text{NOT} \ (\text{mask n}) \)  
by (rule bit-eqI) (simp add: bit-minus-iff bit-not-iff flip: mask-eq-exp-minus-1)

lemma push-bit-minus-one-eq-not-mask [simp]:
\[ \text{push-bit } n \ (\ -1) = \text{NOT} \ (\text{mask } n) \]
by (simp add: push-bit-eq-mult minus-exp-eq-not-mask)

\textbf{lemma} take-bit-not-mask-eq-0:
\[ \langle \text{take-bit } m \ (\text{NOT} \ (\text{mask } n)) = 0 \rangle \text{ if } \langle n \geq m \rangle \]
by (rule bit-eqI) (use that in \langle simp add: bit-take-bit-iff bit-not-iff bit-mask-iff \rangle)

\textbf{lemma} unset-bit-eq-and-not:
\[ \langle \text{unset-bit } n \ a = a \ AND \ NOT \ (\text{push-bit } n \ 1) \rangle \]
by (rule bit-eqI) (auto simp add: bit-simps)

\textbf{lemma} push-bit-Suc-minus-numeral [simp]:
\[ \langle \text{push-bit } (\text{Suc } n) \ (\ -\text{numeral } k) = \text{push-bit } n \ (\ -\text{numeral} \ (\text{Num.Bit0 } k)) \rangle \]
apply (simp only: numeral-Bit0)
apply simp
apply (simp only: numeral-mult mult-2-right numeral-add)
done

\textbf{lemma} push-bit-minus-numeral [simp]:
\[ \langle \text{push-bit } (\text{numeral } k) \ (\ -\text{numeral } k) = \text{push-bit } (\text{pred-numeral } l) \ (\ -\text{numeral} \ (\text{Num.Bit0 } k)) \rangle \]
by (simp only: numeral-eq-Suc push-bit-Suc-minus-numeral)

\textbf{lemma} take-bit-Suc-minus-1-eq:
\[ \langle \text{take-bit } (\text{Suc } n) \ (\ -1) = 2 \ ^ \text{Suc } n \ -1 \rangle \]
by (simp add: mask-eq-exp-minus-1)

\textbf{lemma} take-bit-numeral-minus-1-eq:
\[ \langle \text{take-bit } (\text{numeral } k) \ (\ -1) = 2 \ ^ \text{numeral } k \ -1 \rangle \]
by (simp add: mask-eq-exp-minus-1)

\textbf{lemma} push-bit-mask-eq:
\[ \langle \text{push-bit } m \ (\text{mask } n) = \text{mask } (\text{m } + \text{n}) \ AND \ NOT \ (\text{mask } m) \rangle \]
by (rule bit-eqI) (auto simp add: bit-simps not-less possible-bit-less-imp)

\textbf{lemma} slice-eq-mask:
\[ \langle \text{push-bit } n \ (\text{take-bit } m \ (\text{drop-bit } n \ a)) = a \ AND \ \text{mask } (m \ + \ n) \ AND \ NOT \ (\text{mask } n) \rangle \]
by (rule bit-eqI) (auto simp add: bit-simps)

\textbf{lemma} push-bit-numeral-minus-1 [simp]:
\[ \langle \text{push-bit } (\text{numeral } n) \ (\ -1) = - (2 \ ^ \text{numeral } n) \rangle \]
by (simp add: push-bit-eq-mult)

\textbf{lemma} unset-bit-eq:
\[ \langle \text{unset-bit } n \ a = a - \text{of-bool} \ (\text{bit } a \ n) \ast 2 \ ^ \text{n} \rangle \]
proof -
have \[ \langle \text{NOT } a \ AND \ \text{of-bool} \ (\text{bit } a \ n) \ast 2 \ ^ \text{n} = 0 \rangle \]
by (auto simp add: bit-eq-iff bit-simps)
moreover have \( \text{unset-bit } n \ a = a \ \text{AND NOT} \ (\text{of-bool} \ (\text{bit} \ a \ n) \ * \ 2 \ ^n) \)
by (auto simp add: bit-eq-iff bit-simps)
ultimately show \(?thesis
by (simp add: disjunctive-diff-eq-and-not)
qed

end

68.3 Common algebraic structure

class linordered-euclidean-semiring-bit-operations =
linordered-euclidean-semiring + semiring-bit-operations
begin

lemma possible-bit [simp]:
\( \text{possible-bit TYPE' } (\'a) \ n \)
by (simp add: possible-bit-def)

lemma take-bit-of-exp [simp]:
\( \text{take-bit } m \ (2 \ ^n) = \text{of-bool} \ (n < m) \ * \ 2 \ ^n \)
by (simp add: take-bit-eq-mod exp-mod-exp)

lemma take-bit-of-2 [simp]:
\( \text{take-bit } n \ 2 = \text{of-bool} \ (2 \leq \ n) \ * \ 2 \)
using take-bit-of-exp [of \ n \ 1] by simp

lemma push-bit-eq-0-iff [simp]:
\( \text{push-bit } n \ a = 0 \longleftrightarrow a = 0 \)
by (simp add: push-bit-eq-mult)

lemma take-bit-add:
\( \text{take-bit } n \ (\text{take-bit } n \ a + \text{take-bit } n \ b) = \text{take-bit } n \ (a + b) \)
by (simp add: take-bit-eq-mod simps)

lemma take-bit-of-1-eq-0-iff [simp]:
\( \text{take-bit } n \ 1 = 0 \longleftrightarrow n = 0 \)
by (simp add: take-bit-eq-mod)

lemma drop-bit-Suc-bit0 [simp]:
\( \text{drop-bit } (\text{Suc} \ n) \ (\text{numeral} \ (\text{Num.Bit0} \ k)) = \text{drop-bit } n \ (\text{numeral} \ k) \)
by (simp add: drop-bit-Suc numeral-Bit0-div-2)

lemma drop-bit-Suc-bit1 [simp]:
\( \text{drop-bit } (\text{Suc} \ n) \ (\text{numeral} \ (\text{Num.Bit1} \ k)) = \text{drop-bit } n \ (\text{numeral} \ k) \)
by (simp add: drop-bit-Suc numeral-Bit1-div-2)

lemma drop-bit-numeral-bit0 [simp]:
\( \text{drop-bit } (\text{numeral} \ l) \ (\text{numeral} \ (\text{Num.Bit0} \ k)) = \text{drop-bit } (\text{pred-numeral} \ l) \ (\text{numeral} \ k) \)
by (simp add: drop-bit-rec numeral-Bit0-div-2)

lemma drop-bit-numeral-bit1 [simp]:
\langle drop-bit (numeral l) (numeral (Num.Bit1 k)) = drop-bit (pred-numeral l) (numeral k) \rangle
by (simp add: drop-bit-rec numeral-Bit1-div-2)

lemma take-bit-Suc-1 [simp]:
\langle take-bit (Suc n) 1 = 1 \rangle
by (simp add: take-bit-Suc)

lemma take-bit-Suc-bit0:
\langle take-bit (Suc n) (numeral (Num.Bit0 k)) = take-bit n (numeral k) * 2 \rangle
by (simp add: take-bit-Suc numeral-Bit0-div-2 mod-2-eq-odd)

lemma take-bit-Suc-bit1:
\langle take-bit (Suc n) (numeral (Num.Bit1 k)) = take-bit n (numeral k) * 2 + 1 \rangle
by (simp add: take-bit-Suc numeral-Bit1-div-2)

lemma bit-of-nat-iff-bit [bit-simps]:
\langle bit (of-nat m) n \iff bit m n \rangle
proof –
  have \langle even (m div 2 ^ n) \iff even (of-nat (m div 2 ^ n)) \rangle
    by simp
  also have \langle of-nat (m div 2 ^ n) = of-nat m div of-nat (2 ^ n) \rangle
    by (simp add: of-nat-div)
finally show ?thesis
  by (simp add: bit-iff-odd semiring-bits-class.bit-iff-odd)
qed

lemma drop-bit-mask-eq:
\langle drop-bit m (mask n) = mask (n - m) \rangle
by (rule bit-eqI) (auto simp add: bit-simps possible-bit-def)

lemma bit-push-bit-iff':
theory "Bit-Operations"

lemma mask-half: 
  \( \text{mask } n \div 2 = \text{mask } (n - 1) \)
  by (cases n) (simp-all add: mask-Suc-double one-or-eq)

lemma take-bit-Suc-from-most: 
  \( \text{take-bit } (\text{Suc } n) \ a = 2^n \ast \text{of-bool } (\text{bit } a \ n) + \text{take-bit } n \ a \)
  using mod-mult2-eq \([\text{of } a \ (2^n \ast 2)]\)
  by (simp only: take-bit-eq-mod power-Suc2)
  (simp-all add: bit-iff-odd odd-iff-mod-2-eq-one)

lemma take-bit-nonnegative [simp]: 
  \( \text{0} \leq \text{take-bit } n \ a \)
  using horner-sum-nonnegative by (simp flip: horner-sum-bit-eq-take-bit)

lemma not-take-bit-negative [simp]:
  \( \neg \text{take-bit } n \ a < 0 \)
  by (simp add: not-less)

lemma bit-imp-take-bit-positive:
  \( \text{0} < \text{take-bit } m \ a \) if \( \text{n} < m \) and \( \text{bit } a \ n \)
  proof (rule ccontr)
    assume \( \neg \text{0} < \text{take-bit } m \ a \)
    then have \( \text{take-bit } m \ a = 0 \)
      by (auto simp add: not-less intro: order-antisym)
    then have \( \text{bit } (\text{take-bit } m \ a) \ n = \text{bit } 0 \ n \)
      by simp
    with that show False
      by (simp add: bit-take-bit-iff)
  qed

lemma take-bit-mult:
  \( \text{take-bit } n \ (\text{take-bit } n \ a \ast \text{take-bit } n \ b) = \text{take-bit } n \ (a \ast b) \)
  by (simp add: take-bit-eq-mod mod-mult-eq)

lemma drop-bit-push-bit:
  \( \text{drop-bit } m \ (\text{push-bit } n \ a) = \text{drop-bit } (m - n) \ (\text{push-bit } (n - m) \ a) \)
  by (cases \( m \leq n \))
    (auto simp add: mult.left-commute \([\text{of } (2^n \ast n)]\) mult.commute \([\text{of } (2^n \ast n)]\) mult.assoc
      mult.commute \([\text{of } a]\) drop-bit-eq-div push-bit-eq-mult not-le power-add Orderings.not-le dest!: leSuc-ex less-imp-Suc-add)

end
68.4 Instance \( \text{int} \)

locale fold2-bit-int =
  fixes \( f : (\text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}) \)
begin

context begin

function \( F :: (\text{int} \Rightarrow \text{int} \Rightarrow \text{int}) \)
  where \( F k l = \) (if \( k \in \{0, -1\} \wedge l \in \{0, -1\} \)
      then \( -\text{of-bool} (f (\text{odd} k) (\text{odd} l)) \)
      else \( \text{of-bool} (f (\text{odd} k) (\text{odd} l)) + 2 \ast (F (k \text{ div} 2) (l \text{ div} 2)) \))
  by auto

private termination proof (relation \( \text{measure} (\lambda (k, l). \text{nat} (|k| + |l|)) \))
  have less-eq: \( |k \text{ div} 2| \leq |k| \) for \( k :: \text{int} \)
      by (cases k) (simp-all add: divide-int-def nat-add-distrib)
  then have less: \( |k \text{ div} 2| < |k| \) if \( k \notin \{0, -1\} \) for \( k :: \text{int} \)
    using that by (auto simp add: less-le [of k])
  show \( \text{of} (\text{measure} (\lambda (k, l). \text{nat} (|k| + |l|))) \)
    by simp
  show \( ((k \text{ div} 2, l \text{ div} 2), k, l) \in \text{measure} (\lambda (k, l). \text{nat} (|k| + |l|)) \)
    if \( \neg (k \in \{0, -1\} \wedge l \in \{0, -1\}) \) for \( k \) \( l \)
  proof
    from that have \( \ast : k \notin \{0, -1\} \lor l \notin \{0, -1\} \);
      by simp
    then have \( 0 < |k| + |l| \)
      by auto
    moreover from \( \ast \) have \( |k \text{ div} 2| + |l \text{ div} 2| < |k| + |l| \)
      proof
        assume \( k \notin \{0, -1\} \)
        then have \( |k \text{ div} 2| < |k| \)
          by (rule less)
        with less-eq \( |l| \) show \( \ast \)thesis
          by auto
      next
        assume \( l \notin \{0, -1\} \)
        then have \( |l \text{ div} 2| < |l| \)
          by (rule less)
        with less-eq \( |k| \) show \( \ast \)thesis
          by auto
      qed
    ultimately show \( \ast \)thesis
      by (simp only: in-measure split-def fst-conv snd-conv nat-mono-iff)
    qed
  qed

declare \( F.\text{.simps} \ [\text{simp del}] \)
lemma rec:
\[ F \, k \, l = \text{of-bool} (f (\text{odd} \, k) (\text{odd} \, l)) + 2 \ast (F \, (k \, \text{div} \, 2) \, (l \, \text{div} \, 2)) \]
for \( k \, l :: \text{int} \)

proof (cases \( k \in \{0, \, -1\} \land l \in \{0, \, -1\} \))
  case True
  then show \( ?\text{thesis} \)
    by (auto simp add: F.simps [of 0] F.simps [of \(-1\)])
next
  case False
  then show \( ?\text{thesis} \)
    by (auto simp add: ac-simps F.simps [of k l])

qed

lemma bit-iff:
\[ \text{bit} (F \, k \, l) \, n \longleftrightarrow f (\text{bit} \, k \, n) \, (\text{bit} \, l \, n) \]
for \( k \, l :: \text{int} \)

proof (induction \( n \) arbitrary: \( k \, l \))
  case 0
  then show \( ?\text{case} \)
    by (simp add: rec [of k l] bit-0)
next
  case (Suc \( n \))
  then show \( ?\text{case} \)
    by (simp add: rec [of k l] bit-Suc)

qed

end

end

instantiation \( \text{int} :: \text{ring-bit-operations} \)
begin

definition not-int :: \( \langle \text{int} \Rightarrow \text{int} \rangle \)
  where \( \langle \text{not-int} \, k \rangle = - k - 1 \)

global-interpretation and-int: fold2-bit-int \( \langle \wedge \rangle \)
  defines and-int = and-int.F .

global-interpretation or-int: fold2-bit-int \( \langle \lor \rangle \)
  defines or-int = or-int.F .

global-interpretation xor-int: fold2-bit-int \( \langle \neq \rangle \)
  defines xor-int = xor-int.F .

definition mask-int :: \( \langle \text{nat} \Rightarrow \text{int} \rangle \)
  where \( \langle \text{mask} \, n \rangle = (2 :: \text{int}) \, ^n - 1 \)

definition push-bit-int :: \( \langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \rangle \)
  where \( \langle \text{push-bit-int} \, n \, k \rangle = k \ast 2 \, ^n \)
definition drop-bit-int :: (nat ⇒ int ⇒ int)
  where (drop-bit-int n k = k div 2 ^ n)

definition take-bit-int :: (nat ⇒ int ⇒ int)
  where (take-bit-int n k = k mod 2 ^ n)

definition set-bit-int :: (nat ⇒ int ⇒ int)
  where (set-bit n k = k OR push-bit n 1) for k :: int

definition unset-bit-int :: (nat ⇒ int ⇒ int)
  where (unset-bit n k = k AND NOT (push-bit n 1)) for k :: int

definition flip-bit-int :: (nat ⇒ int ⇒ int)
  where (flip-bit n k = k XOR push-bit n 1) for k :: int

lemma not-int-div-2:
  (NOT k div 2 = NOT (k div 2)) for k :: int
  by (simp add: not-int-def)

lemma bit-not-int-iff:
  (bit (NOT k) n ←→ ¬ bit k n) for k :: int
proof (rule sym, induction n arbitrary: k)
  case 0
  then show ?case
    by (simp add: bit-0 not-int-def)
next
  case (Suc n)
  then show ?case
    by (simp add: bit-Suc not-int-div-2)
qed

instance proof
  fix k l :: int and m n :: nat
  show (unset-bit n k = (k OR push-bit n 1) XOR push-bit n 1)
    by (rule bit-eqI)
      (auto simp add: unset-bit-int-def
        and-int.bit-iff or-int.bit-iff xor-int.bit-iff bit-not-int-iff push-bit-int-def bit-simps)
qed (fact and-int.rec or-int.rec xor-int.rec mask-int-def set-bit-int-def flip-bit-int-def
             push-bit-int-def drop-bit-int-def take-bit-int-def not-int-def)+

end

instance int :: linordered-euclidean-semiring-bit-operations ..

context ring-bit-operations
begin

lemma even-of-int-iff:

THEORY “Bit-Operations”

\[
\begin{align*}
\text{even (of-int } k\text{) } & \iff \text{even } k \\
\text{by } & (\text{induction } k \text{ rule: } \text{int-bit-induct}) \text{ simp-all}
\end{align*}
\]

lemma bit-of-int-iff [bit-simps]:
\[
\text{\langle bit (of-int } k\text{) } n \iff \text{possible-bit TYPE}('a)\text{ } n \land \text{bit } k\text{ } n\rangle
\]
proof (cases \langle \text{possible-bit TYPE}('a)\text{ } n\rangle)
\[
\text{case False}\text{ Then show } \text{thesis}
\quad \text{by (simp add: impossible-bit)}
\]
next
\[
\text{case True}\text{ Then have } \langle \text{bit (of-int } k\text{) } n \iff \text{bit } k\text{ } n\rangle
\]
proof (induction \textit{k arbitrary: } n \text{ rule: } \text{int-bit-induct})
\[
\text{case zero}\text{ Then show } \text{?case}
\quad \text{by simp}
\]
next
\[
\text{case minus}\text{ Then show } \text{?case}
\quad \text{by simp}
\]
next
\[
\text{case (even } k\text{) Then show } \text{?case}
\quad \text{using bit-double-iff [of } \langle \text{of-int } k\text{) } n\rangle \text{ Bit-Operations.bit-double-iff [of } k\text{ } n\rangle
\quad \text{by (cases } n\text{) (auto simp add: ac-simps possible-bit-def dest: mult-not-zero)}
\]
next
\[
\text{case (odd } k\text{) Then show } \text{?case}
\quad \text{using bit-double-iff [of } \langle \text{of-int } k\text{) } n\rangle
\quad \text{by (cases } n\text{)}
\quad \text{(auto simp add: ac-simps bit-double-iff even-bit-succ-iff Bit-Operations.bit-0}
\quad \text{Bit-Operations.bit-Suc}
\quad \text{possible-bit-def dest: mult-not-zero)}
\]
\[
\text{qed}
\]
with True show \text{thesis}
\quad \text{by simp}
\]
\text{qed}

lemma push-bit-of-int:
\[
\langle \text{push-bit } n \text{ (of-int } k\text{)} = \text{of-int (push-bit } n\text{ } k)\rangle
\quad \text{by (simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult)}
\]

lemma of-int-push-bit:
\[
\langle \text{of-int (push-bit } n\text{ } k) = \text{push-bit } n \text{ (of-int } k\text{)}\rangle
\quad \text{by (simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult)}
\]

lemma take-bit-of-int:
\[
\langle \text{take-bit } n \text{ (of-int } k\text{)} = \text{of-int (take-bit } n\text{ } k)\rangle
\quad \text{by (rule bit-eql) (simp add: bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-int-iff)}
\]

lemma of-int-take-bit:
  ‹of-int (take-bit n k) = take-bit n (of-int k)›
  by (rule bit-eqI) (simp add: bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-int-iff)

lemma of-int-not-eq:
  ‹of-int (NOT k) = NOT (of-int k)›
  by (rule bit-eqI) (simp add: bit-not-iff Bit-Operations.bit-not-iff bit-of-int-iff)

lemma of-int-not-numeral:
  ‹of-int (NOT (numeral k)) = NOT (numeral k)›
  by (simp add: local.of-int-not-eq)

lemma of-int-and-eq:
  ‹of-int (k AND l) = of-int k AND of-int l›

lemma of-int-or-eq:
  ‹of-int (k OR l) = of-int k OR of-int l›
  by (rule bit-eqI) (simp add: bit-of-int-iff bit-or-iff Bit-Operations.bit-or-iff)

lemma of-int-xor-eq:
  ‹of-int (k XOR l) = of-int k XOR of-int l›
  by (rule bit-eqI) (simp add: bit-of-int-iff bit-xor-iff Bit-Operations.bit-xor-iff)

lemma of-int-mask-eq:
  ‹of-int (mask n) = mask n›
  by (induction n) (simp-all add: mask-Suc-double Bit-Operations.mask-Suc-double of-int-or-eq)

end

lemma take-bit-int-less-exp [simp]:
  ‹take-bit n k < 2 ^ n› for k :: int
  by (simp add: take-bit-eq-mod)

lemma take-bit-int-eq-self-iff:
  ‹take-bit n k = k ↔ 0 ≤ k ∧ k < 2 ^ n› (is ‹?P ↔ ?Q›)
  for k :: int
  proof
    assume ?P
    moreover note take-bit-int-less-exp [of n k] take-bit-nonnegative [of n k]
    ultimately show ?Q
      by simp
  next
    assume ?Q
    then show ?P
      by (simp add: take-bit-eq-mod)
  qed
lemma `take-bit-int-eq-self`: `\langle \text{take-bit } n \ k = k \rangle \text{ if } \langle 0 \leq k \ \land \ k < 2 \ ^n \rangle \text{ for } k :: \text{int}
``
using that by `(simp add: `take-bit-int-eq-self-iff`)

lemma `mask-nonnegative-int` [simp]:
`\langle \text{mask } n \geq (0::\text{int}) \rangle
by (simp add: `mask-eq-exp-minus-1 \ add-le-imp-le-diff`)

lemma `not-mask-negative-int` [simp]:
`\langle \neg \text{mask } n \ < (0::\text{int}) \rangle
by (simp add: `not-less`)

lemma `not-nonnegative-int-iff` [simp]:
`\langle \neg k \geq 0 \iff k < 0 \rangle \text{ for } k :: \text{int}
by (simp add: `not-int-def`)

lemma `not-negative-int-iff` [simp]:
`\langle \neg k \ < 0 \iff k \geq 0 \rangle \text{ for } k :: \text{int}
by (subst Not-eq-iff [symmetric]) (simp add: `not-less not-le`)

lemma `and-nonnegative-int-iff` [simp]:
`\langle k \ \land \ l \geq 0 \iff k \geq 0 \lor l \geq 0 \rangle \text{ for } k \ l :: \text{int}
proof (induction k arbitrary: l rule: `int-bit-induct`)
case `zero
then show ?case
  by simp
next
case `minus
then show ?case
  by simp
next
case (even k)
then show ?case
  using and-int.rec [of `k \ * \ 2` \ `l`]
  by (simp add: `pos-imp-zdiv-nonneg-iff zero-le-mult-iff`)
next
case (odd k)
from `odd` have `0 \leq k \ \land \ l \ \text{div} \ 2 \iff 0 \leq k \ \lor \ 0 \leq l \ \text{div} \ 2`
  by simp
then have `0 \leq (1 \ + \ k \ * \ 2) \ \text{div} \ 2 \ \land \ l \ \text{div} \ 2 \iff 0 \leq (1 \ + \ k \ * \ 2) \ \text{div} \ 2 \ \lor
  0 \leq l \ \text{div} \ 2`
  by simp
with and-int.rec [of `(1 \ + \ k \ * \ 2)` \ `l`]
show ?case
  by (auto simp add: `zero-le-mult-iff not-le`)
qed

lemma `and-negative-int-iff` [simp]:
\[ \langle k \text{ AND } l < 0 \iff k < 0 \land l < 0 \rangle \text{ for } k l :: \text{int} \]

by (subst Not-eq-iff [symmetric]) (simp add: not-less)

**lemma** and-less-eq:
\[ \langle k \text{ AND } l \leq k \rangle \text{ if } \langle l < 0 \rangle \text{ for } k l :: \text{int} \]

**using** that **proof** (induction \( k \) arbitrary: \( l \) rule: int-bit-induct)

**case** zero
then show ?case
  by simp
next
case minus
then show ?case
  by simp
next
case (even \( k \))
from even.IH [of \( \langle l \text{ div } 2 \rangle \)] even.hyps even.prems
show ?case
  by (simp add: and-int.rec [of - \( l \)])
next
case (odd \( k \))
from odd.IH [of \( \langle l \text{ div } 2 \rangle \)] odd.hyps odd.prems
show ?case
  by (simp add: and-int.rec [of - \( l \)])

**qed**

**lemma** or-nonnegative-int-iff [simp]:
\[ \langle k \text{ OR } l \geq 0 \iff k \geq 0 \land l \geq 0 \rangle \text{ for } k l :: \text{int} \]

by (simp only: or-eq-not-not-and not-nonnegative-int-iff) simp

**lemma** or-negative-int-iff [simp]:
\[ \langle k \text{ OR } l < 0 \iff k < 0 \lor l < 0 \rangle \text{ for } k l :: \text{int} \]

by (subst Not-eq-iff [symmetric]) (simp add: not-less)

**lemma** or-greater-eq:
\[ \langle k \text{ OR } l \geq k \rangle \text{ if } \langle l \geq 0 \rangle \text{ for } k l :: \text{int} \]

**using** that **proof** (induction \( k \) arbitrary: \( l \) rule: int-bit-induct)

**case** zero
then show ?case
  by simp
next
case minus
then show ?case
  by simp
next
case (even \( k \))
from even.IH [of \( \langle l \text{ div } 2 \rangle \)] even.hyps even.prems
show ?case
  by (simp add: or-int.rec [of - \( l \)])
next
case (odd k)
from odd.IH [of \langle l \div 2 \rangle] odd.hyps odd.prems
show ?case
  by (simp add: or-int.rec [of - l])
qed

lemma xor-nonnegative-int-iff [simp]:
\langle k \text{ XOR } l \geq 0 \Longleftrightarrow (k \geq 0 \Longleftrightarrow l \geq 0) \rangle \text{ for } k l :: \text{int}
by (simp only: bitxor_def or-nonnegative-int-iff)

lemma xor-negative-int-iff [simp]:
\langle k \text{ XOR } l < 0 \Longleftrightarrow (k < 0 \neq (l < 0)) \rangle \text{ for } k l :: \text{int}
by (subst Not-eq-iff [symmetric]) (auto simp add: not-less)

lemma OR-upper:
\langle x \text{ OR } y < 2 ^ n \rangle \text{ if } \langle 0 \leq x \rangle \langle x < 2 ^ n \rangle \langle y < 2 ^ n \rangle \text{ for } x y :: \text{int}
using that proof (induction x arbitrary: y n rule: int-bit-induct)
case zero
  then show ?case
  by simp
next
case minus
  then show ?case
  by simp
next
case (even x)
from even.IH [of \langle n - 1 \rangle \langle y \div 2 \rangle] even.prems even.hyps
show ?case
  by (cases n) (auto simp add: or-int.rec [of -$2^2$] elim: oddE)
next
case (odd x)
from odd.IH [of \langle n - 1 \rangle \langle y \div 2 \rangle] odd.prems odd.hyps
show ?case
  by (cases n) (auto simp add: or-int.rec [of $1 + -2^2$], linarith)
qed

lemma XOR-upper:
\langle x \text{ XOR } y < 2 ^ n \rangle \text{ if } \langle 0 \leq x \rangle \langle x < 2 ^ n \rangle \langle y < 2 ^ n \rangle \text{ for } x y :: \text{int}
using that proof (induction x arbitrary: y n rule: int-bit-induct)
case zero
  then show ?case
  by simp
next
case minus
  then show ?case
  by simp
next
case (even x)
from even.IH [of \langle n - 1 \rangle \langle y \div 2 \rangle] even.prems even.hyps
show ?case  
  by (cases n) (auto simp add: xor-int-rec [of `- 2`] elim: oddE) 
next  
  case (odd x)  
  from odd.IH [of `n - 1` `y div 2`] odd.prems odd.hyps  
  show ?case  
    by (cases n) (auto simp add: xor-int-rec [of `1 + `- 2`]) 
qed 

lemma AND-lower [simp]:  
  `0 ≤ x AND y` if `0 ≤ x` for x y :: int  
using that by simp 

lemma OR-lower [simp]:  
  `0 ≤ x OR y` if `0 ≤ x` `0 ≤ y` for x y :: int  
using that by simp 

lemma XOR-lower [simp]:  
  `0 ≤ x XOR y` if `0 ≤ x` `0 ≤ y` for x y :: int  
using that by simp 

lemma AND-upper1 [simp]:  
  `x AND y ≤ x` if `0 ≤ x` for x y :: int  
using that proof (induction x arbitrary: y rule: int-bit-induct)  
  case (odd k)  
  then have `k AND y div 2 ≤ k`  
    by simp  
  then show ?case  
    by (simp add: and-int-rec [of `- 2`]) 
qed (simp add: and-int-rec [of `- 2`]) 

lemma AND-upper1' [simp]:  
  `y AND x ≤ x` if `0 ≤ y` for x y z :: int  
using - `y ≤ z` by (rule order-le-less-trans) (use `0 ≤ y` in simp) 

lemma AND-upper1'' [simp]:  
  `y AND x < z` if `0 ≤ y` `y < z` for x y z :: int  
using - `y < z` by (rule order-le-less-trans) (use `0 ≤ y` in simp) 

lemma AND-upper2 [simp]:  
  `x AND y ≤ y` if `0 ≤ y` for x y :: int  
using that AND-upper1 [of y x] by (simp add: ac-simps) 

lemma AND-upper2' [simp]:  
  `x AND y ≤ z` if `0 ≤ y` `y ≤ z` for x y z :: int  
using that AND-upper1' [of y z x] by (simp add: ac-simps) 

lemma AND-upper2'' [simp]:  
  `x AND y < z` if `0 ≤ y` `y < z` for x y :: int
using that AND-upper1'' [of y z x] by (simp add: ac-simps)

lemma plus-and-or:
\( (x \text{ AND } y) + (x \text{ OR } y) = x + y \) for \( x, y :: \text{int} \)

proof (induction \( x \) arbitrary; \( y \) rule: int-bit-induct)
  case zero
  then show ?case by simp
next
  case minus
  then show ?case by simp
next
  case (even \( x \))
  from even.IH \[ of \( \langle y \text{ div } 2 \rangle \] 
  show ?case
    by (auto simp add: and-int.rec \[ of - y \] or-int.rec \[ of - y \] elim: oddE)
next
  case (odd \( x \))
  from odd.IH \[ of \( \langle y \text{ div } 2 \rangle \] 
  show ?case
    by (auto simp add: and-int.rec \[ of - y \] or-int.rec \[ of - y \] elim: oddE)
qed

lemma push-bit-minus-one:
\( \langle \text{push-bit } n (-1 :: \text{int}) = - (2 \uparrow n) \rangle \)
by (simp add: push-bit-eq-mult)

lemma minus-1-div-exp-eq-int:
\( \langle -1 \text{ div } (2 :: \text{int}) \uparrow n = -1 \rangle \)
by (induction \( n \)) (use div-exp-eq [symmetric, of \( -1 :: \text{int} \) \( 1 \)]) in \( \langle \text{simp-all add: ac-simps} \rangle \)

lemma drop-bit-minus-one [simp]:
\( \langle \text{drop-bit } n (-1 :: \text{int}) = -1 \rangle \)
by (simp add: drop-bit-eq-div minus-1-div-exp-eq-int)

lemma take-bit-minus:
\( \langle \text{take-bit } n (-\text{take-bit } n \ k) = \text{take-bit } n \ (-k) \rangle \)
for \( k :: \text{int} \)
by (simp add: take-bit-eq-mod mod-minus-eq)

lemma take-bit-diff:
\( \langle \text{take-bit } n (\text{take-bit } n \ k - \text{take-bit } n \ l) = \text{take-bit } n \ (k - l) \rangle \)
for \( k,l :: \text{int} \)
by (simp add: take-bit-eq-mod mod-diff-eq)

lemma (in ring-1) of-nat-nat-take-bit-eq [simp]:
\( \langle \text{of-nat } (\text{nat} (\text{take-bit } n \ k)) = \text{of-int } (\text{take-bit } n \ k) \rangle \)
by simp

lemma take-bit-minus-small-eq:
\(<\text{take-bit } n (\sim k) = 2^n - k >\) if \(<0 < k < 2^n >\) for \(<k :: \text{int}>\)

proof –
  define \(<m = \text{nat } k>\)
  with that have \(<k = \text{int } m\) and \(<0 < m < 2^n >\)
  by simp-all
  have \(<(2^n - m) \mod 2^n = 2^n - m >\)
    using \(<0 < m >\) by simp
  then have \(<\text{int } ((2^n - m) \mod 2^n) = \text{int } (2^n - m) >\)
    by simp
  then have \(<(2^n - \text{int } m) \mod 2^n = 2^n - \text{int } m >\)
    using \(<m \leq 2^n >\) by (simp only: of-nat-mod of-nat-diff) simp
  with \(<k = \text{int } m>\) have \(<(2^n - k) \mod 2^n = 2^n - k >\)
    by simp
  then show \(<?\text{thesis}>\)
    by (simp add: take-bit-eq-mod)
qed

lemma push-bit-nonnegative-int-iff [simp]:
\(<\text{push-bit } n k \geq 0 <\leftrightarrow k \geq 0 \text{ for } k :: \text{int}>\)
by (simp add: push-bit-eq-mult zero-le-mult-iff power-le-zero-eq)

lemma push-bit-negative-int-iff [simp]:
\(<\text{push-bit } n k < 0 <\leftrightarrow k < 0 \text{ for } k :: \text{int}>\)
by (subst Not-eq-iff [symmetric]) (simp add: not-less)

lemma drop-bit-nonnegative-int-iff [simp]:
\(<\text{drop-bit } n k \geq 0 <\leftrightarrow k \geq 0 \text{ for } k :: \text{int}>\)
by (auto simp add: drop-bit-Suc drop-bit-half)

lemma drop-bit-negative-int-iff [simp]:
\(<\text{drop-bit } n k < 0 <\leftrightarrow k < 0 \text{ for } k :: \text{int}>\)
by (subst Not-eq-iff [symmetric]) (simp add: not-less)

lemma set-bit-nonnegative-int-iff [simp]:
\(<\text{set-bit } n k \geq 0 <\leftrightarrow k \geq 0 \text{ for } k :: \text{int}>\)
by (simp add: set-bit-eq-or)

lemma set-bit-negative-int-iff [simp]:
\(<\text{set-bit } n k < 0 <\leftrightarrow k < 0 \text{ for } k :: \text{int}>\)
by (simp add: set-bit-eq-or)

lemma unset-bit-nonnegative-int-iff [simp]:
\(<\text{unset-bit } n k \geq 0 <\leftrightarrow k \geq 0 \text{ for } k :: \text{int}>\)
by (simp add: unset-bit-eq-and-not)

lemma unset-bit-negative-int-iff [simp]:
THEORY “Bit-Operations”

unset-bit n k < 0 ↔ k < 0 for k :: int
by (simp add: unset-bit-eq-and-not)

lemma flip-bit-nonnegative-int-iff [simp]:
flip-bit n k ≥ 0 ↔ k ≥ 0 for k :: int
by (simp add: flip-bit-eq-xor)

lemma flip-bit-negative-int-iff [simp]:
flip-bit n k < 0 ↔ k < 0 for k :: int
by (simp add: flip-bit-eq-xor)

lemma set-bit-greater-eq:
set-bit n k ≥ k for k :: int
by (simp add: set-bit-eq-or or-greater-eq)

lemma unset-bit-less-eq:
unset-bit n k ≤ k for k :: int
by (simp add: unset-bit-eq-and-not and-less-eq)

lemma and-int-unfold:
\[ k \land l = (\text{if } k = 0 \lor l = 0 \text{ then } 0 \text{ else if } k = -1 \text{ then } l \text{ else if } l = -1 \text{ then } k \text{ else } (k \mod 2) \ast (l \mod 2) + 2 \ast ((k \div 2) \land (l \div 2))) \]\for k l :: int
by (auto simp add: and-int.rec [of k l] zmult-eq-1-iff elim: oddE)

lemma or-int-unfold:
\[ k \lor l = (\text{if } k = -1 \lor l = -1 \text{ then } -1 \text{ else if } k = 0 \text{ then } l \text{ else if } l = 0 \text{ then } k \text{ else max } (k \mod 2) (l \mod 2) + 2 \ast ((k \div 2) \lor (l \div 2))) \]\for k l :: int
by (auto simp add: or-int.rec [of k l] elim: oddE)

lemma xor-int-unfold:
\[ k \oplus l = (\text{if } k = -1 \text{ then } \neg l \text{ else if } l = -1 \text{ then } \neg k \text{ else if } k = 0 \text{ then } l \text{ else if } l = 0 \text{ then } k \text{ else } |k \mod 2 - l \mod 2| + 2 \ast ((k \div 2) \oplus (l \div 2))) \]\for k l :: int
by (auto simp add: xor-int.rec [of k l] not-int-def elim!: oddE)

lemma bit-minus-int-iff:
\[ \text{bit } (\neg k) n \leftrightarrow \text{bit } (\neg(\neg k - 1)) n \]\for k :: int
by (simp add: bit-simps)

lemma take-bit-incr-eq:
take-bit n (k + 1) = 1 + take-bit n k if take-bit n k ≠ 2 ^ n - 1 for k :: int
proof —
from that have \(2 ^ n \neq k \mod 2 ^ n + 1\)
by (simp add: take-bit-eq-mod)
moreover have \(k \mod 2 ^ n < 2 ^ n\)
by simp
ultimately have \*: \(k \mod 2 ^ n + 1 < 2 ^ n\)
by linarith
have \((k + 1) \mod 2^\odot n = (k \mod 2^\odot n + 1) \mod 2^\odot n\)
by (simp add: mod-simps)
also have \(\ldots = k \mod 2^\odot n + 1\)
using * by (simp add: zmod-trivial-iff)
finally have \((k + 1) \mod 2^\odot n = k \mod 2^\odot n + 1\).
then show ?thesis
by (simp add: take-bit-eq-mod)
qed

lemma take-bit-decr-eq:
\((\text{take-bit } n \ (k - 1)) = \text{ take-bit } n \ k - 1\) if \((\text{take-bit } n \ k \neq 0)\) for \(k :: \text{int}\)
proof -
from that have \((k \mod 2^\odot n \neq 0)\)
by (simp add: take-bit-eq-mod)
moreover have \((k \mod 2^\odot n \geq 0), (k \mod 2^\odot n < 2^\odot n)\)
by simp-all
ultimately have *: \((k \mod 2^\odot n > 0)\)
by linarith
have \((k - 1) \mod 2^\odot n = (k \mod 2^\odot n - 1) \mod 2^\odot n\)
by (simp add: mod-simps)
also have \(\ldots = k \mod 2^\odot n - 1\)
by (simp add: zmod-trivial-iff)
( use \(k \mod 2^\odot n < 2^\odot n\) * in linarith)
finally have \((k - 1) \mod 2^\odot n = k \mod 2^\odot n - 1\).
then show ?thesis
by (simp add: take-bit-eq-mod)
qed

lemma take-bit-int-greater-eq:
\((k + 2^\odot n \leq \text{ take-bit } n \ k)\) if \((k < 0)\) for \(k :: \text{int}\)
proof -
have \((k + 2^\odot n \leq \text{ take-bit } n \ (k + 2^\odot n))\)
proof (cases \(k \geq (2^\odot n)\))
case False
then have \((k + 2^\odot n \leq 0)\)
by simp
also note take-bit-nonnegative
finally show ?thesis.
next
case True
with that have \((0 \leq k + 2^\odot n)\) and \((k + 2^\odot n < 2^\odot n)\)
by simp-all
then show ?thesis
by (simp only: take-bit-eq-mod mod-pos-pos-trivial)
qed
then show ?thesis
by (simp add: take-bit-eq-mod)
qed
lemma take-bit-int-less-eq:
\langle \text{take-bit } n \ k \leq \ k - 2^n \rangle \text{ if } \langle 2^n \leq k \rangle \text{ and } \langle n > 0 \rangle \text{ for } k :: \text{int}
using that zmod-le-nonneg-dividend [of \langle k - 2^n \rangle \langle 2^n \rangle]
by (simp add: take-bit-eq-mod)

lemma take-bit-int-less-eq-self-iff:
\langle \text{take-bit } n \ k \leq \ k \iff 0 \leq k \rangle \text{ (is } \langle ?P \iff ?Q \rangle ) \text{ for } k :: \text{int}

proof
assume ?P
show ?Q
proof (rule ccontr)
assume \langle \neg 0 \leq k \rangle
then have \langle k < 0 \rangle
  by simp
with \langle ?P \rangle
have \langle \text{take-bit } n \ k < 0 \rangle
  by (rule le-less-trans)
then show False
  by simp
qed

next
assume ?Q
then show ?P
  by (simp add: take-bit-eq-mod zmod-le-nonneg-dividend)
qed

lemma take-bit-int-less-self-iff:
\langle \text{take-bit } n \ k < k \iff 0 \leq k \rangle \text{ for } k :: \text{int}

by (auto simp add: less-le take-bit-int-less-eq-self-iff take-bit-int-eq-self-iff
  intro: order-trans [of 0 \langle 2^n \rangle k])

lemma take-bit-int-greater-self-iff:
\langle k < \text{take-bit } n \ k \iff k < 0 \rangle \text{ for } k :: \text{int}
using take-bit-int-less-eq-self-iff [of n k] by auto

lemma take-bit-int-greater-eq-self-iff:
\langle k \leq \text{take-bit } n \ k \iff k < 2^n \rangle \text{ for } k :: \text{int}
by (auto simp add: le-less take-bit-int-greater-self-iff take-bit-int-eq-self-iff
  dest: sym not-sym intro: less-trans [of k 0 \langle 2^n \rangle])

lemma take-bit-tightened-less-eq-int:
\langle \text{take-bit } m \ k \leq \text{take-bit } n \ k \rangle \text{ if } \langle m \leq n \rangle \text{ for } k :: \text{int}

proof
have \langle \text{take-bit } m \ (\text{take-bit } n \ k) \leq \text{take-bit } n \ k \rangle
  by (simp only: take-bit-int-less-eq-self-iff take-bit-nonnegative)
with that show \langle \text{thesis} \rangle
  by simp
qed
lemma \textit{not-exp-less-eq-0-int} [simp]:
\begin{align*}
\lnot 2^\leq n \leq (0::\text{int}) \\
\text{by (simp add: power-le-zero-eq)}
\end{align*}

lemma \textit{int-bit-bound}:
\begin{align*}
\text{fixes } k :: \text{int} \\
\text{obtains } n \text{ where } \forall m. \ n \leq m \implies \text{bit } k \ m \longleftrightarrow \text{bit } k \ n \\
\text{and } \forall n > 0 \implies \text{bit } k \ (n - 1) \neq \text{bit } k \ n
\end{align*}

proof –
\text{obtain } q \text{ where } \forall m. \ n \leq m \implies \text{bit } k \ m \longleftrightarrow \text{bit } k \ q
\begin{align*}
\text{proof (cases } k \geq 0) \\
\text{case True} \\
\text{moreover from power-gt-expt [of 2 (nat } k\text{)]} \\
\text{have } (\text{nat } k < 2 \sim \text{nat } k) \\
\text{by simp} \\
\text{then have } (\text{int } (\text{nat } k) < \text{int } (2 \sim \text{nat } k)) \\
\text{by (simp only: of-nat-less-iff)} \\
\text{ultimately have } \forall m. \ n \leq m \implies \text{bit } k \ m \longleftrightarrow \text{bit } k \ q
\end{align*}

show thesis
\text{proof (rule that [of } (\text{nat } k)\text{])}
\text{fix } m \\
\text{assume } (\text{nat } k \leq m) \\
\text{then show } (\text{bit } k \ m \longleftrightarrow \text{bit } k \ (\text{nat } k))
\text{by (auto simp add: * bit-iff-odd power-add zdiv-zmult2-eq dest!: le-Suc-ex)}
\text{qed}
next
\text{case False}
\text{moreover from power-gt-expt [of 2 (nat } (- k)\text{)]} \\
\text{have } (\text{nat } (- k) < 2 \sim \text{nat } (- k)) \\
\text{by simp} \\
\text{then have } (\text{int } (\text{nat } (- k)) < \text{int } (2 \sim \text{nat } (- k))) \\
\text{by (simp only: of-nat-less-iff)} \\
\text{ultimately have } (- k \div (2 \sim \text{nat } (- k)) = - 1) \\
\text{by (subst div-pos-neg-trivial) simp-all} \\
\text{then have } \forall m. \ n \leq m \implies \text{bit } k \ m \longleftrightarrow \text{bit } k \ (\text{nat } (- k))
\text{by simp} \\
\text{show thesis}
\text{proof (rule that [of } (\text{nat } (- k))\text{])}
\text{fix } m \\
\text{assume } (\text{nat } (- k) \leq m) \\
\text{then show } (\text{bit } k \ m \longleftrightarrow \text{bit } k \ (\text{nat } (- k)))
\text{by (auto simp add: * bit-iff-odd power-add zdiv-zmult2-eq minus-1-div-exp-eq-int dest!: le-Suc-ex)}
\text{qed}
\text{qed}
\text{show thesis}
\text{proof (cases } \forall m. \text{bit } k \ m \longleftrightarrow \text{bit } k \ q\text{)}
\text{case True}
then have \( \langle \text{bit} \ k \ 0 \longleftrightarrow \text{bit} \ k \ q \rangle \)
\[ \text{by} \ \text{blast} \]
with True that [of 0] show thesis
\[ \text{by} \ \text{simp} \]
next
  case False
  then obtain \( r \) where \( \ast \ast \langle \text{bit} \ k \ r \neq \text{bit} \ k \ q \rangle \)
  \[ \text{by} \ \text{blast} \]
  have \( \langle r < q \rangle \)
  \[ \text{by} \ \text{rule ccontr} \ (\text{use \ [of \ } r \ \ast \ast \ \text{in} \ \text{simp})} \]
  define \( N \) where \( \langle N = \{ n. n < q \wedge \text{bit} \ k \ n \neq \text{bit} \ k \ q \} \rangle \)
  moreover have \( \langle \text{finite} \ N, \langle r \in N \rangle \rangle \)
  using \( \ast \ast \text{N-def} \langle r < q \rangle \) by auto
  ultimately have \( \langle \forall m. n \leq m \implies \text{bit} \ k \ m \longleftrightarrow \text{bit} \ k \ n \rangle \)
  apply auto
  \[ \text{apply} \ (\text{metis (full-types, lifting) \ast Max-ge-iff Suc-n-not-le-n \langle \text{finite} \ N \rangle \ all-not-in-conv mem-Collect-eq not-le}) \]
  apply (\text{metis \ \ast \text{Max-ge Suc-n-not-le-n \langle \text{finite} \ N \rangle \ linorder-not-less mem-Collect-eq})
  apply (\text{metis \ \ast \text{Max-ge Suc-n-not-le-n \langle \text{finite} \ N \rangle \ linorder-not-less mem-Collect-eq})
  apply (\text{metis (full-types, lifting) \ast Max-ge-iff Suc-n-not-le-n \langle \text{finite} \ N \rangle \ all-not-in-conv mem-Collect-eq not-le})
  done
  have \( \langle \text{bit} \ k \ (\text{Max} \ N) \neq \text{bit} \ k \ n \rangle \)
  \[ \text{by} \ \text{metis (mono-tags, lifting) \ast Max-in N-def \langle \forall m. n \leq m \implies \text{bit} \ k \ m =} \]
  \[ \text{bit} \ k \ n \ \langle \text{finite} \ N, \langle r \in N \rangle \text{empty-iff le-cases mem-Collect-eq}) \]
  show thesis apply (\text{rule that [of n]})
  using \( \langle \forall m. n \leq m \implies \text{bit} \ k \ m = \text{bit} \ k \ n \rangle \) apply blast
  using \( \langle \text{bit} \ k \ (\text{Max} \ N) \neq \text{bit} \ k \ n \rangle \) n-def by auto
  qed
qed

\[ \text{68.5 Instance nat} \]

instantiation nat :: semiring-bit-operations
begin

definition and-nat :: \( \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle \)
\[ \langle m \ \text{AND} \ n = \text{nat} \ (\text{int} \ m \ \text{AND} \ \text{int} \ n) \rangle \text{ for } m \ n :: \text{nat} \]

definition or-nat :: \( \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle \)
\[ \langle m \ \text{OR} \ n = \text{nat} \ (\text{int} \ m \ \text{OR} \ \text{int} \ n) \rangle \text{ for } m \ n :: \text{nat} \]

definition xor-nat :: \( \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle \)
\[ \langle m \ \text{XOR} \ n = \text{nat} \ (\text{int} \ m \ \text{XOR} \ \text{int} \ n) \rangle \text{ for } m \ n :: \text{nat} \]

definition mask-nat :: \( \langle \text{nat} \Rightarrow \text{nat} \rangle \)
\[ \langle \text{mask} \ n = (2 :: \text{nat}) \ ^{-} \text{\textit{n}} - 1 \rangle \text{ for } m \ n :: \text{nat} \]
definition push-bit-nat :: (nat ⇒ nat ⇒ nat)
where \( (\text{push-bit-nat} \ n \ m = m \times 2 ^ n) \n\)
definition drop-bit-nat :: (nat ⇒ nat ⇒ nat)
where \( (\text{drop-bit-nat} \ n \ m = m \div 2 ^ n) \n\)
definition take-bit-nat :: (nat ⇒ nat ⇒ nat)
where \( (\text{take-bit-nat} \ n \ m = m \mod 2 ^ n) \n\)
definition set-bit-nat :: (nat ⇒ nat ⇒ nat)
where \( (\text{set-bit} \ m \ n = \text{push-bit} \ m \ 1) \n\)
definition unset-bit-nat :: (nat ⇒ nat ⇒ nat)
where \( (\text{unset-bit} \ m \ n = (\text{push-bit} \ m \ 1) \ \text{XOR} \ \text{push-bit} \ m \ 1) \n\)
definition flip-bit-nat :: (nat ⇒ nat ⇒ nat)
where \( (\text{flip-bit} \ m \ n = \text{push-bit} \ m \ 1) \n\)

instance proof
fix \( m \ n :: \text{nat} \n\)
show \( (\text{push-bit} \ n \ (\text{of-nat} \ m) = \text{of-nat} \ (\text{push-bit} \ n \ m) \n\)
by (simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult)

lemma push-bit-of-nat:
\( (\text{of-nat} \ \text{push-bit} \ n \ m = \text{of-nat} \ \text{push-bit} \ n \ m) \n\)
by (simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult)

lemma of-nat-push-bit:
\( (\text{of-nat} \ \text{push-bit} \ n \ m = \text{push-bit} \ m \ (\text{of-nat} \ n) \n\)
by (simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult)

lemma take-bit-of-nat:
\( (\text{of-nat} \ \text{take-bit} \ n \ m = \text{of-nat} \ (\text{take-bit} \ n \ m) \n\)
by (rule bit-eqI) (simp add: bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-nat-iff)

end

instance nat :: linordered-euclidean-semiring-bit-operations ..

context semiring-bit-operations
begin

lemma push-bit-of-nat:
\( (\text{push-bit} \ n \ (\text{of-nat} \ m) = \text{of-nat} \ (\text{push-bit} \ n \ m) \n\)
by (simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult)

lemma of-nat-push-bit:
\( (\text{of-nat} \ \text{push-bit} \ n \ m = \text{push-bit} \ m \ (\text{of-nat} \ n) \n\)
by (simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult)

lemma take-bit-of-nat:
\( (\text{of-nat} \ \text{take-bit} \ n \ m = \text{of-nat} \ (\text{take-bit} \ n \ m) \n\)
by (rule bit-eqI) (simp add: bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-nat-iff)

lemma of-nat-take-bit:
THEORY “Bit-Operations”

(of-nat (take-bit n m) = take-bit n (of-nat m))
by (rule bit-eqI) (simp add: bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-nat-iff)

lemma of-nat-and-eq:
(of-nat (m AND n) = of-nat m AND of-nat n)

lemma of-nat-or-eq:
(of-nat (m OR n) = of-nat m OR of-nat n)
by (rule bit-eqI) (simp add: bit-of-nat-iff bit-or-iff Bit-Operations.bit-or-iff)

lemma of-nat-xor-eq:
(of-nat (m XOR n) = of-nat m XOR of-nat n)
by (rule bit-eqI) (simp add: bit-of-nat-iff bit-xor-iff Bit-Operations.bit-xor-iff)

lemma of-nat-mask-eq:
(of-nat (mask n) = mask n)
by (induction n) (simp-all add: mask-Suc-double Bit-Operations.mask-Suc-double of-nat-or-eq)

end

context linordered-euclidean-semiring-bit-operations
begin

lemma drop-bit-of-nat:
drop-bit n (of-nat m) = of-nat (drop-bit n m)
by (simp add: drop-bit-eq-div Bit-Operations.drop-bit-eq-div of-nat-div [of m 2 ^ n])

lemma of-nat-drop-bit:
of-nat (drop-bit m n) = drop-bit m (of-nat n)

end

lemma take-bit-nat-less-exp [simp]:
take-bit n m < 2 ^ n) for n m :: nat
by (simp add: take-bit-eq-mod)

lemma take-bit-nat-eq-self-iff:
take-bit n m = m (\rightarrow m < 2 ^ n) (is {?P \leftrightarrow ?Q}) for n m :: nat
proof
assume {?P
moreover note take-bit-nat-less-exp [of n m]
ultimately show {?Q
by simp
next
assume {?Q

then show \(?P\)
  by (simp add: take-bit-eq-mod)
qed

lemma take-bit-nat-eq-self:
  \(\{\text{take-bit } n \ m = m\} \ \text{if } \ m < 2 \ ^{\sim} n\) \text{ for } m \ n :: \text{nat}
using that by (simp add: take-bit-nat-eq-self-iff)

lemma take-bit-nat-less-eq-self [simp]:
  \(\{\text{take-bit } n \ m \leq m\} \ \text{for } n \ m :: \text{nat}\)
by (simp add: take-bit-eq-mod)

lemma take-bit-nat-less-self-iff:
  \(\{\text{take-bit } n \ m \ < m\} \ \text{if } \ 2 \ ^{\sim} n \leq m\) \text{ is } \(\{?P \urlsym\?Q\}\) \text{for } m \ n :: \text{nat}
proof
assume \(?P\)
then have \(\{\text{take-bit } n \ m \neq m\}\)
  by simp
then show \(\{?Q\}\)
  by (simp add: take-bit-nat-eq-self-iff)
next
have \(\{\text{take-bit } n \ m \ < 2 \ ^{\sim} n\}\)
  by (fact take-bit-nat-less-exp)
also assume \(?Q\)
finally show \(?P\).
qed

lemma Suc-0-and-eq [simp]:
  \(\{\text{Suc } 0 \ \land \ n = n \ \text{mod } 2\}\)
using one-and-eq [of n] by simp

lemma and-Suc-0-eq [simp]:
  \(\{n \ \land \ \text{Suc } 0 = n \ \text{mod } 2\}\)
using and-one-eq [of n] by simp

lemma Suc-0-or-eq:
  \(\{\text{Suc } 0 \ \lor \ n = n + \text{of-bool} \ (\text{even } n)\}\)
using one-or-eq [of n] by simp

lemma or-Suc-0-eq:
  \(\{n \ \lor \ \text{Suc } 0 = n + \text{of-bool} \ (\text{even } n)\}\)
using or-one-eq [of n] by simp

lemma Suc-0-xor-eq:
  \(\{\text{Suc } 0 \ \text{xor} \ n = n + \text{of-bool} \ (\text{even } n) - \text{of-bool} \ (\text{odd } n)\}\)
using one-xor-eq [of n] by simp

lemma xor-Suc-0-eq:
  \(\{n \ \text{xor} \ \text{Suc } 0 = n + \text{of-bool} \ (\text{even } n) - \text{of-bool} \ (\text{odd } n)\}\)
using xor-one-eq [of n] by simp

lemma and-nat-unfold [code]:
\( \langle m \land n = (if m = 0 \lor n = 0 then 0 else (m \mod 2) \times (n \mod 2) + 2 \times ((m \div 2) \land (n \div 2))) \rangle \)
for m n :: nat
by (auto simp add: and-rec [of m n] elim: oddE)

lemma or-nat-unfold [code]:
\( \langle m \lor n = (if m = 0 then n else if n = 0 then m else max (m \mod 2) (n \mod 2) + 2 \times ((m \div 2) \lor (n \div 2))) \rangle \)
for m n :: nat
by (auto simp add: or-rec [of m n] elim: oddE)

lemma xor-nat-unfold [code]:
\( \langle m \lor n = (if m = 0 \lor n = 0 then m \else (m \mod 2 + \neg n \mod 2) \mod 2 + 2 \times ((m \div 2) \xor (n \div 2))) \rangle \)
for m n :: nat
by (auto simp add: xor-rec [of m n] elim!: oddE)

lemma unset-bit-Suc-0 [simp]:
\( \langle \text{ unset-bit } n \text{ Suc } 0 = 2 \times \neg n \rangle \)
using push-bit-of-Suc-0 [where \( \neg a = \text{ nat} \)] by simp

lemma take-bit-Suc-0 [simp]:
\( \langle \text{ take-bit } n \text{ Suc } 0 = \text{ of-bool } (0 < n) \rangle \)
using take-bit-of-Suc-0 [where \( \neg a = \text{ nat} \)] by simp

lemma drop-bit-Suc-0 [simp]:
\( \langle \text{ drop-bit } n \text{ Suc } 0 = \text{ of-bool } (n = 0) \rangle \)
using drop-bit-of-Suc-0 [where \( \neg a = \text{ nat} \)] by simp

lemma Suc-mask-eq-exp:
\( \langle \text{ Suc } (\text{ mask } n) = 2 \times \neg n \rangle \)
by (simp add: mask-eq-exp-minus-1)

lemma less-eq-mask:
\( \langle n \leq \text{ mask } n \rangle \)
by (simp add: mask-eq-exp-minus-1 le-diff-conv2)
  (metis Suc-mask-eq-exp diff-Suc-1 diff-le-diff-poW diff-zero le-refl not-less-eq-eq power-0)

lemma less-mask:
\( \langle n < \text{ mask } n \rangle \)
if \( \langle \text{ Suc } 0 < n \rangle \)
proof –
  define m where \( m = n - 2 \)
  with that have \(*\!\!: (n = m + 2)\)
    by \simp
  have \( \langle \text{Suc} \ (\text{Suc} \ (\text{Suc} \ m)) < 4 \ast 2 ^{\sim} m \rangle \)
    by \( \text{(induction } m) \ \simp\text{-all} \)
  then have \( \langle \text{Suc} \ (m + 2) < \text{Suc} \ (\text{mask} \ (m + 2)) \rangle \)
    by \( \simp\text{ add: } \text{Suc-mask-eq-exp} \)
  then have \( \langle m + 2 < \text{mask} \ (m + 2) \rangle \)
    by \( \simp\text{ add: } \text{less-le} \)
  with \* show \( ?\text{thesis} \)
    by \simp
  qed

lemma \( \text{mask-nat-less-exp} \ [\simp]; \)
  \( \langle \text{mask} \ n :: \text{nat} \rangle < 2 ^{\sim} n \)
  by \( \simp\text{ add: } \text{mask-eq-exp-minus-1} \)

lemma \( \text{mask-nat-positive-iff} \ [\simp]; \)
  \( \langle 0 :: \text{nat} \rangle < \text{mask} \ n \leftrightarrow 0 < n \)
proof \( \text{(cases } n = 0 \rangle \)
  case True
  then show \( ?\text{thesis} \)
    by \simp
next
  case False
  then have \( \langle 0 < n \rangle \)
    by \simp
  then have \( \langle 0 :: \text{nat} \rangle < \text{mask} \ n \)
    using \( \text{less-eq-mask} \ [\text{of } n] \) by \( \text{rule } \text{order-less-le-trans} \)
  with \( \langle 0 < n \rangle \) show \( ?\text{thesis} \)
    by \simp
  qed

lemma \( \text{take-bit-tightened-less-eq-nat} \);
  \( \langle \text{take-bit} \ m \ q \leq \text{take-bit} \ n \ q \rangle \) if \( \langle m \leq n \rangle \) for \( q :: \text{nat} \)
proof –
  have \( \langle \text{take-bit} \ m \ (\text{take-bit} \ n \ q) \leq \text{take-bit} \ n \ q \rangle \)
    by \( \text{rule } \text{take-bit-nat-less-eq-self} \)
  with that show \( ?\text{thesis} \)
    by \simp
  qed

lemma \( \text{push-bit-nat-eq} \);
  \( \langle \text{push-bit} \ n \ (\text{nat} \ k) = \text{nat} \ (\text{push-bit} \ n \ k) \rangle \)
by \( \text{(cases } k \geq 0 \rangle \) \( \simp\text{-all add: } \text{push-bit-eq-mult nat-mult-distrib not-le mult-nonneg-nonpos2} \)

lemma \( \text{drop-bit-nat-eq} \);
  \( \langle \text{drop-bit} \ n \ (\text{nat} \ k) = \text{nat} \ (\text{drop-bit} \ n \ k) \rangle \)
apply (cases \(k \geq 0\))
apply (simp-all add: drop-bit-eq-div nat-div-distrib nat-power-eq not-le)
apply (simp add: divide-int-def)
done

lemma take-bit-nat-eq:
\(<\text{take-bit } n \ (\text{nat } k) = \text{nat } \ (\text{take-bit } n \ k)\> \ if \ <k \geq 0>
using that by (simp add: take-bit-eq-mod nat-mod-distrib nat-power-eq)

lemma nat-take-bit-eq:
\(<\text{nat } \ (\text{take-bit } n \ k) = \text{take-bit } n \ (\text{nat } k)\>
if \ <k \geq 0>
using that by (simp add: take-bit-eq-mod nat-mod-distrib nat-power-eq)

lemma nat-mask-eq:
\(<\text{nat } \ (\text{mask } n) = \text{mask } n\>
by (simp add: nat-eq-iff of-nat-mask-eq)

68.6 Symbolic computations on numeral expressions

context semiring-bits
begin

lemma not-bit-numeral-Bit0-0 [simp]:
\(<\neg \text{bit } (\text{numeral } (\text{Num.Bit0 } m)) \ 0\>
by (simp add: bit-0)

lemma bit-numeral-Bit1-0 [simp]:
\(<\text{bit } (\text{numeral } (\text{Num.Bit1 } m)) \ 0\>
by (simp add: bit-0)

lemma bit-numeral-Bit0-iff:
\(<\text{bit } (\text{numeral } (\text{num.Bit0 } m)) \ n \leftrightarrow \text{possible-bit } \text{TYPE}'(a) \ n \land n > 0 \land \text{bit } (\text{numeral } m) \ (n - 1)\>
by (simp only: numeral-Bit0-eq-double [of m] bit-simps) simp

lemma bit-numeral-Bit1-Suc-iff:
\(<\text{bit } (\text{numeral } (\text{num.Bit1 } m)) \ (\text{Suc } n) \leftrightarrow \text{possible-bit } \text{TYPE}'(a) \ (\text{Suc } n) \land \text{bit } (\text{numeral } m) \ n\>
using even-bit-succ-iff [of \(2 \ast \text{numeral } m\] \(\text{Suc } n\)]
by (simp only: numeral-Bit1-eq-inc-double [of m] bit-simps) simp

end

context ring-bit-operations
begin

lemma not-bit-minus-numeral-Bit0-0 [simp]:
\(<\neg \text{bit } (- \text{numeral } (\text{Num.Bit0 } m)) \ 0\>

by (simp add: bit-0)

lemma bit-minus-numeral-Bit1-0 [simp]:
  \(\langle \text{bit} \left(- \text{numeral (Num.Bit1 m)}\right) 0 \rangle\)
by (simp add: bit-0)

lemma bit-minus-numeral-Bit0-Suc-iff:
  \(\langle \text{bit} \left(- \text{numeral (num.Bit0 m)\right) (Suc n)\langle \text{possible-bit TYPE('a) (Suc n) \land bit \left(- \text{numeral m} \right) n\rangle\rangle\}\)\)
by (simp only: numeral-Bit0-eq-double \[of m\] minus-mult-right bit-simps) auto

lemma bit-minus-numeral-Bit1-Suc-iff:
  \(\langle \text{bit} \left(- \text{numeral (num.Bit1 m)\right) (Suc n)\langle \text{possible-bit TYPE('a) (Suc n) \land \neg bit \left(- \text{numeral m} \right) n\rangle\rangle\}\)\)
by (simp only: numeral-Bit1-eq-inc-double \[of m\] minus-add-distrib minus-mult-right add-uminus-conv-diff bit-decr-iff bit-double-iff) auto

lemma bit-numeral-BitM-0 [simp]:
  \(\langle \text{bit} \left(\text{numeral (Num.BitM m)\right) 0 \rangle\)\)
by (simp only: numeral-BitM bit-decr-iff not-bit-minus-numeral-Bit0-0 simp)

lemma bit-numeral-BitM-Suc-iff:
  \(\langle \text{bit} \left(\text{numeral (Num.BitM m)\right) (Suc n)\langle \text{possible-bit TYPE('a) (Suc n) \land \neg bit \left(- \text{numeral m} \right) n\rangle\rangle\}\)\)
by (simp-all only: numeral-BitM bit-decr-iff bit-minus-numeral-Bit0-Suc-iff) auto

context linordered-euclidean-semiring-bit-operations
begin

lemma bit-numeral-iff:
  \(\langle \text{bit} \left(\text{numeral m\right) n\langle \text{bit} \left(\text{numeral m :: nat\right) n\rangle\rangle\)\)
using bit-of-nat-iff-bit \[of \text{\langle \text{numeral m\rangle\} \text{n}}\] by simp

lemma bit-numeral-Bit0-Suc-iff [simp]:
  \(\langle \text{bit} \left(\text{numeral (Num.Bit0 m)\right) (Suc n)\langle \text{bit} \left(\text{numeral m\right) n\rangle\rangle\)\)
by (simp add: bit-Suc numeral-Bit0-div-2)

lemma bit-numeral-Bit1-Suc-iff [simp]:
  \(\langle \text{bit} \left(\text{numeral (Num.Bit1 m)\right) (Suc n)\langle \text{bit} \left(\text{numeral m\right) n\rangle\rangle\)\)
by (simp add: bit-Suc numeral-Bit1-div-2)

lemma bit-numeral-rec:
  \(\langle \text{bit} \left(\text{numeral (Num.Bit0 w)\right) n\langle \text{case n of 0 \Rightarrow False | Suc m \Rightarrow bit \left(\text{numeral w\right) m\rangle\rangle\)\)
  \(\langle \text{bit} \left(\text{numeral (Num.Bit1 w)\right) n\langle \text{case n of 0 \Rightarrow True | Suc m \Rightarrow bit \left(\text{numeral w\right) m\rangle\rangle\)\)
end
THEORY “Bit-Operations” 1499

by (cases n; simp add: bit-0)+

lemma bit-numeral-simps [simp]:

\(<\sim bit \text{ (numeral } n)\):
\(<bit \text{ (numeral } \text{Nat.Bit0 } w) \text{ (numeral } n) \iff bit \text{ (numeral } w) \text{ (pred-numeral } n)\>:
\(<bit \text{ (numeral } \text{Nat.Bit1 } w) \text{ (numeral } n) \iff bit \text{ (numeral } w) \text{ (pred-numeral } n)\>:
by (simp-all add: bit-1-iff numeral-eq-Suc)

lemma and-numerals [simp]:
\(<1 \text{ AND numeral } \text{Nat.Bit0 } y = 0\>:
\(<1 \text{ AND numeral } \text{Nat.Bit1 } y = 1\>:
\(<\text{numeral } \text{Nat.Bit0 } x \text{ AND numeral } \text{Nat.Bit0 } y = 2 * \text{ (numeral } x \text{ AND numeral } y)\>:
\(<\text{numeral } \text{Nat.Bit0 } x \text{ AND numeral } \text{Nat.Bit1 } y = 2 * \text{ (numeral } x \text{ AND numeral } y)\>:
\(<\text{numeral } \text{Nat.Bit1 } x \text{ AND numeral } \text{Nat.Bit0 } y = 2 * \text{ (numeral } x \text{ AND numeral } y)\>:
\(<\text{numeral } \text{Nat.Bit1 } x \text{ AND numeral } \text{Nat.Bit1 } y = 1 + 2 * \text{ (numeral } x \text{ AND numeral } y)\>:
by (simp-all add: bit-eq-iff) (simp-all add: bit-0 bit-simps bit-Suc bit-numeral-rec split: nat.splits)

lemma or-numerals [simp]:
\(<1 \text{ OR numeral } \text{Nat.Bit0 } y = \text{ numeral } \text{Nat.Bit1 } y\>:
\(<1 \text{ OR numeral } \text{Nat.Bit1 } y = \text{ numeral } \text{Nat.Bit1 } y\>:
\(<\text{numeral } \text{Nat.Bit0 } x \text{ OR numeral } \text{Nat.Bit0 } y = 2 * \text{ (numeral } x \text{ OR numeral } y)\>:
\(<\text{numeral } \text{Nat.Bit0 } x \text{ OR numeral } \text{Nat.Bit1 } y = 1 + 2 * \text{ (numeral } x \text{ OR numeral } y)\>:
\(<\text{numeral } \text{Nat.Bit1 } x \text{ OR numeral } \text{Nat.Bit0 } y = 1 + 2 * \text{ (numeral } x \text{ OR numeral } y)\>:
\(<\text{numeral } \text{Nat.Bit1 } x \text{ OR numeral } \text{Nat.Bit1 } y = 1 + 2 * \text{ (numeral } x \text{ OR numeral } y)\>:
by (simp-all add: bit-eq-iff) (simp-all add: bit-0 bit-simps bit-Suc bit-numeral-rec split: nat.splits)

lemma xor-numerals [simp]:
\(<1 \text{ XOR numeral } \text{Nat.Bit0 } y = \text{ numeral } \text{Nat.Bit1 } y\>:
\(<1 \text{ XOR numeral } \text{Nat.Bit1 } y = \text{ numeral } \text{Nat.Bit0 } y\>:
\(<\text{numeral } \text{Nat.Bit0 } x \text{ XOR numeral } \text{Nat.Bit0 } y = 2 * \text{ (numeral } x \text{ XOR numeral } y)\>:
\(<\text{numeral } \text{Nat.Bit0 } x \text{ XOR numeral } \text{Nat.Bit1 } y = 1 + 2 * \text{ (numeral } x \text{ XOR numeral } y)\>:
\(<\text{numeral } \text{Nat.Bit1 } x \text{ XOR numeral } \text{Nat.Bit0 } y = 1 + 2 * \text{ (numeral } x \text{ XOR numeral } y)\>:
\(<\text{numeral } \text{Nat.Bit1 } x \text{ XOR numeral } \text{Nat.Bit1 } y = \text{ numeral } \text{Nat.Bit1 } y\)
THEORY "Bit-Operations"

\begin{align*}
\langle \text{numeral (Num.Bit1 x)} \text{ XOR numeral (Num.Bit0 y)} \rangle &= 1 + 2 \ast (\text{numeral x XOR numeral y}) \\
\langle \text{numeral (Num.Bit1 x)} \text{ XOR numeral (Num.Bit1 y)} \rangle &= 2 \ast (\text{numeral x XOR numeral y}) \\
\langle \text{numeral (Num.Bit1 x)} \text{ XOR 1} \rangle &= \text{numeral (Num.Bit0 x)} \\
\text{by (simp-all add: bit-eq-iff) (simp-all add: bit-0 bit-simps bit-Suc bit-numeral-rec split: nat.splits)}
\end{align*}

end

\begin{align*}
\text{lemma drop-bit-Suc-minus-bit0 [simp]:} \\
\langle \text{drop-bit (Suc n)} (- \text{numeral (Num.Bit0 k)}) \rangle &= \text{drop-bit n} (- \text{numeral k :: int}) \\
\text{by (simp add: drop-bit-Suc numeral-Bit0-div-2)}
\end{align*}

\begin{align*}
\text{lemma drop-bit-Suc-minus-bit1 [simp]:} \\
\langle \text{drop-bit (Suc n)} (- \text{numeral (Num.Bit1 k)}) \rangle &= \text{drop-bit n} (- \text{numeral (Num.inc k) :: int}) \\
\text{by (simp add: drop-bit-Suc numeral-Bit1-div-2 add-One)}
\end{align*}

\begin{align*}
\text{lemma drop-bit-numeral-minus-bit0 [simp]:} \\
\langle \text{drop-bit (numeral l)} (- \text{numeral (Num.Bit0 k)}) \rangle &= \text{drop-bit (pred-numeral l)} (- \text{numeral k :: int}) \\
\text{by (simp add: numeral-eq-Suc numeral-Bit0-div-2)}
\end{align*}

\begin{align*}
\text{lemma drop-bit-numeral-minus-bit1 [simp]:} \\
\langle \text{drop-bit (numeral l)} (- \text{numeral (Num.Bit1 k)}) \rangle &= \text{drop-bit (pred-numeral l)} (- \text{numeral (Num.inc k) :: int}) \\
\text{by (simp add: numeral-eq-Suc numeral-Bit1-div-2)}
\end{align*}

\begin{align*}
\text{lemma take-bit-Suc-minus-bit0:} \\
\langle \text{take-bit (Suc n)} (- \text{numeral (Num.Bit0 k)}) \rangle &= \text{take-bit n} (- \text{numeral k) \ast (2 :: int)} \\
\text{by (simp add: take-bit-Suc numeral-Bit0-div-2)}
\end{align*}

\begin{align*}
\text{lemma take-bit-Suc-minus-bit1:} \\
\langle \text{take-bit (Suc n)} (- \text{numeral (Num.Bit1 k)}) \rangle &= \text{take-bit n} (- \text{numeral (Num.inc k) \ast (2 :: int)} + (1 :: int)) \\
\text{by (simp add: take-bit-Suc numeral-Bit1-div-2 add-One)}
\end{align*}

\begin{align*}
\text{lemma take-bit-numeral-minus-bit0:} \\
\langle \text{take-bit (numeral l)} (- \text{numeral (Num.Bit0 k)}) \rangle &= \text{take-bit (pred-numeral l)} (- \text{numeral k) \ast (2 :: int)} \\
\text{by (simp add: numeral-eq-Suc numeral-Bit0-div-2 take-bit-Suc-minus-bit0)}
\end{align*}

\begin{align*}
\text{lemma take-bit-numeral-minus-bit1:} \\
\langle \text{take-bit (numeral l)} (- \text{numeral (Num.Bit1 k)}) \rangle &= \text{take-bit (pred-numeral l)} (- \text{numeral (Num.inc k) \ast (2 :: int)} + (1 :: int)) \\
\text{by (simp add: numeral-eq-Suc numeral-Bit1-div-2 take-bit-Suc-minus-bit1)}
\end{align*}
lemma and-nat-numerals [simp]:
  \[\text{Suc 0 AND numeral (Num.Bit0 y) = 0}\]
  \[\text{Suc 0 AND numeral (Num.Bit1 y) = 1}\]
  \[\text{numeral (Num.Bit0 x) AND Suc 0 = 0}\]
  \[\text{numeral (Num.Bit1 x) AND Suc 0 = 1}\]
  by (simp-all only: and-numerals flip: One-nat-def)

lemma or-nat-numerals [simp]:
  \[\text{Suc 0 OR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)}\]
  \[\text{Suc 0 OR numeral (Num.Bit1 y) = numeral (Num.Bit1 y)}\]
  \[\text{numeral (Num.Bit0 x) OR Suc 0 = numeral (Num.Bit1 x)}\]
  \[\text{numeral (Num.Bit1 x) OR Suc 0 = numeral (Num.Bit0 x)}\]
  by (simp-all only: or-numerals flip: One-nat-def)

lemma xor-nat-numerals [simp]:
  \[\text{Suc 0 XOR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)}\]
  \[\text{Suc 0 XOR numeral (Num.Bit1 y) = numeral (Num.Bit0 y)}\]
  \[\text{numeral (Num.Bit0 x) XOR Suc 0 = numeral (Num.Bit1 x)}\]
  \[\text{numeral (Num.Bit1 x) XOR Suc 0 = numeral (Num.Bit0 x)}\]
  by (simp-all only: xor-numerals flip: One-nat-def)

context ring-bit-operations
begin

lemma minus-numeral-inc-eq:
  \[\text{\neg numeral (Num.inc n) = NOT (numeral n)}\]
  by (simp add: not-eq-complement sub-inc-One-eq add-One)

lemma sub-one-eq-not-neg:
  \[\text{Num.sub n num.One = NOT (\neg numeral n)}\]
  by (simp add: not-eq-complement)

lemma minus-numeral-eq-not-sub-one:
  \[\text{\neg numeral n = NOT (Num.sub n num.One)}\]
  by (simp add: not-eq-complement)

lemma not-numeral-eq [simp]:
  \[\text{NOT (numeral n) = \neg numeral (Num.inc n)}\]
  by (simp add: minus-numeral-inc-eq)

lemma not-minus-numeral-eq [simp]:
  \[\text{NOT (\neg numeral n) = Num.sub n num.One}\]
  by (simp add: sub-one-eq-not-neg)

lemma minus-not-numeral-eq [simp]:
  \[\text{\neg (NOT (numeral n)) = numeral (Num.inc n)}\]
  by simp

lemma not-numeral-BitM-eq:
THEORY "Bit-Operations"

\langle NOT \text{ numeral (Num.BitM } n) = \text{ numeral (num.Bit0 } n) \rangle \\
\text{by simp add: inc-BitM-eq}

\textbf{lemma not-numeral-Bit0-eq:} \\
\langle NOT \text{ numeral (Num.Bit0 } n) = \text{ numeral (num.Bit1 } n) \rangle \\
\text{by simp}

\textbf{end}

\textbf{lemma bit-minus-numeral-int [simp]}: \\
\langle bit (\text{ numeral (num.Bit0 } w) :: int) \text{ numeral } n \leftrightarrow \langle bit (\text{ numeral } w :: int) \text{ pred-numeral } n \rangle \\
\langle bit (\text{ numeral (num.Bit1 } w) :: int) \text{ numeral } n \leftrightarrow \langle bit (\text{ numeral } w :: int) \text{ pred-numeral } n \rangle \\
\text{by simp-all add: bit-minus-iff bit-not-iff numeral-eq-Suc bit-Suc add-One sub-inc-One-eq}

\textbf{lemma bit-minus-numeral-Bit0-Suc-iff [simp]}: \\
\langle bit (\text{ numeral (num.Bit0 } w) :: int) \text{ Suc } n \leftrightarrow \langle bit (\text{ numeral } w :: int) \text{ Suc } n \rangle \\
\text{by simp add: bit-Suc}

\textbf{lemma bit-minus-numeral-Bit1-Suc-iff [simp]}: \\
\langle bit (\text{ numeral (num.Bit1 } w) :: int) \text{ Suc } n \leftrightarrow \langle bit (\text{ numeral } w :: int) \text{ Suc } n \rangle \\
\text{by simp add: bit-Suc add-One flip: bit-not-iff}

\textbf{lemma and-not-numerals:} \\
\langle 1 \text{ AND NOT } 1 = (0 :: int) \rangle \\
\langle 1 \text{ AND NOT } \text{ numeral (Num.Bit0 } n) = (1 :: int) \rangle \\
\langle 1 \text{ AND NOT } \text{ numeral (Num.Bit1 } n) = (0 :: int) \rangle \\
\langle \text{ numeral (Num.Bit0 } m) \text{ AND NOT } (1 :: int) = \text{ numeral (Num.Bit0 } m) \rangle \\
\langle \text{ numeral (Num.Bit0 } m) \text{ AND NOT } \text{ numeral (Num.Bit0 } n) = (2 :: int) \ast \text{ numeral } m \text{ AND NOT } (\text{ numeral } n) \rangle \\
\langle \text{ numeral (Num.Bit0 } m) \text{ AND NOT } \text{ numeral (Num.Bit1 } n) = (2 :: int) \ast \text{ numeral } m \text{ AND NOT } (\text{ numeral } n) \rangle \\
\langle \text{ numeral (Num.Bit0 } m) \text{ AND NOT } (1 :: int) = \text{ numeral (Num.Bit0 } m) \rangle \\
\langle \text{ numeral (Num.Bit1 } m) \text{ AND NOT } (\text{ numeral (Num.Bit0 } n) = 1 + (2 :: int) \ast (\text{ numeral } m \text{ AND NOT } (\text{ numeral } n)) \rangle \\
\langle \text{ numeral (Num.Bit1 } m) \text{ AND NOT } \text{ numeral (Num.Bit1 } n) = (2 :: int) \ast (\text{ numeral } m \text{ AND NOT } (\text{ numeral } n)) \rangle \\
\text{by simp-all add: bit-eq-iff) (auto simp add: bit-0 bit-simps bit-Suc bit-numeral-rec BitM-inc-eq sub-inc-One-eq split: nat.split)

\textbf{fun and-not-num :: (num ⇒ num ⇒ num option)}

\textbf{where} \\
\langle and-not-num num.\text{One } num.\text{One} = \text{None} \rangle \\
\langle and-not-num num.\text{One } num.\text{Bit0 } n = \text{Some num.\text{One}} \rangle \\
\langle and-not-num num.\text{One } num.\text{Bit1 } n = \text{None} \rangle \\
\langle and-not-num (num.\text{Bit0 } m) num.\text{One} = \text{Some (num.\text{Bit0 } m)} \rangle \\
\langle and-not-num (num.\text{Bit0 } m) (num.\text{Bit0 } n) = \text{map-option num.Bit0 (and-not-num m n)} \rangle
\[\langle \text{and-not-num} (\text{num.Bit0} m) (\text{num.Bit1} n) = \text{map-option} \text{num.Bit0} (\text{and-not-num} m n) \rangle\]
\[\langle \text{and-not-num} (\text{num.Bit1} m) \text{num.One} = \text{Some} (\text{num.Bit0} m) \rangle\]
\[\langle \text{and-not-num} (\text{num.Bit1} m) (\text{num.Bit0} n) = (\text{case and-not-num} m n \text{ of None} \Rightarrow \text{Some num.One} \mid \text{Some} n' \Rightarrow \text{Some} (\text{num.Bit1} n')) \rangle\]
\[\langle \text{and-not-num} (\text{num.Bit1} m) (\text{num.Bit1} n) = \text{map-option} \text{num.Bit0} (\text{and-not-num} m n) \rangle\]

**lemma int-numeral-and-not-num:**

\[
\langle \text{numeral} m \text{ AND NOT} \ (\text{numeral} n) = (\text{case and-not-num} m n \text{ of None} \Rightarrow 0 :: \text{int} \mid \text{Some} n' \Rightarrow \text{numeral} n') \rangle
\]

**by (induction m n rule: and-not-num.induct) (simp-all del: not-numeral-eq not-one-eq add: and-not-numerals split: option.split)\]

**lemma int-numeral-not-and-num:**

\[
\langle \text{NOT} (\text{numeral} m) \text{ AND} \text{numeral} n = (\text{case and-not-num} m n \text{ of None} \Rightarrow 0 :: \text{int} \mid \text{Some} n' \Rightarrow \text{numeral} n') \rangle
\]

**using int-numeral-and-not-num [of n m] by (simp add: ac-simps)\]

**lemma and-not-num-eq-None-iff:**

\[
\langle \text{and-not-num} m n = \text{None} \Leftrightarrow \text{numeral} m \text{ AND NOT} (\text{numeral} n) = (0 :: \text{int}) \rangle
\]

**by (simp del: not-numeral-eq add: int-numeral-and-not-num split: option.split)\]

**lemma and-not-num-eq-Some-iff:**

\[
\langle \text{and-not-num} m n = \text{Some} q \leftrightarrow \text{numeral} m \text{ AND NOT} (\text{numeral} n) = (\text{numeral} q :: \text{int}) \rangle
\]

**by (simp del: not-numeral-eq add: int-numeral-and-not-num split: option.split)\]

**lemma and-minus-numerals [simp]:**

\[
\langle 1 \text{ AND} - (\text{numeral} (\text{num.Bit0} n)) = (0::\text{int}) \rangle
\]

\[
\langle 1 \text{ AND} - (\text{numeral} (\text{num.Bit1} n)) = (1::\text{int}) \rangle
\]

\[
\langle \text{numeral} m \text{ AND} - (\text{numeral} (\text{num.Bit0} n)) = (\text{case and-not-num} m (\text{Num.BitM} n) \text{ of None} \Rightarrow 0 :: \text{int} \mid \text{Some} n' \Rightarrow \text{numeral} n') \rangle
\]

\[
\langle \text{numeral} m \text{ AND} - (\text{numeral} (\text{num.Bit1} n)) = (\text{case and-not-num} m (\text{Num.Bit0} n) \text{ of None} \Rightarrow 0 :: \text{int} \mid \text{Some} n' \Rightarrow \text{numeral} n') \rangle
\]

\[
\langle - (\text{numeral} (\text{num.Bit0} n)) \text{ AND} 1 = (0::\text{int}) \rangle
\]

\[
\langle - (\text{numeral} (\text{num.Bit1} n)) \text{ AND} 1 = (1::\text{int}) \rangle
\]

\[
\langle - (\text{numeral} (\text{num.Bit0} n)) \text{ AND} \text{numeral} m = (\text{case and-not-num} m (\text{Num.BitM} n) \text{ of None} \Rightarrow 0 :: \text{int} \mid \text{Some} n' \Rightarrow \text{numeral} n') \rangle
\]

\[
\langle - (\text{numeral} (\text{num.Bit1} n)) \text{ AND} \text{numeral} m = (\text{case and-not-num} m (\text{Num.Bit0} n) \text{ of None} \Rightarrow 0 :: \text{int} \mid \text{Some} n' \Rightarrow \text{numeral} n') \rangle
\]


**lemma and-minus-minus-numerals [simp]:**

\[
\langle - (\text{numeral} m :: \text{int}) \text{ AND} - (\text{numeral} n :: \text{int}) = \text{NOT} ((\text{numeral} m - 1) \text{ OR} (\text{numeral} n - 1)) \rangle
\]

**by (simp add: minus-numeral-eq-not-sub-one)\]

**THEORY “Bit-Operations” 1503**
lemma or-not-numerals:
\[1 \text{ OR } \text{NOT}(0 :: \text{int})\]
\[1 \text{ OR } \text{NOT} \text{(numeral (Num.Bit0 n)) = NOT (numeral (Num.Bit0 n) :: int)}\]
\[1 \text{ OR } \text{NOT} \text{(numeral (Num.Bit1 n)) = NOT (numeral (Num.Bit0 n) :: int)}\]
\[\text{numeral (Num.Bit0 m) OR NOT (1 :: int) = NOT (1 :: int)}\]
\[\text{numeral (Num.Bit0 m) OR NOT (numeral (Num.Bit0 n)) = 1 + (2 :: int) \ast (numeral m OR NOT (numeral n))}\]
\[\text{numeral (Num.Bit0 m) OR NOT (numeral (Num.Bit1 n)) = (2 :: int) \ast (numeral m OR NOT (numeral n))}\]
\[\text{numeral (Num.Bit1 m) OR NOT (numeral (Num.Bit0 n)) = 1 + (2 :: int) \ast (numeral m OR NOT (numeral n))}\]
\[\text{by (simp-all add: bit-0 bit-simps bit-Suc bit-numeral-rec sub-inc-One-eq split: nat.split)}\]

fun or-not-num-neg :: \(\text{num \Rightarrow num \Rightarrow num}\)

where
\[\text{or-not-num-neg num.One num.One = num.One}\]
\[\text{or-not-num-neg num.One (num.Bit0 m) = num.Bit1 m}\]
\[\text{or-not-num-neg num.One (num.Bit1 m) = num.Bit1 m}\]
\[\text{or-not-num-neg (num.Bit0 n) num.One num.One = num.Bit0 num.One}\]
\[\text{or-not-num-neg (num.Bit0 n) (num.Bit0 m) = Num.BitM (or-not-num-neg n m)}\]
\[\text{or-not-num-neg (num.Bit0 n) (num.Bit1 m) = num.Bit0 (or-not-num-neg n m)}\]
\[\text{or-not-num-neg (num.Bit1 m) num.One num.One = num.One}\]
\[\text{or-not-num-neg (num.Bit1 m) (num.Bit0 m) = Num.BitM (or-not-num-neg n m)}\]
\[\text{or-not-num-neg (num.Bit1 m) (num.Bit1 m) = Num.BitM (or-not-num-neg n m)}\]

lemma int-numeral-or-not-num-neg:
\[\text{numeral n OR NOT (numeral n :: int) = - numeral (or-not-num-neg m n)}\]
\[\text{by (induction m n rule: or-not-num-neg_induct) (simp-all del: not-numeral-eq not-one-eq add: or-not-numerals, simp-all)}\]

lemma int-numeral-or-not-num-neg:
\[\text{NOT (numeral m) OR (numeral n :: int) = - numeral (or-not-num-neg n m)}\]
\[\text{using int-numeral-or-not-num-neg [of n m] by (simp add: ac-simps)}\]

lemma numeral-or-not-num-eq:
\[\text{numeral (or-not-num-neg m n) = - (numeral m OR NOT (numeral n :: int))}\]
\[\text{using int-numeral-or-not-num-neg [of m n] by simp}\]

lemma or-minus-numerals [simp]:
\[\text{1 OR - (numeral (num.Bit0 n)) = - (numeral (or-not-num-neg num.One}
THEORY “Bit-Operations” 1505

(Num.BitM n) :: int

k OR (numeral (Num.Bit1 n)) = - (numeral (Num.Bit1 n) :: int)

numeral n OR (numeral (Num.Bit0 n)) = - (numeral (num-or-not-num-neg m
(Num.BitM n)) :: int)

numeral m OR (numeral (Num.Bit1 n)) = - (numeral (or-not-num-neg m
(Num.Bit0 n)) :: int)

(- (numeral (Num.Bit0 n)) OR 1) = - (numeral (or-not-num-neg num.One
(Num.BitM n)) :: int)

(- (numeral (Num.Bit1 n)) OR 1) = - (numeral (Num.Bit1 n) :: int)

(- (numeral (Num.Bit0 n)) OR numeral m) = - (numeral (or-not-num-neg m
(Num.BitM n)) :: int)

(- (numeral (Num.Bit1 n)) OR numeral m) = - (numeral (num-or-not-num-neg m
(Num.Bit0 n)) :: int)

by (simp-all only; or-commute [of - 1] or-commute [of - numeral m]
minus-numeral-eq-not-sub-one or-not-numerals
numeral-or-not-num-eq arith-simps minus-minus-numeral-One)

lemma or-minus-minus-numerals [simp]:

(- (numeral m :: int) OR - (numeral n :: int) = NOT ((numeral m - 1) AND
(numeral n - 1))

by (simp add: minus-numeral-eq-not-sub-one)

lemma xor-minus-numerals [simp]:

(- numeral n XOR k = NOT (neg-numeral-class.sub n num.One XOR k)

k XOR - numeral n = NOT (k XOR (neg-numeral-class.sub n num.One)) for
k :: int

by (simp-all add: minus-numeral-eq-not-sub-one)

definition take-bit-num :: (nat ⇒ num ⇒ num option)

where (take-bit-num n m =
(if take-bit (numeral m :: nat) = 0 then None else Some (num-of-nat (take-bit
n (numeral m :: nat))))

lemma take-bit-num-simps:

(take-bit-num 0 m = None)

(take-bit-num (Suc n) Num.One =
Some Num.One)

(take-bit-num (Suc n) (Num.Bit0 m) =
(case take-bit-num n m of None ⇒ None | Some q ⇒ Some (Num.Bit0 q))

(take-bit-num (Suc n) (Num.Bit1 m) =
Some (case take-bit-num n m of None ⇒ Num.One | Some q ⇒ Num.Bit1 q)

(take-bit-num (numeral r) Num.One =
Some Num.One)

(take-bit-num (numeral r) (Num.Bit0 m) =
(case take-bit-num (pred-numeral r) m of None ⇒ None | Some q ⇒ Some
(Num.Bit0 q))

(take-bit-num (numeral r) (Num.Bit1 m) =
Some (case take-bit-num (pred-numeral r) m of None ⇒ Num.One | Some q ⇒
Num.Bit1 q))
by (auto simp add: take-bit-num-def ac-simps mult-2 num-of-nat-double
   take-bit-Suc-bit0 take-bit-Suc-bit1 take-bit-numeral-bit0 take-bit-numeral-bit1)

lemma take-bit-num-code [code]:
— Ocaml-style pattern matching is more robust wrt. different representations of nat
<take-bit-num n m = (case (n, m)
of (0, -) ⇒ None
  | (Suc n, Num.One) ⇒ Some Num.One
  | (Suc n, Num.Bit0 m) ⇒ (case take-bit-num n m of None ⇒ None | Some q ⇒ Some (Num.Bit0 q))
  | (Suc n, Num.Bit1 m) ⇒ Some (case take-bit-num n m of None ⇒ Num.One
  | Some q ⇒ Num.Bit1 q))>
by (cases n; cases m) (simp-all add: take-bit-num-simps)

context semiring-bit-operations begin

lemma take-bit-num-eq-None-imp:
<take-bit m (numeral n) = 0> if <take-bit-num m n = None>
proof –
from that have <take-bit m (numeral n :: nat) = 0>
by (simp add: take-bit-num-def split: if-splits)
then have <of-nat (take-bit m (numeral n)) = of-nat 0>
by simp
then show ?thesis
by (simp add: of-nat-take-bit)
qed

lemma take-bit-num-eq-None-imp:
<take-bit m (numeral n) = numeral q> if <take-bit-num m n = Some q>
proof –
from that have <take-bit m (numeral n :: nat) = numeral q>
by (auto simp add: take-bit-num-def Num.numeral-num-of-nat-unfold split: if-splits)
then have <of-nat (take-bit m (numeral n)) = of-nat (numeral q)>
by simp
then show ?thesis
by (simp add: of-nat-take-bit)
qed

lemma take-bit-numeral-numeral:
<take-bit (numeral m) (numeral n) =
  (case take-bit-num (numeral m) n of None ⇒ 0 | Some q ⇒ numeral q)>
by (auto split: option.split dest: take-bit-num-eq-None-imp take-bit-num-eq-None-imp)
end

lemma take-bit-numeral-minus-numeral-int:
68.7 Symbolic computations for code generation

**lemma** bit-int-code [code]:

\[ \langle \text{bit} (0 :: \text{int}) \rangle \quad n \quad \longleftrightarrow \quad \text{False} \]
\[ \langle \text{bit} (\text{Int.Neg} \text{ num}.\text{One}) \rangle \quad n \quad \longleftrightarrow \quad \text{True} \]
\[ \langle \text{bit} (\text{Int.Pos} \text{ num}.\text{One}) \rangle \quad 0 \quad \longleftrightarrow \quad \text{True} \]
\[ \langle \text{bit} (\text{Int.Pos} (\text{num}.\text{Bit0} m)) \rangle \quad 0 \quad \longleftrightarrow \quad \text{False} \]
\[ \langle \text{bit} (\text{Int.Pos} (\text{num}.\text{Bit1} m)) \rangle \quad 0 \quad \longleftrightarrow \quad \text{True} \]
\[ \langle \text{bit} (\text{Int.Neg} (\text{num}.\text{Bit0} m)) \rangle \quad 0 \quad \longleftrightarrow \quad \text{False} \]
\[ \langle \text{bit} (\text{Int.Neg} (\text{num}.\text{Bit1} m)) \rangle \quad 0 \quad \longleftrightarrow \quad \text{True} \]
\[ \langle \text{bit} (\text{Int.Pos} \text{ num}.\text{One}) \rangle \quad (\text{Suc} n) \quad \longleftrightarrow \quad \text{False} \]
\[ \langle \text{bit} (\text{Int.Pos} (\text{num}.\text{Bit0} m)) \rangle \quad (\text{Suc} n) \quad \longleftrightarrow \quad \text{bit} (\text{Int.Pos} m) \quad n \]
\[ \langle \text{bit} (\text{Int.Pos} (\text{num}.\text{Bit1} m)) \rangle \quad (\text{Suc} n) \quad \longleftrightarrow \quad \text{bit} (\text{Int.Pos} m) \quad n \]
\[ \langle \text{bit} (\text{Int.Neg} (\text{num}.\text{Bit0} m)) \rangle \quad (\text{Suc} n) \quad \longleftrightarrow \quad \text{bit} (\text{Int.Neg} m) \quad n \]
\[ \langle \text{bit} (\text{Int.Neg} (\text{num}.\text{Bit1} m)) \rangle \quad (\text{Suc} n) \quad \longleftrightarrow \quad \text{bit} (\text{Int.Neg} (\text{Num.inc} m)) \quad n \]

by (simp-all add: Num.add-One bit-0 bit-Suc)

**lemma** not-int-code [code]:

\[ \langle \text{NOT} (0 :: \text{int}) \rangle = -1 \]
\[ \langle \text{NOT} (\text{Int.Pos} n) \rangle = \text{Int.Neg} (\text{Num.inc} n) \]
THEORY "Bit-Operations"

\[ \text{NOT} \ (\text{Int.Neg} \ n) = \text{Num.sub} \ n \ \text{num.One} \]
by (simp-all add: Num.add-One not-int-def)

fun and-num :: \( \text{num} \Rightarrow \text{num} \Rightarrow \text{num} \ option \)
where
\[ \begin{align*}
\text{and-num} \ \text{num.One} \ \text{num.One} &= \text{Some} \ \text{num.One} \\
\text{and-num} \ \text{num.One} \ (\text{num.Bit0} \ n) &= \text{None} \\
\text{and-num} \ (\text{num.Bit0} \ m) \ \text{num.One} &= \text{None} \\
\text{and-num} \ (\text{num.Bit0} \ m) \ (\text{num.Bit0} \ n) &= \text{map-option} \ \text{num.Bit0} \ (\text{and-num} \ m \ n) \\
\text{and-num} \ (\text{num.Bit0} \ m) \ (\text{num.Bit1} \ n) &= \text{map-option} \ \text{num.Bit0} \ (\text{and-num} \ m \ n) \\
\text{and-num} \ (\text{num.Bit1} \ m) \ \text{num.One} &= \text{Some} \ \text{num.One} \\
\text{and-num} \ (\text{num.Bit1} \ m) \ (\text{num.Bit0} \ n) &= \text{map-option} \ \text{num.Bit0} \ (\text{and-num} \ m \ n) \\
\text{and-num} \ (\text{num.Bit1} \ m) \ (\text{num.Bit1} \ n) &= (\text{case and-num} \ m \ n \ \text{of None} \Rightarrow \text{Some} \ \text{num.One} | \ \text{Some} \ n' \Rightarrow \text{Some} \ (\text{num.Bit1} \ n')) \\
\end{align*} \]

context linordered-euclidean-semiring-bit-operations
begin

lemma numeral-and-num:
\[ \text{numeral} \ m \ \text{AND} \ \text{numeral} \ n = (\text{case and-num} \ m \ n \ \text{of None} \Rightarrow 0 | \ \text{Some} \ n' \Rightarrow \text{numeral} \ n') \]
by (induction m n rule: and-num.induct) (simp-all add: split: option.split)

lemma and-num-eq-None-iff:
\[ \text{and-num} \ m \ n = \text{None} \longleftrightarrow \text{numeral} \ m \ \text{AND} \ \text{numeral} \ n = 0 \]
by (simp add: numeral-and-num split: option.split)

lemma and-num-eq-Some-iff:
\[ \text{and-num} \ m \ n = \text{Some} \ q \longleftrightarrow \text{numeral} \ m \ \text{AND} \ \text{numeral} \ n = \text{numeral} \ q \]
by (simp add: numeral-and-num split: option.split)

end

lemma and-int-code [code]:
fixes i j :: int
shows
\[ \begin{align*}
0 \ \text{AND} \ j &= 0, \\
i \ \text{AND} \ 0 &= 0, \\
(\text{Int.Pos} \ n \ \text{AND} \ \text{Int.Pos} \ m) &= (\text{case and-num} \ m \ n \ \text{of None} \Rightarrow 0 | \ \text{Some} \ n' \Rightarrow \text{Int.Pos} \ n') \\
(\text{Int.Pos} \ n \ \text{AND} \ \text{Int.Neg} \ m) &= \text{NOT} \ (\text{Num.sub} \ n \ \text{num.One} \ OR \ \text{Num.sub} \ m \ \text{num.One}) \\
(\text{Int.Pos} \ n \ \text{AND} \ \text{Int.Neg} \ \text{num.One}) &= \text{Int.Pos} \ n \\
(\text{Int.Pos} \ n \ \text{AND} \ \text{Int.Neg} \ (\text{num.Bit0} \ m) \ \text{m}) &= \text{Num.sub} \ (\text{or-not-num-neg} \ (\text{Num.BitM} \ m) \ m) \ \text{num.One} \\
(\text{Int.Pos} \ n \ \text{AND} \ \text{Int.Neg} \ (\text{num.Bit1} \ m) \ \text{m}) &= \text{Num.sub} \ (\text{or-not-num-neg} \ (\text{Num.Bit0} \ m) \ m) \ \text{num.One} \\
(\text{Int.Neg} \ \text{num.One} \ \text{AND} \ \text{Int.Pos} \ m) &= \text{Int.Pos} \ m \\
(\text{Int.Neg} \ (\text{num.Bit0} \ n) \ \text{AND} \ \text{Int.Pos} \ m) &= \text{Num.sub} \ (\text{or-not-num-neg} \ (\text{Num.BitM} \ n) \ m) \ \text{num.One} \\
\end{align*} \]
n) m) num.One
   <Int.Neg (num.Bit1 n) AND Int.Pos m = Num.sub (or-not-num-neg (num.Bit0 n) m) num.One>
   apply (auto simp add: and-num-eq-None-iff [where ?'a = int] and-num-eq-Some-iff 
     split: option.split)
   apply (simp-all only: sub-one-eq-not-neg numeral-or-not-num-eq minus-minus
     ac-simps)
   done

context linordered-euclidean-semiring-bit-operations
begin

fun or-num :: (num ⇒ num ⇒ num)
where
  <or-num num.One num.One = num.One>
| <or-num num.One (num.Bit0 n) = num.Bit1 n>
| <or-num num.One (num.Bit1 n) = num.Bit1 n>
| <or-num (num.Bit0 m) num.One = num.Bit1 m>
| <or-num (num.Bit0 m) (num.Bit0 n) = num.Bit0 (or-num m n)>
| <or-num (num.Bit0 m) (num.Bit1 n) = num.Bit1 (or-num m n)>
| <or-num (num.Bit1 m) num.One = num.Bit1 m>
| <or-num (num.Bit1 m) (num.Bit0 n) = num.Bit1 (or-num m n)>
| <or-num (num.Bit1 m) (num.Bit1 n) = num.Bit1 (or-num m n)>

lemma numeral-or-num:
  <numeral m OR numeral n = numeral (or-num m n)>
  by (induction m n rule: or-num.induct) simp-all

lemma numeral-or-num-eq:
  <numeral (or-num m n) = numeral m OR numeral n>
  by (simp add: numeral-or-num)

end

lemma or-int-code [code]:
  fixes i j :: int shows
  <0 OR j = j>
| i OR 0 = i>
| Int.Pos n OR Int.Pos m = Int.Pos (or-num n m)>
| Int.Neg n OR Int.Neg m = NOT (Num.sub n num.One AND Num.sub m num.One)>
| Int.Pos n OR Int.Pos num.One = Int.Pos num.One>
| Int.Pos n OR Int.Neg (num.Bit0 m) = (case and-not-num (Num.BitM m) n of None ⇒ -1 | Some n' ⇒ Int.Neg (Num.inc n'))>
| Int.Pos n OR Int.Neg (num.Bit1 m) = (case and-not-num (num.Bit0 m) n of None ⇒ -1 | Some n' ⇒ Int.Neg (Num.inc n'))>
\[ \langle \text{Int.Neg } \text{num}.\text{One} \text{ OR } \text{Int.Pos } m = \text{Int.Neg } \text{num}.\text{One} \rangle \]
\[ \langle \text{Int.Neg } (\text{num}.\text{Bit}0 n) \text{ OR } \text{Int.Pos } m = (\text{case and-not-num } (\text{Num.BitM } n) \text{ m of None } \Rightarrow -1 \mid \text{Some } n' \Rightarrow \text{Int.Neg } (\text{Num.inc } n')) \rangle \]
\[ \langle \text{Int.Neg } (\text{num}.\text{Bit}1 n) \text{ OR } \text{Int.Pos } m = (\text{case and-not-num } (\text{num}.\text{Bit}0 n) \text{ m of None } \Rightarrow -1 \mid \text{Some } n' \Rightarrow \text{Int.Neg } (\text{Num.inc } n')) \rangle \]
\[
\text{apply (auto simp add: numeral-or-num-eq split: option.splits)}
\]
\[
\text{apply (simp-all only: and-not-num-eq-None-iff and-not-num-eq-Some-iff and-not-numerals numeral-or-not-num-eq or-eq-not-not-and bit.double-compl ac-simps flip: numeral-eq-iff [where } \forall a = \text{int}]})
\]
\[
\text{apply simp-all }
\]
\[
\text{done}
\]

\begin{lstlisting}[language=Haskell]
fun xor-num :: \text{num }\Rightarrow \text{num }\Rightarrow \text{num option}
where
  \langle xor-num \text{num}.\text{One} \text{num}.\text{One} = \text{None} \rangle
| \langle xor-num \text{num}.\text{One} (\text{num}.\text{Bit}0 n) = \text{Some } (\text{num}.\text{Bit}1 n) \rangle
| \langle xor-num \text{num}.\text{One} (\text{num}.\text{Bit}1 n) = \text{Some } (\text{num}.\text{Bit}0 n) \rangle
| \langle xor-num (\text{num}.\text{Bit}0 \text{m}) \text{num}.\text{One} = \text{Some } (\text{num}.\text{Bit}1 \text{m}) \rangle
| \langle xor-num (\text{num}.\text{Bit}0 \text{m}) (\text{num}.\text{Bit}0 n) = \text{map-option } \text{num}.\text{Bit}0 (\text{xor-num } m \text{n}) \rangle
| \langle xor-num (\text{num}.\text{Bit}0 \text{m}) (\text{num}.\text{Bit}1 n) = \text{Some } (\text{case xor-num } m \text{n of None } \Rightarrow \text{num}.\text{One } \mid \text{Some } n' \Rightarrow \text{num}.\text{Bit}1 n') \rangle
| \langle xor-num (\text{num}.\text{Bit}1 \text{m}) \text{num}.\text{One} = \text{Some } (\text{num}.\text{Bit}0 \text{m}) \rangle
| \langle xor-num (\text{num}.\text{Bit}1 \text{m}) (\text{num}.\text{Bit}0 n) = \text{Some } (\text{case xor-num } m \text{n of None } \Rightarrow \text{num}.\text{One } \mid \text{Some } n' \Rightarrow \text{num}.\text{Bit}1 n') \rangle
| \langle xor-num (\text{num}.\text{Bit}1 \text{m}) (\text{num}.\text{Bit}1 n) = \text{map-option } \text{num}.\text{Bit}0 (\text{xor-num } m \text{n}) \rangle
\end{lstlisting}

context linordered-euclidean-semiring-bit-operations
begin

lemma numeral-xor-num:
  \langle numeral m XOR numeral n = (case xor-num m n of None \Rightarrow 0 \mid Some n' \Rightarrow numeral n') \rangle
  by (induction m n rule: xor-num.induct) (simp-all split: option.split)

lemma xor-num-eq-None-iff:
  \langle xor-num m n = None \longleftrightarrow numeral m XOR numeral n = 0 \rangle
  by (simp add: numeral-xor-num split: option.split)

lemma xor-num-eq-Some-iff:
  \langle xor-num m n = Some q \longleftrightarrow numeral m XOR numeral n = numeral q \rangle
  by (simp add: numeral-xor-num split: option.split)

end

lemma xor-int-code [code]:
  fixes i j :: int
  shows \langle 0 XOR j = j \rangle
  \langle i XOR 0 = i \rangle
58.8 More properties

lemma take-bit-eq-mask-iff:
\[ \text{take-bit } n k = \text{mask } n \leftrightarrow \text{take-bit } n (k + 1) = 0, \quad (\text{is } ?P \leftrightarrow ?Q) \]

for \( k :: \text{int} \)

proof

assume ?P

then have \( \text{take-bit } n (\text{take-bit } n k + \text{take-bit } n 1) = 0 \)

by (simp add: mask-eq-exp-minus-1 take-bit-eq-0-iff)

then show ?Q

by (simp only: take-bit-add)

next

assume ?Q

then have \( \text{take-bit } n (k + 1) - 1 = -1 \)

by simp

then have \( \text{take-bit } n (\text{take-bit } n (k + 1) - 1) = \text{take-bit } n (-1) \)

by simp

moreover have \( \text{take-bit } n (\text{take-bit } n (k + 1) - 1) = \text{take-bit } n k \)

by (simp add: take-bit-eq-mod mod-simps)

ultimately show ?P

by simp

qed

lemma take-bit-eq-mask-iff-exp-ded:
THEORY “Bit-Operations”

\[
\text{take-bit } n \ k = \text{mask } n \longleftrightarrow 2^{-n} \text{ dvd } k + 1
\]

for \( k :: \text{int} \)
by (simp add: take-bit-eq-mask-iff flip: take-bit-eq-0-iff)

\[68.9\] Bit concatenation

definition concat-bit :: \( \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \)
where \( \text{concat-bit } n \ k \ l = \text{take-bit } n \ k \ \text{OR} \ \text{push-bit } n \ l \)

lemma bit-concat-bit-iff [bit-simps]:
\( \text{bit } (\text{concat-bit } m \ k \ l) n \longleftrightarrow n < m \land \text{bit } k \ n \lor m \leq n \land \text{bit } l \ (n - m) \)
by (simp add: concat-bit-def bit-or-iff bit-and-iff bit-take-bit-iff bit-push-bit-iff ac-simps)

lemma concat-bit-eq:
\( \text{concat-bit } n \ k \ l = \text{take-bit } n \ k + \text{push-bit } n \ l \)

proof –
  have \( \text{take-bit } n \ k \ \text{AND} \ \text{push-bit } n \ l = 0 \)
  by (simp add: bit-eq-iff bit-simps)
  then show ?thesis
  by (simp add: bit-eq-iff bit-simps disjunctive-add-eq-or)
qed

lemma concat-bit-0 [simp]:
\( \text{concat-bit } 0 \ k \ l = l \)
by (simp add: concat-bit-def)

lemma concat-bit-Suc:
\( \text{concat-bit } (\text{Suc } n) \ k \ l = k \ \text{mod } 2 + 2 * \text{concat-bit } n \ (k \ \text{div } 2) \ l \)
by (simp add: concat-bit-eq take-bit-Suc push-bit-double)

lemma concat-bit-of-zero-1 [simp]:
\( \text{concat-bit } n \ 0 \ l = \text{push-bit } n \ l \)
by (simp add: concat-bit-def)

lemma concat-bit-of-zero-2 [simp]:
\( \text{concat-bit } n \ k \ 0 = \text{take-bit } n \ k \)
by (simp add: concat-bit-def take-bit-eq-mask)

lemma concat-bit-nonnegative-iff [simp]:
\( \text{concat-bit } n \ k \ l \geq 0 \longleftrightarrow l \geq 0 \)
by (simp add: concat-bit-def)

lemma concat-bit-negative-iff [simp]:
\( \text{concat-bit } n \ k \ l < 0 \longleftrightarrow l < 0 \)
by (simp add: concat-bit-def)

lemma concat-bit-assoc:
\( \text{concat-bit } n \ k \ (\text{concat-bit } m \ l \ r) = \text{concat-bit } (m + n) \ (\text{concat-bit } n \ k \ l) \ r \)
by (rule bit-eqI) (auto simp add: bit-concat-bit-iff ac-simps)

lemma concat-bit-assoc-sym:
\[(\text{concat-bit} \ m \ (\text{concat-bit} \ n \ k \ l) = \text{concat-bit} \ (\text{min} \ m \ n) \ k \ (\text{concat-bit} \ (m - n) \ l \ r))\]
by (rule bit-eqI) (auto simp add: bit-concat-bit-iff ac-simps min-def)

lemma concat-bit-eq-iff:
\[\text{concat-bit} \ n \ k \ l = \text{concat-bit} \ n \ r \ s \iff \text{take-bit} \ n \ k = \text{take-bit} \ n \ r \land l = s\]
(is \(?P \iff \?Q\) )
proof
  assume \(?Q\)
  then show \(?P\)
  by (simp add: concat-bit-def)
next
  assume \(?P\)
  then have \(*\):
    \[\text{bit} \ (\text{concat-bit} \ n \ k \ l) = \text{bit} \ (\text{concat-bit} \ n \ r \ s) \] for \(m\)
    by (simp add: bit-eq-iff)
  have \(\text{take-bit} \ n \ k = \text{take-bit} \ n \ r\)
  proof (rule bit-eqI)
    fix \(m\)
    from \(*\) [of \(m\)]
    show \(\text{bit} \ (\text{take-bit} \ n \ k) = \text{bit} \ (\text{take-bit} \ n \ r)\)
    by (auto simp add: bit-take-bit-iff bit-concat-bit-iff)
  qed
moreover have \(\text{push-bit} \ n \ l = \text{push-bit} \ n \ s\)
  proof (rule bit-eqI)
    fix \(m\)
    from \(*\) [of \(m\)]
    show \(\text{bit} \ (\text{push-bit} \ n \ l) = \text{bit} \ (\text{push-bit} \ n \ s)\)
    by (auto simp add: bit-push-bit-iff bit-concat-bit-iff)
  qed
ultimately show \(?Q\)
  by (simp add: concat-bit-def)
qed

lemma take-bit-concat-bit-eq:
\[\text{take-bit} \ m \ (\text{concat-bit} \ n \ k \ l) = \text{concat-bit} \ (\text{min} \ m \ n) \ k \ (\text{take-bit} \ (m - n) \ l)\]
by (rule bit-eqI)
  (auto simp add: bit-take-bit-iff bit-concat-bit-iff min-def)

lemma concat-bit-take-bit-eq:
\[\text{concat-bit} \ n \ (\text{take-bit} \ n \ b) = \text{concat-bit} \ n \ b\]
by (simp add: concat-bit-def [abs-def])
68.10 Taking bits with sign propagation

context ring-bit-operations
begin

definition signed-take-bit :: \( \text{nat} \Rightarrow 'a \Rightarrow 'a \)
where \( \text{signed-take-bit} \ n \ a = \text{take-bit} \ n \ a \ \text{OR} \ \text{of-bool} \ (\text{bit} \ a \ n) \ \text{NOT} \ \text{mask} \ n \)\)

lemma signed-take-bit-eq-if-positive:
\( \text{signed-take-bit} \ n \ a = \text{take-bit} \ n \ a \) if \( \neg \text{bit} \ a \ n \)
using that by (simp add: signed-take-bit-def)

lemma signed-take-bit-eq-if-negative:
\( \text{signed-take-bit} \ n \ a = \text{take-bit} \ n \ a \ \text{OR} \ \text{NOT} \ \text{mask} \ n \) if \( \text{bit} \ a \ n \)
using that by (simp add: signed-take-bit-def)

lemma even-signed-take-bit-iff:
\( \text{even} \ (\text{signed-take-bit} \ m \ a) \iff \text{even} \ a \)
by (auto simp add: bit-0 signed-take-bit-def even-or-iff even-mask-iff bit-double-iff)

lemma bit-signed-take-bit-iff [bit-simps]:
\( \text{bit} \ (\text{signed-take-bit} \ m \ a) \ n \iff \text{possible-bit} \ TYPE(\text{'}a\text{'} \ n \ \land \text{bit} \ a \ (\text{min} \ m \ n)) \)
by (simp add: signed-take-bit-def bit-take-bit-iff bit-or-iff bit-not-iff bit-mask-iff min-def not-le
(blast dest: bit-imp-possible-bit)

lemma signed-take-bit-0 [simp]:
\( \text{signed-take-bit} \ 0 \ a = - (a \ mod \ 2) \)
by (simp add: bit-0 signed-take-bit-def odd-iff-mod-2-eq-one)

lemma signed-take-bit-Suc:
\( \text{signed-take-bit} \ (\text{Suc} \ n) \ a = a \ mod \ 2 + 2 \ast \text{signed-take-bit} \ n \ (a \ div \ 2) \)
by (simp add: bit-eq-iff bit-sam-mult-2-cases bit-simps bit-0 possible-bit-less-imp flip: bit-Suc min-Suc-Suc)

lemma signed-take-bit-of-0 [simp]:
\( \text{signed-take-bit} \ n \ 0 = 0 \)
by (simp add: signed-take-bit-def)

lemma signed-take-bit-of-minus-1 [simp]:
\( \text{signed-take-bit} \ n \ (-1) = -1 \)
by (simp add: signed-take-bit-def mask-eq-exp-minus-1 possible-bit-def)

lemma signed-take-bit-Suc-1 [simp]:
\( \text{signed-take-bit} \ (\text{Suc} \ n) \ 1 = 1 \)
by (simp add: signed-take-bit-Suc)

lemma signed-take-bit-numeral-of-1 [simp]:
\( \text{signed-take-bit} \ (\text{numeral} \ k) \ 1 = 1 \)
by \((\text{simp add: } \text{bit-1-iff } \text{signed-take-bit-if-positive})\)

**Lemma** \(\text{signed-take-bit-rec}:

\[\text{signed-take-bit } n \ a = (\text{if } n = 0 \text{ then } (a \mod 2) \text{ else } a \mod 2 + 2 \ast \text{signed-take-bit } (n - 1) (a \div 2))\]

by \((\text{cases } n)\) \((\text{simp-all add: } \text{signed-take-bit-Suc})\)

**Lemma** \(\text{signed-take-bit-eq-iff-take-bit-eq}:

\[\text{signed-take-bit } n \ a = \text{signed-take-bit } n \ b \iff \text{take-bit } (\text{Suc } n) \ a = \text{take-bit } (\text{Suc } n) \ b\]

proof –

have \(\text{bit } (\text{signed-take-bit } n \ a) = \text{bit } (\text{signed-take-bit } n \ b) \iff \text{bit } (\text{take-bit } (\text{Suc } n) \ b)\)

by \((\text{simp add: } \text{fun-eq-iff } \text{bit-signed-take-bit-iff } \text{bit-take-bit-iff } \text{not-le } \text{less-Suc-eq-ле } \text{min-def})\)

(use \text{bit-imp-possible-bit} in \text{fastforce})

then show \(?\text{thesis}\)

by \((\text{auto simp add: } \text{fun-eq-iff intro: } \text{bit-eqI})\)

qed

**Lemma** \(\text{signed-take-bit-signed-take-bit}\)

\[[\text{simp}]:\]

\[\text{signed-take-bit } m \ (\text{signed-take-bit } n \ a) = \text{signed-take-bit } (\text{min } m \ n) \ a\]

by \((\text{auto simp add: } \text{bit-eq-iff } \text{bit-simps } \text{ac-simps})\)

**Lemma** \(\text{signed-take-bit-take-bit}:

\[\text{signed-take-bit } m \ (\text{take-bit } n \ a) = (\text{if } n \leq m \text{ then } \text{take-bit } n \text{ else } \text{signed-take-bit } m) \ a\]

by \((\text{rule } \text{bit-eqI})\) \((\text{auto simp add: } \text{bit-signed-take-bit-iff } \text{bit-take-bit-iff})\)

**Lemma** \(\text{take-bit-signed-take-bit}:

\[\text{take-bit } m \ (\text{signed-take-bit } n \ a) = \text{take-bit } m \ a \iff \langle m \leq \text{Suc } n \rangle\]

using that by \((\text{rule } \text{le-SucE}; \text{intro } \text{bit-eqI})\)

\((\text{auto simp add: } \text{bit-take-bit-iff } \text{bit-signed-take-bit-iff } \text{min-def } \text{less-Suc-eq})\)

**Lemma** \(\text{signed-take-bit-eq-take-bit-add}:

\[\text{signed-take-bit } n \ k = \text{take-bit } (\text{Suc } n) \ k + \text{NOT } (\text{mask } (\text{Suc } n)) \ast \text{of-bool } (\text{bit } k \ n)\]

proof \((\text{cases } (\text{bit } k \ n))\)

case False

case False

show \(?\text{thesis}\)

by \((\text{rule } \text{bit-eqI})\) \((\text{simp add: } \text{False } \text{bit-simps } \text{min-def } \text{less-Suc-eq})\)

next

case True

have \(\langle \text{signed-take-bit } n \ k = \text{take-bit } (\text{Suc } n) \ k \text{ OR } \text{NOT } (\text{mask } (\text{Suc } n))\rangle\)

by \((\text{rule } \text{bit-eqI})\) \((\text{auto simp add: } \text{bit-signed-take-bit-iff } \text{bit-take-bit-iff } \text{bit-or-iff } \text{bit-not-iff } \text{bit-mask-iff } \text{less-Suc-eq-True})\)

also have \(\langle \ldots = \text{take-bit } (\text{Suc } n) \ k + \text{NOT } (\text{mask } (\text{Suc } n))\rangle\)

by \((\text{simp add: } \text{disjunctive-add-eq-or } \text{bit-eq-iff } \text{bit-simps})\)

finally show \(?\text{thesis}\)
**THEORY “Bit-Operations”**

by (simp add: True)
qed

**lemma signed-take-bit-eq-take-bit-minus:**
\[ \text{signed-take-bit } n \ k = \text{take-bit } (\text{Suc } n) \ k - 2 \cdot (\text{Suc } n) * \text{of-bool } (\text{bit } k n) \]
by (simp add: signed-take-bit-eq-take-bit-add flip: minus-exp-eq-not-mask)
end

Modulus centered around 0

**lemma signed-take-bit-eq-concat-bit:**
\[ \text{signed-take-bit } n \ k = \text{concat-bit } n \ k \ (- \text{of-bool } (\text{bit } k n)) \]
by (simp add: concat-bit-def signed-take-bit-def)

**lemma signed-take-bit-add:**
\[ \text{signed-take-bit } n \ ((\text{signed-take-bit } n \ k + \text{signed-take-bit } n \ l)) = \text{signed-take-bit } n \ (k + l) \]
for \( k \ l :: \text{int} \)
proof –
have \( \text{take-bit } (\text{Suc } n) \)
\( (\text{take-bit } (\text{Suc } n) \ (\text{signed-take-bit } n \ k) + \text{take-bit } (\text{Suc } n) \ (\text{signed-take-bit } n \ l)) = \text{take-bit } (\text{Suc } n) \ (k + l) \)
by (simp add: take-bit-signed-take-bit take-bit-add)
then show ?thesis
by (simp only: signed-take-bit-eq-iff-take-bit-eq take-bit-add)
qed

**lemma signed-take-bit-diff:**
\[ \text{signed-take-bit } n \ ((\text{signed-take-bit } n \ k - \text{signed-take-bit } n \ l)) = \text{signed-take-bit } n \ (k - l) \]
for \( k \ l :: \text{int} \)
proof –
have \( \text{take-bit } (\text{Suc } n) \)
\( (\text{take-bit } (\text{Suc } n) \ (\text{signed-take-bit } n \ k) - \text{take-bit } (\text{Suc } n) \ (\text{signed-take-bit } n \ l)) = \text{take-bit } (\text{Suc } n) \ (k - l) \)
by (simp add: take-bit-signed-take-bit take-bit-diff)
then show ?thesis
by (simp only: signed-take-bit-eq-iff-take-bit-eq take-bit-diff)
qed

**lemma signed-take-bit-minus:**
\[ \text{signed-take-bit } n \ (- \text{signed-take-bit } n \ k) = \text{signed-take-bit } n \ (- \ k) \]
for \( k :: \text{int} \)
proof –
have \( \text{take-bit } (\text{Suc } n) \)
\( (- \text{take-bit } (\text{Suc } n) \ (\text{signed-take-bit } n \ k)) = \text{take-bit } (\text{Suc } n) \ (- \ k) \)
by (simp add: take-bit-signed-take-bit take-bit-minus)
then show ?thesis
  by (simp only: signed-take-bit-eq-iff-take-bit-eq take-bit-minus)
qed

lemma signed-take-bit-mult:
  \langle \text{signed-take-bit } n \text{ } (\text{signed-take-bit } n \text{ } k \text{ } \ast \text{ } \text{signed-take-bit } n \text{ } l) = \text{signed-take-bit } n \text{ } (k \text{ } \ast \text{ } l) \rangle
  for \text{k l :: int}
proof
  have \langle \text{take-bit } (\text{Suc } n) \text{ } (\text{signed-take-bit } n \text{ } k) \text{ } \ast \text{ } \text{take-bit } (\text{Suc } n) \text{ } (\text{signed-take-bit } n \text{ } l) = \text{take-bit } (\text{Suc } n) \text{ } (k \text{ } \ast \text{ } l) \rangle
    by (simp add: take-bit-signed-take-bit take-bit-mult)
  then show ?thesis
    by (simp only: signed-take-bit-eq-iff-take-bit-eq take-bit-mult)
qed

lemma signed-take-bit-eq-take-bit-shift:
  \langle \text{signed-take-bit } n \text{ } k = \text{take-bit } (\text{Suc } n) \text{ } (k + 2 ^ n) \text{ } \ast \text{ } \text{NOT} \text{ } \text{mask } n \rangle
  for \text{k :: int}
proof
  have \langle \text{take-bit } n \text{ } k \text{ } \& \text{ } \& \text{ } \text{2 } ^ \text{n } = \text{0} \rangle
    by (rule bit-eqI) (simp add: bit-simps)
  then have * \langle \text{take-bit } n \text{ } k \text{ } \or \text{ } \text{take-bit } n \text{ } k \text{ } \text{OF-BOOL} \text{ } \text{bit } k \text{ } n \text{ } \text{+} \text{ } \text{take-bit } n \text{ } k \rangle
    by (simp add: disjunctive-add-eq-or)
  have \langle \text{take-bit } n \text{ } k \text{ } \text{+} \text{ } \text{2 } ^ \text{n } = \text{take-bit } n \text{ } k \text{ } \text{+} \text{ } \text{NOT} \text{ } \text{mask } n \text{ } \rangle
    by (simp add: minus-exp-eq-not-mask)
  also have \langle \ldots \text{ = take-bit } n \text{ } k \text{ } \text{OR} \text{ } \text{NOT} \text{ } \text{mask } n \text{ } \rangle
    by (rule disjunctive-add-eq-or) (simp add: bit-eq-iff bit-simps)
  finally have ** \langle \text{take-bit } n \text{ } k \text{ } \text{-} \text{2 } ^ \text{n } = \text{take-bit } n \text{ } k \text{ } \text{OR} \text{ } \text{NOT} \text{ } \text{mask } n \text{ } \rangle
  have \langle \text{take-bit } (\text{Suc } n) \text{ } (k + 2 ^ n) = \text{take-bit } (\text{Suc } n) \text{ } \text{take-bit } (\text{Suc } n) \text{ } \text{+} \text{ } \text{take-bit } (\text{Suc } n) \text{ } \text{2 } ^ \text{n } \rangle
    by (simp only: take-bit-add)
  also have \langle \text{take-bit } (\text{Suc } n) \text{ } k = 2 ^ n \text{ } \ast \text{ } \text{of-bool} \text{ } \text{bit } k \text{ } n \text{ } + \text{ } \text{take-bit } n \text{ } k \rangle
    by (simp add: take-bit-Suc-from-most)
  finally have \langle \text{take-bit } (\text{Suc } n) \text{ } (k + 2 ^ n) = \text{take-bit } (\text{Suc } n) \text{ } (2 ^ n \text{ } \ast \text{ } \text{of-bool} \text{ } \text{bit } k \text{ } n) \text{ } + \text{ } \text{take-bit } n \text{ } k \rangle
    by (simp add: ac-simps)
  also have \langle \text{2 } ^ \text{n} \text{ } \ast \text{ } \text{of-bool} \text{ } \text{bit } k \text{ } n \text{ } + \text{ } \text{take-bit } n \text{ } k = 2 ^ n \text{ } \ast \text{ } \text{of-bool} \text{ } \text{bit } k \text{ } n \text{ } \text{OR} \text{ } \text{take-bit } n \text{ } k \text{ } \rangle
    by (rule disjunctive-add-eq-or, rule bit-eqI) (simp add: bit-simps)
  finally show ?thesis
    using * ** by (simp add: signed-take-bit-def concat-bit-Suc-min-def ac-simps)
qed

lemma signed-take-bit-nonnegative-iff [simp]:
  \langle 0 \leq \text{signed-take-bit } n \text{ } k \text{ } \iff \text{-} \text{bit } k \text{ } n \rangle
for \( k :: \text{int} \)
by (simp add: signed-take-bit-def not-less concat-bit-def)

lemma signed-take-bit-negative-iff [simp]:
\[
\langle \text{signed-take-bit } n \; k < 0 \longleftrightarrow \text{bit } k \; n \rangle
\]
for \( k :: \text{int} \)
by (simp add: signed-take-bit-def not-less concat-bit-def)

lemma signed-take-bit-int-greater-eq-minus-exp [simp]:
\[
\langle - (2 ^ n) \leq \text{signed-take-bit } n \; k \rangle
\]
for \( k :: \text{int} \)
by (simp add: signed-take-bit-eq-take-bit-shift)

lemma signed-take-bit-int-less-exp [simp]:
\[
\langle \text{signed-take-bit } n \; k < 2 ^ n \rangle
\]
for \( k :: \text{int} \)
using take-bit-int-less-exp [of \( \text{Suc } n \)]
by (simp add: signed-take-bit-eq-take-bit-shift)

lemma signed-take-bit-int-eq-self-iff:
\[
\langle \text{signed-take-bit } n \; k = k \longleftrightarrow - (2 ^ n) \leq k \land k < 2 ^ n \rangle
\]
for \( k :: \text{int} \)
by (auto simp add: signed-take-bit-eq-take-bit-shift take-bit-int-less-eq-self-iff algebra-simps)

lemma signed-take-bit-int-eq-self:
\[
\langle \text{signed-take-bit } n \; k = k \rangle \text{ if } \langle - (2 ^ n) \leq k \land k < 2 ^ n \rangle
\]
for \( k :: \text{int} \)
using that by (simp add: signed-take-bit-eq-take-bit-shift)

lemma signed-take-bit-int-less-eq-self-iff:
\[
\langle \text{signed-take-bit } n \; k \leq k \longleftrightarrow - (2 ^ n) \leq k \rangle
\]
for \( k :: \text{int} \)
by (simp add: signed-take-bit-eq-take-bit-shift take-bit-int-less-eq-self-iff algebra-simps) linarith

lemma signed-take-bit-int-greater-self-iff:
\[
\langle k < \text{signed-take-bit } n \; k \longleftrightarrow k < - (2 ^ n) \rangle
\]
for \( k :: \text{int} \)
by (simp add: signed-take-bit-eq-take-bit-shift take-bit-int-greater-self-iff algebra-simps) linarith

lemma signed-take-bit-int-greater-eq-self-iff:
\[
\langle k \leq \text{signed-take-bit } n \; k \longleftrightarrow k < 2 ^ n \rangle
\]
for $k :: \text{int}$
by (simp add: signed-take-bit-eq-take-bit-shift take-bit-int-greater-eq-self-iff algebra-simps)

**lemma** signed-take-bit-int-greater-eq:
\[ k + 2 ^ \text{Suc} n \leq \text{signed-take-bit} n k \text{ if } k < (2 ^ n) \]
for $k :: \text{int}$
using that take-bit-int-greater-eq [of $k + 2 ^ n$ [Suc n]]
by (simp add: signed-take-bit-eq-take-bit-shift)

**lemma** signed-take-bit-int-less-eq:
\[ \text{signed-take-bit} n k \leq k - 2 ^ \text{Suc} n \text{ if } k \geq 2 ^ n \]
for $k :: \text{int}$
using that take-bit-int-less-eq [of [Suc n] $k + 2 ^ n$]
by (simp add: signed-take-bit-eq-take-bit-shift)

**lemma** signed-take-bit-Suc-bit0 [simp]:
\[ \text{signed-take-bit} (\text{Suc} n) (\text{numeral} (\text{Num.Bit0} k)) = \text{signed-take-bit} n (\text{numeral} k) \times (2 :: \text{int}) \]
by (simp add: signed-take-bit-Suc)

**lemma** signed-take-bit-Suc-bit1 [simp]:
\[ \text{signed-take-bit} (\text{Suc} n) (\text{numeral} (\text{Num.Bit1} k)) = \text{signed-take-bit} n (\text{numeral} k) \times 2 + (1 :: \text{int}) \]
by (simp add: signed-take-bit-Suc)

**lemma** signed-take-bit-Suc-minus-bit0 [simp]:
\[ \text{signed-take-bit} (\text{Suc} n) (- \text{numeral} (\text{Num.Bit0} k)) = \text{signed-take-bit} n (- \text{numeral} k) \times (2 :: \text{int}) \]
by (simp add: signed-take-bit-Suc)

**lemma** signed-take-bit-Suc-minus-bit1 [simp]:
\[ \text{signed-take-bit} (\text{Suc} n) (- \text{numeral} (\text{Num.Bit1} k)) = \text{signed-take-bit} n (- \text{numeral} k - 1) \times 2 + (1 :: \text{int}) \]
by (simp add: signed-take-bit-Suc)

**lemma** signed-take-bit-numeral-bit0 [simp]:
\[ \text{signed-take-bit} (\text{numeral} l) (\text{numeral} (\text{Num.Bit0} k)) = \text{signed-take-bit} \text{bit} (\text{pred-numeral} l) (\text{numeral} k) \times (2 :: \text{int}) \]
by (simp add: signed-take-bit-rec)

**lemma** signed-take-bit-numeral-bit1 [simp]:
\[ \text{signed-take-bit} (\text{numeral} l) (\text{numeral} (\text{Num.Bit1} k)) = \text{signed-take-bit} \text{bit} (\text{pred-numeral} l) (\text{numeral} k) \times 2 + (1 :: \text{int}) \]
by (simp add: signed-take-bit-rec)

**lemma** signed-take-bit-numeral-minus-bit0 [simp]:
\[ \text{signed-take-bit} (\text{numeral} l) (- \text{numeral} (\text{Num.Bit0} k)) = \text{signed-take-bit} \text{bit} (\text{pred-numeral} l) (- \text{numeral} k) \times (2 :: \text{int}) \]
by (simp add: signed-take-bit-rec)

lemma signed-take-bit-numeral-minus-bit1 [simp]:
\langle \text{signed-take-bit \ (numeral \ l)} \ (- \ \text{numeral \ (Nat.Bit1 \ k)}) = \text{signed-take-bit \ (pred-numeral \ l)} \ (- \ \text{numeral \ k} - 1) \ast 2 + (1 :: \text{int}) \rangle
by (simp add: signed-take-bit-rec)

lemma signed-take-bit-code [code]:
\langle \text{signed-take-bit \ n \ a} =
\quad (\text{let \ l = take-bit \ (Suc \ n) \ a \ in \ if \ bit \ l \ n \ then \ l + push-bit \ (Suc \ n) \ (- \ 1) \ else \ l)} \rangle
by (simp add: signed-take-bit-eq-take-bit-add bit-simps)

68.11 Key ideas of bit operations

When formalizing bit operations, it is tempting to represent bit values as explicit lists over a binary type. This however is a bad idea, mainly due to the inherent ambiguities in representation concerning repeating leading bits.

Hence this approach avoids such explicit lists altogether following an algebraic path:

- Bit values are represented by numeric types: idealized unbounded bit values can be represented by type \text{int}, bounded bit values by quotient types over \text{int}.

- (A special case are idealized unbounded bit values ending in 0 which can be represented by type \text{nat} but only support a restricted set of operations).

- From this idea follows that
  - multiplication by 2 is a bit shift to the left and
  - division by 2 is a bit shift to the right.

- Concerning bounded bit values, iterated shifts to the left may result in eliminating all bits by shifting them all beyond the boundary. The property \(2^n \neq 0\) represents that \(n\) is \textit{not} beyond that boundary.

- The projection on a single bit is then \(\text{bit \ a \ n} = \text{odd} \ (a \ \text{div} \ 2^n)\).

- This leads to the most fundamental properties of bit values:
  - Equality rule: \((\wedge \text{type \ TYPE} \ (\text{int}) \ n \Rightarrow \text{bit \ a \ n} = \text{bit \ b}) \Rightarrow a = b\)
  - Induction rule: \([\wedge \text{a \ \text{div} \ 2} = a \Rightarrow P \ a; \wedge \text{a \ \text{div} \ 2} = a \Rightarrow P \ (\text{of-boool \ b} + 2 \ast a)] \Rightarrow P \ (\text{of-boool \ b} + 2 \ast a) \Rightarrow P \ a\)
• Typical operations are characterized as follows:

  – Singleton \( n \)th bit: \( 2^n \)
  – Bit mask upto bit \( n \): \( \text{mask} n = 2^n - 1 \)
  – Left shift: \( \text{push-bit} n a = a \ast 2^n \)
  – Right shift: \( \text{drop-bit} n a = a \div 2^n \)
  – Truncation: \( \text{take-bit} n a = a \mod 2^n \)
  – Negation: \( \text{bit} (\text{NOT} a) n = (\text{possible-bit TYPE}(\text{int}) n \land \neg \text{bit} a n) \)
  – And: \( \text{bit} (a \text{ AND} b) n = (\text{bit} a n \land \text{bit} b n) \)
  – Or: \( \text{bit} (a \text{ OR} b) n = (\text{bit} a n \lor \text{bit} b n) \)
  – Xor: \( \text{bit} (a \text{ XOR} b) n = (\text{bit} a n \neq \text{bit} b n) \)
  – Set a single bit: \( \text{set-bit} n a = a \text{ OR push-bit} n 1 \)
  – Unset a single bit: \( \text{unset-bit} n a = a \text{ AND NOT} (\text{push-bit} n 1) \)
  – Flip a single bit: \( \text{flip-bit} n a = a \text{ XOR push-bit} n 1 \)
  – Signed truncation, or modulus centered around 0: \( \text{signed-take-bit} n a = \text{take-bit} n a \lor \text{of-bool} (\text{bit} a n) \ast \text{NOT} (\text{mask} n) \)
  – Bit concatenation: \( \text{concat-bit} n k l = \text{take-bit} n k \text{ OR push-bit} n l \)
  – (Bounded) conversion from and to a list of bits: \( \text{horner-sum of-bool 2 (map (bit a) [0..<n])} = \text{take-bit} n a \)

68.12 Lemma duplicates and other

context semiring-bits

begin

lemma exp-div-exp-eq:
  \( \langle 2 ^ m \text{ div} 2 ^ n = \text{of-bool} (2 ^ m \neq 0 \land m \geq n) \ast 2 ^ (m - n) \rangle \)
  apply (rule bit-eqI)
  using bit-exp-iff div-exp-eq apply (auto simp add: bit-iff-odd possible-bit-def)
  done

lemma bits-1-div-2:
  \( \langle 1 \text{ div} 2 = 0 \rangle \)
  by (fact half-1)

lemma bits-1-div-exp:
  \( \langle 1 \text{ div} 2 ^ n = \text{of-bool} (n = 0) \rangle \)
  using div-exp-eq [of 1 1] by (cases n) simp-all
lemma exp-add-not-zero-imp:
\(2^m \neq 0\) and \(2^n \neq 0\) if \(2^{(m+n)} \neq 0\)
proof
have \((\neg (2^m = 0 \lor 2^n = 0))\)
proof (rule notI)
assume \(2^m = 0 \lor 2^n = 0\)
then have \(2^{(m+n)} = 0\)
by (rule disjE) (simp-all add: power-add)
with that show False ..
qed
then show \(2^m \neq 0\) and \(2^n \neq 0\)
by simp-all
qed

lemma
exp-add-not-zero-imp-left: \(2^m \neq 0\)
and exp-add-not-zero-imp-right: \(2^n \neq 0\)
if \(2^{(m+n)} \neq 0\)
proof
have \((\neg (2^m = 0 \lor 2^n = 0))\)
proof (rule notI)
assume \(2^m = 0 \lor 2^n = 0\)
then have \(2^{(m+n)} = 0\)
by (rule disjE) (simp-all add: power-add)
with that show False ..
qed
then show \(2^m \neq 0\) and \(2^n \neq 0\)
by simp-all
qed

lemma exp-not-zero-imp-exp-diff-not-zero:
\(2^{(n-m)} \neq 0\) if \(2^n \neq 0\)
proof (cases \((m \leq n)\))
case True
moreover define \(q\) where \(q = n - m\)
ultimately have \(n = m + q\)
by simp
with that show ?thesis
by (simp add: exp-add-not-zero-imp-right)
next
case False
with that show ?thesis
by simp
qed

lemma exp-eq-0-imp-not-bit:
\((\neg \text{bit } a n)\) if \(2^n = 0\)
using that by (simp add: bit-iff-odd)
lemma bit-disjunctive-add-iff:
\langle bit \ (a + b) \ n \longleftrightarrow bit \ a \ n \lor bit \ b \ n \rangle
if \ \langle \land \ n. \ \neg bit \ a \ n \lor \neg bit \ b \ n \rangle
proof (cases \langle \text{possible-bit TYPE('a) n} \rangle)
\begin{proof}
case False
then show \ ?thesis
by (auto dest: impossible-bit)
next
\begin{proof}
case True
with that show \ ?thesis proof (induction n arbitrary: a b)
\begin{proof}
case 0
from 0.prems(1) [of 0] show \ ?case
by (auto simp add: bit-0)
next
\begin{proof}
case (Suc n)
from Suc.prems(1) [of 0] have even: \langle even \ a \lor even \ b \rangle
by (auto simp add: bit-0)
\begin{proof}
have bit: \langle \neg bit \ (a \ div \ 2) \ n \lor \neg bit \ (b \ div \ 2) \ n \rangle \ \text{for n}
using Suc.prems(1) [of \langle Suc n \rangle] by (simp add: bit-Suc)
\begin{proof}
from Suc.prems(2) have \langle possible-bit TYPE('a) (Suc n) \rangle \ \langle possible-bit TYPE('a) n \rangle
by (simp-all flip: possible-bit-less-imp)
\begin{proof}
\begin{proof}
\begin{proof}
have \langle a + b = (a \ div \ 2 \ast 2 + a \ mod \ 2) + (b \ div \ 2 \ast 2 + b \ mod \ 2) \rangle
\\begin{proof}
also have \langle ... \longleftrightarrow \langle \langle bit \ (a \ div \ 2) \ n \lor bit \ (b \ div \ 2) \ n \rangle \rangle \rangle
using \langle possible-bit TYPE('a) \ (Suc n) \rangle \ \langle possible-bit TYPE('a) n \rangle \by (simp-all flip: bit-Suc add: bit-double-iff possible-bit-def)
\\begin{proof}
also have \langle ... \longleftrightarrow \langle bit \ (a \ div \ 2) \ n \lor bit \ (b \ div \ 2) \ n \rangle \rangle
using \langle possible-bit TYPE('a) \ (Suc n) \rangle \ \langle possible-bit TYPE('a) n \rangle \by (rule Suc.IH)
\\begin{proof}
finally show \ ?case
by (simp add: bit-Suc)
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof}
\end{proof
lemma mult-exp-mod-exp-eq:
\[ m \leq n \implies (a \ast 2^m) \mod (2^n) = (a \mod 2^{n-m}) \ast 2^m \]
by (simp flip: push-bit-eq-mult take-bit-eq-mod add: push-bit-take-bit)

lemma div-exp-mod-exp-eq:
\[ a \div 2^n \mod 2^m = a \mod (2^{n+m}) \div 2^n \]
by (simp flip: drop-bit-eq-div take-bit-eq-mod add: drop-bit-take-bit)

lemma even-mult-exp-div-exp-iff:
\[ \text{even } (a \ast 2^m \div 2^n) \iff m > n \lor 2^n = 0 \lor (m \leq n \land \text{even } (a \div 2^{n-m}))) \]

lemma mod-exp-div-exp-eq-0:
\[ a \mod 2^n \div 2^n = 0 \]
by (simp flip: take-bit-eq-mod drop-bit-eq-div add: drop-bit-take-bit)

lemmas bits-one-mod-two-eq-one = one-mod-two-eq-one
lemmas set-bit-def = set-bit-eq-or
lemmas unset-bit-def = unset-bit-eq-and-not
lemmas flip-bit-def = flip-bit-eq-xor

lemma disjunctive-add:
\[ a + b = a \text{ OR } b \text{ if } \forall n. \neg \text{bit } a n \lor \neg \text{bit } b n \]
by (rule disjunctive-add-eq-or) (use that in \{simp add: bit-iff eq bit-simps\})

lemma even-mod-exp-div-exp-iff:
\[ \text{even } (a \mod 2^m \div 2^n) \iff m \leq n \lor \text{even } (a \div 2^n) \]
by (auto simp add: even-drop-bit-iff-not-bit bit-simps simp flip: drop-bit-eq-div take-bit-eq-mod)

disjunctive-diff:
\[ a - b = a \text{ AND NOT } b \text{ if } \forall n. \text{bit } b n \implies \text{bit } a n \]
proof -
have \{NOT a + b = NOT a OR b\}
  by (rule disjunctive-add) (auto simp add: bit-iff dest: that)
then have \{NOT (NOT a + b) = NOT (NOT a OR b)\}
  by simp
then show \?thesis
by (simp add: not-add-distrib)
qed

end

lemma and-nat-rec:
  \( m \land n = \text{of-bool (odd } m \land \text{odd } n) + 2 \times ((m \div 2) \land (n \div 2)) \) for \( m, n \) :: nat
  by (fact and-rec)

lemma or-nat-rec:
  \( m \lor n = \text{of-bool (odd } m \lor \text{odd } n) + 2 \times ((m \div 2) \lor (n \div 2)) \) for \( m, n \) :: nat
  by (fact or-rec)

lemma xor-nat-rec:
  \( m \oplus n = \text{of-bool (odd } m \neq \text{odd } n) + 2 \times ((m \div 2) \oplus (n \div 2)) \) for \( m, n \) :: nat
  by (fact xor-rec)

lemma bit-push-bit-iff-nat:
  \( \text{bit (push-bit } m q \text{)} n \leftrightarrow m \leq n \land \text{bit } q (n - m) \) for \( q :: \text{nat} \)
  by (fact bit-push-bit-iff)

lemma mask-half-int:
  \( \text{mask } n \div 2 = (\text{mask } (n - 1) :: \text{int}) \)
  by (fact mask-half)

lemma not-int-rec:
  \( \text{NOT } k = \text{of-bool (even } k) + 2 \ast \text{NOT } (k \div 2) \) for \( k :: \text{int} \)
  by (fact not-rec)

lemma even-not-iff-int:
  \( \text{even } (\text{NOT } k) \leftrightarrow \text{odd } k \) for \( k :: \text{int} \)
  by (fact even-not-iff)

lemma bit-not-int-iff:
  \( \text{bit } (- k - 1) n \leftrightarrow \neg \text{bit } k n \) for \( k :: \text{int} \)
  by (simp flip: not-eq-complement add: bit-simps)

lemmas and-int-rec = and-int.rec
lemmas even-and-iff-int:
  \( \text{even } (k \land l) \leftrightarrow \text{even } k \land \text{even } l \) for \( k, l :: \text{int} \)
  by (fact even-and-iff)
lemmas bit-and-int-iff = and-int.bit-iff
lemmas or-int-rec = or-int.rec
lemmas bit-or-int-iff = or-int.bit-iff

lemmas xor-int-rec = xor-int.rec

lemmas bit-xor-int-iff = xor-int.bit-iff

lemma drop-bit-push-bit-int:
\langle drop-bit m (push-bit n k) = drop-bit (m - n) (push-bit (n - m) k) \rangle \text{ for } k :: \mathtt{int}
by \hspace{1pt} \text{(fact drop-bit-push-bit)}

lemma bit-push-bit-iff-int:
\langle \text{bit (push-bit m k) n } \leftrightarrow \text{ m } \leq n \land \text{ bit k (n - m)} \rangle \text{ for } k :: \mathtt{int}
by \hspace{1pt} \text{(fact bit-push-bit-iff')}

no-notation
not \hspace{1pt} \langle \langle \text{NOT} \rangle \rangle
and \hspace{1pt} \text{ and \hspace{1pt} (infixr \hspace{1pt} \langle \langle \text{AND} \rangle \rangle \hspace{1pt} \text{64})}
or \hspace{1pt} \text{ or \hspace{1pt} (infixr \hspace{1pt} \langle \langle \text{OR} \rangle \rangle \hspace{1pt} \text{59})}
xor \hspace{1pt} \text{ xor \hspace{1pt} (infixr \hspace{1pt} \langle \langle \text{XOR} \rangle \rangle \hspace{1pt} \text{59})}

bundle bit-operations-syntax
begin

notation
not \hspace{1pt} \langle \langle \text{NOT} \rangle \rangle
and \hspace{1pt} \text{ and \hspace{1pt} (infixr \hspace{1pt} \langle \langle \text{AND} \rangle \rangle \hspace{1pt} \text{64})}
or \hspace{1pt} \text{ or \hspace{1pt} (infixr \hspace{1pt} \langle \langle \text{OR} \rangle \rangle \hspace{1pt} \text{59})}
xor \hspace{1pt} \text{ xor \hspace{1pt} (infixr \hspace{1pt} \langle \langle \text{XOR} \rangle \rangle \hspace{1pt} \text{59})}

end
end

69  Numeric types for code generation onto target language numerals only

theory Code-Numeral
imports Lifting Bit-Operations
begin

69.1 Type of target language integers

typedef integer = UNIV :: int set
morfisms int-of-integer integer-of-int ..

setup-lifting \langle \langle \text{type-definition-integer} \rangle \rangle
lemma integer-eq-iff:
  \( k = l \iff \text{int-of-integer } k = \text{int-of-integer } l \)
by transfer rule

lemma integer-eqI:
  \( \text{int-of-integer } k = \text{int-of-integer } l \implies k = l \)
using integer-eq-iff [of \( k \) \( l \)] by simp

lemma int-of-integer-integer-of-int [simp]:
  \( \text{int-of-integer } (\text{integer-of-int } k) = k \)
by transfer rule

lemma integer-of-int-int-of-integer [simp]:
  \( \text{integer-of-int } (\text{int-of-integer } k) = k \)
by transfer rule

instantiation integer :: ring-1
begin

lift-definition zero-integer :: integer
  is 0 :: int

declare zero-integer.rep-eq [simp]

lift-definition one-integer :: integer
  is 1 :: int

declare one-integer.rep-eq [simp]

lift-definition plus-integer :: integer \Rightarrow integer \Rightarrow integer
  is plus :: int \Rightarrow int \Rightarrow int

declare plus-integer.rep-eq [simp]

lift-definition uminus-integer :: integer \Rightarrow integer
  is uminus :: int \Rightarrow int

declare uminus-integer.rep-eq [simp]

lift-definition minus-integer :: integer \Rightarrow integer \Rightarrow integer
  is minus :: int \Rightarrow int \Rightarrow int

declare minus-integer.rep-eq [simp]
lift-definition times-integer :: integer ⇒ integer ⇒ integer
  is times :: int ⇒ int ⇒ int .

declare times-integer.rep-eq [simp]

instance proof
qed (transfer, simp add: algebra-simps+)

end

instance integer :: Rings.dvd ..

context
  includes lifting-syntax
  notes transfer-rule-numeral [transfer-rule]
begin

lemma [transfer-rule]:
  (pcr-integer ===> pcr-integer ===> (↔→)) (dvd) (dvd)
  by (unfold dvd-def) transfer-prover

lemma [transfer-rule]:
  ((↔→) ===> pcr-integer) of-bool of-bool
  by (unfold of-bool-def) transfer-prover

lemma [transfer-rule]:
  ((=) ===> pcr-integer) int of-nat
  by (rule transfer-rule-of-nat) transfer-prover+

lemma [transfer-rule]:
  ((=) ===> pcr-integer) (λk. k) of-int
proof –
  have ((=) ===> pcr-integer) of-int of-int
    by (rule transfer-rule-of-int) transfer-prover+
  then show ?thesis by (simp add: id-def)
qed

lemma [transfer-rule]:
  ((=) ===> pcr-integer) numeral numeral
  by transfer-prover

lemma [transfer-rule]:
  ((=) ===> (=) ===> pcr-integer) Num.sub Num.sub
  by (unfold Num.sub-def) transfer-prover

lemma [transfer-rule]:
  (pcr-integer ===> (=) ===> pcr-integer) (⊤) (⊤)
  by (unfold power-def) transfer-prover
end

\textbf{lemma} int-of-integer-of-nat [simp]:
\texttt{int-of-integer (of-nat n) = of-nat n}
\texttt{by transfer rule}

\textbf{lift-definition} integer-of-nat :: nat \Rightarrow integer
\texttt{is of-nat :: nat \Rightarrow int}
\textbf{.}

\textbf{lemma} integer-of-nat-eq-of-nat [code]:
\texttt{integer-of-nat = of-nat}
\texttt{by transfer rule}

\textbf{lemma} int-of-integer-integer-of-nat [simp]:
\texttt{int-of-integer (integer-of-nat n) = of-nat n}
\texttt{by transfer rule}

\textbf{lift-definition} nat-of-integer :: integer \Rightarrow nat
\texttt{is Int.nat}
\textbf{.}

\textbf{lemma} nat-of-integer-of-nat [simp]:
\texttt{nat-of-integer (of-nat n) = n}
\texttt{by transfer simp}

\textbf{lemma} int-of-integer-of-int [simp]:
\texttt{int-of-integer (of-int k) = k}
\texttt{by transfer simp}

\textbf{lemma} nat-of-integer-integer-of-nat [simp]:
\texttt{nat-of-integer (integer-of-nat n) = n}
\texttt{by transfer simp}

\textbf{lemma} integer-of-int-eq-of-int [simp, code-abbrev]:
\texttt{integer-of-int = of-int}
\texttt{by transfer \{simp add: fun-eq-iff\}}

\textbf{lemma} of-int-integer-of [simp]:
\texttt{of-int (int-of-integer k) = (k :: integer)}
\texttt{by transfer rule}

\textbf{lemma} int-of-integer-numeral [simp]:
\texttt{int-of-integer (numeral k) = numeral k}
\texttt{by transfer rule}

\textbf{lemma} int-of-integer-sub [simp]:
\texttt{int-of-integer (Num.sub k l) = Num.sub k l}
by transfer rule

**definition** integer-of-num :: num ⇒ integer
**where** [simp]: integer-of-num = numeral

**lemma** integer-of-num [code]:
integer-of-num Num.One = 1
integer-of-num (Num.Bit0 n) = (let k = integer-of-num n in k + k)
integer-of-num (Num.Bit1 n) = (let k = integer-of-num n in k + k + 1)
by (simp-all only; integer-of-num-def numeral.simps Let-def)

**lemma** integer-of-num-triv:
integer-of-num Num.One = 1
integer-of-num (Num.Bit0 Num.One) = 2
by simp-all

**instantiation** integer :: equal
begin

**lift-definition** equal-integer :: ‹integer ⇒ integer ⇒ bool›
is ‹HOL.equal :: int ⇒ int ⇒ bool›
.

**instance**
by (standard; transfer) (fact equal-eq)
end

**instantiation** integer :: linordered-idom

begin

**lift-definition** abs-integer :: ‹integer ⇒ integer›
is ‹abs :: int ⇒ int›
.

**declare** abs-integer.rep-eq [simp]

**lift-definition** sgn-integer :: ‹integer ⇒ integer›
is ‹sgn :: int ⇒ int›
.

**declare** sgn-integer.rep-eq [simp]

**lift-definition** less-eq-integer :: ‹integer ⇒ integer ⇒ bool›
is ‹less-eq :: int ⇒ int ⇒ bool›
.

**lemma** integer-less-eq-iff:
<k ≤ l ←→ int-of-integer k ≤ int-of-integer l>
by (fact less-eq-integer.rep-eq)

lift-definition less-integer :: (integer ⇒ integer ⇒ bool)
is (less :: int ⇒ int ⇒ bool)
.

lemma integer-less-iff:
  \( k < l \iff \text{int-of-integer } k < \text{int-of-integer } l \)
by (fact less-integer.rep-eq)

instance
by (standard; transfer)
  (simp-all add: algebra-simps less-le-not-le [symmetric] mult-strict-right-mono
  linear)
end

instance integer :: discrete-linordered-semidom
by (standard; transfer)
  (fact less-iff-succ-less-eq)

context
  includes lifting-syntax
begin

lemma [transfer-rule]:
  \( \langle \text{pcr-integer } \Longrightarrow \text{pcr-integer } \Longrightarrow \text{pcr-integer} \rangle \ \text{min} \ \text{min} \)
by (unfold min-def) transfer-prover

lemma [transfer-rule]:
  \( \langle \text{pcr-integer } \Longrightarrow \text{pcr-integer } \Longrightarrow \text{pcr-integer} \rangle \ \text{max} \ \text{max} \)
by (unfold max-def) transfer-prover
end

lemma int-of-integer-min [simp]:
int-of-integer (min k l) = min (int-of-integer k) (int-of-integer l)
by transfer rule

lemma int-of-integer-max [simp]:
int-of-integer (max k l) = max (int-of-integer k) (int-of-integer l)
by transfer rule

lemma nat-of-integer-non-positive [simp]:
k ≤ 0 ⇒ nat-of-integer k = 0
by transfer simp

lemma of-nat-of-integer [simp]:
of-nat (nat-of-integer k) = max 0 k
by transfer auto

instantiation integer :: unique-euclidean-ring
begin

lift-definition divide-integer :: integer ⇒ integer ⇒ integer
  is divide :: int ⇒ int ⇒ int
  .

declare divide-integer.rep-eq [simp]

lift-definition modulo-integer :: integer ⇒ integer ⇒ integer
  is modulo :: int ⇒ int ⇒ int
  .

declare modulo-integer.rep-eq [simp]

lift-definition euclidean-size-integer :: integer ⇒ nat
  is euclidean-size :: int ⇒ nat
  .

declare euclidean-size-integer.rep-eq [simp]

lift-definition division-segment-integer :: integer ⇒ integer
  is division-segment :: int ⇒ int
  .

declare division-segment-integer.rep-eq [simp]

instance
  apply (standard; transfer)
  apply (use mult-le-mono2 [of 1] in \auto simp add: sgn-mult-abs abs-mult sgn-mult
    abs-mod-less sgn-mod nat-mult-distrib
    division-segment-mult division-segment-mod)
  apply (simp add: division-segment-int-def split: if-splits)
  done
end

lemma [code]:
  euclidean-size = nat-of-integer ◦ abs
by (simp add: fun-eq-iff nat-of-integer.rep-eq)

lemma [code]:
  division-segment (k :: integer) = (if k ≥ 0 then 1 else −1)
by transfer (simp add: division-segment-int-def)

instance integer :: linordered-euclidean-semiring
  by (standard; transfer) (simp-all add: of-nat-div division-segment-int-def)
instatiation integer :: ring-bit-operations  
begin  
lift-definition bit-integer :: (integer ⇒ nat ⇒ bool)  
is bit .  
lift-definition not-integer :: (integer ⇒ integer)  
is not .  
lift-definition and-integer :: (integer ⇒ integer ⇒ integer)  
is ⟨and⟩ .  
lift-definition or-integer :: (integer ⇒ integer ⇒ integer)  
is or .  
lift-definition xor-integer :: (integer ⇒ integer ⇒ integer)  
is xor .  
lift-definition mask-integer :: (nat ⇒ integer)  
is mask .  
lift-definition set-bit-integer :: (nat ⇒ integer ⇒ integer)  
is set-bit .  
lift-definition unset-bit-integer :: (nat ⇒ integer ⇒ integer)  
is unset-bit .  
lift-definition flip-bit-integer :: (nat ⇒ integer ⇒ integer)  
is flip-bit .  
lift-definition push-bit-integer :: (nat ⇒ integer ⇒ integer)  
is push-bit .  
lift-definition drop-bit-integer :: (nat ⇒ integer ⇒ integer)  
is drop-bit .  
lift-definition take-bit-integer :: (nat ⇒ integer ⇒ integer)  
is take-bit .  
instance by (standard; transfer)  
(fact bit-induct div-by-0 div-by-1 div-0 even-half-succ-eq  
  half-div-exp-eq even-double-div-exp-iff bits-mod-div-trivial  
  bit-iff-odd push-bit-eq-mult drop-bit-eq-div take-bit-eq-mod  
  and-rec or-rec xor-rec mask-eq-exp-minus-1  
  set-bit-eq-or unset-bit-eq-or-xor flip-bit-eq-xor not-eq-complement)+  
end
instance integer :: linordered-euclidean-semiring-bit-operations ..

context
  includes bit-operations-syntax
begin

lemma [code]:
  \( \langle \text{bit } k \ n \iff \text{odd (drop-bit } n \ k)\rangle \)
  \( \langle \text{NOT } k = -k - 1 \rangle \)
  \( \langle \text{mask } n = 2 \sim n - (1 :: \text{integer})\rangle \)
  \( \langle \text{set-bit } n \ k = k \text{ OR push-bit } n \ 1 \rangle \)
  \( \langle \text{unset-bit } n \ k = k \text{ AND NOT (push-bit } n \ 1)\rangle \)
  \( \langle \text{flip-bit } n \ k = k \text{ XOR push-bit } n \ 1 \rangle \)
  \( \langle \text{push-bit } n \ k = k \times 2 \sim n \rangle \)
  \( \langle \text{drop-bit } n \ k = k \div 2 \sim n \rangle \) \text{ for } k :: \text{integer}
  \text{ by (fact bit-iff-odd-drop-bit not-eq-complement mask-eq-exp-minus-1}
  \text{ set-bit-eq-or unset-bit-eq-and-not flip-bit-eq-xor push-bit-eq-mult drop-bit-eq-div}
  \text{ take-bit-eq-mod)}+

lemma [code]:
  \( \langle \text{k AND l = (if k = 0 \lor l = 0 \text{ then } 0 \text{ else if } k = -1 \text{ then } l \text{ else if } l = -1\text{ then } k}
  \text{ else (k mod } 2\text{) \times (l mod } 2\text{) + 2 \times ((k div } 2\text{) AND (l div } 2\text{))})\rangle \) \text{ for } k \ l :: \text{integer}
  \text{ by transfer (fact and-int-unfold)}

lemma [code]:
  \( \langle \text{k OR l = (if k = -1 \lor l = -1 \text{ then } -1 \text{ else if } k = 0 \text{ then } l \text{ else if } l = 0 \text{ then } k}
  \text{ else max (k mod } 2\text{) (l mod } 2\text{) + 2 \times ((k div } 2\text{) OR (l div } 2\text{))})\rangle \) \text{ for } k \ l :: \text{integer}
  \text{ by transfer (fact or-int-unfold)}

lemma [code]:
  \( \langle \text{k XOR l = (if k = -1 \text{ then NOT l else if } l = -1 \text{ then NOT k else if } k = 0 \text{ then l else if } l = 0 \text{ then k}
  \text{ else max (k mod } 2\text{) (l mod } 2\text{) + 2 \times ((k div } 2\text{) XOR (l div } 2\text{))})\rangle \) \text{ for } k \ l :: \text{integer}
  \text{ by transfer (fact xor-int-unfold)}

end

instantiation integer :: linordered-euclidean-semiring-division
begin

definition divmod-integer :: num \Rightarrow num \Rightarrow integer \times integer
where
  \text{divmod-integer'}-def: \text{divmod-integer } m \ n = (\text{numeral } m \div \text{numeral } n, \text{numeral } m \mod \text{numeral } n)
definition divmod-step-integer :: integer ⇒ integer × integer ⇒ integer × integer
where
divmod-step-integer l qr = (let (q, r) = qr
    in if |l| ≤ |r| then (2 * q + 1, r - l)
    else (2 * q, r))

instance by standard
(auto simp add: divmod-integer'-def divmod-step-integer-def integer-less-eq-iff)
end

declare divmod-algorithm-code [where ?'a = integer,
folded integer-of-num-def, unfolded integer-of-num-triv,
code]

lemma integer-of-nat-0: integer-of-nat 0 = 0
by transfer simp

lemma integer-of-nat-1: integer-of-nat 1 = 1
by transfer simp

lemma integer-of-nat-numeral:
integer-of-nat (numeral n) = numeral n
by transfer simp

69.2 Code theorems for target language integers

Constructors

definition Pos :: num ⇒ integer
where
[simp, code-post]: Pos = numeral

context
includes lifting-syntax

begin

lemma [transfer-rule]:
(=) ===> per-integer numeral Pos
by simp transfer-prover

end

lemma Pos-fold [code-unfold]:
numeral Num.One = Pos Num.One
numeral (Num.Bit0 k) = Pos (Num.Bit0 k)
numeral (Num.Bit1 k) = Pos (Num.Bit1 k)
by simp-all

definition Neg :: num ⇒ integer
where
[simp, code-abbrev]: \text{Neg } n = - \text{Pos } n

context
  includes lifting-syntax
begin

lemma [transfer-rule]:

\langle (\equiv) \equiv\equiv \Rightarrow \text{pcr-integer} \rangle (\lambda n. - \text{numeral } n) \text{Neg}

by (unfold Neg-def) transfer-prover

end

code-datatype 0::integer Pos Neg

A further pair of constructors for generated computations

context
begin

qualified definition positive :: num \Rightarrow integer
  where [simp]: positive = numeral

qualified definition negative :: num \Rightarrow integer
  where [simp]: negative = uminus \circ numeral

lemma [code-computation-unfold]:

numeral = positive
Pos = positive
Neg = negative

by (simp-all add: fun-eq-iff)

end

Auxiliary operations

lift-definition dup :: integer \Rightarrow integer
  is \lambda k::int. k + k

lemma dup-code [code]:

dup 0 = 0
dup (Pos n) = Pos (Num.Bit0 n)
dup (Neg n) = Neg (Num.Bit0 n)

by (transfer, simp only: numeral-Bit0 minus-add-distrib)

lift-definition sub :: num \Rightarrow num \Rightarrow integer
  is \lambda m n. numeral m - numeral n :: int

lemma sub-code [code]:
sub Num.One Num.One = 0
sub (Num.Bit0 m) Num.One = Pos (Num.BitM m)
sub (Num.Bit1 m) Num.One = Pos (Num.Bit0 m)
sub Num.One (Num.Bit0 n) = Neg (Num.BitM n)
sub Num.One (Num.Bit1 n) = Neg (Num.Bit0 n)
sub (Num.Bit0 m) (Num.Bit0 n) = dup (sub m n)
sub (Num.Bit1 m) (Num.Bit1 n) = dup (sub m n)
sub (Num.Bit1 m) (Num.Bit0 n) = dup (sub m n) + 1
sub (Num.Bit0 m) (Num.Bit1 n) = dup (sub m n) − 1
by (transfer, simp add: dbl-def dbl-inc-def dbl-dec-def)+

Implementations

lemma one-integer-code [code, code-unfold]:
1 = Pos Num.One
by simp

lemma plus-integer-code [code]:
k + 0 = (k::integer)
0 + l = (l::integer)
Pos m + Pos n = Pos (m + n)
Pos m + Neg n = sub m n
Neg m + Pos n = sub n m
Neg m + Neg n = Neg (m + n)
by (transfer, simp)+

lemma uminus-integer-code [code]:
uminus 0 = (0::integer)
uminus (Pos m) = Neg m
uminus (Neg m) = Pos m
by simp-all

lemma minus-integer-code [code]:
k − 0 = (k::integer)
0 − l = uminus (l::integer)
Pos m − Pos n = sub m n
Pos m − Neg n = Pos (m + n)
Neg m − Pos n = Neg (m + n)
Neg m − Neg n = sub n m
by (transfer, simp)+

lemma abs-integer-code [code]:
|k| = (if (k::integer) < 0 then − k else k)
by simp

lemma sgn-integer-code [code]:
sgn k = (if k = 0 then 0 else if (k::integer) < 0 then − 1 else 1)
by simp

lemma times-integer-code [code]:
THEORY “Code-Numeral”

\[ k \times 0 = (0::\text{integer}) \]
\[ 0 \times l = (0::\text{integer}) \]
Pos \( m \times Pos n = Pos (m \times n) \]
Pos \( m \times Neg n = Neg (m \times n) \]
Neg \( m \times Pos n = Neg (m \times n) \]
Neg \( m \times Neg n = Pos (m \times n) \]
by simp-all

definition divmod-integer :: integer ⇒ integer ⇒ integer × integer
where
\[ \text{divmod-integer } k \ l = (k \div l, k \mod l) \]

lemma fst-divmod-integer [simp]:
\[ \text{fst } (\text{divmod-integer } k \ l) = k \div l \]
by (simp add: divmod-integer-def)

lemma snd-divmod-integer [simp]:
\[ \text{snd } (\text{divmod-integer } k \ l) = k \mod l \]
by (simp add: divmod-integer-def)

definition divmod-abs :: integer ⇒ integer ⇒ integer × integer
where
\[ \text{divmod-abs } k \ l = (|k| \div |l|, |k| \mod |l|) \]

lemma fst-divmod-abs [simp]:
\[ \text{fst } (\text{divmod-abs } k \ l) = |k| \div |l| \]
by (simp add: divmod-abs-def)

lemma snd-divmod-abs [simp]:
\[ \text{snd } (\text{divmod-abs } k \ l) = |k| \mod |l| \]
by (simp add: divmod-abs-def)

lemma divmod-abs-code [code]:
\[ \text{divmod-abs } (\text{Pos } k) \ (\text{Pos } l) = \text{divmod } k \ l \]
\[ \text{divmod-abs } (\text{Neg } k) \ (\text{Neg } l) = \text{divmod } k \ l \]
\[ \text{divmod-abs } (\text{Neg } k) \ (\text{Pos } l) = \text{divmod } k \ l \]
\[ \text{divmod-abs } (\text{Pos } k) \ (\text{Neg } l) = \text{divmod } k \ l \]
\[ \text{divmod-abs } j \ (0) = (0, |j|) \]
\[ \text{divmod-abs } 0 \ j = (0, 0) \]
by (simp-all add: prod-eq-iff)

lemma divmod-integer-eq-cases:
\[ \text{divmod-integer } k \ l =\]
\[ \begin{cases} (\text{apsnd } \circ \text{times } \circ \text{sgn}) \ l \ (\text{if sgn } k = \text{sgn } l \text{ then divmod-abs } k \ l \\ \text{else let } (r, s) = \text{divmod-abs } k \ l \text{ in} \\ \text{if } s = 0 \text{ then } (-r, 0) \text{ else } (-r - 1, |l| - s)) \end{cases} \]
proof –
have ∗: \( \text{sgn } k = \text{sgn } l \iff k = 0 \land l = 0 \lor 0 < l \land 0 < k \lor l < 0 \land k < 0 \)
for \( k \ l :: \text{int} \)
  by (auto simp add: sgn-if)

have ∗∗: \(- k = l \cdot q \iff k = -(l \cdot q)\) for \( k \ l \ q :: \text{int} \)
  by auto

show ?thesis
  by (simp add: divmod-integer-def divmod-abs-def)
    (transfer, auto simp add: ∗∗∗ not-less zdiv-zminus1-eq-if zmod-zminus1-eq-if
div-minus-right mod-minus-right)

qed

lemma divmod-integer-code [code]:
  divmod-integer \( k \ l =\)
    (if \( k = 0 \) then \((0, 0)\)
      else if \( l > 0 \) then
        (if \( k > 0 \) then Code-Numeral.divmod-abs \( k \ l \)
        else case Code-Numeral.divmod-abs \( k \ l \) of \((r, s)\) ⇒
          if \( s = 0 \) then \((- r, 0)\) else \((- r - 1, l - s)\))
      else if \( l = 0 \) then \((0, k)\)
      else apsnd uminus
        (if \( k < 0 \) then Code-Numeral.divmod-abs \( k \ l \)
        else case Code-Numeral.divmod-abs \( k \ l \) of \((r, s)\) ⇒
          if \( s = 0 \) then \((- r, 0)\) else \((- r - 1, - l - s)\)))
    by (cases \( l \ 0 :: \text{integer} \) rule: linorder-cases)

    (auto split: prod.splits simp add: divmod-integer-eq-cases)

lemma div-integer-code [code]:
  \( k \ div \ l = \text{fst} \ (\text{divmod-integer } k \ l) \)
  by simp

lemma mod-integer-code [code]:
  \( k \ mod \ l = \text{snd} \ (\text{divmod-integer } k \ l) \)
  by simp

definition bit-cut-integer :: \text{integer} ⇒ \text{integer} × bool
  where bit-cut-integer \( k = (k \ div \ 2, \text{odd } k) \)

lemma bit-cut-integer-code [code]:
  bit-cut-integer \( k = (\text{if } k = 0 \text{ then } (0, \text{False}) \)
  else let \( (r, s) = \text{Code-Numeral.divmod-abs } k \ 2 \)
  in \( (\text{if } k > 0 \text{ then } r \text{ else } - r - s, s = 1) \))

proof –
  have bit-cut-integer \( k = (\text{let } (r, s) = \text{divmod-integer } k \ 2 \text{ in } (r, s = 1)) \)
    by (simp add: divmod-integer-def bit-cut-integer-def odd-iff-mod-2-eq-one)
  then show ?thesis
    by (simp add: divmod-integer-code) (auto simp add: split-def)

qed

lemma equal-integer-code [code]:
HOL.equal 0 (0::integer) \(\leftrightarrow\) True
HOL.equal 0 (Pos l) \(\leftrightarrow\) False
HOL.equal 0 (Neg l) \(\leftrightarrow\) False
HOL.equal (Pos k) 0 \(\leftrightarrow\) False
HOL.equal (Pos k) (Pos l) \(\leftrightarrow\) HOL.equal k l
HOL.equal (Pos k) (Neg l) \(\leftrightarrow\) False
HOL.equal (Neg k) 0 \(\leftrightarrow\) False
HOL.equal (Neg k) (Pos l) \(\leftrightarrow\) False
HOL.equal (Neg k) (Neg l) \(\leftrightarrow\) HOL.equal k l
by (simp-all add: equal)

lemma equal-integer-refl [code nbe]:
HOL.equal (k::integer) k \(\leftrightarrow\) True
by (fact equal-refl)

lemma less-eq-integer-code [code]:
0 \(\leq\) (0::integer) \(\leftrightarrow\) True
0 \(\leq\) Pos l \(\leftrightarrow\) True
0 \(\leq\) Neg l \(\leftrightarrow\) False
Pos k \(\leq\) 0 \(\leftrightarrow\) False
Pos k \(\leq\) Pos l \(\leftrightarrow\) k \(\leq\) l
Pos k \(\leq\) Neg l \(\leftrightarrow\) False
Neg k \(\leq\) 0 \(\leftrightarrow\) True
Neg k \(\leq\) Pos l \(\leftrightarrow\) True
Neg k \(\leq\) Neg l \(\leftrightarrow\) l \(\leq\) k
by simp-all

lemma less-integer-code [code]:
0 < (0::integer) \(\leftrightarrow\) False
0 < Pos l \(\leftrightarrow\) True
0 < Neg l \(\leftrightarrow\) False
Pos k < 0 \(\leftrightarrow\) False
Pos k < Pos l \(\leftrightarrow\) k \(<\) l
Pos k < Neg l \(\leftrightarrow\) False
Neg k < 0 \(\leftrightarrow\) True
Neg k < Pos l \(\leftrightarrow\) True
Neg k < Neg l \(\leftrightarrow\) l \(<\) k
by simp-all

lift-definition num-of-integer :: integer \(\Rightarrow\) num
is num-of-nat \(\circ\) nat
.

lemma num-of-integer-code [code]:
num-of-integer k = (if k \(\leq\) 1 then Num.One
else let
  (l, j) = divmod-integer k 2;
  l' = num-of-integer l;
  l'' = l' + l'

by simp-all
in if j = 0 then l" else l" + Num.One)

proof {
  assume int-of-integer k mod 2 = 1
  then have nat (int-of-integer k mod 2) = nat 1 by simp
  moreover assume *: 1 < int-of-integer k
  ultimately have **: nat (int-of-integer k) mod 2 = 1 by (simp add: nat-mod-distrib)
    have num-of-nat (nat (int-of-integer k)) = num-of-nat (2 * (nat (int-of-integer k) div 2) + nat (int-of-integer k) mod 2)
      by simp
    then have num-of-nat (nat (int-of-integer k)) = num-of-nat (nat (int-of-integer k) div 2 + nat (int-of-integer k) div 2 + nat (int-of-integer k) mod 2)
      by (simp add: mult-2)
    with ** have num-of-nat (nat (int-of-integer k)) = num-of-nat (nat (int-of-integer k) div 2 + nat (int-of-integer k) div 2 + 1)
      by simp
}

note aux = this

show ?thesis
  by (auto simp add: num-of-integer-def nat-of-integer-def Let-def case-prod-beta not-le integer-eq-iff less-eq-integer-def
    nat-mult-distrib nat-div-distrib num-of-nat-One num-of-nat-plus-distrib
    mult-2 [where 'a=nat] aux add-One)

qed

lemma nat-of-integer-code [code]:
  nat-of-integer k = (if k ≤ 0 then 0
    else let
      (l, j) = divmod-integer k 2;
      l' = nat-of-integer l;
      l" = l' + l'
      in if j = 0 then l" else l" + 1)

proof –
  obtain j where: k = integer-of-int j

proof
  show k = integer-of-int (int-of-integer k) by simp

qed

have *: nat j mod 2 = nat-of-integer (of-int j mod 2) if j ≥ 0
  using that by transfer (simp add: nat-mod-distrib)

from k show ?thesis
  by (auto simp add: split-def Let-def nat-of-integer-def nat-div-distrib mult-2 [symmetric]
    minus-mod-eq-mult-div [symmetric] *)

qed

lemma int-of-integer-code [code]:
  int-of-integer k = (if k < 0 then − (int-of-integer (− k))
    else if k = 0 then 0
else let
    (l, j) = \text{divmod-integer} k 2;
    l' = 2 * \text{int-of-integer} l
in if j = 0 then l' else l' + 1)
by (auto simp add: split-def Let-def integer-eq-iff minus-mod-eq-mult-div [symmetric])

\text{lemma} \ \text{integer-of-int-code} [\text{code}]:
integer-of-int k = (if k < 0 then \ (-\ \text{integer-of-int}\ (-k))
else if k = 0 then 0
else let
    l = 2 * \text{integer-of-int} (k \text{ div} 2);
    j = k \mod 2
in if j = 0 then l else l + 1)
by (auto simp add: split-def Let-def integer-eq-iff minus-mod-eq-mult-div [symmetric])

\text{hide-const} (open) \ \text{Pos} \ \text{Neg} \ \text{sub} \ \text{dup} \ \text{divmod-abs}

69.3 Serializer setup for target language integers

\text{code-reserved} \ \text{Eval} \ \text{int} \ \text{Integer} \ \text{abs}

\text{code-printing}
\text{type-constructor} \ \text{integer} \rightarrow
\begin{align*}
\text{(SML)} & \ \text{IntInf.int} \hfill \\
\text{and} \ \text{(OCaml)} & \ \text{Z.t} \\
\text{and} \ \text{(Haskell)} & \ \text{Integer} \\
\text{and} \ \text{(Scala)} & \ \text{BigInt} \\
\text{and} \ \text{(Eval)} & \ \text{int}
\end{align*}
\begin{itemize}
\item \text{class-instance} \ \text{integer :: equal} \rightarrow
\text{(Haskell)} \ 
\end{itemize}

\text{code-printing}
\text{constant} \ \text{0::integer} \rightarrow
\begin{align*}
\text{(SML)} & \ !(0/::\ \text{IntInf.int}) \\
\text{and} \ \text{(OCaml)} & \ Z.zero \\
\text{and} \ \text{(Haskell)} & \ !(0/::\ \text{Integer}) \\
\text{and} \ \text{(Scala)} & \ \text{BigInt}(0)
\end{align*}

\text{setup} \ 
\text{fold} (fn \ \text{target} =>
\ \text{Numeral.add-code} \ \text{const-name} \ \text{Code-Numeral.Pos} \ 1 \ \text{Code-Printer.literal-numeral}
\ \text{target}
\ #> \ \text{Numeral.add-code} \ \text{const-name} \ \text{Code-Numeral.Neg} \ (-) \ \text{Code-Printer.literal-numeral}
\ \text{target}
\ [\text{SML, OCaml, Haskell, Scala}]
\)

\text{code-printing}
\text{constant} \ \text{plus :: integer} \Rightarrow \ - \Rightarrow - \Rightarrow
(SML) IntInf.+ ((-), (-))
and (OCaml) Z.add
and (Haskell) infixl 6 +
and (Scala) infixl 7 +
and (Eval) infixl 8 +

| constant aminus :: integer ⇒ - →
  | (SML) IntInf.~
  | and (OCaml) Z.neg
  | and (Haskell) negate
  | and (Scala) !((- -)
  | and (Eval) ~ / -

| constant minus :: integer ⇒ - →
  | (SML) IntInf.- ((-), (-))
  | and (OCaml) Z.sub
  | and (Haskell) infixl 6 -
  | and (Scala) infixl 7 -
  | and (Eval) infixl 8 -

| constant Code-Numeral.dup →
  | (SML) IntInf.* (2, (-))
  | and (OCaml) Z.shift'-'-left/ -/ 1
  | and (Haskell) !(2 * -)
  | and (Scala) !(2 * -)
  | and (Eval) !(2 * -)

| constant Code-Numeral.sub →
  | (SML) !((IntInf zero) ? IntInf.(-)
  | and (OCaml) failwith/ sub
  | and (Haskell) error/ sub
  | and (Scala) !sys.error(sub)

| constant times :: integer ⇒ - ⇒ -⇒
  | (SML) IntInf.* ((-), (-))
  | and (OCaml) Z.mul
  | and (Haskell) infixl 7 *
  | and (Scala) infixl 8 *
  | and (Eval) infixl 9 *

| constant Code-Numeral.divmod-abs →
  | (SML) IntInf.divMod/ (IntInf.abs -. IntInf.abs -)
  | and (OCaml) ![fun k l ->/ if Z.equal Z.zero l then/ (Z.zero, l) else/ Z.div'-rem/ (Z.abs k)/ (Z.abs l)]
  | and (Haskell) divMod/ (abs -)/ (abs -)
  | and (Scala) !(((k: BigInt) ===> (l: BigInt) =>/ l == 0 match { case true =>
  | (BigInt(0), k) case false => (k.abs '/% l.abs) })
  | and (Eval) Integer.div'-mod/ (abs -)/ (abs -)

| constant HOL.equal :: integer ⇒ - ⇒ bool →
  | (SML) ![((.: IntInf.int) = -)
  | and (OCaml) Z.equal
  | and (Haskell) infix 4 ==
  | and (Scala) infixl 5 ==
  | and (Eval) infixl 6 =

| constant less-eq :: integer ⇒ - ⇒ bool →
(SML) IntInf.<= ((-), (-))
and (OCaml) Z.leq
and (Haskell) infix 4 <=
and (Scala) infixl 4 <=
and (Eval) infixl 6 <=
| constant less :: integer ⇒ - ⇒ bool ⇒
  (SML) IntInf.< ((-), (-))
and (OCaml) Z.lt
and (Haskell) infix 4 <
and (Scala) infixl 4 <
and (Eval) infixl 6 <
| constant abs :: integer ⇒ - ⇒
  (SML) IntInf.abs
and (OCaml) Z.abs
and (Haskell) Prelude.abs
and (Scala) -.abs
and (Eval) abs

code-identifier
code-module Code-Numeral → (SML) Arith and (OCaml) Arith and (Haskell) Arith

69.4 Type of target language naturals
typedef natural = UNIV :: nat set
  morphisms nat-of-natural natural-of-nat ..

setup-lifting type-definition-natural

lemma natural-eq-iff [termination-simp]:
  m = n ←→ nat-of-natural m = nat-of-natural n
  by transfer rule

lemma natural-eqI:
  nat-of-natural m = nat-of-natural n ⇒ m = n
  using natural-eq-iff [of m n] by simp

lemma nat-of-natural-of-nat-inverse [simp]:
  nat-of-natural (natural-of-nat n) = n
  by transfer rule

lemma natural-of-natural-of-natural-inverse [simp]:
  natural-of-nat (nat-of-natural n) = n
  by transfer rule

instantiation natural :: {comm-monoid-diff, semiring-1}
begin

lift-definition zero-natural :: natural
is 0 :: nat .

declare zero-natural.rep-eq [simp]

lift-definition one-natural :: natural
  is 1 :: nat .

declare one-natural.rep-eq [simp]

lift-definition plus-natural :: natural ⇒ natural ⇒ natural
  is plus :: nat ⇒ nat ⇒ nat .

declare plus-natural.rep-eq [simp]

lift-definition minus-natural :: natural ⇒ natural ⇒ natural
  is minus :: nat ⇒ nat ⇒ nat .

declare minus-natural.rep-eq [simp]

lift-definition times-natural :: natural ⇒ natural ⇒ natural
  is times :: nat ⇒ nat ⇒ nat .

declare times-natural.rep-eq [simp]

instance proof
  qed (transfer, simp add: algebra-simps)+

end

instance natural :: Rings.dvd ..

context
  includes lifting-syntax
begin

  lemma [transfer-rule]:
  \((\text{pcr-natural} === \text{pcr-natural} === \text{pcr-natural}) (\text{dvd}) (\text{dvd})\)
  by (unfold dvd-def) transfer-prover

  lemma [transfer-rule]:
  \(((\text{dvd}) === \text{pcr-natural}) \text{ of-bool} \text{ of-bool})
  by (unfold of-bool-def) transfer-prover

  lemma [transfer-rule]:
THEORY "Code-Numeral"

((=) ===> pcr-natural) (λn. n) of-nat

proof –
  have rel-fun HOL.eq pcr-natural (of-nat :: nat ⇒ nat) (of-nat :: nat ⇒ natural)
    by (unfold of-nat-def) transfer-prover
  then show ?thesis by (simp add: id-def)
qed

lemma [transfer-rule]:
  ((=) ===> pcr-natural) numeral numeral

proof –
  have ((=) ===> pcr-natural) numeral (λn. of-nat (numeral n))
    by transfer-prover
  then show ?thesis by simp
qed

lemma [transfer-rule]:
  (pcr-natural ===> (=) ===> pcr-natural) (^) (^)
  by (unfold power-def) transfer-prover

end

lemma nat-of-natural-of-nat [simp]:
  nat-of-natural (of-nat n) = n
  by transfer rule

lemma natural-of-nat-of-nat [simp, code-abbrev]:
  natural-of-nat = of-nat
  by transfer rule

lemma of-nat-of-natural [simp]:
  of-nat (nat-of-natural n) = n
  by transfer rule

lemma nat-of-natural-numeral [simp]:
  nat-of-natural (numeral k) = numeral k
  by transfer rule

instantiation natural :: {linordered-semiring, equal}
begin

lift-definition less-eq-natural :: natural ⇒ natural ⇒ bool
  is less-eq :: nat ⇒ nat ⇒ bool
.

declare less-eq-natural.rep-eq [termination-simp]

lift-definition less-natural :: natural ⇒ natural ⇒ bool
  is less :: nat ⇒ nat ⇒ bool
.

declare less-natural.rep-eq [termination-simp]

lift-definition equal-natural :: natural ⇒ natural ⇒ bool
  is HOL.equal :: nat ⇒ nat ⇒ bool
.

instance proof
qed (transfer, simp add: algebra-simps equal less-le-not-le [symmetric] linear)+
end

context
  includes lifting-syntax
begin

lemma [transfer-rule]:
  ⟨(pcr-natural ===> pcr-natural ===> pcr-natural) min min⟩
  by (unfold min-def) transfer-prover

lemma [transfer-rule]:
  ⟨(pcr-natural ===> pcr-natural ===> pcr-natural) max max⟩
  by (unfold max-def) transfer-prover
end

lemma nat-of-natural-min [simp]:
  nat-of-natural (min k l) = min (nat-of-natural k) (nat-of-natural l)
  by transfer rule

lemma nat-of-natural-max [simp]:
  nat-of-natural (max k l) = max (nat-of-natural k) (nat-of-natural l)
  by transfer rule

instantiation natural :: unique-euclidean-semiring
begin

lift-definition divide-natural :: natural ⇒ natural ⇒ natural
  is divide :: nat ⇒ nat ⇒ nat
  .

declare divide-natural.rep-eq [simp]

lift-definition modulo-natural :: natural ⇒ natural ⇒ natural
  is modulo :: nat ⇒ nat ⇒ nat
  .

declare modulo-natural.rep-eq [simp]
lift-definition euclidean-size-natural :: natural ⇒ nat
  is euclidean-size :: nat ⇒ nat
  .

declare euclidean-size-natural.rep-eq [simp]

lift-definition division-segment-natural :: natural ⇒ natural
  is division-segment :: nat ⇒ nat
  .

declare division-segment-natural.rep-eq [simp]

instance
  by (standard; transfer)
    (auto simp add: algebra-simps unit-factor-nat-def gr0-conv-Suc)
end

lemma [code]:
  euclidean-size = nat-of-natural
  by (simp add: fun-eq-iff)

lemma [code]:
  division-segment (n::natural) = 1
  by (simp add: natural-eq-iff)

instance natural :: discrete-linordered-semidom
  by (standard; transfer) (simp-all add: Suc-le-eq)

instance natural :: linordered-euclidean-semiring
  by (standard; transfer) simp-all

instantiation natural :: semiring-bit-operations
begin

lift-definition bit-natural :: (natural ⇒ nat ⇒ bool)
  is bit .

lift-definition and-natural :: (natural ⇒ natural ⇒ natural)
  is (and) .

lift-definition or-natural :: (natural ⇒ natural ⇒ natural)
  is or .

lift-definition xor-natural :: (natural ⇒ natural ⇒ natural)
  is xor .

lift-definition mask-natural :: (nat ⇒ natural)
  is mask .
lift-definition set-bit-natural :: (nat ⇒ natural ⇒ natural) is set-bit.

lift-definition unset-bit-natural :: (nat ⇒ natural ⇒ natural) is unset-bit.

lift-definition flip-bit-natural :: (nat ⇒ natural ⇒ natural) is flip-bit.

lift-definition push-bit-natural :: (nat ⇒ natural ⇒ natural) is push-bit.

lift-definition drop-bit-natural :: (nat ⇒ natural ⇒ natural) is drop-bit.

lift-definition take-bit-natural :: (nat ⇒ natural ⇒ natural) is take-bit.

instance by (standard; transfer)
(fact bit-induct div-by-0 div-by-1 div-0 even-half-succ-eq
half-div-exp-eq even-double-div-exp-iff bits-mod-div-trivial
bit-iff-odd push-bit-bq-mult drop-bit-bq-div take-bit-bq-mod
and-rec or-rec xor-rec mask-eq-exp-minus-1
set-bit-bq-or unset-bit-bq-or-xor flip-bit-bq-xor not-eq-complement)+

end

instance natural :: linordered-euclidean-semiring-bit-operations ..

case
  includes bit-operations-syntax
begin

lemma [code]:
  \{bit m n \leftrightarrow odd (drop-bit n m)\}
  \{mask n = 2 \sim n - (1 :: natural)\}
  \{set-bit n m = m OR push-bit n 1\}
  \{flip-bit n m = m XOR push-bit n 1\}
  \{push-bit n m = m \cdot 2 \sim n\}
  \{drop-bit n m = m div 2 \sim n\} for m :: natural
by (fact bit-iff-odd-drop-bit mask-eq-exp-minus-1
  set-bit-eq-or flip-bit-eq-xor push-bit-bq-mult drop-bit-bq-div take-bit-bq-mod)+

lemma [code]:
  \{ m AND n = (if m = 0 \lor n = 0 then 0
               else (m mod 2) \cdot (n mod 2) + 2 \cdot ((m div 2) AND (n div 2)))\} for m n ::
natural
THEORY “Code-Numeral”

by transfer (fact and-nat-unfold)

lemma [code]:
\[
\begin{align*}
  m \lor n &= (\text{if } m = 0 \text{ then } n \text{ else if } n = 0 \text{ then } m \\
  &\quad \text{ else max } (m \mod 2) (n \mod 2) + 2 * ((m \div 2) \lor (n \div 2))
\end{align*}
\]
for \( m, n :: \text{natural} \)
by transfer (fact or-nat-unfold)

lemma [code]:
\[
\begin{align*}
  m \xor n &= (\text{if } m = 0 \text{ then } n \text{ else if } n = 0 \text{ then } m \\
  &\quad \text{ else } (m \mod 2 + n \mod 2) \mod 2 + 2 * ((m \div 2) \xor (n \div 2))
\end{align*}
\]
for \( m, n :: \text{natural} \)
by transfer (fact xor-nat-unfold)

lemma [code]:
\[
\begin{align*}
  \text{unset-bit } 0 m &= 2 * (m \div 2) \\
  \text{unset-bit } (\text{Suc } n) m &= m \mod 2 + 2 * \text{unset-bit } n (m \div 2)
\end{align*}
\]
for \( m :: \text{natural} \)
by (transfer; simp add: unset-bitSuc)

end

lift-definition natural-of-integer :: integer ⇒ natural
  is nat :: int ⇒ nat

lift-definition integer-of-natural :: natural ⇒ integer
  is of-nat :: nat ⇒ int

lemma natural-of-integer-of-natural [simp]:
  natural-of-integer (integer-of-natural n) = n
by transfer simp

lemma integer-of-natural-of-integer [simp]:
  integer-of-natural (natural-of-integer k) = max 0 k
by transfer auto

lemma int-of-integer-of-natural [simp]:
  int-of-integer (integer-of-natural n) = of-nat (nat-of-natural n)
by transfer rule

lemma integer-of-natural-of-nat [simp]:
  integer-of-natural (of-nat n) = of-nat n
by transfer rule

lemma [measure-function]:
  is-measure nat-of-natural
by (rule is-measure-trivial)
69.5 Inductive representation of target language naturals

lift-definition Suc :: natural ⇒ natural
    is Nat.Suc
.

declare Suc.rep-eq [simp]

old-rep-datatype 0::natural Suc
    by (transfer, fact nat.induct nat.inject nat.distinct)+

lemma natural-cases [case-names nat, cases type: natural]:
    fixes m :: natural
    assumes \( \forall n. \, m = \text{of-nat} \, n \Rightarrow P \)
    shows P
    using assms by transfer blast

instantiation natural :: size
begin

definition size-nat where [simp, code]: size-nat = nat-of-natural

instance ..
end

lemma natural-decr [termination-simp]:
    \( n \neq 0 \Rightarrow \text{nat-of-natural} \, n - \text{Nat.Suc} \, 0 < \text{nat-of-natural} \, n \)
    by transfer simp

lemma natural-zero-minus-one: (0::natural) \(-\) 1 = 0
    by (rule zero-diff)

lemma Suc-natural-minus-one: Suc n \(-\) 1 = n
    by transfer simp

hide-const (open) Suc

69.6 Code refinement for target language naturals

lift-definition Nat :: integer ⇒ natural
    is nat
.

lemma [code-post]:
    Nat 0 = 0
    Nat 1 = 1
    Nat (numeral k) = numeral k
    by (transfer, simp)+
lemma [code abstype]:
Nat (integer-of-natural n) = n
by transfer simp

lemma [code]:
natural-of-nat n = natural-of-integer (integer-of-nat n)
by transfer simp

lemma [code abstract]:
integer-of-natural (natural-of-integer k) = max 0 k
by simp

lemma [code]:
\langle integer-of-natural (mask n) = mask n \rangle
by transfer (simp add: mask-eq-exp-minus-1 of-nat-diff)

lemma [code-abbrev]:
natural-of-integer (Code-Numeral.Pos k) = numeral k
by transfer simp

lemma [code abstract]:
integer-of-natural 0 = 0
by transfer simp

lemma [code abstract]:
integer-of-natural 1 = 1
by transfer simp

lemma [code abstract]:
integer-of-natural (Code-Numeral.Suc n) = integer-of-natural n + 1
by transfer simp

lemma [code]:
nat-of-natural = nat-of-integer o integer-of-natural
by transfer (simp add: fun-eq-iff)

lemma [code, code-unfold]:
case-natural f g n = (if n = 0 then f else g (n - 1))
by (cases n rule: natural.exhaust) (simp-all, simp add: Suc-def)

declare natural.rec [code del]

lemma [code abstract]:
integer-of-natural (m + n) = integer-of-natural m + integer-of-natural n
by transfer simp

lemma [code abstract]:
integer-of-natural (m - n) = max 0 (integer-of-natural m - integer-of-natural n)
by transfer simp

lemma [code abstract]:
\[ \text{integer-of-natural } (m * n) = \text{integer-of-natural } m * \text{integer-of-natural } n \]
by transfer simp

lemma [code abstract]:
\[ \text{integer-of-natural } (m \div n) = \text{integer-of-natural } m \div \text{integer-of-natural } n \]
by transfer (simp add: zdiv-int)

lemma [code abstract]:
\[ \text{integer-of-natural } (m \mod n) = \text{integer-of-natural } m \mod \text{integer-of-natural } n \]
by transfer (simp add: zmod-int)

lemma [code]:
\[ \text{HOL.equal } m n \longleftrightarrow \text{HOL.equal } (\text{integer-of-natural } m) (\text{integer-of-natural } n) \]
by transfer (simp add: equal)

lemma [code nbe]: \text{HOL.equal } n (n::natural) \longleftrightarrow \text{True}
by (rule equal-class.equal-refl)

lemma [code]: \[ m \leq n \longleftrightarrow \text{integer-of-natural } m \leq \text{integer-of-natural } n \]
by transfer simp

lemma [code]: \[ m < n \longleftrightarrow \text{integer-of-natural } m < \text{integer-of-natural } n \]
by transfer simp

hide-const (open) \text{Nat}

code-reflect Code-Numeral
datatypes natural
functions Code-Numeral.Suc 0 :: natural 1 :: natural
    plus :: natural \Rightarrow - minus :: natural \Rightarrow -
    times :: natural \Rightarrow - divide :: natural \Rightarrow -
    modulo :: natural \Rightarrow -
    \text{integer-of-natural } \text{natural-of-integer}

lifting-update integer.lifting
lifting-forget integer.lifting

lifting-update natural.lifting
lifting-forget natural.lifting

data
begin

70.1 Auxiliary functions

fun log :: natural ⇒ natural ⇒ natural where
    log b i = (if b ≤ 1 ∨ i < b then 1 else 1 + log b (i div b))

definition inc-shift :: natural ⇒ natural ⇒ natural where
    inc-shift v k = (if v = k then 1 else k + 1)

definition minus-shift :: natural ⇒ natural ⇒ natural ⇒ natural where
    minus-shift r k l = (if k < l then r + k − l else k − l)

70.2 Random seeds

type-synonym seed = natural × natural

primrec next :: seed ⇒ natural × seed where
    next (v, w) = (let
        k = v div 53668;
        v' = minus-shift 2147483563 ((v mod 5368) * 40014) (k * 12211);
        l = w div 52774;
        w' = minus-shift 2147483399 ((w mod 52774) * 40692) (l * 3791);
        z = minus-shift 2147483562 v' (w' + 1) + 1
    in (z, (v', w')))

definition split-seed :: seed ⇒ seed × seed where
    split-seed s = (let
        (v, w) = s;
        (v', w') = snd (next s);
        v'' = inc-shift 2147483562 v;
        w'' = inc-shift 2147483398 w
    in ((v'', w'), (v', w')))

70.3 Base selectors

context
    includes state-combinator-syntax
begin

fun iterate :: natural ⇒ ('b ⇒ 'a ⇒ 'b × 'a) ⇒ 'b ⇒ 'a ⇒ 'b × 'a where
    iterate k f x = (if k = 0 then Pair x else f x o→ iterate (k − 1) f)

definition range :: natural ⇒ seed ⇒ natural × seed where
    range k = iterate (log 2147483561 k)
        (λl, next o→ (λv. Pair (v + l * 2147483561))) 1
        o→ (λv. Pair (v mod k))

lemma range:
    k > 0 ⇒ fst (range k s) < k
THEORY "Random"

by (simp add: range-def split-def less-natural-def del; log.simps iterate.simps)

definition select :: 'a list ⇒ seed ⇒ 'a × seed where
  select xs = range (natural-of-nat (length xs))
  o→ (λk. Pair (nth xs (nat-of-natural k)))

lemma select:
  assumes xs ≠ []
  shows fst (select xs s) ∈ set xs
proof –
  from assms have natural-of-nat (length xs) > 0 by (simp add: less-natural-def)
  with range have fst (range (natural-of-nat (length xs)) s) < natural-of-nat (length xs) by best
  then have nat-of-natural (fst (range (natural-of-nat (length xs)) s)) < length xs by (simp add: less-natural-def)
  then show ?thesis by (simp add: split-beta select-def)
qed

primrec pick :: (natural × 'a) list ⇒ natural ⇒ 'a where
  pick (x # xs) i = (if i < fst x then snd x else pick xs (i - fst x))

lemma pick-member:
  i < sum-list (map fst xs) ⇒ pick xs i ∈ set (map snd xs)
by (induct xs arbitrary: i) (simp-all add: less-natural-def)

lemma pick-drop-zero:
  pick (filter (λ(k, -). k > 0) xs) = pick xs
by (induct xs) (auto simp add: fun-eq-iff less-natural-def minus-natural-def)

lemma pick-same:
  l < length xs ⇒ Random.pick (map (Pair 1) xs) (natural-of-nat l) = nth xs l
proof (induct xs arbitrary: l)
  case Nil then show ?case by simp
next
  case (Cons x xs) then show ?case by (cases l) (simp-all add: less-natural-def)
qed

definition select-weight :: (natural × 'a) list ⇒ seed ⇒ 'a × seed where
  select-weight xs = range (sum-list (map fst xs))
  o→ (λk. Pair (pick xs k))

lemma select-weight-member:
  assumes 0 < sum-list (map fst xs)
  shows fst (select-weight xs s) ∈ set (map snd xs)
proof –
  from range assms
    have fst (range (sum-list (map fst xs))) s < sum-list (map fst xs) .
with pick-member
  have pick xs (fst (range (sum-list (map fst xs)) s)) ∈ set (map snd xs) .
  then show ?thesis by (simp add: select-weight-def scomp-def split-def)
qed

lemma select-weight-cons-zero:
  select-weight ((0, x) # xs) = select-weight xs
  by (simp add: select-weight-def less-natural-def)

lemma select-weight-drop-zero:
  select-weight (filter (λ(k, -). k > 0) xs) = select-weight xs
proof –
  have sum-list (map fst [(k, -)← xs . 0 < k]) = sum-list (map fst xs)
    by (induct xs) (auto simp add: less-natural-def natural-eq-iff)
  then show ?thesis by (simp only: select-weight-def pick-drop-zero)
qed

lemma select-weight-select:
  assumes xs ≠ []
  shows select-weight (map (Pair 1) xs) = select xs
proof –
  have less: ∃s. fst (range (natural-of-nat (length xs)) s) < natural-of-nat (length xs)
    using assms by (intro range) (simp add: less-natural-def)
  moreover have sum-list (map fst (map (Pair 1) xs)) = natural-of-nat (length xs)
    by (induct xs) simp-all
  ultimately show ?thesis
  by (auto simp add: select-weight-def select-def scomp-def split-def
               fun-eq-iff pick-same [symmetric] less-natural-def)
qed

end

70.4 ML interface

code-reflect Random-Engine
  functions range select select-weight

ML (<
  structure Random-Engine =
    struct

    open Random-Engine;

    type seed = Code-Numeral.natural * Code-Numeral.natural;

    local
val seed = Unsynchronized.ref

(let
  val now = Time.toMilliseconds (Time.now ());
  val (q, s1) = IntInf.divMod (now, 2147483562);
  val s2 = q mod 2147483398;
  in apply2 Code-Numerals.natural-of-integer (s1 + 1, s2 + 1) end);

fun next-seed () = 
  let
    val (seed1, seed') = @
        {code split-seed} (! seed)
    val - = seed := seed'
    in
      seed1
    end

fun run f = 
  let
    val (x, seed') = f (! seed);
    val - = seed := seed'
    in x end;
  end;

> hide-type (open) seed
> hide-const (open) inc-shift minus-shift log next split-seed
>  iterate range select pick select-weight
> hide-fact (open) range-def
> end

71 Maps

theory Map
  imports List
  abbrevs (= = ⊆m
begin

  type-synonym ('a, 'b) map = 'a ⇒ ('b option (infixr → 0))

  abbreviation (input)
    empty :: 'a ⇒ 'b where
    empty ≡ λx. None

  definition
map-comp :: (′b → ′c) ⇒ (′a → ′b) ⇒ (′a → ′c) \text{(infixl} \circ_m 55) \text{ where}
\begin{align*}
f \circ_m g &= (\lambda k. \text{case } g k \text{ of } \text{None} \Rightarrow \text{None} | \text{Some } v \Rightarrow f v) \\
\end{align*}

\text{definition}
map-add :: (′a → ′b) ⇒ (′a → ′b) ⇒ (′a → ′b) \text{(infixl} ++ 100) \text{ where}
m_1 ++ m_2 = (\lambda x. \text{case } m_2 x \text{ of } \text{None} \Rightarrow m_1 x | \text{Some } y \Rightarrow \text{Some } y)

\text{definition}
restrict-map :: (′a ↵′b) ⇒ ′a set \text{ where}
m↾A = (\lambda x. \text{if } x \in A \text{ then } m x \text{ else } \text{None})

\text{notation (latex output)}
\begin{align*}
\text{restrict-map} &\iff [-. \{111,110\} 110]
\end{align*}

\text{definition}
dom :: (′a ↵′b) ⇒ ′a set \text{ where}
dom m = \{a. m a \neq \text{None}\}

\text{definition}
ran :: (′a → ′b) ⇒ ′b set \text{ where}
ran m = \{b. \exists a. m a = \text{Some } b\}

\text{definition}
graph :: (′a → ′b) ⇒ (′a × ′b) set \text{ where}
graph m = \{(a, b) | a b. m a = \text{Some } b\}

\text{definition}
map-le :: (′a → ′b) ⇒ (′a → ′b) ⇒ bool \text{ (infix} \subseteq_m 50) \text{ where}
(m_1 \subseteq_m m_2) \iff (\forall a \in \text{dom } m_1. m_1 a = m_2 a)

\begin{align*}
\text{Function update syntax } f(x := y, \ldots) \text{ is extended with } x \mapsto y, \text{ which is} \\
\text{short for } x := \text{Some } y. := \text{ and } \mapsto \text{ can be mixed freely. The syntax } [x \mapsto y, \\
\ldots] \text{ is short for } \text{Map.empty}(x \mapsto y, \ldots) \text{ but must only contain } \mapsto \text{, not } :=,
\end{align*}
because \([x:=y]\) clashes with the list update syntax \text{xs[i:=x]}.

\text{nonterminal} maplet and maplets

\text{syntax}
\begin{align*}
\text{-maplet} &:: [′a, ′a] \Rightarrow \text{maplet} \quad (-/\mapsto/\ -) \\
\text{:: maplet } \Rightarrow \text{updbind} \quad (-) \\
\text{:: maplet } \Rightarrow \text{maplets} \quad (-) \\
\text{-Maplets} &:: [\text{maplet, maplets}] \Rightarrow \text{maplets} \quad (-/\cdot\ -) \\
\text{-Map} &:: \text{maplets } \Rightarrow ′a \mapsto ′b \quad ((f[-]))
\end{align*}

\text{syntax (ASCII)}
\begin{align*}
\text{-maplet} &:: [′a, ′a] \Rightarrow \text{maplet} \quad (-/\mapsto/\ -) \\
\end{align*}

\text{translations}
\begin{align*}
\text{-Update } f &\iff f(x := \text{CONST } \text{Some } y)
\end{align*}
THEORY "Map"

-Maplets m ms → - updbinds m ms
-Map ms ← - Update (CONST empty) ms

-Map (-maplet x y) ← - Update (λu. CONST None) (-maplet x y)
-Map (- updbinds m (- maplet x y)) ← - Update (- Map m) (-maplet x y)

Updating with lists:

primrec map-of :: ('a × 'b) list ⇒ 'a ⇒ 'b where
map-of [] = empty
| map-of (p # ps) = (map-of ps)(fst p ↦→ snd p)

lemma map-of-Cons-code [code]:
map-of [] k = None
map-of ((l, v) # ps) k = (if l = k then Some v else map-of ps k)
by simp-all

definition map-upds :: ('a ⇒ 'b) ⇒ 'a list ⇒ 'b list ⇒ 'a ⇒ 'b where
map-upds m xs ys = m ++ map-of (rev (zip xs ys))

There is also the more specialized update syntax xs [↦→] ys for lists xs and ys.

syntax
-maplets :: ['a, 'a] ⇒ maplet  (- /[↦→]/ -)
syntax (ASCII)
-maplets :: ['a, 'a] ⇒ maplet  (- /[↦→]/ -)

translations
-Update m (-maplets xs ys) ⇌ CONST map-upds m xs ys
-Map (-maplets xs ys) ← - Update (λu. CONST None) (-maplets xs ys)
-Map (- updbinds m (-maplets xs ys)) ← - Update (- Map m) (-maplets xs ys)

71.1 empty

lemma empty-upd-none [simp]: empty(x := None) = empty
by (rule ext) simp

71.2 map-upd

lemma map-upd-triv: t k = Some x ⇒ t(k ↦→ x) = t
by (rule ext) simp

lemma map-upd-nonempty [simp]: t(k ↦→ x) ≠ empty
proof
assume t(k ↦→ x) = empty
then have (t(k ↦→ x)) k = None by simp
then show False by simp
THEORY "Map"

qed

lemma map-upd-eqD1:
  assumes m(a→x) = n(a→y)
  shows x = y
proof
  from assms have (m(a→x)) a = (n(a→y)) a by simp
  then show ?thesis by simp
qed

lemma map-upd-Some-unfold:
  (m(a→b)) x = Some y = (x = a ∧ b = y ∨ x ≠ a ∧ m x = Some y)
by auto

lemma image-map-upd [simp]: x ∉ A ⇒ m(x→y) ' A = m' A
by auto

lemma finite-range-updI:
  assumes finite (range f)
  shows finite (range (f(a→b)))
proof
  have range (f(a→b)) ⊆ insert (Some b) (range f)
  by auto
  then show ?thesis
  by (rule finite-subset) (use assms in auto)
qed

71.3 map-of

lemma map-of-eq-empty-iff [simp]:
  map-of xys = empty ←→ xys = []
proof
  show map-of xys = empty ⇒ xys = []
  by (induction xys) simp-all
qed simp

lemma empty-eq-map-of-iff [simp]:
  empty = map-of xys ←→ xys = []
by (subst eq-commute) simp

lemma map-of-eq-None-iff:
  (map-of xys x = None) = (x ∉ fst ' (set xys))
by (induct xys) simp-all

lemma map-of-eq-Some-iff [simp]:
  distinct(map fst xys) ⇒ (map-of xys x = Some y) = ((x,y) ∈ set xys)
proof (induct xys)
  case (Cons xy xys)
  then show ?case
  by (cases xy) (auto simp flip: map-of-eq-None-iff)
qed auto

lemma Some-eq-map-of-iff [simp]:
distinct(map fst xys) ⇒ (Some y = map-of xys x) = ((x,y) ∈ set xys)
by (auto simp del: map-of-eq-Some-iff simp: map-of-eq-Some-iff [symmetric])

lemma map-of-is-SomeI [simp]:
distinct(map fst xys); (x,y) ∈ set xys] ⇒ map-of xys x = Some y
by simp

lemma map-of-zip-is-None [simp]:
length xs = length ys =⇒ (map-of (zip xs ys) x = None) = (x /∈ set xs)
by (induct rule: list-induct2) simp-all

lemma map-of-zip-is-Some:
assumes length xs = length ys
shows x ∈ set xs ↔ (∃ y. map-of (zip xs ys) x = Some y)
using assms by (induct rule: list-induct2) simp-all

lemma map-of-zip-upd:
fixes x :: 'a and xs :: 'a list and ys zs :: 'b list
assumes length ys = length xs
and length zs = length xs
and x /∈ set xs
and (map-of (zip xs ys))(x ↦ y) = (map-of (zip xs zs))(x ↦ z)
shows map-of (zip xs ys) = map-of (zip xs zs)
proof
fix x' :: 'a
show map-of (zip xs ys) x' = map-of (zip xs zs) x'
proof (cases x = x')
case True
from assms True map-of-zip-is-None [of xs ys x']
have map-of (zip xs ys) x' = None by simp
moreover from assms True map-of-zip-is-None [of xs zs x']
have map-of (zip xs zs) x' = None by simp
ultimately show ?thesis by simp
next
case False from assms
have (((map-of (zip xs ys))(x ↦ y)) x' = (((map-of (zip xs zs))(x ↦ z)) x' by auto
with False show ?thesis by simp
qed
qed

lemma map-of-zip-inject:
assumes length ys = length xs
and length zs = length xs
and dist: distinct xs
and map-of: map-of (zip xs ys) = map-of (zip xs zs)
shows $ys = zs$
using assms(1) assms(2)[symmetric]
using dist map-of
proof (induct ys xs zs rule: list-induct3)
  case Nil show ?case by simp
next
  case (Cons y ys x xs z zs)
  from $\langle \text{map-of} \ (\text{zip} \ (x\#xs) \ (y\#ys)) \rangle = \text{map-of} \ (\text{zip} \ (x\#xs) \ (z\#zs))\rangle$
  have $\text{map-of} \ (\text{map-of} \ (\text{zip} \ xs \ ys))(x \mapsto y) = (\text{map-of} \ (\text{zip} \ xs \ zs))(x \mapsto z)$ by simp
  from Cons have $\text{length} \ ys = \text{length} \ xs$ and $\text{length} \ zs = \text{length} \ xs$
  and $x \notin \text{set} \ xs$ by simp-all
  then have $\text{map-of} \ (\text{zip} \ xs \ ys) = \text{map-of} \ (\text{zip} \ xs \ zs)$ using $\text{map-of}$ by (rule map-of-zip-upd)
  with Cons.hyps $\langle \text{distinct} \ (x \# xs) \rangle$ have $ys = zs$ by simp
  moreover from $\text{map-of}$ have $y = z$ by (rule map-upd-eqD1)
  ultimately show ?case by simp
qed

lemma map-of-zip-nth:
  assumes $\text{length} \ xs = \text{length} \ ys$
  assumes $\text{distinct} \ xs$
  assumes $i < \text{length} \ ys$
  shows $\text{map-of} \ (\text{zip} \ xs \ ys) \ (xs!i) = \text{Some} \ (ys!i)$
using assms proof (induct arbitrary: $i$ rule: list-induct2)
  case Nil then show ?case by simp
next
  case (Cons x xs y ys)
  then show ?case using less-Suc-eq-0-disj by auto
qed

lemma map-of-zip-map:
  $\text{map-of} \ (\text{zip} \ xs \ (\text{map} \ f \ xs)) = (\lambda x. \text{if} \ x \in \text{set} \ xs \text{ then Some} \ (f \ x) \text{ else None})$
by (induct xs) (simp-all add: fun-eq-iff)

lemma finite-range-map-of: $\text{finite} \ (\text{range} \ (\text{map-of} \ ys))$
proof (induct ys)
  case (Cons a ys)
  then show ?case using finite-range-updI by fastforce
qed auto

lemma map-of-SomeD: $\text{map-of} \ xs \ k = \text{Some} \ y \Longrightarrow (k, y) \in \text{set} \ xs$
by (induct xs) (auto split: if-splits)

lemma map-of-mapk-SomeI:
  $\text{inj} \ f \Longrightarrow \text{map-of} \ t \ k = \text{Some} \ x \Longrightarrow$
map-of (map (case-prod (λk. Pair (f k))) t) (f k) = Some x
by (induct t) (auto simp: inj-eq)

lemma weak-map-of-SomeI: (k, x) ∈ set l ⇒ ∃x. map-of l k = Some x
by (induct l) auto

lemma map-of-filter-in:
  map-of xs k = Some z ⇒ P k z ⇒ map-of (filter (case-prod P) xs) k = Some z
by (induct xs) auto

lemma map-of-map:
  map-of (map (λ(k, v). (k, f v)) xs) = map-option f ∘ map-of xs
by (induct xs) (auto simp: fun-eq-iff)

lemma dom-map-option:
  dom (λk. map-option (f k) (m k)) = dom m
by (simp add: dom-def)

lemma dom-map-option-comp [simp]:
  dom (map-option g ∘ m) = dom m
using dom-map-option [of λ- g m] by (simp add: comp-def)

71.4 map-option related

lemma map-option-o-empty [simp]: map-option f ∘ empty = empty
by (rule ext) simp

lemma map-option-o-map-upd [simp]:
  map-option f ∘ m(a→b) = (map-option f ∘ m)(a→f b)
by (rule ext) simp

71.5 map-comp related

lemma map-comp-empty [simp]:
  m o_m empty = empty
  empty o_m m = empty
by (auto simp: map-comp-def split: option.splits)

lemma map-comp-simps [simp]:
  m2 k = None ⇒ (m1 o_m m2) k = None
  m2 k = Some k' ⇒ (m1 o_m m2) k = m1 k'
by (auto simp: map-comp-def)

lemma map-comp-Some-iff:
  ((m1 o_m m2) k = Some v) = (∃k'. m2 k = Some k' ∧ m1 k' = Some v)
by (auto simp: map-comp-def split: option.splits)

lemma map-comp-None-iff:
\[(m_1 \circ_m m_2) k = \text{None} = (m_2 k = \text{None} \lor (\exists k'. m_2 k = \text{Some} k' \land m_1 k' = \text{None}))\]
by (auto simp: map-comp-def split: option.splits)

71.6 \[\text{++}\]

**lemma** map-add-empty [simp]: \[m ++ \text{empty} = m\]
by (simp add: map-add-def)

**lemma** empty-map-add [simp]: \[\text{empty} ++ m = m\]
by (rule ext) (simp add: map-add-def split: option.split)

**lemma** map-add-assoc [simp]: \[m_1 ++ (m_2 ++ m_3) = (m_1 ++ m_2) ++ m_3\]
by (rule ext) (simp add: map-add-def split: option.split)

**lemma** map-add-Some-iff:
\[((m ++ n) k = \text{Some} x) = (n k = \text{Some} x \lor n k = \text{None} \land m k = \text{Some} x)\]
by (simp add: map-add-def split: option.split)

**lemma** map-add-SomeD [dest!]:
\[(m ++ n) k = \text{Some} x \implies n k = \text{Some} x \lor n k = \text{None} \land m k = \text{Some} x\]
by (rule map-add-Some-iff [THEN iffD1])

**lemma** map-add-find-right [simp]: \[n k = \text{Some} x \implies (m ++ n) k = \text{Some} x\]
by (subst map-add-Some-iff)

**lemma** map-add-None [iff]: \[((m ++ n) k = \text{None}) = (n k = \text{None} \land m k = \text{None})\]
by (simp add: map-add-def split: option.split)

**lemma** map-add-upd [simp]: \[f ++ g(x \mapsto y) = (f ++ g)(x \mapsto y)\]
by (rule ext) (simp add: map-add-def)

**lemma** map-add-upds [simp]: \[m_1 ++ (m_2(xs \mapsto ys)) = (m_1 ++ m_2)(xs \mapsto ys)\]
by (simp add: map-upds-def)

**lemma** map-add-upd-left: \[m \notin \text{dom} e_2 \implies e_1(m \mapsto u_1) ++ e_2 = (e_1 ++ e_2)(m \mapsto u_1)\]
by (rule ext) (auto simp: map-add-def dom-def split: option.split)

**lemma** map-of-append [simp]: \[\text{map-of} (xs @ ys) = \text{map-of} ys ++ \text{map-of} xs\]
unfolding map-add-def

**proof** (induct xs)
  case (Cons a xs)
  then show \(?case\)
    by (force split: option.split)

**qed** auto

**lemma** finite-range-map-of-map-add:
\[\text{finite} (\text{range} f) \implies \text{finite} (\text{range} (f ++ \text{map-of} l))\]
proof (induct l)
case (Cons a l)
  then show ?case
    by (metis finite-range-updI map-add-upd map-of.simps(2))
qed auto

lemma inj-on-map-add-dom [iff]:
  inj-on (m ++ m') (dom m') = inj-on m' (dom m')
by (fastforce simp: map-add-def dom-def inj-on-def split: option.splits)

lemma map-upds-fold-map-upd:
  m (ks[↦→]vs) = foldl (λm (k, v). m(k ⇒ v)) m (zip ks vs)
unfolding map-ups-def proof (rule sym, rule zip-obtain-same-length)
  fix ks :: 'a list and vs :: 'b list
  assume length ks = length vs
  then show foldl (λm (k, v). m(k⇒v)) m (zip ks vs) = m ++ map-of (rev (zip ks vs))
    by (induct arbitrary: m rule: list-induct2) simp-all
qed

lemma map-add-map-of-foldr:
  m ++ map-of ps = foldr (λ(k, v). m(k⇒v)) ps m
by (induct ps) (auto simp: fun-eq-iff map-add-def)

71.7 restrict-map

lemma restrict-map-to-empty [simp]: m|’{}= empty
  by (simp add: restrict-map-def)

lemma restrict-map-insert: f |’(insert a A) = (f |’ A)(a := f a)
  by (auto simp: restrict-map-def)

lemma restrict-map-empty [simp]: empty|’D = empty
  by (simp add: restrict-map-def)

lemma restrict-in [simp]: x ∈ A ⇒ (m|’A) x = m x
  by (simp add: restrict-map-def)

lemma restrict-out [simp]: x ∉ A ⇒ (m|’A) x = None
  by (simp add: restrict-map-def)

lemma ran-restrictD: y ∈ ran (m|’A) ⇒ ∃ x ∈ A. m x = Some y
  by (auto simp: restrict-map-def ran-def split: if-split-asm)

lemma dom-restrict [simp]: dom (m|’A) = dom m ∩ A
  by (auto simp: restrict-map-def dom-def split: if-split-asm)

lemma restrict-upd-same [simp]: m(x⇒y)|’(− {x}) = m|’(− {x})
  by (rule ext) (auto simp: restrict-map-def)
lemma restrict-restrict [simp]: \( m\upharpoonright A\cap B = m\upharpoonright (A\cap B) \)
by (rule ext) (auto simp: restrict-map-def)

lemma restrict-fun-upd [simp]:
\( m(x := y)\upharpoonright D = (if x \in D then (m\upharpoonright (D - \{x\}))(x := y) else m\upharpoonright D) \)
by (simp add: restrict-map-def fun-eq-iff)

lemma fun-upd-restrict [simp]:
\( (m\upharpoonright D)(x := y) = (m\upharpoonright (D - \{x\}))(x := y) \)
by (rule fun-upd-restrict)

lemma map-of-map-restrict [simp]:
map_of (map (\lambda k. (k, f k)) ks) = (Some \circ f)\upharpoonright \text{set} ks
by (induct ks) (simp-all add: fun-eq-iff restrict-map-insert)

lemma restrict-complement-singleton-eq:
\( f\upharpoonright (- \{x\}) = f(x := \text{None}) \)
by auto

71.8 \( \text{map-upds} \)

lemma map-upds-Nil1 [simp]: \( m(\bar[] \mapsto bs) = m \)
by (simp add: map-upds-def)

lemma map-upds-Nil2 [simp]: \( m(as \mapsto \bar[]) = m \)
by (simp add:map-upds-def)

lemma map-upds-Cons [simp]: \( m(a\#as \mapsto b\#bs) = (m(a\mapsto b))(as\mapsto bs) \)
by (simp add:map-upds-def)

lemma map-upds-append1 [simp]:
size xs < size ys \implies m(xs@\bar[x] \mapsto ys) = m(xs \mapsto ys, x \mapsto ys\upharpoonright \text{size} xs)\)
proof (induct xs arbitrary: ys m)
case Nil
then show ?case
by (auto simp: neq-Nil-conv)
next
case (Cons a xs)
then show ?case
by (cases ys) auto
qed
lemma map-upds-list-update2-drop [simp]:
size xs ≤ i ⇒ m(xs[→]ys[i:=y]) = m(xs[→]ys)
proof (induct xs arbitrary: m ys i)
  case Nil
  then show ?case by auto
next
  case (Cons a xs)
  then show ?case
  proof (cases ys)
    case (Cons z zs)
    then show ?thesis using Cons. hyps apply fastforce
  done
qed auto

Something weirdly sensitive about this proof, which needs only four lines in apply style

lemma map-upd-upds-conv-if:
  (f(x→y))(xs[→]ys) =
  (if x ∈ set(take (length ys) xs) then f(xs[→]ys)
    else (f(xs[→]ys))(x→y))
proof (induct xs arbitrary: x y ys f)
  case (Cons a xs)
  show ?case
  proof (cases ys)
    case (Cons z zs)
    then show ?thesis using Cons. hyps apply fastforce
  done
qed auto

lemma fun-upds-append-drop [simp]:
size xs = size ys ⇒ m(xs@zs[→]ys) = m(xs[→]ys)
proof (induct xs arbitrary: ys)
case (Cons a xs)

lemma map-upds-twist [simp]:
a /∈ set as ⇒ m(a→b, as[→]bs) = m(as[→]bs, a→b)
using set-take-subset by (fastforce simp add: map-upd-upds-conv-if)

lemma map-ups-apply-nontin [simp]:
x /∈ set xs ⇒ f(xs[→]ys) = f x
proof (induct xs arbitrary: ys)
case (Cons a xs)
then show ?case
  by (cases ys) (auto simp: map-upd-upds-conv-if)
qed auto
then show ?case
  by (cases ys) (auto simp: map-upd-upds-conv-if)
qed auto

lemma fun-upds-append2-drop [simp]:
  size xs = size ys \implies m(xs[\mapsto]ys@zs) = m(xs[\mapsto]ys)
proof (induct xs arbitrary: ys)
  case (Cons a xs)
  then show ?case
  by (cases ys) (auto simp: map-upd-upds-conv-if)
qed auto

lemma restrict-map-upds [simp]:
[ size xs = length ys; set xs \subseteq D ]
  \implies m(xs[\mapsto]ys)[D = (m|^{-} set xs)](xs[\mapsto]ys)
proof (induct xs arbitrary: m ys)
  case (Cons a xs)
  then show ?case
  proof (cases ys)
    case (Cons z zs)
    with Cons.hyps Cons.prems show ?thesis
    apply (simp add: insert-absorb flip: Diff-insert)
    apply (auto simp add: map-upd-upds-conv-if)
    done
  qed auto
qed auto

71.9 dom

lemma dom-eq-empty-conv [simp]: dom f = {} \iff f = empty
by (auto simp: dom-def)

lemma domI: m a = Some b \implies a \in dom m
by (simp add: dom-def)

lemma domD: a \in dom m \implies \exists b. m a = Some b
by (cases m a) (auto simp add: dom-def)

lemma domIff [iff, simp del, code-unfold]: a \in dom m \iff m a \neq None
by (simp add: dom-def)

lemma dom-empty [simp]: dom empty = {}
by (simp add: dom-def)

lemma dom-fun-upd [simp]:
  dom(f(x := y)) = (if y = None then dom f - {x} else insert x (dom f))
by (auto simp: dom-def)
lemma dom-if:
  \(\text{dom} (\lambda x. \text{if } P x \text{ then } f x \text{ else } g x) = \text{dom } f \cap \{x. \ P x\} \cup \text{dom } g \cap \{x. \neg P x\}\)
by (auto split: if-splits)

lemma dom-map-of-cone-image-fst:
  \(\text{dom} (\text{map-of } xys) = \text{fst} \cdot \text{set xys}\)
by (induct xys) (auto simp add: dom-if)

lemma dom-map-of-zip [simp]:
  \(\text{length } xs = \text{length } ys \implies \text{dom} (\text{map-of } (\text{zip } xs \ ys)) = \text{set } xs\)
by (induct rule: list-induct2) (auto simp: dom-if)

lemma finite-dom-map-of: finite (\(\text{dom} (\text{map-of } l)\))
by (induct l) (auto simp: dom-def Collect [symmetric])

lemma dom-map-upds [simp]:
  \(\text{dom}(\text{m}(\text{xs}[\mapsto]ys)) = \text{set}(\text{take}(\text{length } ys) \ xs) \cup \text{dom } m\)
proof (induct xs arbitrary: ys)
  case (Cons a xs)
  then show ?case
  by (cases ys) (auto simp: map-upd-upds-conv-if)
qed auto

lemma dom-map-add [simp]:
  \(\text{dom } (m ++ n) = \text{dom } n \cup \text{dom } m\)
by (auto simp: dom-def)

lemma dom-override-on [simp]:
  \(\text{dom} (\text{override-on } f g A) =\)
  \((\text{dom } f - \{a. \ a \in A - \text{dom } g\}) \cup \{a. \ a \in A \cap \text{dom } g\}\)
by (auto simp: dom-def override-on-def)

lemma map-add-comm: dom m1 \cap dom m2 = {} \implies m1 ++ m2 = m2 ++ m1
by (rule ext) (force simp: map-add-def dom-def split: option.split)

lemma map-add-dom-app-simps:
  \(m \in \text{dom } l2 \implies (l1 ++ l2) m = l2 m\)
  \(m \notin \text{dom } l1 \implies (l1 ++ l2) m = l2 m\)
  \(m \notin \text{dom } l2 \implies (l1 ++ l2) m = l1 m\)
by (auto simp add: map-add-def split: option.split_asm)

lemma dom-const [simp]:
  \(\text{dom} (\lambda x. \text{Some } (f x)) = \text{UNIV}\)
by auto

lemma finite-map-freshness:
  finite (\(\text{dom} (f :: 'a \rightarrow 'b)\)) \implies \neg \text{finite } (\text{UNIV :: 'a set}) \implies
  \exists x. f x = \text{None}\)
by (bestsimp dest: ex-new-if-finite)

lemma dom-minus:
\[ f(x) = \text{None} \implies \text{dom}(f) - \text{insert}(x, A) = \text{dom}(f) - A \]
unfolding dom-def by simp

lemma insert-dom:
\[ f(x) = \text{Some}(y) \implies \text{insert}(x, \text{dom}(f)) = \text{dom}(f) \]
unfolding dom-def by auto

lemma map-of-map-keys:
\[ \text{set}(xs) = \text{dom}(m) \implies \text{map-of}(\map{\lambda k. (k, \text{the}(m(k)))}(xs)) = m \]
by (rule ext) (auto simp add: map-of-map-restrict restrict-map-def)

lemma map-of-eq1:
\[ \text{assumes } \text{set-eq}: \text{set}(\map{\text{fst}}(xs)) = \text{set}(\map{\text{fst}}(ys)) \]
\[ \text{assumes } \text{map-eq}: \forall k \in \text{set}(\map{\text{fst}}(xs)). \map{\text{map-of}}(xs, k) = \map{\text{map-of}}(ys, k) \]
\[ \text{shows } \map{\text{map-of}}(xs) = \map{\text{map-of}}(ys) \]
proof (rule ext)
\[ \text{fix } k \text{ show } \map{\text{map-of}}(xs, k) = \map{\text{map-of}}(ys, k) \]
proof (cases \map{\text{map-of}}(xs, k))
\[ \text{case } \text{None} \]
then have \[ k \notin \text{set}(\map{\text{fst}}(xs)) \] by (simp add: map-of-eq-None-iff)
with \text{set-eq} have \[ k \notin \text{set}(\map{\text{fst}}(ys)) \] by simp
then have \[ \map{\text{map-of}}(ys, k) = \text{None} \] by (simp add: map-of-eq-None-iff)
with \text{None} show \[ ?\text{thesis} \] by simp
next
\[ \text{case } (\text{Some } v) \]
then have \[ k \in \text{set}(\map{\text{fst}}(xs)) \] by (auto simp add: dom-map-of-conv-image-fst [symmetric])
with \text{map-eq} show \[ ?\text{thesis} \] by auto
qed

lemma map-of-eq-dom:
\[ \text{assumes } \map{\text{map-of}}(xs) = \map{\text{map-of}}(ys) \]
\[ \text{shows } \map{\text{fst}} \cdot \text{set}(xs) = \map{\text{fst}} \cdot \text{set}(ys) \]
proof
\[ \text{from } \text{assms } \text{have } \text{dom}(\map{\text{map-of}}(xs)) = \text{dom}(\map{\text{map-of}}(ys)) \text{ by simp} \]
then show \[ ?\text{thesis} \] by (simp add: dom-map-of-conv-image-fst)
qed

lemma finite-set-of-finite-maps:
\[ \text{assumes } \text{finite } A \text{ finite } B \]
\[ \text{shows } \text{finite } \{ m. \text{ dom } m = A \land \text{ ran } m \subseteq B \} \text{ (is finite } ?S) \]
proof
\[ \text{let } ?S' = \{ m. \forall x. (x \in A \implies m(x) \in \text{Some } B) \land (x \notin A \implies m(x) = \text{None}) \} \]
\[ \text{have } ?S = ?S' \]
proof

show \(?S \subseteq \?S'\) by (auto simp: dom-def ran-def image-def)
show \(?S' \subseteq \?S\)
proof
  fix \(m\) assume \(m \in \?S'\)
  hence 1: \(\text{dom } m = A\) by force
  hence 2: \(\text{ran } m \subseteq B\) using \(m \in \?S'\) by (auto simp: dom-def ran-def)
  from 1 2 show \(m \in \?S\) by blast
qed
qed

with assms show ?thesis by (simp add: finite-set-of-finite-funs)
qed

71.10 \(\text{ran}\)

lemma ranI: \(m \ a = \text{Some } b \implies b \in \text{ran } m\)
by (auto simp: ran-def)

lemma ran-empty [simp]: \(\text{ran } \text{empty} = \{\}\)
by (auto simp: ran-def)

lemma ran-map-upd [simp]: \(m \ a = \text{None} \implies \text{ran}(m(a \mapsto b)) = \text{insert } b \ (\text{ran } m)\)
unfolding ran-def
by force

lemma fun-upd-None-if-notin-dom[simp]: \(k \notin \text{dom } m \implies m(k := \text{None}) = m\)
by auto

lemma ran-upd-None-if-notin-dom: \([ m \ x = \text{Some } y; \text{inj-on } m \ (\text{dom } m)\); \(z \notin \text{ran } m\] \implies \text{ran}(m(x := \text{Some } z)) = \text{ran } m - \{y\} \cup \{z\}\)
by (force simp add: ran-def domI inj-onD)

lemma ran-map-add:
  assumes dom m1 \(\cap\) dom m2 = \{\}
  shows \(\text{ran } (m1 ++ m2) = \text{ran } m1 \cup \text{ran } m2\)
proof
  show \(\text{ran } (m1 ++ m2) \subseteq \text{ran } m1 \cup \text{ran } m2\)
  unfolding ran-def by auto
next
  show \(\text{ran } m1 \cup \text{ran } m2 \subseteq \text{ran } (m1 ++ m2)\)
  proof
    have \((m1 ++ m2) \ x = \text{Some } y \text{ if } m1 \ x = \text{Some } y \text{ for } x y\)
      using assms map-add-comm that by fastforce
    moreover have \((m1 ++ m2) \ x = \text{Some } y \text{ if } m2 \ x = \text{Some } y \text{ for } x y\)
      using assms that by auto
    ultimately show ?thesis
    unfolding ran-def by blast
  qed
**THEORY “Map”**

qed

**lemma** finite-ran:
  **assumes** finite (dom p)
  **shows** finite (ran p)
  **proof**
    have ran p = (λx. the (p x)) ∘ dom p
    unfolding ran-def by force
    from this (finite (dom p)) show ?thesis by auto
  qed

**lemma** ran-distinct:
  **assumes** dist: distinct (map fst al)
  **shows** ran (map-of al) = snd ' set al
  using assms
  **proof** (induct al)
  case Nil
  then show ?case by simp
  next
  case (Cons kv al)
  then have ran (map-of al) = snd ' set al by simp
  moreover from Cons.prems have map-of al (fst kv) = None
    by (simp add: map-of-eq-None-iff)
  ultimately show ?case by (simp only: map-of.simps ran-map-upd) simp
  qed

**lemma** ran-map-of-zip:
  **assumes** length xs = length ys distinct xs
  **shows** ran (map-of (zip xs ys)) = set ys
  using assms by (simp add: ran-distinct set-map[symmetric])

**lemma** ran-map-option:
  **shows** ran (λx. map-option f (m x)) = f ' ran m
  by (auto simp add: ran-def)

**71.11** graph

**lemma** graph-empty[simp]: graph empty = {}
  unfolding graph-def by simp

**lemma** in-graphI: m k = Some v ⇒ (k, v) ∈ graph m
  unfolding graph-def by blast

**lemma** in-graphD: (k, v) ∈ graph m ⇒ m k = Some v
  unfolding graph-def by blast

**lemma** graph-map-upd[simp]: graph (m(k ↦ v)) = insert (k, v) (graph (m(k := None)))
  unfolding graph-def by (auto split: if-splits)
lemma graph-fun-upd-None: graph (m(k := None)) = { e ∈ graph m. fst e ≠ k} 
unfolding graph-def by (auto split: if-splits)

lemma graph-restrictD:  
  assumes (k, v) ∈ graph (m |' A)  
  shows k ∈ A and m k = Some v  
  using assms unfolding graph-def  
  by (auto simp: restrict-map-def split: if-splits)

lemma graph-map-comp [simp]: graph (m1 ◦_m m2) = graph m2 O graph m1  
unfolding graph-def by (auto simp: map-comp-Some-iff relcomp-unfold)

lemma graph-map-add: dom m1 ∩ dom m2 = {} ⇒ graph (m1 ++ m2) = graph m1 ∪ graph m2  
unfolding graph-def using map-add-comm by force

lemma graph-eq-to-snd-dom: graph m = (λx. (x, the (m x))) |' dom m  
unfolding graph-def dom-def by force

lemma graph-domD: x ∈ graph m ⇒ fst x ∈ dom m  
using fst-graph-eq-dom by (metis imageI)

lemma snd-graph-ran: snd |' graph m = ran m  
unfolding graph-def ran-def by force

lemma graph-ranD: x ∈ graph m ⇒ snd x ∈ ran m  
using snd-graph-ran by (metis imageI)

lemma finite-graph-map-of: finite (graph (map-of al))  
unfolding graph-eq-to-snd-dom finite-dom-map-of  
using finite-dom-map-of by blast

lemma graph-map-of-if-distinct-dom: distinct (map fst al) ⇒ graph (map-of al) = set al  
unfolding graph-def by auto

lemma finite-graph-if-finite-dom[simp]: finite (graph m) = finite (dom m)  
by (metis graph-eq-to-snd-dom finite-imageI fst-graph-eq-dom)

lemma inj-on-fst-graph: inj-on fst (graph m)  
unfolding graph-def inj-on-def by force

71.12 map-le

lemma map-le-empty [simp]: empty ⊆_m g  
by (simp add: map-le-def)
lemma upd-None-map-le \[\text{simp}]: \( f(x := \text{None}) \subseteq_m f \) by (force simp add: map-le-def)

lemma map-le-upd \[\text{simp}]: \( f \subseteq_m g \rightleftharpoons f(a := b) \subseteq_m g(a := b) \) by (fastforce simp add: map-le-def)

lemma map-le-imp-upd-le \[\text{simp}]: \( m_1 \subseteq m \to m_2 \rightleftharpoons m_1(x := \text{None}) \subseteq_m m_2(x \mapsto y) \) by (force simp add: map-le-def)

lemma map-le-upds \[\text{simp}]: \( f \subseteq m \to g \rightleftharpoons f(as \mapsto bs) \subseteq_m g(as \mapsto bs) \) by (cases bs) (use Cons in auto)

proof (induct as arbitrary: \( f \to g \to bs \))
    case (Cons a as)
    then show \( \text{?case} \) by (cases bs)
        (use Cons in auto)
    qed auto

lemma map-le-implies-dom-le: \( f \subseteq_m g \rightleftharpoons \text{dom } f \subseteq \text{dom } g \) by (fastforce simp add: map-le-def dom-def)

lemma map-le-refl \[\text{simp}]: \( f \subseteq_m f \) by (simp add: map-le-def)

lemma map-le-trans \[\text{trans}]: \( m_1 \subseteq_m m_2; m_2 \subseteq_m m_3 \rightleftharpoons m_1 \subseteq_m m_3 \) by (auto simp add: map-le-def dom-def)

lemma map-le-antisym: \( f \subseteq_m g; g \subseteq_m f \rightleftharpoons f = g \) unfolding map-le-def by (metis ext domIff)

lemma map-le-map-add \[\text{simp}]: \( f \subseteq_m g + + f \) by (fastforce simp: map-le-def)

lemma map-le-iff-map-add-commute: \( f \subseteq_m f + + g \rightleftharpoons f + + g = g + + f \) by (fastforce simp: map-add-def map-le-def fun-eq-iff split: option.splits)

lemma map-add-le-mapE: \( f + + g \subseteq_m h \rightleftharpoons g \subseteq_m h \) by (fastforce simp: map-le-def map-add-def dom-def)

lemma map-add-le-mapI: \( f + + g \subseteq_m h \rightleftharpoons f + + g \subseteq_m h \) by (auto simp: map-le-def map-add-def dom-def split: option.splits)

lemma map-add-subsumed1: \( f \subseteq_m g \rightleftharpoons f + + g = g \) by (simp add: map-add-le-mapI map-le-antisym)

lemma map-add-subsumed2: \( f \subseteq_m g \rightleftharpoons g + + f = g \) by (metis map-add-subsumed1 map-le-iff-map-add-commute)
lemma dom-eq-singleton-conv:
\[ \text{dom } f = \{ x \} \iff (\exists v. f = [x \mapsto v]) \]
(is \(?\text{lhs} \iff ?\text{rhs}\))

\textbf{proof}
\begin{itemize}
  \item assume \(?\text{rhs}\)
  \begin{itemize}
    \item then show \(?\text{lhs}\) by (auto split: if-split-asm)
  \end{itemize}
\end{itemize}

\textbf{next}
\begin{itemize}
  \item assume \(?\text{lhs}\)
  \begin{itemize}
    \item then obtain \(v\) where \(v\) \(f x = \text{Some } v\) by auto
    \begin{itemize}
      \item show \(?\text{rhs}\) proof
      \begin{itemize}
        \item show \(f = [x \mapsto v]\) proof
        \begin{itemize}
          \item (rule map-le-antisym)
          \begin{itemize}
            \item show \([x \mapsto v] \subseteq_m f\)
            \begin{itemize}
              \item using \(v\) by (auto simp add: map-le-def)
              \begin{itemize}
                \item show \(f \subseteq_m [x \mapsto v]\)
                \begin{itemize}
                  \item using \(\text{dom } f = \{ x \}\), \(f x = \text{Some } v\) by (auto simp add: map-le-def)
                \end{itemize}
              \end{itemize}
            \end{itemize}
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}
\end{itemize}

qed

lemma map-add-eq-empty-iff[simp]:
\[(f +\mapsto g = \text{empty}) \iff f = \text{empty} \land g = \text{empty}\]
by (metis map-add-None)

lemma empty-eq-map-add-iff[simp]:
\[(\text{empty} = f +\mapsto g) \iff f = \text{empty} \land g = \text{empty}\]
by (subst map-add-eq-empty-iff[symmetric])(rule eq-commute)

71.13 Various

lemma set-map-of-compr:
\begin{itemize}
  \item assumes \text{distinct}: \text{distinct} \(\mapst{\text{map fst xs}}\)
  \item shows \(\mapst{\text{set xs}} = \{(k, v). \mapst{\text{map of xs k = Some } v}\}\)
  \begin{itemize}
    \item using \text{assms}
  \end{itemize}
\end{itemize}

\textbf{proof (induct \text{xs})}
\begin{itemize}
  \item case \(\text{Nil}\)
  \begin{itemize}
    \item then show \(?\text{case by simp}\)
  \end{itemize}
\end{itemize}

\textbf{next}
\begin{itemize}
  \item case \(\text{Cons } x \text{ xs}\)
  \begin{itemize}
    \item obtain \(k\ v\) where \(x = (k, v)\) by (cases \(x\) blast
    \begin{itemize}
      \item with \text{Cons},\text{prems} have \(k \notin \text{dom } (\mapst{\text{map of xs}})\)
      \begin{itemize}
        \item by (simp add: \mapst{\text{dom-map-of-conv-image-fst}})
      \end{itemize}
    \end{itemize}
    \begin{itemize}
      \item then have \(*\): \text{insert } (k, v) \{(k, v). \mapst{\text{map of xs k = Some } v}\} = \\
            \{(k', v'). (\mapst{\text{map of xs}}(k \mapsto v)) k' = \text{Some } v'\}
      \begin{itemize}
        \item by (auto split: if-splits)
      \end{itemize}
    \end{itemize}
  \end{itemize}
  \begin{itemize}
    \item from \text{Cons} have \(\mapst{\text{set xs}} = \{(k, v). \mapst{\text{map of xs k = Some } v}\}\) by simp
    \begin{itemize}
      \item with \(*\): \(x = (k, v)\), show \(?\text{case by simp}\)
    \end{itemize}
  \end{itemize}
\end{itemize}

qed
lemma \text{eq-key-imp-eq-value}:
\begin{align*}
v1 &= v2 \\
\text{if } \text{distinct} \ (\text{map } \text{fst } \text{xs}) \ (k, v1) \in \text{set } \text{xs} \ (k, v2) \in \text{set } \text{xs}
\end{align*}
proof
from that have \text{inj-on } \text{fst } \text{(set } \text{xs)}
by (simp add: \text{distinct-map})
moreover have \text{fst } (k, v1) = \text{fst } (k, v2)
by simp
ultimately have \ (k, v1) = \ (k, v2)
by (rule \text{inj-onD}) (\text{fact that})+
then show \ ?\text{thesis}
by simp
qed

lemma \text{map-of-inject-set}:
assumes \text{distinct: distinct } \text{(map } \text{fst } \text{xs}) \ \text{distinct } \text{(map } \text{fst } \text{ys})
shows \text{map-of } \text{xs} = \text{map-of } \text{ys} \longleftrightarrow \text{set } \text{xs} = \text{set } \text{ys (is } \ ?\text{lhs } \longleftrightarrow \ ?\text{rhs)}
proof
assume \ ?\text{lhs}
moreover from \ \langle \text{distinct } \text{(map } \text{fst } \text{xs})\rangle \ \text{have } \text{set } \text{xs} = \{\langle k, v \rangle. \text{map-of } \text{xs} \ k = \text{Some } v\}
by (rule \text{set-map-of-compr})
moreover from \ \langle \text{distinct } \text{(map } \text{fst } \text{ys})\rangle \ \text{have } \text{set } \text{ys} = \{\langle k, v \rangle. \text{map-of } \text{ys} \ k = \text{Some } v\}
by (rule \text{set-map-of-compr})
ultimately show \ ?\text{rhs by simp}
next
assume \ ?\text{rhs show } \ ?\text{lhs}
proof
fix \ k
show \text{map-of } \text{xs} \ k = \text{map-of } \text{ys} \ k
proof \ (cases \text{map-of } \text{xs} \ k)
case \text{None}
with \ ?\text{rhs} \ have \text{map-of } \text{ys} \ k = \text{None}
by (simp add: \text{map-of-eq-None-iff})
with \text{None show } \ ?\text{thesis by simp}
next
case \text{Some } v
with \ ?\text{rhs} \ have \text{map-of } \text{ys} \ k = \text{Some } v
by simp
with \text{Some show } ?\text{thesis by simp}
qed
qed

lemma \text{finite-Map-induct}[\text{consumes 1}, \text{case-names empty update}]:
assumes \text{finite } \ (\text{dom } \text{m})
assumes \text{P } \text{Map.empty}
assumes \( \forall k \, v. \text{finite } (\text{dom } m) \implies k \notin \text{dom } m \implies P \, m \implies P \,(m(k \mapsto v)) \)
shows \( P \, m \)
using assms(1)
proof(induction \text{dom } m \text{ arbitrary}; \text{m rule: finite-induct})
case empty
then show ?case using assms(2) unfolding \text{dom-def} by simp
next
case (insert \, x \, F)
then have finite \((\text{dom } (m(x:=\text{None})))\) \( x \notin \text{dom } (m(x:=\text{None})) \) \( P \,(m(x:=\text{None})) \)
by (metis \text{Diff-insert-absorb} \text{dom-fun-upd}+)
with assms(3)[OF this] show ?case
by (metis \text{fun-upd-triv} \text{fun-upd-upd} \text{option.exhaust})
qed

hide-const (open) \text{Map.empty} \text{Map.graph}

end

72 Finite types as explicit enumerations

theory Enum
imports Map Groups-List
begin

72.1 Class enum

class enum =
fixes enum :: \('a\ list
fixes enum-all :: ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}
fixes enum-ex :: ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}
assumes \text{UNIV-enum: } \text{UNIV} = \text{set enum}
and \text{enum-distinct: } \text{distinct enum}
assumes \text{enum-all-UNIV: } \text{enum-all} P \iff \text{Ball UNIV } P
assumes \text{enum-ex-UNIV: } \text{enum-ex} P \iff \text{Bex UNIV } P
— tailored towards simple instantiation
begin

subclass finite proof
qed (simp add: \text{UNIV-enum})

lemma \text{enum-UNIV:}
set enum = \text{UNIV}
by (simp only: \text{UNIV-enum})

lemma \text{in-enum: } x \in \text{set enum}
by (simp add: \text{enum-UNIV})

lemma \text{enum-eq-I:}
assumes \( \forall x. x \in \text{set } xs \)
shows set enum = set xs

proof –
  from assms UNIV-eq-I have UNIV = set xs by auto
  with enum-UNIV show ?thesis by simp
qed

lemma card-UNIV-length-enum:
  card (UNIV :: 'a set) = length enum
  by (simp add: UNIV-enum distinct-card enum-distinct)

lemma enum-all [simp]:
  enum-all = HOL.All
  by (simp add: fun-eq-iff enum-all-UNIV)

lemma enum-ex [simp]:
  enum-ex = HOL.Ex
  by (simp add: fun-eq-iff enum-ex-UNIV)

end

72.2 Implementations using enum

72.2.1 Unbounded operations and quantifiers

lemma Collect-code [code]:
  Collect P = set (filter P enum)
  by (simp add: enum-UNIV)

lemma vimage-code [code]:
  f −' B = set (filter (λx. f x ∈ B) enum-class.enum)
  unfolding vimage-def Collect-code ..

definition card-UNIV :: 'a itself ⇒ nat
where
  [code del]: card-UNIV TYPE('a) = card (UNIV :: 'a set)

lemma [code]:
  card-UNIV TYPE('a :: enum) = card (set (Enum.enum :: 'a list))
  by (simp only: card-UNIV-def enum-UNIV)

lemma all-code [code]: (∀x. P x) ↔ enum-all P
  by simp

lemma exists-code [code]: (∃x. P x) ↔ enum-ex P
  by simp

lemma exists1-code [code]: (∃!x. P x) ↔ list-ex1 P enum
  by (auto simp add: list-ex1-iff enum-UNIV)
72.2.2 An executable choice operator

definition [code del]: enum-the = The

lemma [code]:
The $P = (\text{case filter } P \text{ enum of } [x] \Rightarrow x \mid - \Rightarrow \text{enum-the } P)$
proof 
{
  fix $a$
  assume filter-enum: filter $P$ enum = $[a]$
  have $P = a$
    proof (rule the-equality)
      fix $x$
      assume $P x$
      show $x = a$
        proof (rule ccontr)
          assume $x \neq a$
          from filter-enum obtain $us \ vdashvs$
            where enum-eq: enum = $us \uplus [a] \uplus vs$
            and $\forall x \in \text{set } us. \neg P x$
            and $\forall x \in \text{set } vs. \neg P x$
            and $P a$
          by (auto simp add: filter-eq-Cons-iff) (simp only: filter-empty-conv[symmetric])
          with $\langle P x \rangle$ in-enum[of $x$, unfolded enum-eq] $\langle x \neq a \rangle$
          show False by auto
        qed
    next
    from filter-enum show $P a$ by (auto simp add: filter-eq-Cons-iff)
    qed
\}
from this show radioactive unfolding enum-the-def by (auto split: list.split)
qed

declare [[code abort: enum-the]]

code-printing
  constant enum-the \to \Eval (fn \_ \Rightarrow \raise Match)

72.2.3 Equality and order on functions

instantiation fun :: (enum, equal) equal
begin

definition
  $HOL.\text{equal } f \, g \iff (\forall x \in \text{set enum}. \, f \, x = g \, x)$

instance proof
  qed (simp-all add: equal-fun-def fun-eq-iff enum-UNIV)
lemma [code]:
HOL.equal f g \iff\ enum-all (\%x. f x = g x)
by (auto simp add: equal fun-eq-iff)

lemma [code nbe]:
HOL.equal (f :: 'a\Rightarrow 'b) f \iff True
by (fact equal-refl)

lemma order-fun [code]:
fixes f g :: 'a::enum \Rightarrow 'b::order
shows f \le\ g \iff\ enum-all (\%x. f x \le g x)
and f < g \iff f \le g \wedge\ enum-ex (\%x. f x \neq g x)
by (simp-all add: fun-eq-iff le-fun-def order-less-le)

72.2.4 Operations on relations

lemma [code]:
Id = \image (\%x. (x, x)) (set Enum.enum)
by (auto intro: imageI in-enun)

lemma tranclp-unfold [code]:
\tranclp r a b \iff (a, b) \in \trancl \{(x, y). r x y\}
by (simp add: trancl-def)

lemma rtranclp-rtrancl-eq [code]:
rtranclp r x y \iff (x, y) \in \rtrancl \{(x, y). r x y\}
by (simp add: rtrancl-def)

lemma max-ext-eq [code]:
max-ext R = \{(x, y). \finite X \land \finite Y \land Y \neq \{} \land (\\forall x. x \in X \rightarrow (\\exists xa \in Y. (x, xa) \in R))\}
by (auto simp add: max-ext-simps)

lemma max-extp-eq [code]:
max-extp r x y \iff (x, y) \in \max-ext \{(x, y). r x y\}
by (simp add: max-ext-def)

lemma mlex-eq [code]:
f <\ast\mlex\ast> R = \{(x, y). f x < f y \lor (f x \le f y \land (x, y) \in R)\}
by (auto simp add: mlex-prod-def)

72.2.5 Bounded accessible part

primrec bacc :: ('a \times 'a) set \Rightarrow nat \Rightarrow 'a set
where
bacc r 0 = \{x. \forall y. (y, x) \notin r\}
| bacc r (Suc n) = (bacc r n \cup \{x. \forall y. (y, x) \in r \rightarrow y \in \bacc r n\})
lemma bacc-subseteq-acc:
  bacc r n ⊆ Wellfounded.acc r
by (induct n) (auto intro: acc.intros)

lemma bacc-mono:
  n ≤ m ⇒ bacc r n ⊆ bacc r m
by (induct rule: dec-induct) auto

lemma bacc-upper-bound:
  bacc (r :: ('a × 'a) set) (card (UNIV :: 'a::finite set)) = (⋃ n. bacc r n)
proof
  have mono (bacc r) unfolding mono-def by (simp add: bacc-mono)
  moreover have ∀ n. bacc r n = bacc r (Suc n) → bacc r (Suc n) = bacc r (Suc n)
  by auto
  moreover have finite (range (bacc r)) by auto
  ultimately show ?thesis
  by (intro finite-mono-strict-prefix-implies-finite-fixpoint)
    (auto intro: finite-mono-remains-stable-implies-strict-prefix)
qed

lemma acc-subseteq-bacc:
  assumes finite r
  shows Wellfounded.acc r ⊆ (⋃ n. bacc r n)
proof
  fix x
  assume x ∈ Wellfounded.acc r
  then have ∃ n. x ∈ bacc r n
proof (induct x arbitrary: rule: acc.induct)
  case (accI x)
  then have ∀ y. ∃ n. (y, x) ∈ r → y ∈ bacc r n by simp
  from choice[OF this] obtain n where n: ∀ y. (y, x) ∈ r → y ∈ bacc r (n y)
  ..
  obtain n where ∀ y. (y, x) ∈ r → y ∈ bacc r n
proof
  fix y assume y: (y, x) ∈ r
  with n have y ∈ bacc r (n y) by auto
  moreover have n y ≤ Max ((λ(y, x). n y) :: r)
  using y (finite r) by (auto intro!: Max-ge)
  note bacc-mono[OF this, of r]
  ultimately show y ∈ bacc r (Max ((λ(y, x). n y) :: r)) by auto
qed
  then show y ∈ bacc r (Max ((λ(y, x). n y) :: r)) by auto
qed

lemma acc-bacc-eq:
  fixes A :: ('a :: finite × 'a) set
assumes finite $A$
shows Wellfounded.acc $A = \text{bacc } A (\text{card } \text{UNIV} :: 'a \text{ set})$
using assms by (metis acc-subseteq-bacc bacc-subseteq-acc bacc-upper-bound order-eq-iff)

lemma [code]:
fixes $xs :: ('a :: finite \times 'a) \text{ list}$
shows \(\text{Wellfounded}\text. acc (\text{set } xs) = \text{bacc } (\text{set } xs) (\text{card-UNIV } \text{TYPE}'a)\)
by (simp add: card-UNIV-def acc-bacc-eq)

72.3 Default instances for \text{enum}

lemma map-of-zip-enum-is-Some:
assumes \(\text{length } ys = \text{length } (\text{enum } :: 'a::\text{enum list})\)
shows \(\exists y. \text{map-of} (\text{zip } (\text{enum } :: 'a::\text{enum list}) ys) x = \text{Some } y\)
proof –
from assms have \(x \in \text{set } (\text{enum } :: 'a::\text{enum list}) \iff (\exists y. \text{map-of} (\text{zip } (\text{enum } :: 'a::\text{enum list}) ys) x = \text{Some } y)\)
by (auto intro!: map-of-zip-is-Some)
then show \(?\text{thesis} \text{ using } \text{enum-UNIV} \text{ by auto}\)
qed

lemma map-of-zip-enum-inject:
fixes $xs ys :: 'l::\text{enum list}$
assumes \(\text{length } \text{length } xs = \text{length } (\text{enum } :: 'a::\text{enum list}) \text{ and } \text{map-of} (\text{the } \circ \text{map-of} (\text{zip } (\text{enum } :: 'a::\text{enum list}) xs) = \text{the } \circ \text{map-of} (\text{zip } (\text{enum } :: 'a::\text{enum list}) ys)\)
shows \(xs = ys\)
proof –
have \(\text{map-of} (\text{zip } (\text{enum } :: 'a list) xs) = \text{map-of} (\text{zip } (\text{enum } :: 'a list) ys)\)
proof
fix $x :: 'a$
from \(\text{length } \text{map-of-zip-enum-is-Some} \text{ obtain } y1 y2\)
where \(\text{map-of} (\text{zip } (\text{enum } :: 'a list) xs) x = \text{Some } y1 \text{ and } \text{map-of} (\text{zip } (\text{enum } :: 'a list) ys) x = \text{Some } y2\) by blast
moreover from \(\text{map-of}\)
have \(\text{the } (\text{map-of} (\text{zip } (\text{enum } :: 'a::\text{enum list}) xs) x) = \text{the } (\text{map-of} (\text{zip } (\text{enum } :: 'a::\text{enum list}) ys) x)\)
by (auto dest: fun-cong)
ultimately show \(\text{map-of} (\text{zip } (\text{enum } :: 'a::\text{enum list}) xs) x = \text{map-of} (\text{zip } (\text{enum } :: 'a::\text{enum list}) ys) x\)
by simp
qed
with \(\text{length } \text{enum-distinct} \text{ show } xs = ys \text{ by } (\text{rule } \text{map-of-zip-inject})\)
qed

definition \text{all-n-lists} :: \((('a :: \text{enum} \text{ list } \Rightarrow \text{bool}) \Rightarrow \text{nat } \Rightarrow \text{bool} \text{ where}\)
all-n-lists P n $\iff$ ($\forall$ xs $\in$ set (List.n-lists n enum). P xs)

lemma [code]:
  all-n-lists P n $\iff$ (if n = 0 then P [] else enum-all (\%x. all-n-lists (\%xs. P (x # xs)) (n - 1)))
  unfolding all-n-lists-def enum-all
  by (cases n) (auto simp add: enum-UNIV)

definition ex-n-lists :: (('a :: enum) list $\Rightarrow$ bool) $\Rightarrow$ nat $\Rightarrow$ bool where
  ex-n-lists P n $\iff$ ($\exists$ xs $\in$ set (List.n-lists n enum). P xs)

lemma [code]:
  ex-n-lists P n $\iff$ (if n = 0 then P [] else enum-ex (\%x. ex-n-lists (\%xs. P (x # xs)) (n - 1)))
  unfolding ex-n-lists-def enum-ex
  by (cases n) (auto simp add: enum-UNIV)

instantiation fun :: (enum, enum) enum begin

definition
  enum = map (\ys. the $\circ$ map-of (zip (enum::'a list) ys)) (List.n-lists (length (enum::'a::enum list)) enum)

definition
  enum-all P = all-n-lists (\lbs. P (the $\circ$ map-of (zip enum bs))) (length (enum :: 'a list))

definition
  enum-ex P = ex-n-lists (\lbs. P (the $\circ$ map-of (zip enum bs))) (length (enum :: 'a list))

instance proof
  show UNIV $=$ set (enum :: ('a $\Rightarrow$ 'b) list)
  proof (rule UNIV-eq-I)
    fix f :: ('a $\Rightarrow$ 'b)
    have f = the $\circ$ map-of (zip (enum :: 'a::enum list) (map f enum))
      by (auto simp add: map-of-zip-map fun-eq_iff intro: in_enum)
    then show f $\in$ set enum
      by (auto simp add: enum-fun-def set-n-lists intro: in_enum)
  qed

next
  from map-of-zip-enum-inject
  show distinct (enum :: ('a $\Rightarrow$ 'b) list)
    by (auto intro!: inj_onI simp add: enum-fun-def
distinct-map distinct-n-lists enum-distinct set-n-lists)

next
  fix P
show enum-all \((P :: (\alpha \Rightarrow \beta) \Rightarrow \text{bool}) = \text{Ball \ UNIV \ P}\)

proof
  assume enum-all P
  show Ball UNIV P
  proof
    fix \(f :: \alpha \Rightarrow \beta\)
    have \(f = \text{the} \circ \text{map-of} \ (\text{zip} \ (\text{enum} :: \alpha :: \text{enum \ list}) \ (\text{map} \ f \ \text{enum}))\)
      by (auto simp add: map-of-zip-map fun-eq-iff intro: in-enum)
    from \(\text{enum-all \ P} \) have \(\text{P} \ (\text{the} \circ \text{map-of} \ (\text{zip} \ \text{enum} \ (\text{map} \ f \ \text{enum})))\)
      unfolding enum-all-fun-def all-n-lists-def
      apply (simp add: set-n-lists)
      apply (erule-tac x=map f enum in allE)
      apply (auto intro!: in-enum)
    done
    from this \(f\) show \(P \ f\) by auto
  qed
next
  assume Ball UNIV P
  from this show enum-all P
  unfolding enum-all-fun-def all-n-lists-def by auto
qed

next
fix \(P\)
show enum-ex \((P :: (\alpha \Rightarrow \beta) \Rightarrow \text{bool}) = \text{Bex \ UNIV \ P}\)

proof
  assume enum-ex P
  from this show Bex UNIV P
  unfolding enum-ex-fun-def ex-n-lists-def by auto
next
  assume Bex UNIV P
  from this obtain \(f\) where \(P \ f\ ..\)
  have \(f = \text{the} \circ \text{map-of} \ (\text{zip} \ (\text{enum} :: \alpha :: \text{enum \ list}) \ (\text{map} \ f \ \text{enum}))\)
    by (auto simp add: map-of-zip-map fun-eq-iff intro: in-enum)
  from \(P \ f\) this have \(\text{P} \ (\text{the} \circ \text{map-of} \ (\text{zip} \ (\text{enum} :: \alpha :: \text{enum \ list}) \ (\text{map} \ f \ \text{enum})))\)
    by auto
  from this show enum-ex P
  unfolding enum-ex-fun-def ex-n-lists-def
  apply (auto simp add: set-n-lists)
  apply (rule-tac x=map f enum in exI)
  apply (auto intro!: in-enum)
  done
qed

end

lemma enum-fun-code [code]: \(\text{enum} = (\text{let \ enum-a = (\text{enum} :: \alpha::\{\text{enum, equal}\} \ list})\)
in map (\ys. the ◦ map-of (zip enum-a \ys)) (List.n-lists (length enum-a) enum))
by (simp add: enum-fun-def Let-def)

lemma enum-all-fun-code [code]:
enum-all P = (let enum-a = (enum :: 'a::{enum, equal} list)
in all-n-lists (\bs. P (the ◦ map-of (zip enum-a \bs))) (length enum-a))
by (simp only: enum-all-fun-def Let-def)

lemma enum-ex-fun-code [code]:
enum-ex P = (let enum-a = (enum :: 'a::{enum, equal} list)
in ex-n-lists (\bs. P (the ◦ map-of (zip enum-a \bs))) (length enum-a))
by (simp only: enum-ex-fun-def Let-def)

instantiation set :: (enum) enum
begin

definition enum = map set (subseqs enum)

definition enum-all P ←→ (\A\in set enum. P (A::'a set))

definition enum-ex P ←→ (\exists A\in set enum. P (A::'a set))

instance proof
qed (simp-all add: enum-set-def enum-all-set-def enum-ex-set-def subseqs-powset
distinct-set-subseqs
enum-distinct enum-UNIV)

end

instantiation unit :: enum
begin

definition enum = [()]

definition enum-all P = P ()

definition enum-ex P = P ()

instance proof
qed (auto simp add: enum-unit-def enum-all-unit-def enum-ex-unit-def)

end
instance proof
qed (simp-all only: enum-bool-def enum-all-bool-def enum-ex-bool-def UNIV-bool,
simp-all)

end

instance instantiation prod :: (enum, enum) enum
begin

definition
enum = List.product enum enum

definition
enum-all P = enum-all (\x. enum-all (\y. P (x, y)))
definition
enum-ex P = enum-ex (\x. enum-ex (\y. P (x, y)))

instance
by standard
  (simp-all add: enum-prod-def distinct-product
   enum-UNIV enum-distinct enum-all-prod-def enum-ex-prod-def)

end

instance instantiation sum :: (enum, enum) enum
begin

definition
enum = map Inl enum @ map Inr enum

definition
enum-all P = enum-all (\x. P (Inl x)) \& enum-all (\x. P (Inr x))
definition
enum-ex P = enum-ex (\x. P (Inl x)) \lor enum-ex (\x. P (Inr x))
instance proof
\textbf{qed} (simp-all only: enum-sum-def enum-all-sum-def enum-ex-sum-def UNIV-sum,
auto simp add: enum-UNIV distinct-map enum-distinct)
end

\textbf{instantiation} \texttt{option :: (enum) enum} \\
\textbf{begin}

definition \texttt{enum} = \texttt{None \# map Some enum}

definition \texttt{enum-all \ P} \iff \texttt{P None \land enum-all (\lx. \ P (Some \ x))}

definition \texttt{enum-ex \ P} \iff \texttt{P None \lor enum-ex (\lx. \ P (Some \ x))}

\textbf{instance proof}
\textbf{qed} (simp-all only: enum-option-def enum-all-option-def enum-ex-option-def UNIV-option-conv, 
auto simp add: distinct-map enum-UNIV enum-distinct)
end

\subsection*{72.4 Small finite types}

We define small finite types for use in Quickcheck

\textbf{datatype} (plugins only: code quickcheck extraction) \texttt{finite-1 = a_1}

\textbf{notation} (output) \texttt{a_1 \ (a_1)}

\textbf{lemma} \texttt{UNIV-finite-1:}
\texttt{UNIV = \{a_1\}} \\
by (auto intro: finite-1.exhaust)

\textbf{instantiation} \texttt{finite-1 :: enum} \\
\textbf{begin}

definition \texttt{enum} = [a_1]

definition \texttt{enum-all \ P} = \texttt{P a_1}

definition \texttt{enum-ex \ P} = \texttt{P a_1}
THEORY “Enum”

instance proof
qed (simp-all only: enum-finite-1-def enum-all-finite-1-def enum-ex-finite-1-def UNIV-finite-1, simp-all)

end

instantiation finite-1 :: linorder
begin

definition less-finite-1 :: finite-1 ⇒ finite-1 ⇒ bool
where
  \( x < (y :: finite-1) \) ≜ False

definition less-eq-finite-1 :: finite-1 ⇒ finite-1 ⇒ bool
where
  \( x ≤ (y :: finite-1) \) ≜ True

instance
apply (intro-classes)
apply (auto simp add: less-finite-1-def less-eq-finite-1-def)
apply (metis (full-types) finite-1.exhaust)
done

end

instance finite-1 :: {dense-linorder, wellorder}
by intro-classes (simp-all add: less-finite-1-def)

instantiation finite-1 :: complete-lattice
begin

definition [simp]: Inf = (λ-. a₁)
definition [simp]: Sup = (λ-. a₁)
definition [simp]: bot = a₁
definition [simp]: top = a₁
definition [simp]: inf = (λ-. a₁)
definition [simp]: sup = (λ--. a₁)

instance by intro-classes(simp-all add: less-eq-finite-1-def)
end

instance finite-1 :: complete-distrib-lattice
by standard simp-all

instance finite-1 :: complete-linorder ..

lemma finite-1-eq: \( x = a₁ \)
by(cases \( x \)) simp
simproc-setup finite-1-eq (x::finite-1) = 
K (K (fn ct =>
  (case Thm.term-of ct of
    Const (const-name (a1), _) => NONE
    | _ => SOME (mk-meta-eq @{thm finite-1-eq})))
),

instantiation finite-1 :: complete-boolean-algebra
begin
definition [simp]: (−) = (λ−. a1)
definition [simp]: uminus = (λ−. a1)
instance by intro-classes simp-all
end

instantiation finite-1 ::
  {linordered-ring-strict, linordered-comm-ring-strict, ordered-comm-ring,
   ordered-cancel-comm-monoid-diff, comm-monoid-mult, ordered-ring-abs,
   one, modulo, sgn, inverse}
begindefinition [simp]: Groups.zero = a1
definition [simp]: Groups.one = a1
definition [simp]: (+) = (λ−. a1)
definition [simp]: (∗) = (λ−. a1)
definition [simp]: (mod) = (λ−. a1)
definition [simp]: abs = (λ−. a1)
definition [simp]: sgn = (λ−. a1)
definition [simp]: inverse = (λ−. a1)
definition [simp]: divide = (λ−. a1)
instance by intro-classes(simp-all add: less-finite-1-def)
end

declare [[simproc del: finite-1-eq]]
hide-const (open) a1
datatype (plugins only: code quickcheck extraction) finite-2 =
  a1 | a2
notation (output) a1 (a1)
notation (output) a2 (a2)

lemma UNIV-finite-2:
  UNIV = {a1, a2}
  by (auto intro: finite-2.exhaust)

instantiation finite-2 :: enum
begin

definition
enum = [a₁, a₂]

definition
enum-all P ←→ P a₁ ∧ P a₂

definition
enum-ex P ←→ P a₁ ∨ P a₂

instance proof
qed (simp-all only: enum-finite-2-def enum-all-finite-2-def enum-ex-finite-2-def UNIV-finite-2, simp-all)

end

instantiation finite-2 :: linorder
begin

definition less-finite-2 :: finite-2 ⇒ finite-2 ⇒ bool
where
x < y ←→ x = a₁ ∧ y = a₂

definition less-eq-finite-2 :: finite-2 ⇒ finite-2 ⇒ bool
where
x ≤ y ←→ x = y ∨ x < (y :: finite-2)

instance
apply (intro-classes)
apply (auto simp add: less-finite-2-def less-eq-finite-2-def)
apply (metis finite-2.nchotomy)+
done

end

instance finite-2 :: wellorder
by (rule wf-wellorderI)(simp add: less-finite-2-def, intro-classes)

instantiation finite-2 :: complete-lattice
begin

definition ⋂ A = (if a₁ ∈ A then a₁ else a₂)
definition ⋃ A = (if a₂ ∈ A then a₂ else a₁)
definition [simp]: bot = a₁
definition [simp]: top = a₂
definition x ⋂ y = (if x = a₁ ∨ y = a₁ then a₁ else a₂)
definition x ⋃ y = (if x = a₂ ∨ y = a₂ then a₂ else a₁)

lemma neq-finite-2-a1-iff [simp]: x ≠ a₁ ←→ x = a₂
by (cases x) simp-all
lemma neq-finite-2-a1-iff [simp]: $a_1 \neq x \longleftrightarrow x = a_2$
by (cases $x$) simp-all

lemma neq-finite-2-a2-iff [simp]: $x \neq a_2 \longleftrightarrow x = a_1$
by (cases $x$) simp-all

lemma neq-finite-2-a1-iff' [simp]: $a_2 \neq x \longleftrightarrow x = a_1$
by (cases $x$) simp-all

instance
proof
fix $x :: \text{finite-2}$ and $A$
assume $x \in A$
then show $\bigcap A \leq x \leq \bigcup A$
  by (cases $x$; auto simp add: less-eq-finite-2-def less-finite-2-def Inf-finite-2-def Sup-finite-2-def)+
qed (auto simp add: less-eq-finite-2-def less-finite-2-def inf-finite-2-def sup-finite-2-def Inf-finite-2-def Sup-finite-2-def)
end

instance finite-2 :: complete-linorder ..

instance finite-2 :: complete-distrib-lattice ..

instantiation finite-2 :: \{field, idom-abs-sgn, idom-modulo\} begin

definition [simp]: $0 = a_1$
definition [simp]: $1 = a_2$
definition $x + y = (case \ (x, y) \ of \ (a_1, a_1) \Rightarrow a_1 \mid (a_2, a_2) \Rightarrow a_1 \mid - \Rightarrow a_2)$
definition uminus = $(\lambda x :: \text{finite-2} . x)$
definition ($-\) = $((+) :: \text{finite-2} \Rightarrow \cdot)$
definition $x * y = (case \ (x, y) \ of \ (a_2, a_2) \Rightarrow a_2 \mid - \Rightarrow a_1)$
definition inverse = $(\lambda x :: \text{finite-2} . x)$
definition divide = $((\cdot) :: \text{finite-2} \Rightarrow \cdot)$
definition $x \ mod \ y = (case \ (x, y) \ of \ (a_2, a_1) \Rightarrow a_2 \mid - \Rightarrow a_1)$
definition abs = $(\lambda x :: \text{finite-2} . x)$
definition sgn = $(\lambda x :: \text{finite-2} . x)$
instance
by standard
(subproofs
  simp-all add: plus-finite-2-def uminus-finite-2-def minus-finite-2-def
times-finite-2-def
inverse-finite-2-def divide-finite-2-def modulo-finite-2-def
abs-finite-2-def sgn-finite-2-def
split: finite-2.splits)
end

lemma two-finite-2 [simp]:
$2 = a_1$
by (simp add: numeral.simps plus-finite-2-def)
lemma dvd-finite-2-unfold:
\[ x \text{ dvd } y \iff x = a_2 \lor y = a_1 \]
by (auto simp add: dvd-def times-finite-2-def split: finite-2.splits)

instantiation finite-2 :: {normalization-semidom, unique-euclidean-semiring} begin
  definition [simp]: normalize = (id :: finite-2 ⇒ -)
  definition [simp]: unit-factor = (id :: finite-2 ⇒ -)
  definition [simp]: euclidean-size x = (case x of a₁ ⇒ 0 | a₂ ⇒ 1)
  definition [simp]: division-segment (x :: finite-2) = 1
  instance by standard (subproofs
    (auto simp add: divide-finite-2-def times-finite-2-def dvd-finite-2-unfold
    split: finite-2.splits))
end

hide-const (open) a₁ a₂

datatype (plugins only: code quickcheck extraction) finite-3 =
  a₁ | a₂ | a₃

notation (output) a₁ (a₁)
notation (output) a₂ (a₂)
notation (output) a₃ (a₃)

lemma UNIV-finite-3:
\[ \text{UNIV} = \{a₁, a₂, a₃\} \]
by (auto intro: finite-3.exhaust)

instantiation finite-3 :: enum begin
  definition enum = [a₁, a₂, a₃]

  definition enum-all P ⇔ P a₁ ∧ P a₂ ∧ P a₃

  definition enum-ex P ⇔ P a₁ ∨ P a₂ ∨ P a₃

  instance proof
  qed (simp-all only: enum-finite-3-def enum-all-finite-3-def enum-ex-finite-3-def UNIV-finite-3, simp-all)
end
lemma finite-3-not-eq-unfold:
  \( x \neq a_1 \iff x \in \{ a_2, a_3 \} \)
  \( x \neq a_2 \iff x \in \{ a_1, a_3 \} \)
  \( x \neq a_3 \iff x \in \{ a_1, a_2 \} \)
  by (cases x; simp)+

instantiation finite-3 :: linorder
begin

definition less-finite-3 :: finite-3 \Rightarrow finite-3 \Rightarrow bool
where
  \( x < y \equiv \text{case } x \text{ of } a_1 \Rightarrow y \neq a_1 \mid a_2 \Rightarrow y = a_3 \mid a_3 \Rightarrow \text{False} \)

definition less-eq-finite-3 :: finite-3 \Rightarrow finite-3 \Rightarrow bool
where
  \( x \leq y \equiv x = y \lor x < (\text{y :: finite-3}) \)

instance proof (intro-classes)
qed (auto simp add: less-finite-3-def less-eq-finite-3-def split: finite-3.split-asm)

end

instance finite-3 :: wellorder
proof (rule wf-wellorderI)
  have inv-image less-than (case-finite-3 0 1 2) = \{ (x, y). x < y \}
    by (auto simp add: less-finite-3-def split: finite-3.splits)
  from this[ symmetric] show wf \dots \ by simp
qed intro-classes

class finite-lattice = finite + lattice + Inf + Sup + bot + top +
  assumes Inf-finite-empty: Inf \{\} = Sup UNIV
  assumes Inf-finite-insert: Inf (insert a A) = a \cap Inf A
  assumes Sup-finite-empty: Sup \{\} = Inf UNIV
  assumes Sup-finite-insert: Sup (insert a A) = a \sqcup Sup A
  assumes bot-finite-def: bot = Inf UNIV
  assumes top-finite-def: top = Sup UNIV
begin

subclass complete-lattice
proof
  fix \( x \ A \)
  show \( x \in A \Longrightarrow \bigcap A \leq x \)
    by (metis Set.set-insert abel-semigroup.commute local.Inf-finite-insert local.inf.abel-semigroup-axioms
     local.inf.left-idem local.inf.order1)
  show \( x \in A \Longrightarrow x \leq \bigcup A \)
    by (metis Set.set-insert insert-absorb2 local.Sup-finite-insert local.sup.absorb-iff2)
next
  fix \( A \ z \)
have \( \bigcup \) UNIV = z \( \sqcup \) \( \bigcup \) UNIV 
  by (subst Sup-finite-insert [symmetric], simp add: insert-UNIV)
from this have \([\text{simpl}]\): z \( \leq \) \( \bigcup \) UNIV
  using local.le_iff_sup by auto
have \((\forall x. x \in A \rightarrow z \leq x) \rightarrow z \leq \bigcap A\)
  by (rule finite-induct \([\text{of A A}]. (\forall x. x \in A \rightarrow z \leq x) \rightarrow z \leq \bigcap A\)\])
(simp-all add: Inf-finite-empty Inf-finite-insert)
from this show \((\\forall x. x \in A \Rightarrow x \leq z) \Rightarrow z \leq \bigcap A\)
  by simp

next
show \( \bigcap \) \{\} = \top
  by (simp add: Inf-finite-empty top-finite-def)
show \( \bigcup \) \{\} = \bot
  by (simp add: Sup-finite-empty bot-finite-def)
qed

end

class finite-distrib-lattice = finite-lattice + distrib-lattice
begin
lemma finite-inf-Sup: \( a \sqcap (\text{Sup A}) = \text{Sup} \{a \sqcap b \mid b \in A\}\)
proof (rule finite-induct \([\text{of A A}]. a \sqcap (\text{Sup A}) = \text{Sup} \{a \sqcap b \mid b \in A\}\], simp-all)
  fix x::'a
  fix F
  assume x \notin F
  assume \([\text{simpl}]\): a \( \sqcap \) \( \bigcup \) F = \( \bigcup \) \{a \( \sqcap \) b \mid b \in F\}
  have \([\text{simpl}]\): insert (a \( \sqcap \) x) \{a \( \sqcap \) b \mid b \in F\} = \{a \( \sqcap \) b \mid b = x \( \lor \) b \in F\}
    by blast
  have a \( \sqcap \) (x \( \sqcup \) \( \bigcup \) F) = a \( \sqcap \) x \( \sqcup \) a \( \sqcap \) \( \bigcup \) F
    by (simp add: inf-sup-distrib1)
  also have \( \ldots \) = a \( \sqcap \) x \( \sqcup \) \( \bigcup \) \{a \( \sqcap \) b \mid b \in F\}
    by simp
  also have \( \ldots \) = \( \bigcup \) \{a \( \sqcap \) b \mid b = x \( \lor \) b \in F\}
    by (unfold Sup-insert[THEN sym], simp)
  finally show a \( \sqcap \) (x \( \sqcup \) \( \bigcup \) F) = \( \bigcup \) \{a \( \sqcap \) b \mid b = x \( \lor \) b \in F\}
    by simp
qed

lemma finite-Inf-Sup: \( \bigcap \) (\text{Sup ' A}) \( \leq \) \( \bigcup \) (\text{Inf ' \{f ' A \mid \forall Y \in A. f Y \in Y\}})
proof (rule finite-induct [of A λA. ∩(Sup ' F) ≤ ∪(Inf ' {f ' F | f ∈ Y})], simp-all add: finite-UnionD)

fix x:a set

fix F

assume x ∉ F

have [simp]: {∪ x ∩ b | b ∈ Inf ' {f ' F | f ∈ Y}} = {∪ x ∩ (Inf (f ' F)) | f ∈ Y}

by auto

define fa where fa = (λ b: 'a) f Y . (if Y = x then b else f Y)

have ∀ f ∈ F. f Y ∈ Y → b ∈ x → insert b (f (F ∩ {Y. Y ≠ x})) = insert (fa b f x) (fa b f ' F ∧ fa b f x ∈ x ∧ (∀ Y ∈ F. fa b f Y ∈ Y))

by (auto simp add: fa-def)

from this have B: ∀ b. ∀ Y ∈ F. f Y ∈ Y → b ∈ x → fa b f ' F ∈ {insert (f x) (f ' F) | f. f x ∈ x ∧ (∀ Y ∈ F. f Y ∈ Y)}

by blast

have [simp]: ∩ f b. f Y ∈ Y → b ∈ x → b ∩ (⋃ x ∈ F. f x) ≤ ∪ (Inf ' {insert (f x) (f ' F) | f. f x ∈ x ∧ (∀ Y ∈ F. f Y ∈ Y)})

using B apply (rule SUP-upper2)

using (x ∉ F); apply (simp-all add: fa-def Inf-union-distrib)

apply (simp add: image-mono Inf-superset-mono inf.coboundedI2)

done

assume (∩ (Sup ' F) ≤ ∪ (Inf ' {f ' F | f ∈ Y})

from this have ∪ x ∩ (∩ (Sup ' F) ≤ ∪ x ∩ (∪ (Inf ' {f ' F | f ∈ Y})

using inf.coboundedI2 by auto

also have ... = Sup (⋃ x ∩ (Inf (f ' F)) ∩ (∀ Y ∈ F. f Y ∈ Y))

by (simp add: finite-inf-Sup)

also have ... = Sup {Inf (f ' F) ∩ b | b ∈ x} | f . (∀ Y ∈ F. f Y ∈ Y)}

by (subst inf-commute) (simp add: finite-inf-Sup)

also have ... ≤ (∪ (Inf ' {insert (f x) (f ' F) | f. f x ∈ x ∧ (∀ Y ∈ F. f Y ∈ Y))}

apply (rule Sup-least,clarsimp)+

apply (subst inf-commute, simp)

done

finally show ∪ x ∩ (∩ (Sup ' F) ≤ ∪ (Inf ' {insert (f x) (f ' F) | f. f x ∈ x ∧ (∀ Y ∈ F. f Y ∈ Y)})

by simp

qed

subclass complete-distrib-lattice

by (standard, rule finite-Inf-Sup)

end

instantiation finite-3 :: finite-lattice

begin

definition ∩ A = (if a1 ∈ A then a1 else if a2 ∈ A then a2 else a3)
\begin{verbatim}
THEORY "Enum"

definition \( \bigcup A = \) (if \( a_3 \in A \) then \( a_3 \) else if \( a_2 \in A \) then \( a_2 \) else \( a_1 \))
definition \([\text{simp}]\): \( \text{bot} = a_1 \)
definition \([\text{simp}]\): \( \text{top} = a_3 \)
definition \([\text{simp}]\): \( \text{inf} = (\text{min} :: \text{finite-3} \Rightarrow \cdot) \)
definition \([\text{simp}]\): \( \text{sup} = (\text{max} :: \text{finite-3} \Rightarrow \cdot) \)

instance proof qed (auto simp add: Inf-finite-3-def Sup-finite-3-def max-def min-def less-eq-finite-3-def less-finite-3-def split: finite-3).
end

instance finite-3 :: complete-lattice ..

instance finite-3 :: finite-distrib-lattice proof qed (auto simp add: min-def max-def).

instance finite-3 :: complete-distrib-lattice ..

instance finite-3 :: complete-linorder ..

instantiation finite-3 :: \{field, idom-abs-sgn, idom-modulo\} begin
definition \([\text{simp}]\): \( 0 = a_1 \)
definition \([\text{simp}]\): \( 1 = a_2 \)
definition \( x + y = (\text{case } (x, y) \text{ of} \) 
  \( (a_1, a_1) \Rightarrow a_1 \mid (a_2, a_3) \Rightarrow a_1 \mid (a_3, a_2) \Rightarrow a_1 \)
  \( | (a_1, a_2) \Rightarrow a_2 \mid (a_2, a_1) \Rightarrow a_2 \mid (a_3, a_3) \Rightarrow a_2 \)
  \( | - \Rightarrow a_3 \) \)
definition \(- x = (\text{case } x \text{ of} a_1 \Rightarrow a_1 \mid a_2 \Rightarrow a_3 \mid a_3 \Rightarrow a_2 \)\)
definition \( x \times y = (\text{case } (x, y) \text{ of} \) 
  \( (a_2, a_2) \Rightarrow a_2 \mid (a_3, a_3) \Rightarrow a_2 \mid (a_2, a_3) \Rightarrow a_3 \)
  \( | (a_3, a_2) \Rightarrow a_3 \mid - \Rightarrow a_1 \) \)
definition inverse = (\( \lambda x :: \text{finite-3}. \ x \) )
definition \( x \div y = x \times \text{inverse} \ (y :: \text{finite-3}) \)
definition \( x \mod y = (\text{case } y \text{ of} a_1 \Rightarrow x \mid - \Rightarrow a_1 \) \)
definition abs = (\( \lambda x. \text{ case } x \text{ of} a_3 \Rightarrow a_2 \mid \cdot \Rightarrow x \) )
definition sgn = (\( \lambda x :: \text{finite-3}. \ x \) )
instance by standard (subproofs <\( \text{simp-all add: plus-finite-3-def uminus-finite-3-def minus-finite-3-def times-finite-3-def inverse-finite-3-def divide-finite-3-def modulo-finite-3-def abs-finite-3-def sgn-finite-3-def less-finite-3-def split: finite-3.splits\>)
end
\end{verbatim}
lemma two-finite-3 [simp]:
  2 = a₃
  by (simp add: numeral.simps plus-finite-3-def)

lemma dvd-finite-3-unfold:
  x dvd y <-> x = a₂ ∨ x = a₃ ∨ y = a₁
  by (cases x) (auto simp add: dvd-def times-finite-3-def split: finite-3.splits)

instantiation finite-3 :: {normalization-semidom, unique-euclidean-semiring} begin
  definition [simp]: normalize x = (case x of a₃ ⇒ a₂ | - ⇒ x)
  definition [simp]: unit-factor = (id :: finite-3 ⇒ -)
  definition [simp]: euclidean-size x = (case x of a₁ ⇒ 0 | - ⇒ 1)
  definition [simp]: division-segment (x :: finite-3) = 1

instance proof
  fix x :: finite-3
  assume x ≠ 0
  then show is-unit (unit-factor x)
    by (cases x) (simp-all add: dvd-finite-3-unfold)
  qed
  (subproofs
    auto simp add: divide-finite-3-def times-finite-3-def
    dvd-finite-3-unfold inverse-finite-3-def plus-finite-3-def
    split: finite-3.splits)
end

hide-const (open) a₁ a₂ a₃

datatype (plugins only: code quickcheck extraction) finite-4 =
  a₁ | a₂ | a₃ | a₄

notation (output) a₁ · a₁
notation (output) a₂ · a₂
notation (output) a₃ · a₃
notation (output) a₄ · a₄

lemma UNIV-finite-4:
  UNIV = {a₁, a₂, a₃, a₄}
  by (auto intro: finite-4.exhaust)

instantiation finite-4 :: enum begin

  definition enum = [a₁, a₂, a₃, a₄]

  definition
enum-all \( P \) \( \iff \) \( P \ a_1 \land P \ a_2 \land P \ a_3 \land P \ a_4 \)

**definition**

\( \text{enum-ex} \ P \) \( \iff \) \( P \ a_1 \lor P \ a_2 \lor P \ a_3 \lor P \ a_4 \)

**instance proof**

**qed (simp-all only: enum-finite-4-def enum-all-finite-4-def enum-ex-finite-4-def UNIV-finite-4, simp-all)**

end

**instantiation** finite-4 :: finite-distrib-lattice begin

\( a_1 < a_2, a_3 < a_4 \), but \( a_2 \) and \( a_3 \) are incomparable.

**definition**

\( x < y \) \( \iff \) \( \text{(case (x, y) of} \)

\( (a_1, a_1) \Rightarrow \text{False} | (a_1, -) \Rightarrow \text{True} \)

\( (a_2, a_1) \Rightarrow \text{True} \)

\( (a_3, a_1) \Rightarrow \text{True} | - \Rightarrow \text{False} \)

**definition**

\( x \leq y \) \( \iff \) \( \text{(case (x, y) of} \)

\( (a_1, -) \Rightarrow \text{True} \)

\( (a_2, a_2) \Rightarrow \text{True} | (a_2, a_4) \Rightarrow \text{True} \)

\( (a_3, a_2) \Rightarrow \text{True} | (a_4, a_4) \Rightarrow \text{True} \)

\( (a_4, a_4) \Rightarrow \text{True} | - \Rightarrow \text{False} \)

**definition**

\( \bigcap A = (\text{if } a_1 \in A \lor a_2 \in A \land a_3 \in A \text{ then } a_1 \text{ else if } a_2 \in A \text{ then } a_2 \text{ else if } a_3 \in A \text{ then } a_3 \text{ else } a_4) \)

**definition**

\( \bigcup A = (\text{if } a_4 \in A \lor a_2 \in A \land a_3 \in A \text{ then } a_4 \text{ else if } a_2 \in A \text{ then } a_2 \text{ else if } a_3 \in A \text{ then } a_3 \text{ else } a_4) \)

**definition** [simp]: \( \text{bot} = a_1 \)

**definition** [simp]: \( \text{top} = a_4 \)

**definition**

\( x \cap y = \text{(case (x, y) of} \)

\( (a_1, -) \Rightarrow a_1 | (\_, a_1) \Rightarrow a_1 | (a_2, a_3) \Rightarrow a_1 | (a_3, a_2) \Rightarrow a_1 \)

\( (a_2, -) \Rightarrow a_2 | (\_, a_2) \Rightarrow a_2 \)

\( (a_3, -) \Rightarrow a_3 | (\_, a_3) \Rightarrow a_3 \)

\( - \Rightarrow a_4 \)

**definition**

\( x \cup y = \text{(case (x, y) of} \)

\( (a_2, -) \Rightarrow a_4 | (\_, a_4) \Rightarrow a_4 | (a_2, a_3) \Rightarrow a_4 | (a_3, a_2) \Rightarrow a_4 \)

\( (a_3, -) \Rightarrow a_4 | (\_, a_4) \Rightarrow a_4 \)

\( - \Rightarrow a_4 \)

**instance**
by standard
(subproofs
(auto simp add: less-finite-4-def less-eq-finite-4-def Inf-finite-4-def Sup-finite-4-def
inf-finite-4-def sup-finite-4-def split: finite-4.splits)
end

instance finite-4 :: complete-lattice ..

instance finite-4 :: complete-distrib-lattice ..

instantiation finite-4 :: complete-boolean-algebra begin
definition − x = (case x of a1 ⇒ a4 | a2 ⇒ a3 | a3 ⇒ a2 | a4 ⇒ a1)
definition x − y = x ⊓− (y :: finite-4)
instance by standard
(subproofs
(simp-all add: inf-finite-4-def sup-finite-4-def uminus-finite-4-def minus-finite-4-def
split: finite-4.splits)
end

hide-const (open) a1 a2 a3 a4

datatype (plugins only: code quickcheck extraction) finite-5 =
a1 | a2 | a3 | a4 | a5

notation (output) a1 (a1)
notation (output) a2 (a2)
notation (output) a3 (a3)
notation (output) a4 (a4)
notation (output) a5 (a5)

lemma UNIV-finite-5:
UNIV = {a1, a2, a3, a4, a5}
by (auto intro: finite-5.exhaust)

instantiation finite-5 :: enum begin

definition enum = [a1, a2, a3, a4, a5]

definition enum-all P ↔ P a1 ∧ P a2 ∧ P a3 ∧ P a4 ∧ P a5

definition enum-ex P ↔ P a1 ∨ P a2 ∨ P a3 ∨ P a4 ∨ P a5
instance proof
qed (simp-all only: enum-finite-5-def enum-all-finite-5-def enum-ex-finite-5-def UNIV-finite-5, simp-all)

end

instantiation finite-5 :: finite-lattice
begin

The non-distributive pentagon lattice $N_5$

definition
$x < y$ \iff (case $(x, y)$ of
  $(a_1, a_1) \Rightarrow \text{False} | (a_1, -) \Rightarrow \text{True}$
  | $(a_2, a_3) \Rightarrow \text{True} | (a_2, a_3) \Rightarrow \text{True}$
  | $(a_3, a_5) \Rightarrow \text{True}$
  | $(a_4, a_5) \Rightarrow \text{True} | - \Rightarrow \text{False}$)

definition
$x \leq y$ \iff (case $(x, y)$ of
  $(a_1, -) \Rightarrow \text{True}$
  | $(a_2, a_2) \Rightarrow \text{True} | (a_2, a_3) \Rightarrow \text{True} | (a_2, a_5) \Rightarrow \text{True}$
  | $(a_3, a_3) \Rightarrow \text{True} | (a_4, a_5) \Rightarrow \text{True}$
  | $(a_4, a_4) \Rightarrow \text{True} | (a_4, a_5) \Rightarrow \text{True}$
  | $(a_5, a_5) \Rightarrow \text{True} | - \Rightarrow \text{False}$)

definition
$\bigwedge A =$
  (if $a_1 \in A \lor a_4 \in A \land (a_2 \in A \lor a_3 \in A)$ then $a_1$
    else if $a_2 \in A$ then $a_2$
    else if $a_3 \in A$ then $a_3$
    else if $a_4 \in A$ then $a_4$
    else $a_5$)

definition
$\bigvee A =$
  (if $a_5 \in A \lor a_4 \in A \land (a_2 \in A \lor a_3 \in A)$ then $a_5$
    else if $a_3 \in A$ then $a_3$
    else if $a_2 \in A$ then $a_2$
    else if $a_4 \in A$ then $a_4$
    else $a_1$)

definition [simp]: $\bot = a_1$
definition [simp]: $\top = a_5$
definition
$x \sqcap y = (case (x, y)$ of
  $(a_1, -) \Rightarrow a_1 | (-, a_1) \Rightarrow a_1 | (a_2, a_4) \Rightarrow a_1 | (a_4, a_2) \Rightarrow a_1 | (a_3, a_4) \Rightarrow a_1 |
  (a_4, a_3) \Rightarrow a_1$
  | $(a_2, -) \Rightarrow a_2 | (-, a_2) \Rightarrow a_2$
  | $(a_3, -) \Rightarrow a_3 | (-, a_3) \Rightarrow a_3$
  | $(a_4, -) \Rightarrow a_4 | (-, a_4) \Rightarrow a_4$
  | - \Rightarrow a_5)$
theory "String"

definition
\[ x \sqcup y = \text{case } (x, y) \text{ of}\]
\[ (a_5, -) \Rightarrow a_5 | (-, a_5) \Rightarrow a_5 | (a_2, a_4) \Rightarrow a_5 | (a_4, a_2) \Rightarrow a_5 | (a_3, a_4) \Rightarrow a_5 \]
\[ (a_4, a_3) \Rightarrow a_5 \]
\[ (-, -) \Rightarrow a_3 | (-, a_3) \Rightarrow a_3 \]
\[ (a_2, -) \Rightarrow a_2 | (-, a_2) \Rightarrow a_2 \]
\[ (a_4, -) \Rightarrow a_4 | (-, a_4) \Rightarrow a_4 \]
\[ (-) \Rightarrow a_1 \]

instance
by standard
(subproofs
(auto simp add: less-eq-finite-5-def less-finite-5-def inf-finite-5-def sup-finite-5-def
Inf-finite-5-def Sup-finite-5-def split: finite-5.splits if-split-asm)
end

instance finite-5 :: complete-lattice ..

hide-const (open) a_1 a_2 a_3 a_4 a_5

72.5 Closing up
hide-type (open) finite-1 finite-2 finite-3 finite-4 finite-5
hide-const (open) enum enum-all enum-ex all-n-lists ex-n-lists ntrancl

end

73 Character and string types

theory String
imports Enum Bit-Operations Code-Numeral
begin

73.1 Strings as list of bytes

When modelling strings, we follow the approach given in https://utf8everywhere.org/:

- Strings are a list of bytes (8 bit).
- Byte values from 0 to 127 are US-ASCII.
- Byte values from 128 to 255 are uninterpreted blobs.
73.1.1 Bytes as datatype

datatype char =
  Char (digit0: bool) (digit1: bool) (digit2: bool) (digit3: bool)
  (digit4: bool) (digit5: bool) (digit6: bool) (digit7: bool)

class comm-semiring-1
begin

definition of-char :: 'a 
  where of-char c = horner-sum of-bool 2 [digit0 c, digit1 c, digit2 c, digit3 c,
  digit4 c, digit5 c, digit6 c, digit7 c] 

lemma of-char-Char [simp]: 
  of-char (Char b0 b1 b2 b3 b4 b5 b6 b7) = 
  horner-sum of-bool 2 [b0, b1, b2, b3, b4, b5, b6, b7] 
by (simp add: of-char-def)

end

lemma (in comm-semiring-1) of-nat-of-char:
  of-nat (of-char c) = of-char c 
by (cases c) simp

lemma (in comm-ring-1) of-int-of-char:
  of-int (of-char c) = of-char c 
by (cases c) simp

lemma nat-of-char [simp]: 
  nat (of-char c) = of-char c 
by (cases c) (simp only: of-char-Char nat-horner-sum)

context linordered-euclidean-semiring-bit-operations
begin

definition char-of :: 'a 
  where char-of n = Char (bit n 0) (bit n 1) (bit n 2) (bit n 3) (bit n 4) (bit n 5)
  (bit n 6) (bit n 7) 

lemma char-of-take-bit-eq: 
  char-of (take-bit n m) = char-of m if n ≥ 8 
using that by (simp add: char-of-def bit-take-bit-iff)

lemma char-of-char [simp]: 
  char-of (of-char c) = c 
by (simp only: of-char-def char-of-def bit-horner-sum-bit-iff) simp

lemma char-of-comp-of-char [simp]: 
  char-of o of-char = id
by (simp add: fun-eq-iff)

lemma inj-of-char:
  ⟨inj of-char⟩
proof (rule injI)
  fix c d
  assume of-char c = of-char d
  then have char-of (of-char c) = char-of (of-char d)
    by simp
  then show c = d
    by simp
qed

lemma of-char-eqI:
  ⟨c = d⟩ if ⟨of-char c = of-char d⟩
using that inj-of-char by (simp add: inj-eq)

lemma of-char-eq-iff [simp]:
  ⟨of-char c = of-char d ⟷ c = d⟩
by (auto intro: of-char-eqI)

lemma of-char-of [simp]:
  ⟨of-char (char-of a) = a mod 256⟩
proof
  have ⟨{0..<8} = [0, Suc 0, 2, 3, 4, 5, 6, 7 :: nat}⟩
    by (simp add: upt-eq-Cons-cone)
  then have ⟨[bit a 0, bit a 1, bit a 2, bit a 3, bit a 4, bit a 5, bit a 6, bit a 7] =
    map (bit a) [0..<8]⟩
    by simp
  then have ⟨of-char (char-of (of-char c)) = take-bit 8 a⟩
    by (simp only: char-of-def of-char-def char.sel horner-sum-bit-eq-take-bit)
  then show ?thesis
    by (simp add: take-bit-eq-mod)
qed

lemma char-of-mod-256 [simp]:
  ⟨char-of (n mod 256) = char-of n⟩
by (rule of-char-eqI) simp

lemma of-char-mod-256 [simp]:
  ⟨of-char c mod 256 = of-char c⟩
proof
  have ⟨of-char (char-of (of-char c)) mod 256 = of-char (char-of (of-char c))⟩
    by (simp only: of-char-of) simp
  then show ?thesis
    by simp
qed

lemma char-of-quasi-inj [simp]:
THEORY "String"

\langle\text{char-of } m = \text{char-of } n \longleftrightarrow m \mod 256 = n \mod 256; \text{(is } \langle P \longleftrightarrow Q \rangle)\rangle

\text{proof}
\begin{itemize}
    \item assume \( ?Q \)
    \item then show \( ?P \)
        \begin{itemize}
            \item by (auto intro: of-char-eqI)
        \end{itemize}
\end{itemize}

\text{next}
\begin{itemize}
    \item assume \( ?P \)
    \item then have \( \langle \text{of-char } (\text{char-of } m) = \text{of-char } (\text{char-of } n) \rangle \)
        \begin{itemize}
            \item by simp
        \end{itemize}
    \item then show \( ?Q \)
        \begin{itemize}
            \item by simp
        \end{itemize}
\end{itemize}

\text{qed}

\text{lemma char-of-eq-iff:}
\langle\text{char-of } n = c \longleftrightarrow \text{take-bit } 8 n = \text{of-char } c\rangle
\text{by (auto intro: of-char-eqI simp add: take-bit-eq-mod)}

\text{lemma char-of-nat [simp]:}
\langle\text{char-of } (\text{of-nat } n) = \text{char-of } n\rangle
\text{by (simp add: char-of-def String.char-of-def drop-bit-of-nat bit-simps possible-bit-def)}

end

\text{lemma inj-on-char-of-nat [simp]:}
\langle\text{inj-on } \text{char-of } \{0::nat..<256\}\rangle
\text{by (rule inj-onI simp)}

\text{lemma nat-of-char-less-256 [simp]:}
\langle\text{of-char } c < (256 :: nat)\rangle
\text{proof}
\begin{itemize}
    \item have \( \text{of-char } c \mod (256 :: nat) < 256 \)
        \begin{itemize}
            \item by arith
        \end{itemize}
    \item then show \( ?\text{thesis} \)
        \begin{itemize}
            \item by simp
        \end{itemize}
\end{itemize}

\text{qed}

\text{lemma range-nat-of-char:
\langle}\text{range of-char} = \{0::nat..<256\}\rangle
\text{proof (rule; rule)
\begin{itemize}
    \item fix \( n :: \text{nat} \)
    \item assume \( n \in \text{range of-char} \)
    \item then show \( n \in \{0..<256\} \)
        \begin{itemize}
            \item by auto
        \end{itemize}
\end{itemize}
\text{next}
\begin{itemize}
    \item fix \( n :: \text{nat} \)
    \item assume \( n \in \{0..<256\} \)
    \item then have \( n = \text{of-char } (\text{char-of } n) \)
        \begin{itemize}
            \item by simp
        \end{itemize}
    \item then show \( n \in \text{range of-char} \)
        \begin{itemize}
            \item by (rule range-eqI)
        \end{itemize}
\end{itemize}
\text{end}
qed

lemma UNIV-char-of-nat:
UNIV = char-of ‘ {0::nat..<256}
proof –
  have range (of-char :: char ⇒ nat) = of-char ‘ char-of ‘ {0::nat..<256}
    by (auto simp add: range-nat-of-char intro: image-eqI)
with inj-of-char [where ¡a = nat] show ?thesis
  by (simp add: inj-image-eq-iff)
qed

lemma card-UNIV-char:
card (UNIV :: char set) = 256
by (auto simp add: UNIV-char-of-nat card-image)

context
  includes lifting-syntax integer.lifting natural.lifting
begin

lemma [transfer-rule]:
〈(pcr-integer ===> (=)) char-of char-of〉
by (unfold char-of-def) transfer-prover

lemma [transfer-rule]:
〈(=) ===> pcr-integer) of-char of-char〉
by (unfold of-char-def) transfer-prover

lemma [transfer-rule]:
〈(per-natural ===> (=)) char-of char-of〉
by (unfold char-of-def) transfer-prover

lemma [transfer-rule]:
〈(=) ===> per-natural) of-char of-char〉
by (unfold of-char-def) transfer-prover

end

lifting-update integer.lifting
lifting-forget integer.lifting
lifting-update natural.lifting
lifting-forget natural.lifting

lemma size-char-eq-0 [simp, code]:
〈size c = 0〉 for c :: char
by (cases c) simp

lemma size'-char-eq-0 [simp, code]:
〈size-char c = 0〉
THEORY "String"

by (cases c) simp

syntax
  - Char :: str-position ⇒ char  (CHR -)
  - Char-ord :: num-const ⇒ char  (CHR -)

type-synonym string = char list

syntax
  - String :: str-position ⇒ string  (-)

ML-file <Tools/string-syntax.ML>
THEORY "String"

CHR "\"", CHR "\", CHR "~", CHR 0x7F,
CHR 0x80, CHR 0x81, CHR 0x82, CHR 0x83,
CHR 0x84, CHR 0x85, CHR 0x86, CHR 0x87,
CHR 0x88, CHR 0x89, CHR 0x8A, CHR 0x8B,
CHR 0x8C, CHR 0x8D, CHR 0x8E, CHR 0x8F,
CHR 0x90, CHR 0x91, CHR 0x92, CHR 0x93,
CHR 0x94, CHR 0x95, CHR 0x96, CHR 0x97,
CHR 0x98, CHR 0x99, CHR 0x9A, CHR 0x9B,
CHR 0x9C, CHR 0x9D, CHR 0x9E, CHR 0x9F,
CHR 0xA0, CHR 0xA1, CHR 0xA2, CHR 0xA3,
CHR 0xA4, CHR 0xA5, CHR 0xA6, CHR 0xA7,
CHR 0xA8, CHR 0xA9, CHR 0xAA, CHR 0xAB,
CHR 0xAC, CHR 0xAD, CHR 0xAE, CHR 0xAF,
CHR 0xB0, CHR 0xB1, CHR 0xB2, CHR 0xB3,
CHR 0xB4, CHR 0xB5, CHR 0xB6, CHR 0xB7,
CHR 0xB8, CHR 0xB9, CHR 0xBA, CHR 0xBB,
CHR 0xBC, CHR 0xBD, CHR 0xBE, CHR 0xBF,
CHR 0xC0, CHR 0xC1, CHR 0xC2, CHR 0xC3,
CHR 0xC4, CHR 0xC5, CHR 0xC6, CHR 0xC7,
CHR 0xC8, CHR 0xC9, CHR 0xCA, CHR 0xCB,
CHR 0xCC, CHR 0xCD, CHR 0xCE, CHR 0xCF,
CHR 0xD0, CHR 0xD1, CHR 0xD2, CHR 0xD3,
CHR 0xD4, CHR 0xD5, CHR 0xD6, CHR 0xD7,
CHR 0xD8, CHR 0xD9, CHR 0xDA, CHR 0xDB,
CHR 0xDC, CHR 0xDD, CHR 0xDE, CHR 0xDF,
CHR 0xE0, CHR 0xE1, CHR 0xE2, CHR 0xE3,
CHR 0xE4, CHR 0xE5, CHR 0xE6, CHR 0xE7,
CHR 0xE8, CHR 0xE9, CHR 0xEA, CHR 0xEB,
CHR 0xEC, CHR 0xED, CHR 0xEE, CHR 0xEF,
CHR 0xF0, CHR 0xF1, CHR 0xF2, CHR 0xF3,
CHR 0xF4, CHR 0xF5, CHR 0xF6, CHR 0xF7,
CHR 0xF8, CHR 0xF9, CHR 0xFA, CHR 0xFB,
CHR 0xFC, CHR 0xFD, CHR 0xFE, CHR 0xFF

definition
Enum.enum-all P ⇔ list-all P (Enum.enum :: char list)

definition
Enum.enum-ex P ⇔ list-ex P (Enum.enum :: char list)

lemma enum-char-unfold:
Enum.enum = map char-of [0..<256]

proof
  have map (of-char :: char ⇒ nat) Enum.enum = [0..<256]
    by (simp add: enum-char-def of-char-def apl-conv-Cons-Cons numeral-2-eq-2
      [symmetric])
  then have map char-of (map (of-char :: char ⇒ nat) Enum.enum) =
    map char-of [0..<256]
    by simp
then show thesis
  by simp
qed

instance proof
show UNIV: UNIV = set (Enum.enum :: char list)
  by (simp add: enum-char-unfold UNIV-char-of-nat atLeast0LessThan)
show distinct (Enum.enum :: char list)
  by (auto simp add: enum-char-unfold distinct-map intro: inj-onI)
show ∃ P. Enum.enum-all P ←→ Ball (UNIV :: char set) P
  by (simp add: UNIV enum-all-char-def list-all-iff)
show ∃ P. Enum.enum-ex P ←→ Bex (UNIV :: char set) P
  by (simp add: UNIV enum-ex-char-def list-ex-iff)
qed

end

lemma linorder-char:
  class.linorder ((λ c d. of-char c ≤ (of-char d :: nat)) (λ c d. of-char c < (of-char d :: nat))
  by standard auto

Optimized version for execution

definition char-of-integer :: integer ⇒ char
  where [code-abbrev]: char-of-integer = char-of
definition integer-of-char :: char ⇒ integer
  where [code-abbrev]: integer-of-char = of-char

lemma char-of-integer-code [code]:
  char-of-integer k = (let
    (q0, b0) = bit-cut-integer k;
    (q1, b1) = bit-cut-integer q0;
    (q2, b2) = bit-cut-integer q1;
    (q3, b3) = bit-cut-integer q2;
    (q4, b4) = bit-cut-integer q3;
    (q5, b5) = bit-cut-integer q4;
    (q6, b6) = bit-cut-integer q5;
    (-, b7) = bit-cut-integer q6
  in Char b0 b1 b2 b3 b4 b5 b6 b7)
  by (simp add: bit-cut-integer-def char-of-integer-def char-of-def div-mult2-numeral-eq
  bit-iff-odd-drop-bit drop-bit-eq-div)

lemma integer-of-char-code [code]:
  integer-of-char (Char b0 b1 b2 b3 b4 b5 b6 b7) =
  (((((of_bool b7 ∗ 2 + of_bool b6) ∗ 2 +
    of_bool b5) ∗ 2 + of_bool b4) ∗ 2 +
    of_bool b3) ∗ 2 + of_bool b2) ∗ 2 +
  of_bool b1) ∗ 2 + of_bool b0
73.2 Strings as dedicated type for target language code generation

73.2.1 Logical specification

context

begin

qualified definition ascii-of :: char ⇒ char
  where ascii-of c = Char (digit0 c) (digit1 c) (digit2 c) (digit3 c) (digit4 c) (digit5 c) (digit6 c) False

qualified lemma ascii-of-Char [simp]:
  ascii-of (Char b0 b1 b2 b3 b4 b5 b6 b7) = Char b0 b1 b2 b3 b4 b5 b6 False
  by (simp add: ascii-of-def)

qualified lemma digit0-ascii-of-iff [simp]:
  digit0 (String.ascii-of c) ⇔ digit0 c
  by (simp add: String.ascii-of-def)

qualified lemma digit1-ascii-of-iff [simp]:
  digit1 (String.ascii-of c) ⇔ digit1 c
  by (simp add: String.ascii-of-def)

qualified lemma digit2-ascii-of-iff [simp]:
  digit2 (String.ascii-of c) ⇔ digit2 c
  by (simp add: String.ascii-of-def)

qualified lemma digit3-ascii-of-iff [simp]:
  digit3 (String.ascii-of c) ⇔ digit3 c
  by (simp add: String.ascii-of-def)

qualified lemma digit4-ascii-of-iff [simp]:
  digit4 (String.ascii-of c) ⇔ digit4 c
  by (simp add: String.ascii-of-def)

qualified lemma digit5-ascii-of-iff [simp]:
  digit5 (String.ascii-of c) ⇔ digit5 c
  by (simp add: String.ascii-of-def)

qualified lemma digit6-ascii-of-iff [simp]:
  digit6 (String.ascii-of c) ⇔ digit6 c
  by (simp add: String.ascii-of-def)

qualified lemma not-digit7-ascii-of [simp]:
  ¬ digit7 (ascii-of c)
  by (simp add: ascii-of-def)
qualified lemma ascii-of-idem:
  ascii-of c = c if ¬ digit7 c
  using that by (cases c) simp

qualified typedef literal = \{ cs. \forall c \in set cs. ¬ digit7 c \}
morphisms explode Abs-literal
proof
  show [] \in \{ cs. \forall c \in set cs. ¬ digit7 c \}
  by simp
qed

qualified setup-lifting type-definition-literal

qualified lift-definition implode :: string \Rightarrow literal
  is map ascii-of
  by auto

qualified lemma implode-explode-eq [simp]:
  String.implode (String.explode s) = s
proof transfer
  fix cs
  show map ascii-of cs = cs if \forall c \in set cs. ¬ digit7 c
  using that
  by (induction cs) (simp-all add: ascii-of-idem)
qed

qualified lemma explode-implode-eq [simp]:
  String.explode (String.implode cs) = map ascii-of cs
  by transfer rule
end

context linordered-euclidean-semiring-bit-operations
begin

context
begin

qualified lemma char-of-ascii-of [simp]:
  char-of (String.ascii-of c) = take-bit 7 (char-of c)
  by (cases c) (simp only: String.ascii-of-Char char-of-Char take-bit-horner-sum-bit-eq, simp)

qualified lemma ascii-of-char-of:
  String.ascii-of (char-of a) = char-of (take-bit 7 a)
  by (simp add: char-of-def bit-simps)
end
73.2.2 Syntactic representation

Logical ground representations for literals are:

1. 0 for the empty literal;
2. Literal $b_0 \ldots b_6 s$ for a literal starting with one character and continued by another literal.

Syntactic representations for literals are:

3. Printable text as string prefixed with $STR$;
4. A single ascii value as numerical hexadecimal value prefixed with $STR$.

instantiate $String.literal :: zero$

begin

context

begin

qualified lift-definition zero-literal :: $String.literal$

is Nil

by simp

instance ..

end

end

context

begin

qualified abbreviation (output) empty-literal :: $String.literal$

where empty-literal $\equiv$ 0

qualified lift-definition Literal :: bool $\Rightarrow$ bool $\Rightarrow$ bool $\Rightarrow$ bool $\Rightarrow$ bool $\Rightarrow$ bool $\Rightarrow$ bool $\Rightarrow$ String.literal $\Rightarrow$ String.literal

is $\lambda b_0 b_1 b_2 b_3 b_4 b_5 b_6 cs. \text{Char } b_0 b_1 b_2 b_3 b_4 b_5 b_6 \text{ False } \# cs$

by auto

qualified lemma Literal-eq-iff [simp]:

$\text{Literal } b_0 b_1 b_2 b_3 b_4 b_5 b_6 s = \text{Literal } c_0 c_1 c_2 c_3 c_4 c_5 c_6 t$

$\iff (b_0 \leftrightarrow c_0) \land (b_1 \leftrightarrow c_1) \land (b_2 \leftrightarrow c_2) \land (b_3 \leftrightarrow c_3)$

$\land (b_4 \leftrightarrow c_4) \land (b_5 \leftrightarrow c_5) \land (b_6 \leftrightarrow c_6) \land s = t$

by transfer simp
qualified lemma empty-neq-Literal [simp];
  empty-literal ≠ Literal b0 b1 b2 b3 b4 b5 b6 s
  by transfer simp

qualified lemma Literal-neq-empty [simp];
  Literal b0 b1 b2 b3 b4 b5 b6 s ≠ empty-literal
  by transfer simp

end

code-datatype 0 :: String.literal String.Literal

syntax
  -Literal :: str-position ⇒ String.literal (STR -)
  -Ascii :: num-const ⇒ String.literal (STR -)

ML-file ⟨Tools/literal.ML⟩

73.2.3 Operations

instantiation String.literal :: plus
begin

context
begin

qualified lift-definition plus-literal :: String.literal ⇒ String.literal ⇒ String.literal
  is (@)
  by auto

instance ..

end

end

instance String.literal :: monoid-add
  by (standard; transfer) simp-all

instantiation String.literal :: size
begin

context
  includes literal.lifting
begin

lift-definition size-literal :: String.literal ⇒ nat
  is length .


end

instance ..
end

instantiation String.literal :: equal
begin
context
begin
qualified lift-definition equal-literal :: String.literal ⇒ String.literal ⇒ bool
is HOL.equal .

instance
by (standard; transfer) (simp add: equal)
end
end

instantiation String.literal :: linorder
begin
context
begin
qualified lift-definition less-eq-literal :: String.literal ⇒ String.literal ⇒ bool
is ord.lexordp-eq (λc d. of-char c < (of-char d :: nat)) .

qualified lift-definition less-literal :: String.literal ⇒ String.literal ⇒ bool
is ord.lexordp (λc d. of-char c < (of-char d :: nat)) .

instance proof –
from linorder-char interpret linorder ord.lexordp-eq (λc d. of-char c < (of-char d :: nat))
ord.lexordp (λc d. of-char c < (of-char d :: nat)) :: string ⇒ string ⇒ bool
by (rule linorder.lexordp-linorder)
show PROP ?thesis
by (standard; transfer) (simp-all add: less-le-not-le linear)
qed
end
end
lemma infinite-literal:
infinite (UNIV :: String.literal set)
proof -
define S where S = range (λn. replicate n CHR "A")
have inj-on String.implode S
proof (rule inj-onI)
fix cs ds
assume String.implode cs = String.implode ds
then have String.explode (String.implode cs) = String.explode (String.implode ds)
  by simp
moreover assume cs ∈ S and ds ∈ S
ultimately show cs = ds
  by (auto simp add: S-def)
qed
moreover have infinite S
  by (auto simp add: S-def dest: finite-range-image1[of - length])
ultimately have infinite (String.implode  S)
  by (simp add: finite-image-iff)
then show thesis
  by (auto intro: finite-subset)
qed

73.2.4 Executable conversions
context
begin

qualified lift-definition asciis-of-literal :: String.literal ⇒ integer list
  is map of-char

qualified lemma asciis-of-zero [simp, code]:
  asciis-of-literal 0 = []
  by transfer simp

qualified lemma asciis-of-Literal [simp, code]:
  asciis-of-literal (String.Literal b0 b1 b2 b3 b4 b5 b6 s) =
    of-char (Char b0 b1 b2 b3 b4 b5 b6 False) # asciis-of-literal s
  by transfer simp

qualified lift-definition literal-of-ascii :: integer list ⇒ String.literal
  is map (String.ascii-of ∘ char-of)
  by auto

qualified lemma literal-of-ascii-Nil [simp, code]:
  literal-of-ascii [] = 0
  by transfer simp
qualified lemma literal-of-asciis-Cons [simp, code]:
  literal-of-asciis (k # ks) = (case char-of k
    of Char b0 b1 b2 b3 b4 b5 b6 b7 ⇒ String.Literal b0 b1 b2 b3 b4 b5 b6
    (literal-of-asciis ks))
  by (simp add: char-of-def) (transfer, simp add: char-of-def)

qualified lemma literal-of-asciis-of-literal [simp]:
  literal-of-asciis (asciis-of-literal s) = s
proof transfer
  fix cs
  assume ∀ c ∈ set cs. ¬ digit? c
  then show map (String.ascii-of ◦ char-of) (map of-char cs) = cs
    by (induction cs) (simp-all add: String.ascii-of-idem)
qed

qualified lemma explode-code [code]:
  String.explode s = map char-of (asciis-of-literal s)
  by transfer simp

qualified lemma implode-code [code]:
  String.implode cs = literal-of-asciis (map of-char cs)
  by transfer simp

qualified lemma equal-literal [code]:
  HOL.equal (String.Literal b0 b1 b2 b3 b4 b5 b6 s)
  (String.Literal a0 a1 a2 a3 a4 a5 a6 r)
  ⇐ (b0 ←→ a0) ∧ (b1 ←→ a1) ∧ (b2 ←→ a2) ∧ (b3 ←→ a3)
  ∧ (b4 ←→ a4) ∧ (b5 ←→ a5) ∧ (b6 ←→ a6) ∧ (s = r)
  by (simp add: equal)

end

73.2.5 Technical code generation setup

Alternative constructor for generated computations

context
begin

qualified definition Literal' :: bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool
  ⇒ String.literal ⇒ String.literal
  where [simp]: Literal' = String.Literal

lemma [code]:
  "Literal' b0 b1 b2 b3 b4 b5 b6 s = String.literal-of-asciis
  [foldr (λb k. of-bool b + k * 2) [b0, b1, b2, b3, b4, b5, b6] 0] + s"
proof
  have "foldr (λb k. of-bool b + k * 2) [b0, b1, b2, b3, b4, b5, b6] 0 = of-char
  (Char b0 b1 b2 b3 b4 b5 b6 False)"
by simp
moreover have \((\text{Literal'}\ b0\ b1\ b2\ b3\ b4\ b5\ b6\ s =\ \text{String.literal-of-asciis}
\[\text{of-char}\ (\text{Char}\ b0\ b1\ b2\ b3\ b4\ b5\ b6\ \text{False})\] + s)\)
by (unfold \text{Literal'}-def) \((\text{transfer, simp only: list.simps comp-apply char-of-char, simp})\)
ultimately show \(\theory\)
by simp
qed

lemma [code-computation-unfold]:
\(\text{String.Literal} = \text{Literal'}\)
by simp
end

\text{code-reserved SML string String Char Str-Literal}
\text{code-reserved OCaml string String Char Str-Literal}
\text{code-reserved Haskell Prelude}
\text{code-reserved Scala string}

code-identifier
\text{code-module String \(\rightarrow\)}
\(\text{(SML) Str and (OCaml) Str and (Haskell) Str and (Scala) Str}\)

code-printing
type-constructor \text{String.literal \(\rightarrow\)}
\(\text{(SML) String and (OCaml) String and (Haskell) String and (Scala) String}\)
constant \text{STR'}
\(\text{(SML) and (OCaml) and (Haskell) and (Scala)}\)

setup \((\text{fold Literal.add-code [SML, OCaml, Haskell, Scala]}\))

\text{code-printing}
\text{code-module Str-Literal \(\rightarrow\)}
\(\text{(SML) structure Str-Literal =}\)
struct
\(\text{fun map f} \ [\] = \ []
| \text{map f} \ (x :: xs) = f x :: \text{map f} \ xs; \text{(* deliberate clone not relying on List.- module *)}\)
fun check-ascii (k : IntInf.int) =  
if 0 <= k andalso k < 128  
  then k  
else raise Fail Non-ASCII character in literal;  

val char-of-ascii = Char.chr o IntInf.toInt o check-ascii;  

val ascii-of-char = check-ascii o IntInf.fromInt o Char.ord;  

val literal-of-asciis = String.implode o map char-of-ascii;  

val ascii-of-literal = map ascii-of-char o String.explode;  

end;  

for constant String.literal-of-asciis String.ascii-of-literal  
  and (OCaml) module Str-Literal =  
  struct  
  let implode f xs =  
    let rec length xs = match xs with  
      | [] -> 0  
      | x :: xs -> 1 + length xs in  
    let rec nth xs n = match xs with  
      | (x :: xs) -> if n <= 0 then x else nth xs (n - 1)  
      in String.init (length xs) (fun n -> f (nth xs n));;  
  let explode f s =  
    let rec map-range f n =  
      if n <= 0 then [] else map-range f (n - 1) @ [ f n]  
    in map-range (fun n -> f (String.get s n)) (String.length s);;  
  let z-128 = Z.of-int 128;;  
  let check-ascii (k : Z.t) =  
    if Z.leq Z.zero k && Z.lt k z-128  
    then k  
    else failwith Non-ASCII character in literal;;  
  let char-of-ascii k = Char.chr (Z.to-int (check-ascii k));;  
  let ascii-of-char c = check-ascii (Z.of-int (Char.code c));;  
  let literal-of-asciis ks = implode char-of-ascii ks;;  
  let ascii-of-literal s = explode ascii-of-char s;;  
end;  

for constant String.literal-of-asciis String.ascii-of-literal  
  and (OCaml) infixl 18 ~  
  and (SML) infixr 6 ~
and (Haskell) infixr 5 ++
and (Scala) infixl 7 +
| constant String.literal-of-ascii →
(SML) Str'-.Literal.literal'-of'-ascii
and (OCaml) Str'-.Literal.ascii'-of'-ascii
and (Haskell) map/ (let chr k | (0 <= k & k < 128) = Prelude.toEnum k ::
Prelude.Char in chr . Prelude.fromInteger)
and (Scala) -map((k: BigInt) => BigInt(0) <= k & k < BigInt(128) match {
case true => k.charValue case false => sys.error(Non-ASCII character in literal)
}).mkString
| constant String.ascii-of-literal →
(SML) Str'-.Literal.ascii'-of'-literal
and (OCaml) Str'-.Literal.ascii'-of'-literal
and (Haskell) map/ (let ord k | (k < 128) = Prelude.toInteger k in ord .
(Prelude.fromEnum :: Prelude.Char -> Prelude.Int))
and (Scala) !(.-toList.map(c => { val k: Int = c.toInt; k < 128 match {
case true => BigInt(k) case false => sys.error(Non-ASCII character in literal) } )))
| class-instance String.literal :: equal →
(Haskell) –
| constant HOL.equal :: String.literal ⇒ String.literal ⇒ bool →
(SML) !(.- : string) = -
and (OCaml) !(- : string) = -
and (Haskell) infix 4 ==
and (Scala) infixl 5 ==
| constant (<=) :: String.literal ⇒ String.literal ⇒ bool →
(SML) !(.- : string) <= -
and (OCaml) !(- : string) <= -
and (Haskell) infix 4 <=
— Order operations for String.literal work in Haskell only if no type class instance needs to be generated, because String = [Char] in Haskell and char list need not have the same order as String.literal.
and (Scala) infixl 4 <=
and (Eval) infixl 6 <=
| constant (<) :: String.literal ⇒ String.literal ⇒ bool →
(SML) !(.- : string) < -
and (OCaml) !(- : string) < -
and (Haskell) infix 4 <
and (Scala) infixl 4 <
and (Eval) infixl 6 <

73.2.6 Code generation utility
setup «Sign.map-naming (Name-Space.mandatory-path Code)»

definition abort :: String.literal ⇒ (unit ⇒ 'a) ⇒ 'a
where [simp]: abort _ f = f ()
declare [[code drop: Code.abort]]
lemma abort-cong:
  \( \text{msg} = \text{msg}' \implies \text{Code.abort} \text{msg} f = \text{Code.abort} \text{msg}' f \)
by simp

setup (Sign.map-naming Name-Space.parent-path)

setup (Code-Simp.map-ss (Simplifier.add-cong @{thm Code.abort-cong} ))

code-printing
constant Code.abort →
  (SML) !(raise/ Fail/ -)
and (OCaml) failwith
and (Haskell) !(error/ ::/ forall a./ String -- ((()) -> a) -> a)
and (Scala) !{/ sys.error((·));/ ((·)).apply((·))/ }

73.2.7 Finally

lifting-update literal.lifting
lifting-forget literal.lifting

end

74 Reflecting Pure types into HOL

theory Typerep
  imports String
begin

datatype typerep = Typerep String.literal typerep list

class typerep =
  fixes typerep :: 'a itself ⇒ typerep
begin

definition typerep-of :: 'a ⇒ typerep where
  [simp]: typerep-of x = typerep TYPE('a)

end

syntax
-TYPEREP :: type ⇒ logic ((1TYPEREP/(1'(·))))

parse-translation ( let
  fan typerep-tr (*-TYPEREP*) [ty] =
  Syntax.const const-syntax (typerep) $
  (Syntax.const syntax-const (-constrain) $ Syntax.const const-syntax (Pure.type) $
   (Syntax.const type-syntax (itself) $ ty))
THEORY "Typerep"

| typerep-tr ("-TYPEREP") ts = raise TERM (typerep-tr, ts); in [(syntax-const (-TYPEREP), K typerep-tr)] end |

| | |
| | |
| | |
| typed-print-translation |
| let |
| fan typerep-tr' ctxt ("typerep") Type (fun Type (itself T) -› |
| (Const (const-syntax:Pure.type, _) :: ts) = |
| Term.list-comb |
| (Syntax.const syntax-const (-TYPEREP) $ Syntax-Phases.term-of-typ |
| ctxt T, ts) |
| (typerep-tr' - T ts = raise Match; |
| in [(const-syntax typerep, typerep-tr')] end |
| ) |

| setup |
| let |
| fun add-typerep tyco thy = |
| let |
| val sorts = replicate (Sign.arity-number thy tyco) sort typerep; |
| val vs = Name.invent-names Name.context 'a sorts; |
| val ty = Type (tyco, map TFree vs); |
| val lhs = Const (typerep ty) $ Free (T, Term.itselfT ty); |
| val rhs = Const (Typerep) $ HOLogic.mk-literal tyco |
| $ HOLogic.mk-list Type (map (HOLogic.mk-typerep o TFree) vs); |
| val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs)); |
| in |
| thy |
| > Class.instantiation ([tyco], vs, sort typerep) |
| > (fn lthy => Syntax.check-term lthy eq) |
| -> (fn eq => Specification.definition NONE [] (Binding.empty-atts, eq)) |
| > snd |
| > Class.prove-instantiation-exit (fn ctxt => Class.intro-classes-tac ctxt []) | |
| end; |

| fun ensure-typerep tyco thy = |
| if not (Sorts.has-instance (Sign.classes-of thy) tyco sort typerep) |
| andalso Sorts.has-instance (Sign.classes-of thy) tyco sort (type) |
| then add-typerep tyco thy else thy; |
| in |
| add-typerep type-name (fan) |
| #> Typedef.interpretation (Local-Theory.background-theory o ensure-typerep) |
| #> Code.type-interpretation ensure-typerep |
| end |
| ) |
theory Predicate

imports String

begin

75 Predicates as enumerations

theory Predicate
imports String
begin

75.1 The type of predicate enumerations (a monad)

datatype (plugins only: extraction) (dead 'a) pred = Pred (eval: 'a ⇒ bool)

lemma pred-eqI: 
(∀w. eval P w ⇔ eval Q w) ⇒ P = Q
by (cases P, cases Q) (auto simp add: fun-eq-iff)

lemma pred-eq-iff: 
P = Q ⇒ (∀w. eval P w ⇔ eval Q w)
by (simp add: pred-eqI)

instantiation pred :: (type) complete-lattice
begin

definition P ≤ Q ←→ eval P ≤ eval Q

definition P < Q ←→ eval P < eval Q

definition ⊥ = Pred ⊥
lemma eval-bot [simp]:
  \text{eval } \bot = \bot
by (simp add: bot-pred-def)

definition
\top = \text{Pred } \top

lemma eval-top [simp]:
  \text{eval } \top = \top
by (simp add: top-pred-def)

definition
P \sqcap Q = \text{Pred } (\text{eval } P \sqcap \text{eval } Q)

lemma eval-inf [simp]:
  \text{eval } (P \sqcap Q) = \text{eval } P \sqcap \text{eval } Q
by (simp add: inf-pred-def)

definition
P \sqcup Q = \text{Pred } (\text{eval } P \sqcup \text{eval } Q)

lemma eval-sup [simp]:
  \text{eval } (P \sqcup Q) = \text{eval } P \sqcup \text{eval } Q
by (simp add: sup-pred-def)

definition
\bigcap A = \text{Pred } (\bigcap (\text{eval } ' A))

lemma eval-Inf [simp]:
  \text{eval } (\bigcap A) = \bigcap (\text{eval } ' A)
by (simp add: Inf-pred-def)

definition
\bigcup A = \text{Pred } (\bigcup (\text{eval } ' A))

lemma eval-Sup [simp]:
  \text{eval } (\bigcup A) = \bigcup (\text{eval } ' A)
by (simp add: Sup-pred-def)

instance proof
qed (auto intro!: pred-eqI simp add: less-eq-pred-def less-pred-def le-fun-def less-fun-def)

end

lemma eval-INF [simp]:
  \text{eval } (\bigcap (f ' A)) = \bigcap ((\text{eval } \circ f) ' A)
by (simp add: image-comp)
lemma eval-SUP [simp]:
  \[ \text{eval} \left( \bigsqcup (f \cdot A) \right) = \bigsqcup \left( \text{eval} \circ f \right) \cdot A \]
  by (simp add: image-comp)

instantiation pred :: (type) complete-boolean-algebra
begin

definition \( \neg P \) = \( \text{Pred} \left( \neg \text{eval} P \right) \)

lemma eval-compl [simp]:
  \[ \text{eval} (\neg P) = \neg \text{eval} P \]
  by (simp add: uminus-pred-def)

definition \( P \neg Q \) = \( \text{Pred} \left( \text{eval} P \neg \text{eval} Q \right) \)

lemma eval-minus [simp]:
  \[ \text{eval} (P \neg Q) = \text{eval} P \neg \text{eval} Q \]
  by (simp add: minus-pred-def)

instance proof
  fix A :: 'a pred set set
  show \( \text{Sup } A \leq \bigsqcup \left( \text{Inf } \{ f \cdot A \mid \forall Y \in A. f Y \in Y \} \right) \)
  proof (simp add: less-eq-pred-def Sup-fun-def Inf-fun-def, safe)
    fix w
    assume A: \( \forall x \in A. \exists f \in x. \text{eval} f w \)
    define F where F = \( \lambda x. \text{SOME } f. f \in x \wedge \text{eval} f w \)
    have [simp]: \( \forall f \in (F \cdot A) . \text{eval} f w \)
      by (metis (no-types, lifting) A F-def image-iff some-eq-ex)
    have (\( \exists f. f \cdot A = f \cdot A \land (\forall Y \in A. f Y \in Y) \)) \& (\( \forall f \in (F \cdot A) . \text{eval} f w \))
      using A by (simp, metis (no-types, lifting) F-def someI)
    from this show \( \exists x. (\exists f. x = f \cdot A \land (\forall Y \in A. f Y \in Y) ) \land (\forall f \in x. \text{eval} f w ) \)
      by (rule exI [of - F \cdot A])
    qed
  qed (auto intro!: pred-eqI)
end

definition single :: 'a \Rightarrow 'a pred where
  \( \text{single } x = \text{Pred} \left( (\hat{} x) \right) \)

lemma eval-single [simp]:
  \[ \text{eval} (\text{single } x) = (\hat{} x) \]
  by (simp add: single-def)

definition bind :: 'a pred \Rightarrow ('a \Rightarrow 'b pred) \Rightarrow 'b pred (infixl \( \triangleright \triangleright \)) where
  \( P \triangleright \triangleright f = (\bigsqcup \left( f \cdot \{ x. \text{eval} P x \} \right) ) \)
lemma eval-bind [simp]:
  eval (P △ f) = eval (∪ (f ' {x. eval P x}))
by (simp add: bind-def)

lemma bind-bind:
  (P △ Q) △ R = P △ (λx. Q x △ R)
by (rule pred-eqI) auto

lemma bind-single:
  P △ single = P
by (rule pred-eqI) auto

lemma single-bind:
  single x △ P = P x
by (rule pred-eqI) auto

lemma bottom-bind:
  ⊥ △ P = ⊥
by (rule pred-eqI) auto

lemma sup-bind:
  (P ⊔ Q) △ R = P △ R ⊔ Q △ R
by (rule pred-eqI) auto

lemma Sup-bind:
  (∪ A △ f) = (∪ ((λx. x △ f) ' A))
by (rule pred-eqI) auto

lemma pred-iffI:
assumes ∏ x. eval A x ⇒ eval B x
and ∏ x. eval B x ⇒ eval A x
shows A = B
using assms by (auto intro: pred-eqI)

lemma singleI: eval (single x) x
by simp

lemma singleI-unit: eval (single ()) x
by simp

lemma singleE: eval (single x) y ⇒ (y = x ⇒ P) ⇒ P
by simp

lemma singleE': eval (single x) y ⇒ (x = y ⇒ P) ⇒ P
by simp

lemma bindI: eval P x ⇒ eval (Q x) y ⇒ eval (P △ Q) y
by auto
lemma bindE: eval \( R \Rightarrow Q \) \( y \Rightarrow (x. \) eval \( R x \Rightarrow \) eval \( Q x \) \( y \Rightarrow P \) \) \( \Rightarrow P \)
by auto

lemma botE: eval \( \bot \) \( x \Rightarrow P \)
by auto

lemma supI1: eval \( A x \Rightarrow \) eval \( (A \sqcup B) x \)
by auto

lemma supI2: eval \( B x \Rightarrow \) eval \( (A \sqcup B) x \)
by auto

lemma supE: eval \( (A \sqcup B) x \Rightarrow (x. \) eval \( A x \Rightarrow P \) \) \( \Rightarrow (x. \) eval \( B x \Rightarrow P \) \) \( \Rightarrow P \)
by auto

lemma single-not-bot [simp]:
\( \) single \( x \neq \bot \)
by (auto simp add: single-def bot-pred-def fun-eq-iff)

lemma not-bot:
\( \) assumes \( A \neq \bot \)
\( \) obtains \( x \) where eval \( A x \)
using assms by (cases \( A \)) (auto simp add: bot-pred-def)

75.2 Emptiness check and definite choice

definition is-empty :: 'a pred \( \Rightarrow \) bool where
is-empty \( A \leftarrow \) \( A = \bot \)

lemma is-empty-bot:
is-empty \( \bot \)
by (simp add: is-empty-def)

lemma not-is-empty-single:
\( \neg \) is-empty \( (\) single \( x \) \( ) \)
by (auto simp add: is-empty-def single-def bot-pred-def fun-eq-iff)

lemma is-empty-sup:
is-empty \( (A \sqcup B) \leftarrow \) is-empty \( A \land \) is-empty \( B \)
by (auto simp add: is-empty-def)

definition singleton :: (unit \( \Rightarrow \) 'a) \( \Rightarrow \) 'a pred \( \Rightarrow \) 'a where
singleton \( \) default \( A \) = (if \( \exists !x. \) eval \( A x \) \( \) then \( \) THE \( x. \) eval \( A x \) else default \( ) \) \) for \( \) default

lemma singleton-eqI:
\( \exists !x. \) eval \( A x \) \Rightarrow eval \( A x \Rightarrow \) singleton \( \) default \( A = x \) for \( \) default
by (auto simp add: singleton-def)
lemma eval-singletonI:
\exists! x. eval A x \implies eval A (singleton default A) for default
proof
  assume assm: \exists! x. eval A x
  then obtain x where x: eval A x ..
  with assm have singleton default A = x by (rule singleton-eqI)
  with x show \?thesis by simp
qed

lemma single-singleton:
\exists! x. eval A x \implies single (singleton default A) = A for default
proof
  assume assm: \exists! x. eval A x
  then have eval A (singleton default A) by (rule eval-singletonI)
  moreover from assm have \(\forall x. \) eval A x \implies singleton default A = x
  by (rule singleton-eqI)
  ultimately have eval (single (singleton default A)) = eval A
    by (simp (no-asm-use) add: single-def fun-eq-iff) blast
  then have \(\forall x. \) eval (single (singleton default A)) x = eval A x
    by simp
  then show \?thesis by (rule pred-eqI)
qed

lemma singleton-undefinedI:
\neg (\exists! x. eval A x) \implies singleton default A = default () for default
by (simp add: singleton-def)

lemma singleton-bot:
singleton default \bot = default () for default
by (auto simp add: bot-pred-def intro: singleton-undefinedI)

lemma singleton-single:
singleton default (single x) = x for default
by (auto simp add: intro: singleton-eqI singleI elim: singleE)

lemma singleton-sup-single-single:
singleton default (single x \sqcup single y) = (if x = y then x else default ()) for default
proof (cases x = y)
  case True then show \?thesis by (simp add: singleton-single)
next
  case False
  have eval (single x \sqcup single y) x
    and eval (single x \sqcup single y) y
    by (auto intro: supI1 supI2 singleI)
  with False have \neg (\exists! z. eval (single x \sqcup single y) z)
    by blast
then have singleton default (single x ⊔ single y) = default ()
  by (rule singleton-undefinedI)
with False show ?thesis by simp
qed

lemma singleton-sup-aux:
  singleton default (A ⊔ B) = (if A = ⊥ then singleton default B
  else if B = ⊥ then singleton default A
  else singleton default
      (single (singleton default A) ⊔ single (singleton default B)))
  for default
proof (cases (∃!x. eval A x) ∧ (∃!y. eval B y))
  case True then show ?thesis
    by (simp add: single-singleton)
next
  case False
from False have A-or-B:
  singleton default A = default () ∨ singleton default B = default ()
  by (auto intro: singleton-undefinedI)
then have rhs:
  singleton default (single (singleton default A) ⊔ single (singleton default B)) = default ()
  by (auto simp add: singleton-sup-single-single singleton-single)
from False
  have not-unique:
  ¬ (∃!x. eval A x) ∨ ¬ (∃!y. eval B y)
  by simp
show ?thesis proof (cases A ≠ ⊥ ∧ B ≠ ⊥)
  case True
  then obtain a b where: a: eval A a and b: eval B b
    by (blast elim: not-bot)
  with True not-unique have ¬ (∃!x. eval (A ⊔ B) x)
    by (auto simp add: sup-pred-def bot-pred-def)
  then have singleton default (A ⊔ B) = default ()
    by (rule singleton-undefinedI)
  with True rhs show ?thesis by simp
next
  case False then show ?thesis by auto
qed
qed

lemma singleton-sup:
  singleton default (A ⊔ B) = (if A = ⊥ then singleton default B
  else if B = ⊥ then singleton default A
  else if singleton default A = singleton default B then singleton default A else default ()))
  for default
using singleton-sup-aux [of default A B] by (simp only: singleton-sup-single-single)

75.3 Derived operations

definition if-pred :: bool ⇒ unit pred where
  if-pred-eq: if-pred b = (if b then single () else ⊥)

definition holds :: unit pred ⇒ bool where
  holds-eq: holds P = eval P ()
definition not-pred :: unit pred ⇒ unit pred where
not-pred-eq: not-pred P = (if eval P () then ⊥ else single ())

lemma if-predI: P ⇒ eval (if-pred P) ()
unfolding if-pred-eq by (auto intro: singleI)

lemma if-predE: eval (if-pred b) x ⇒ (b ⇒ x = () ⇒ P) ⇒ P
unfolding if-pred-eq by (cases b) (auto elim: botE)

lemma not-predI: ¬ P ⇒ eval (not-pred (Pred (λu. P))) ()
unfolding not-pred-eq by (auto intro: singleI)

lemma not-predI’: ¬ eval P () ⇒ eval (not-pred P) ()
unfolding not-pred-eq by (auto intro: singleI)

lemma not-predE: eval (not-pred (Pred (λu. P))) x ⇒ (¬ P ⇒ thesis) ⇒ thesis
unfolding not-pred-eq
by (auto split: if-split-asm elim: botE)

lemma not-predE’: eval (not-pred P) x ⇒ (¬ eval P x ⇒ thesis) ⇒ thesis
unfolding not-pred-eq
by (auto split: if-split-asm elim: botE)

lemma f () = False ∨ f () = True
by simp

lemma closure-of-bool-cases [no-atp]:
fixes f :: unit ⇒ bool
assumes f = (λu. False) ⇒ P f
assumes f = (λu. True) ⇒ P f
shows P f
proof –
have f = (λu. False) ∨ f = (λu. True)
apply (cases f ()
apply (rule disjI2)
apply (rule ext)
apply (simp add: unit-eq)
apply (rule disjI1)
apply (rule ext)
apply (simp add: unit-eq)
done
from this assms show ?thesis by blast
qed

lemma unit-pred-cases:
assumes P ⊥
assumes P (single ())
shows P Q
using assms unfolding bot-pred-def bot-fun-def bot-bool-def empty-def single-def

proof (cases Q)
  fix f
  assume P (Pred (\lambda u. False)) P (Pred (\lambda u. () = u))
  then have P (Pred f)
    by (cases - f rule: closure-of-bool-cases) simp-all
  moreover assume Q = Pred f
  ultimately show P Q by simp
qed

lemma holds-if-pred:
  holds (if-pred b) = b
unfolding if-pred-eq holds-eq
by (cases b) (auto intro: singleI elim: botE)

lemma if-pred-holds:
  if-pred (holds P) = P
unfolding if-pred-eq holds-eq
by (rule unit-pred-cases) (auto intro: singleI elim: botE)

lemma is-empty-holds:
  is-empty P \iff \neg holds P
unfolding is-empty-def holds-eq
by (rule unit-pred-cases) (auto elim: botE intro: singleI)

definition map :: ('a \Rightarrow 'b) \Rightarrow 'a pred \Rightarrow 'b pred
where map f P = 
    P >> (\lambda x. f x)

lemma eval-map [simp]:
  eval (map f P) = \bigcup x \in \{x. eval P x\}. (\lambda y. f x = y)
by (simp add: map-def comp-def image-comp)

functor map: map
by (rule ext, rule pred-eqI, auto)+

75.4 Implementation

datatype (plugins only: code extraction) (dead 'a) seq =
  Empty
| Insert 'a 'a pred
| Join 'a pred 'a seq

primrec pred-of-seq :: 'a seq \Rightarrow 'a pred
where
  pred-of-seq Empty = \bot
| pred-of-seq (Insert x P) = single x \b P
| pred-of-seq (Join P xq) = P \b pred-of-seq xq

definition Seq :: (unit \Rightarrow 'a seq) \Rightarrow 'a pred
where
Seq f = pred-of-seq (f ()))
code-datatype Seq

primrec member :: 'a seq ⇒ 'a ⇒ bool where
  member Empty x ←→ False
  | member (Insert y P) x ←→ x = y ⊔ eval P x
  | member (Join P xq) x ←→ eval P x ⊔ member xq x

lemma eval-member:
  member xq = eval (pred-of-seq xq)
proof (induct xq)
  case Empty show ?case
    by (auto simp add: fun-eq-iff elim: botE)
next
  case Insert show ?case
    by (auto simp add: fun-eq-iff elim: supE singleE intro: supI1 supI2 singleI)
next
  case Join then show ?case
    by (auto simp add: fun-eq-iff elim: supE intro: supI1 supI2)
qed

lemma eval-code [code]: eval (Seq f) = member (f ())
unfolding Seq-def by (rule sym, rule eval-member)

lemma single-code [code]:
  single x = Seq (λu. Insert x ⊥)
unfolding Seq-def by simp

primrec apply :: ('a ⇒ 'b pred) ⇒ 'a seq ⇒ 'b seq where
  apply f Empty = Empty
  | apply f (Insert x P) = Join (f x) (Join (P ≃ f) Empty)
  | apply f (Join P xq) = Join (P ≃ f) (apply f xq)

lemma apply-bind:
  pred-of-seq (apply f xq) = pred-of-seq xq ≃ f
proof (induct xq)
  case Empty show ?case
    by (simp add: bottom-bind)
next
  case Insert show ?case
    by (simp add: single-bind sup-bind)
next
  case Join then show ?case
    by (simp add: sup-bind)
qed

lemma bind-code [code]:
  Seq g ≃ f = Seq (λu. apply f (g ()))
unfolding Seq-def by (rule sym, rule apply-bind)
lemma bot-set-code [code]:
⊥ = Seq (λu. Empty)
unfolding Seq-def by simp

primrec adjunct :: 'a pred ⇒ 'a seq ⇒ 'a seq where
adjunct P Empty = Join P Empty
| adjunct P (Insert x Q) = Insert x (Q △ P)
| adjunct P (Join Q xq) = Join Q (adjunct P xq)

lemma adjunct-sup:
pred-of-seq (adjunct P xq) = P △ pred-of-seq xq
by (induct xq) (simp-all add: sup-assoc sup-commute sup-left-commute)

lemma sup-code [code]:
Seq f △ Seq g = Seq (λu. case f () of Empty ⇒ g ()
| Insert x P ⇒ Insert x (P △ Seq g)
| Join P xq ⇒ adjunct (Seq g) (Join P xq))
proof (cases f ())
case Empty
  thus ?thesis
  unfolding Seq-def by (simp add: sup-commute [of ⊥])
next
case Insert
  thus ?thesis
  unfolding Seq-def by (simp add: sup-assoc)
next
case Join
  thus ?thesis
  unfolding Seq-def
  by (simp add: adjunct-sup sup-assoc sup-commute sup-left-commute)
qed

primrec contained :: 'a seq ⇒ 'a pred ⇒ bool where
contained Empty Q ←→ True
| contained (Insert x P) Q ←→ eval Q x ∧ P ≤ Q
| contained (Join P xq) Q ←→ P ≤ Q ∧ contained xq Q

lemma single-less-eq-eval:
single x ≤ P ←→ eval P x
by (auto simp add: less-eq-pred-def le-fun-def)

lemma contained-less-eq:
contained xq Q ←→ pred-of-seq xq ≤ Q
by (induct xq) (simp-all add: single-less-eq-eval)

lemma less-eq-pred-code [code]:
Seq f ≤ Q = (case f ()

THEORY “Predicate” 1631
of Empty ⇒ True
| Insert x P ⇒ eval Q x ∧ P ≤ Q
| Join P xq ⇒ P ≤ Q ∧ contained xq Q)
by (cases f (f))
   (simp-all add: Seq-def single-less-eq-eval contained-less-eq)

instantiation pred :: (type) equal
begin

definition equal-pred
where [simp]: HOL.equal P Q ←→ P = (Q :: 'a pred)

instance by standard simp

end

lemma [code]:
   HOL.equal P Q ←→ P ≤ Q ∧ Q ≤ P for P Q :: 'a pred
   by auto

lemma [code nbe]:
   HOL.equal P P ←→ True for P :: 'a pred
   by (fact equal-refl)

lemma [code]:
   case-pred f P = f (eval P)
   by (fact pred.case-eq-if)

lemma [code]:
   rec-pred f P = f (eval P)
   by (cases P) simp

inductive eq :: 'a ⇒ 'a ⇒ bool where eq x x

lemma eq-is-eq: eq x y ≡ (x = y)
   by (rule eq-reflection) (auto intro: eq.intros elim: eq.cases)

primrec null :: 'a seq ⇒ bool where
   null Empty ←→ True
   | null (Insert x P) ←→ False
   | null (Join P xq) ←→ is-empty P ∧ null xq

lemma null-is-empty:
   null xq ←→ is-empty (pred-of-seq xq)
   by (induct xq) (simp-all add: is-empty-bot not-is-empty-single is-empty-sup)

lemma is-empty-code [code]:
   is-empty (Seq f) ←→ null (f (f))
   by (simp add: null-is-empty Seq-def)
primrec the-only :: (unit ⇒ 'a) ⇒ 'a seq ⇒ 'a where
  the-only default Empty = default () for default
| the-only default (Insert x P) =
  (if is-empty P then x else let y = singleton default P in if x = y then x else default ()) for default
| the-only default (Join P xq) =
  (if is-empty P then the-only default xq else if null xq then singleton default P
   else let x = singleton default P; y = the-only default xq in
   if x = y then x else default ()) for default

lemma the-only-singleton:
the-only default xq = singleton default (pred-of-seq xq) for default
by (induct xq)
(auto simp add: singleton-bot singleton-single is-empty-def
 null-is-empty Let-def singleton-sup)

lemma singleton-code [code]:
singleton default (Seq f) =
(case f () of
  Empty ⇒ default ()
| Insert x P ⇒ if is-empty P then x
   else let y = singleton default P in
   if x = y then x else default ()
| Join P xq ⇒ if is-empty P then the-only default xq
   else if null xq then singleton default P
   else let x = singleton default P; y = the-only default xq in
   if x = y then x else default ()) for default
by (cases f ()
(auto simp add: Seq-def the-only-singleton is-empty-def
 null-is-empty singleton-bot singleton-single singleton-sup Let-def)

definition the :: 'a pred ⇒ 'a where
the A = (THE x. eval A x)

lemma the-eqI:
(THE x. eval P x) = x ⇒ the P = x
by (simp add: the-def)

lemma the-eq [code]: the A = singleton (λx. Code.abort (STR "not-unique") (λ- the A)) A
by (rule the-eqI) (simp add: singleton-def the-def)

code-reflect Predicate
datatypes pred = Seq and seq = Empty | Insert | Join

ML "
signature PREDICATE =
sig
val anamorph: ('a -> ('b * 'a option) -> int -> 'a -> 'b list * 'a

datatype 'a pred = Seq of (unit -> 'a seq)

and 'a seq = Empty | Insert of ('a * 'a pred | Join of 'a pred * 'a seq

val map: ('a -> 'b) -> 'a pred -> 'b pred
val yield: 'a pred -> ('a * 'a pred) option
val yieldn: int -> 'a pred -> 'a list * 'a pred
end;

structure Predicate : PREDICATE =

fun anamorph f k x = (if k = 0 then ([], x)
else case f x
  of NONE => ([], x)
   | SOME (v, y) => let
     val k' = k - 1;
     val (vs, z) = anamorph f k' y
     in (v :: vs, z) end);

datatype pred = datatype Predicate.pred

datatype seq = datatype Predicate.seq

fun map f = @{code Predicate.map} f;

fun yield (Seq f) = next (f ())
and next Empty = NONE
  | next (Insert (x, P)) = SOME (x, P)
  | next (Join (P, xq)) = (case yield P
    of NONE => next xq
     | SOME (x, Q) => SOME (x, Seq (fn - => Join (Q, xq)))));

fun yieldn k = anamorph yield k;

end;

Conversion from and to sets

definition pred-of-set :: 'a set ⇒ 'a pred where
  pred-of-set = Pred o (λA x. x ∈ A)

lemma eval-pred-of-set [simp]:
  eval (pred-of-set A) x ←→ x ∈ A
by (simp add: pred-of-set-def)

definition set-of-pred :: 'a pred ⇒ 'a set where
  set-of-pred = Collect o eval

lemma member-set-of-pred [simp]:
\[ x \in \text{set-of-pred } P \iff \text{Predicate.eval } P \, x \]
by \text{(simp add: set-of-pred-def)}

**definition** set-of-seq :: 'a seq \Rightarrow 'a set where
\[ \text{set-of-seq } = \text{set-of-pred } \circ \text{pred-of-seq} \]

**lemma** member-set-of-seq [simp]:
\[ x \in \text{set-of-seq } xq = \text{Predicate.member } xq \, x \]
by \text{(simp add: set-of-seq-def eval-member)}

**lemma** of-pred-code [code]:
\[ \text{set-of-pred } (\text{Predicate}.\text{Seq } f) = \text{case } f (\text{()) of} \]
\[ \text{Predicate.Empty } \Rightarrow \{\} \]
\[ | \text{Predicate.Insert } x \, P \Rightarrow \text{insert } x \, (\text{set-of-pred } P) \]
\[ | \text{Predicate.Join } P \, xq \Rightarrow \text{set-of-pred } P \cup \text{set-of-seq } xq \]
by \text{(auto split: seq.split simp add: eval-code)}

**lemma** of-seq-code [code]:
\[ \text{set-of-seq } \text{Predicate.Empty } = \{\} \]
\[ \text{set-of-seq } (\text{Predicate.Insert } x \, P) = \text{insert } x \, (\text{set-of-pred } P) \]
\[ \text{set-of-seq } (\text{Predicate.Join } P \, xq) = \text{set-of-pred } P \cup \text{set-of-seq } xq \]
by \text{auto}

Lazy Evaluation of an indexed function

**function** iterate-upto :: \((\text{natural } \Rightarrow 'a) \Rightarrow \text{natural } \Rightarrow \text{natural } \Rightarrow 'a \text{ Predicate.pred}\)
where
\[ \text{iterate-upto } f \, n \, m = \]
\[ \text{Predicate.Seq } (\%u. \text{if } n > m \text{ then } \text{Predicate.Empty} \]
\[ \text{else } \text{Predicate.Insert } (f \, n) \, (\text{iterate-upto } f \, (n + 1) \, m)) \]
by \text{pat-completeness auto}

**termination** by \text{(relation measure \((% (f, \, n, \, m). \text{nat-of-natural } (m + 1 - n))) \} \}
\text{(auto simp add: less-natural-def)}

Misc

**declare** Inf-set-fold [where \('a = 'a \text{ Predicate.pred}, \text{ code}\]
**declare** Sup-set-fold [where \('a = 'a \text{ Predicate.pred}, \text{ code}\]

**lemma** pred-of-set-fold-sup:
\[ \text{assumes finite } A \]
\[ \text{shows pred-of-set } A = \text{Finite-Set.fold } sup \, \text{bot } (\text{Predicate.single } \cdot ' A) \, (\text{is } \?lhs = \?rhs) \]
\text{proof \text{(rule sym)}}
\[ \text{interpret comp-fun-idem sup :: 'a \text{ Predicate.pred } \Rightarrow 'a \text{ Predicate.pred } \Rightarrow 'a \text{ Predicate.pred} \]
\text{by \text{(fact comp-fun-idem-sup)}}
\text{from \text{finite } A \text{ show } \?rhs = \?lhs \text{ by \text{(induct } A) \text{ (auto intro: pred-eqI))}}
THEORY “Lazy-Sequence”

qed

lemma pred-of-set-set-fold-sup:
  pred-of-set (set xs) = fold sup (List.map Predicate.single xs) bot
proof -
  interpret comp-fun-idem sup :: 'a Predicate.pred ⇒ 'a Predicate.pred ⇒ 'a Predicate.pred
    by (fact comp-fun-idem-sup)
  show ?thesis by (simp add: pred-of-set-fold-sup fold-set-fold [symmetric])
qed

lemma pred-of-set-set-foldr-sup [code]:
  pred-of-set (set xs) = foldr sup (List.map Predicate.single xs) bot
by (simp add: pred-of-set-set-fold-sup ac-simps foldr-fold fun-eq-iff)

no-notation
  bind (infixl ≫= 70)
hide-type (open) pred seq
hide-const (open) Pred eval single bind is-empty singleton if-pred not-pred holds
  Empty Insert Join Seq member pred-of-seq apply adjunct null the-only eq map the
iterate-upto
hide-fact (open) null-def member-def
end

76 Lazy sequences

theory Lazy-Sequence
imports Predicate
begin

76.1 Type of lazy sequences

datatype (plugins only: code extraction) (dead 'a) lazy-sequence =
  lazy-sequence-of-list 'a list
primrec list-of-lazy-sequence :: 'a lazy-sequence ⇒ 'a list
where
  list-of-lazy-sequence (lazy-sequence-of-list xs) = xs
lemma lazy-sequence-of-list-of-lazy-sequence [simp]:
  lazy-sequence-of-list (list-of-lazy-sequence xq) = xq
by (cases xq) simp-all
lemma lazy-sequence-eql:
  list-of-lazy-sequence xq = list-of-lazy-sequence yq ⇒ xq = yq
by (cases xq, cases yq) simp
lemma lazy-sequence-eq-iff:
$xq = yq \iff \text{list-of-lazy-sequence } xq = \text{list-of-lazy-sequence } yq$
by (auto intro: lazy-sequence-eqI)

lemma case-lazy-sequence [simp]:
case-lazy-sequence $f$ \ $xq = f$ (list-of-lazy-sequence \ $xq$
by (cases \ $xq$) auto

lemma rec-lazy-sequence [simp]:
rec-lazy-sequence $f$ \ $xq = f$ (list-of-lazy-sequence \ $xq$
by (cases \ $xq$) auto

definition Lazy-Sequence :: \ $(\text{unit} \Rightarrow \ (\ 'a \times \ 'a \text{ lazy-sequence}) \ \text{option}) \Rightarrow \ 'a \text{ lazy-sequence}$
where
Lazy-Sequence $f = \text{lazy-sequence-of-list } (\text{case } f () \ of$
| None $\Rightarrow []$
| Some ($x$, \ $xq$) $\Rightarrow x \# \text{list-of-lazy-sequence } xq$

code-datatype Lazy-Sequence

declare list-of-lazy-sequence.simps [code del]
declare lazy-sequence.case [code del]
declare lazy-sequence.rec [code del]

lemma list-of-Lazy-Sequence [simp]:
list-of-lazy-sequence (Lazy-Sequence \ $f$) = (case \ $f ()$ \ of
| None \ $\Rightarrow []$
| Some ($x$, \ $xq$) \ $\Rightarrow x \# \text{list-of-lazy-sequence } xq$
by (simp add: Lazy-Sequence-def)

definition yield :: \ 'a \text{ lazy-sequence} \Rightarrow (\ 'a \times \ 'a \text{ lazy-sequence}) \ \text{option}$
where
yield \ $xq = (\text{case } \text{list-of-lazy-sequence } xq \ of$
| [] \ $\Rightarrow \text{None}$
| $x \# \text{xs} \Rightarrow \text{Some } (x, \ \text{lazy-sequence-of-list } \text{xs})$)

lemma yield-Seq [simp, code]:
yield \ (Lazy-Sequence \ $f$) = \ $f ()$
by (cases \ $f ()$) (simp-all add: yield-def split-def)

lemma case-yield-eq [simp]: \ case-option \ $g \ \text{h} \ (\text{yield } xq) =$
case-list \ $g$ (\ $x$. \ curry \ $h \ \text{x} \circ \text{lazy-sequence-of-list}$) (list-of-lazy-sequence \ $xq$
by (cases list-of-lazy-sequence \ $xq$) (simp-all add: yield-def)

lemma equal-lazy-sequence-code [code]:
HOL.equal \ $xq \ yq = (\text{case } \text{yield } xq, \ \text{yield } yq ) \ \text{of}$
| (None, \ None) \ $\Rightarrow$ True
| (Some ($x$, \ $x'$), \ Some ($y$, \ $yq'$)) \ $\Rightarrow$ HOL.equal \ $x$ \ $y$ \ \text{and} \ \text{HOL.equal } xq \ yq$
| - \ $\Rightarrow$ False
by (simp-all add: lazy-sequence-eq-iff equal-eq split: list.splits)

lemma [code nbe]:
  HOL.equal (x :: 'a lazy-sequence) x ⟷ True
by (fact equal-refl)

definition empty :: 'a lazy-sequence
where
  empty = lazy-sequence-of-list []

lemma list-of-lazy-sequence-empty [simp]:
  list-of-lazy-sequence empty = []
by (simp add: empty-def)

lemma empty-code [code]:
  empty = Lazy-Sequence (λ-. None)
by (simp add: lazy-sequence-eq-iff)

definition single :: 'a ⇒ 'a lazy-sequence
where
  single x = lazy-sequence-of-list [x]

lemma list-of-lazy-sequence-single [simp]:
  list-of-lazy-sequence (single x) = [x]
by (simp add: single-def)

lemma single-code [code]:
  single x = Lazy-Sequence (λ. Some (x, empty))
by (simp add: lazy-sequence-eq-iff)

definition append :: 'a lazy-sequence ⇒ 'a lazy-sequence ⇒ 'a lazy-sequence
where
  append xq yq = lazy-sequence-of-list (list-of-lazy-sequence xq @ list-of-lazy-sequence yq)

lemma list-of-lazy-sequence-append [simp]:
  list-of-lazy-sequence (append xq yq) = list-of-lazy-sequence xq @ list-of-lazy-sequence yq
by (simp add: append-def)

lemma append-code [code]:
  append xq yq = Lazy-Sequence (λ. case yield xq of
  None ⇒ yield yq
| Some (x, xq') ⇒ Some (x, append xq' yq))
by (simp-all add: lazy-sequence-eq-iff split: list.splits)

definition map :: ('a ⇒ 'b) ⇒ 'a lazy-sequence ⇒ 'b lazy-sequence
where
  map f xq = lazy-sequence-of-list (List.map f (list-of-lazy-sequence xq))
lemma list-of-lazy-sequence-map [simp]:
list-of-lazy-sequence (map f xq) = List.map f (list-of-lazy-sequence xq)
by (simp add: map-def)

lemma map-code [code]:
map f xq =
Lazy-Sequence (λx. map-option (λx'. (f x, map f x')) (yield x))
by (simp-all add: lazy-sequence-eq-iff split: list.splits)

definition flat :: 'a lazy-sequence lazy-sequence ⇒ 'a lazy-sequence
where
flat xqq = lazy-sequence-of-list (concat (List.map list-of-lazy-sequence (list-of-lazy-sequence xqq)))

lemma list-of-lazy-sequence-flat [simp]:
list-of-lazy-sequence (flat xqq) = concat (List.map list-of-lazy-sequence (list-of-lazy-sequence xqq))
by (simp add: flat-def)

lemma flat-code [code]:
flat xqq = Lazy-Sequence (λx. case yield xqq of
  None ⇒ None
| Some (xq, xqq') ⇒ yield (append xq (flat xqq')))
by (simp add: lazy-sequence-eq-iff split: list.splits)

definition bind :: 'a lazy-sequence ⇒ ('a ⇒ 'b lazy-sequence) ⇒ 'b lazy-sequence
where
bind xq f = flat (map f xq)

definition if-seq :: bool ⇒ unit lazy-sequence
where
if-seq b = (if b then single () else empty)

definition those :: 'a option lazy-sequence ⇒ 'a lazy-sequence option
where
those xq = map-option lazy-sequence-of-list (List.those (list-of-lazy-sequence xq))

function iterate-upto :: (natural ⇒ 'a) ⇒ natural ⇒ natural ⇒ 'a lazy-sequence
where
iterate-upto f n m =
Lazy-Sequence (λx. if n > m then None else Some (f n, iterate-upto f (n + 1) m))
by pat-completeness auto

termination by (relation measure (λ(f, n, m). nat-of-natural (m + 1 - n)))
(auto simp add: less-natural-def)

definition not-seq :: unit lazy-sequence ⇒ unit lazy-sequence

THEORY "Lazy-Sequence"

where
not-seq xq = (case yield xq of
    None ⇒ single ()
  | Some ((() , xq)) ⇒ empty)

76.2 Code setup

code-reflect Lazy-Sequence
datatypes lazy-sequence = Lazy-Sequence

ML
signature LAZY-SEQUENCE =
sig
datatype 'a lazy-sequence = Lazy-Sequence of (unit -> ('a * 'a Lazy-Sequence.lazy-sequence) option)
val map: ('a -> 'b) -> 'a lazy-sequence -> 'b lazy-sequence
val yield: 'a lazy-sequence -> ('a * 'a lazy-sequence) option
val yieldn: int -> 'a lazy-sequence -> 'a list * 'a lazy-sequence
end;
structure Lazy-Sequence : LAZY-SEQUENCE =
struct
datatype lazy-sequence = datatype Lazy-Sequence.lazy-sequence;

fun map f = @
{code Lazy-Sequence.map} f;
fun yield P = @
{code Lazy-Sequence.yield} P;
fun yieldn k = Predicate.anamorph yield k;
end;

76.3 Generator Sequences

76.3.1 General lazy sequence operation
definition product :: 'a lazy-sequence ⇒ 'b lazy-sequence ⇒ ('a × 'b) lazy-sequence
where
product s1 s2 = bind s1 (λa. bind s2 (λb. single (a, b)))

76.3.2 Small lazy typeclasses
class small-lazy =
  fixes small-lazy :: natural ⇒ 'a lazy-sequence
instantiation unit :: small-lazy
begin
THEORY "Lazy-Sequence"

definition small-lazy d = single ()

instance ..

end

instantiation int :: small-lazy
begin
maybe optimise this expression -> append (single x) xs == cons x xs Performance difference?

function small-lazy' :: int ⇒ int ⇒ int lazy-sequence
where
  small-lazy' d i = (if d < i then empty
                      else append (single i) (small-lazy' d (i + 1)))
  by pat-completeness auto

termination
  by (relation measure (%(d, i). nat (d + 1 − i))) auto

definition small-lazy d = small-lazy' (int (nat-of-natural d)) (− (int (nat-of-natural d)))

instance ..

end

instantiation prod :: (small-lazy, small-lazy) small-lazy
begin

definition
  small-lazy d = product (small-lazy d) (small-lazy d)

instance ..

end

instantiation list :: (small-lazy) small-lazy
begin

fun small-lazy-list :: natural ⇒ 'a list lazy-sequence
where
  small-lazy-list d = append (single [])
  (if d > 0 then bind (product (small-lazy (d − 1))
                       (small-lazy (d − 1))) (λ(x, xs). single (x # xs)) else empty)

instance ..

end
76.4 With Hit Bound Value

assuming in negative context

**type-synonym** 'a hit-bound-lazy-sequence = 'a option lazy-sequence

**definition** hit-bound :: 'a hit-bound-lazy-sequence

where
hit-bound = Lazy-Sequence (λ-. Some (None, empty))

**lemma** list-of-lazy-sequence-hit-bound [simp]:
list-of-lazy-sequence hit-bound = [None]

by (simp add: hit-bound-def)

**definition** hb-single :: 'a ⇒ 'a hit-bound-lazy-sequence

where
hb-single x = Lazy-Sequence (λ-. Some ((Some x), empty))

**definition** hb-map :: ('a ⇒ 'b) ⇒ 'a hit-bound-lazy-sequence ⇒ 'b hit-bound-lazy-sequence

where
hb-map f xq = map (map-option f) xq

**lemma** hb-map-code [code]:
hb-map f xq = Lazy-Sequence (λ-. map-option (λ(x, xq). (map-option f x, hb-map f xq')) (yield xq))

using map-code [of map-option f xq] by (simp add: hb-map-def)

**definition** hb-flat :: 'a hit-bound-lazy-sequence hit-bound-lazy-sequence ⇒ 'a hit-bound-lazy-sequence

where
hb-flat xqq = lazy-sequence-of-list (concat (List.map ((Ax. case x of None ⇒ [None] | Some xs ⇒ xs) o map-option list-of-lazy-sequence) (list-of-lazy-sequence xqq)))

**lemma** list-of-lazy-sequence-hb-flat [simp]:
list-of-lazy-sequence (hb-flat xqq) = concat (List.map ((Ax. case x of None ⇒ [None] | Some xs ⇒ xs) o map-option list-of-lazy-sequence) (list-of-lazy-sequence xqq))

by (simp add: hb-flat-def)

**lemma** hb-flat-code [code]:
hb-flat xqq = Lazy-Sequence (λ-. case yield xqq of
  None ⇒ None
  | Some (xq, xqq') ⇒ yield (append (case xq of None ⇒ hit-bound | Some xq ⇒ xq) (hb-flat xqq'))) by (simp add: lazy-sequence-eq-iff split: list.splits option.splits)

**definition** hb-bind :: 'a hit-bound-lazy-sequence ⇒ ('a ⇒ 'b hit-bound-lazy-sequence) ⇒ 'b hit-bound-lazy-sequence

where


\[ \text{hb-bind } xq f = \text{hb-flat } (\text{hb-map } f xq) \]

**definition** \( \text{hb-if-seq} :: \text{bool } \Rightarrow \text{unit hit-bound-lazy-sequence} \)
where
\[ \text{hb-if-seq } b = (\text{if } b \text{ then } \text{hb-single } () \text{ else empty}) \]

**definition** \( \text{hb-not-seq} :: \text{unit hit-bound-lazy-sequence } \Rightarrow \text{unit lazy-sequence} \)
where
\[ \text{hb-not-seq } xq = (\text{case } \text{yield } xq \text{ of} \]
\[ \text{None } \Rightarrow \text{single } () \]
\[ \text{| Some } (x, xq) \Rightarrow \text{empty} \]

**hide-const** *(open)* \( \text{yield empty single append flat map bind if-seq those iterate-upto not-seq product} \)

**hide-fact** *(open)* \( \text{yield-def empty-def single-def append-def flat-def map-def bind-def if-seq-def those-def not-seq-def product-def} \)

end

77 Depth-Limited Sequences with failure element

theory Limited-Sequence
imports Lazy-Sequence
begin

77.1 Depth-Limited Sequence

type-synonym \( \alpha \text{ dseq} = \text{natural } \Rightarrow \text{bool } \Rightarrow \text{a lazy-sequence option} \)

**definition** \( \text{empty} :: \text{a dseq} \)
where
\[ \text{empty} = (\lambda - -. \text{Some Lazy-Sequence.empty}) \]

**definition** \( \text{single} :: \text{a } \Rightarrow \text{a dseq} \)
where
\[ \text{single } x = (\lambda - -. \text{Some } (\text{Lazy-Sequence.single } x)) \]

**definition** \( \text{eval} :: \text{a dseq } \Rightarrow \text{natural } \Rightarrow \text{bool } \Rightarrow \text{a lazy-sequence option} \)
where
\[ \text{[simp]}: \text{eval } f i \text{ pol } = f i \text{ pol} \]

**definition** \( \text{yield} :: \text{a dseq } \Rightarrow \text{natural } \Rightarrow \text{bool } \Rightarrow (\text{a } \times \text{a dseq }) \text{ option} \)
where
\[ \text{yield } f i \text{ pol } = (\text{case } \text{eval } f i \text{ pol of} \]
\[ \text{None } \Rightarrow \text{None} \]
\[ \text{| Some } s \Rightarrow (\text{map-option } \circ \text{apsnd}) (\lambda r - -. \text{Some } r) (\text{Lazy-Sequence.yield } s) \]

**definition** \( \text{map-seq} :: (\text{a } \Rightarrow \text{b dseq}) \Rightarrow \text{a lazy-sequence } \Rightarrow \text{b dseq} \)
where
\[ \text{map-seq } f \ xq i \ pol = \text{map-option Lazy-Sequence.flat} \]
\[ (\text{Lazy-Sequence.}\text{those (Lazy-Sequence.map (}\lambda x. f x \ i \ pol) \ xq)) \]

lemma map-seq-code [code]:
\[ \text{map-seq } f \ xq i \ pol = (\text{case Lazy-Sequence.yield } xq \text{ of} \]
\[ \text{None } \Rightarrow \text{Some Lazy-Sequence.empty} \]
\[ \text{Some } (x, xq') \Rightarrow (\text{case eval } (f x) \ i \ pol \text{ of} \]
\[ \text{None } \Rightarrow \text{None} \]
\[ \text{Some } yq \Rightarrow (\text{case map-seq } f \ xq' \ i \ pol \text{ of} \]
\[ \text{None } \Rightarrow \text{None} \]
\[ \text{Some } zq \Rightarrow \text{Some (Lazy-Sequence.append } yq zq))) \]
by (cases xq)
(auto simp add: map-seq-def Lazy-Sequence.those-def lazy-sequence-eq-iff split: list.splits simp add: option.splits)

definition bind :: \(' a \ dseq \Rightarrow ('a \Rightarrow 'b \ dseq) \Rightarrow 'b \ dseq'
where
bind x f = (\lambda i pol.
  if i = 0 then
    (if pol then Some Lazy-Sequence.empty else None)
  else
    (case x (i - 1) pol of
      None \Rightarrow \text{None} \]
    | Some xq \Rightarrow \text{map-seq } f xq i pol))

definition union :: \(' a \ dseq \Rightarrow 'a \ dseq \Rightarrow 'a \ dseq'
where
union x y = (\lambda i pol. \text{case } (x i pol, y i pol) \text{ of}
  (Some xq, Some yq) \Rightarrow \text{Some (Lazy-Sequence.append } xq yq)
  | - \Rightarrow \text{None})

definition if-seq :: bool \Rightarrow unit \ dseq
where
if-seq b = (if b then single () else empty)

definition not-seq :: unit \ dseq \Rightarrow unit \ dseq
where
not-seq x = (\lambda i pol. \text{case } x i (\neg pol) \text{ of}
  None \Rightarrow \text{Some Lazy-Sequence.empty} \]
  | Some xq \Rightarrow (\text{case Lazy-Sequence.yield } xq \text{ of}
    None \Rightarrow \text{Some (Lazy-Sequence.single ())} \]
  | Some - \Rightarrow \text{Some (Lazy-Sequence.empty())})

definition map :: ('a \Rightarrow 'b) \Rightarrow 'a \ dseq \Rightarrow 'b \ dseq
where
map f g = (\lambda i pol. \text{case } g i pol \text{ of}
  None \Rightarrow \text{None} \]
  | Some xq \Rightarrow \text{Some (Lazy-Sequence.map } f xq))
77.2 Positive Depth-Limited Sequence

type-synonym `'a pos-dseq = natural ⇒ 'a Lazy-Sequence.lazy-sequence

definition pos-empty :: `'a pos-dseq
  where
  pos-empty = (λi. Lazy-Sequence.empty)

definition pos-single :: `'a ⇒ 'a pos-dseq
  where
  pos-single x = (λi. Lazy-Sequence.single x)

definition pos-bind :: `'a pos-dseq ⇒ ('a ⇒ 'b pos-dseq) ⇒ 'b pos-dseq
  where
  pos-bind x f = (λi. Lazy-Sequence.bind (x i) (λa. f a i))

definition pos-decr-bind :: `'a pos-dseq ⇒ ('a ⇒ 'b pos-dseq) ⇒ 'b pos-dseq
  where
  pos-decr-bind x f = (λi.
    if i = 0 then
      Lazy-Sequence.empty
    else
      Lazy-Sequence.bind (x (i − 1)) (λa. f a i))

definition pos-union :: `'a pos-dseq ⇒ 'a pos-dseq ⇒ 'a pos-dseq
  where
  pos-union xq yq = (λi. Lazy-Sequence.append (xq i) (yq i))

definition pos-if-seq :: bool ⇒ unit pos-dseq
  where
  pos-if-seq b = (if b then pos-single () else pos-empty)

definition pos-iterate-upto :: (natural ⇒ 'a) ⇒ natural ⇒ natural ⇒ 'a pos-dseq
  where
  pos-iterate-upto f n m = (λi. Lazy-Sequence.iterate-upto f n m)

definition pos-map :: ('a ⇒ 'b) ⇒ 'a pos-dseq ⇒ 'b pos-dseq
  where
  pos-map f xq = (λi. Lazy-Sequence.map f (xq i))

77.3 Negative Depth-Limited Sequence

type-synonym `'a neg-dseq = natural ⇒ 'a Lazy-Sequence.hit-bound-lazy-sequence

definition neg-empty :: `'a neg-dseq
  where
  neg-empty = (λi. Lazy-Sequence.empty)

definition neg-single :: `'a ⇒ 'a neg-dseq
  where
THEORY "Limited-Sequence"

\( \text{neg-single } x = (\lambda i. \text{Lazy-Sequence.hb-single } x) \)

definition neg-bind :: 'a neg-dseq \( \Rightarrow \) ('a \( \Rightarrow \) 'b neg-dseq) \( \Rightarrow \) 'b neg-dseq
where
\( \text{neg-bind } x f = (\lambda i. \text{hb-bind } (x i) (\lambda a. f a i)) \)

definition neg-decr-bind :: 'a neg-dseq \( \Rightarrow \) ('a \( \Rightarrow \) 'b neg-dseq) \( \Rightarrow \) 'b neg-dseq
where
\( \text{neg-decr-bind } x f = (\lambda i. \text{if } i = 0 \text{ then Lazy-Sequence.hit-bound } \text{ else hb-bind } (x (i - 1)) (\lambda a. f a i)) \)

definition neg-union :: 'a neg-dseq \( \Rightarrow \) 'a neg-dseq \( \Rightarrow \) 'a neg-dseq
where
\( \text{neg-union } x y = (\lambda i. \text{Lazy-Sequence.append } (x i) (y i)) \)

definition neg-if-seq :: bool \( \Rightarrow \) unit neg-dseq
where
\( \text{neg-if-seq } b = (\text{if } b \text{ then neg-single } () \text{ else neg-empty}) \)

definition neg-iterate-upto
where
\( \text{neg-iterate-upto } f n m = (\lambda i. \text{Lazy-Sequence.iterate-upto } (\lambda i. \text{Some } (f i)) n m) \)

definition neg-map :: ('a \( \Rightarrow \) 'b) \( \Rightarrow \) 'a neg-dseq \( \Rightarrow \) 'b neg-dseq
where
\( \text{neg-map } f xq = (\lambda i. \text{Lazy-Sequence.hb-map } f (xq i)) \)

77.4 Negation

definition pos-not-seq :: unit neg-dseq \( \Rightarrow \) unit pos-dseq
where
\( \text{pos-not-seq } xq = (\lambda i. \text{Lazy-Sequence.hb-not-seq } (xq (3 \ast i))) \)

definition neg-not-seq :: unit pos-dseq \( \Rightarrow \) unit neg-dseq
where
\( \text{neg-not-seq } x = (\lambda i. \text{case Lazy-Sequence.yield } (x i) \text{ of None } \Rightarrow \text{Lazy-Sequence.hb-single } () | \text{Some } ((), xq) \Rightarrow \text{Lazy-Sequence.empty}) \)

ML
signature LIMITED-SEQUENCE =
sig
  type 'a dseq = Code-Numerical.natural \( \Rightarrow \) bool \( \Rightarrow \) 'a Lazy-Sequence.lazy-sequence
option
val map : ('a \( \Rightarrow \) 'b) \( \Rightarrow \) 'a dseq \( \Rightarrow \) 'b dseq
THEORY “Code-Evaluation”

val yield : 'a dseq -> Code-Numeral.natural -> bool -> ('a * 'a dseq) option
val yieldn : int -> 'a dseq -> Code-Numeral.natural -> bool -> 'a list * 'a dseq
end;

structure Limited-Sequence : LIMITED-SEQUENCE =
  struct
type 'a dseq = Code-Numeral.natural -> bool -> 'a Lazy-Sequence.lazy-sequence
  option
fun map f = @{code Limited-Sequence.map} f;
fun yield f = @{code Limited-Sequence.yield} f;
fun yieldn n f i pol = (case f i pol of
  NONE => ([], fn - => fn - => NONE)
  | SOME s => let val (xs, s') = Lazy-Sequence.yieldn n s in (xs, fn - => fn - => SOME s') end);
end;

hide-const (open) yield empty single eval map-seq bind union if-seq not-seq map
  pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-seq pos-iterate-upto
  pos-not-seq pos-map
  neg-empty neg-single neg-bind neg-decr bind neg-union neg-if-seq neg-iterate-upto
  neg-not-seq neg-map

hide-fact (open) yield-def empty-def single-def eval-def map-seq-def bind-def union-def
  if-seq-def not-seq-def map-def
  pos-not-seq-def pos-map-def
  neg-empty-def neg-single-def neg-bind-def neg-decr-def neg-union-def neg-if-seq-def neg-iterate-upto-def
  neg-not-seq-def neg-map-def
end

78 Term evaluation using the generic code generator

theory Code-Evaluation
imports Typerep Limited-Sequence
keywords value :: diag
begin
78.1  Term representation

78.1.1  Terms and class term-of

datatype (plugins only: extraction) term = dummy-term

definition Const :: String.literal ⇒ typerep ⇒ term where
  Const - - = dummy-term

definition App :: term ⇒ term ⇒ term where
  App - - = dummy-term

definition Abs :: String.literal ⇒ typerep ⇒ term ⇒ term where
  Abs - - - = dummy-term

definition Free :: String.literal ⇒ typerep ⇒ term where
  Free - - = dummy-term

code-datatype Const App Abs Free

class term-of = typerep +

fixes term-of :: 'a ⇒ term

lemma term-of-anything: term-of x ≡ t
  by (rule eq-reflection) (cases term-of x, cases t, simp)

definition valapp :: ('a ⇒ 'b) × (unit ⇒ term)
⇒ 'a × (unit ⇒ term) ⇒ 'b × (unit ⇒ term) where
  valapp f x = (fst f (fst x), λu. App (snd f ())) (snd x ()))

lemma valapp-code [code, code-unfold]:
  valapp (f, tf) (x, tx) = (f x, λu. App (tf ()) (tx ()))
  by (simp only: valapp-def fst-conv snd-conv)

78.1.2  Syntax

definition termify :: 'a ⇒ term where
  [code del]: termify x = dummy-term

abbreviation valtermify :: 'a ⇒ 'a × (unit ⇒ term) where
  valtermify x ≡ (x, λu. termify x)

bundle term-syntax
begin

notation App (infixl <·> 70)
  and valapp (infixl {·} 70)

end
78.2 Tools setup and evaluation

context
begin

qualified definition TERM-OF :: 'a::term-of itself
where
TERM-OF = snd (Code-Evaluation.term-of :: 'a ⇒ -, TYPE('a))

qualified definition TERM-OF-EQUAL :: 'a::term-of itself
where
TERM-OF-EQUAL = snd (λ(a::'a). (Code-Evaluation.term-of a, HOL.eq a), TYPE('a))

end

lemma eq-eq-TrueD:
fixes x y :: 'a::{}
assumes (x ≡ y) ≡ Trueprop True
shows x ≡ y
using assms by simp

code-printing
type-constructor term → (Eval) Term.term
| constant Const → (Eval) Term.Const/ ((-), (-))
| constant App → (Eval) Term.$/ ((-), (-))
| constant Abs → (Eval) Term.Abs/ ((-), (-), (-))
| constant Free → (Eval) Term.Free/ ((-), (-))

ML-file ⟨Tools/code-evaluation.ML⟩

code-reserved Eval Code-Evaluation

ML-file ⟨~/src/HOL/Tools/value-command.ML⟩

78.3 Dedicated term-of instances

instantiation fun :: (typerep, typerep) term-of
begin

definition term-of (f :: 'a ⇒ 'b) =
  Const (STR "Pure.dummy-pattern")
  (Typerep.Typerep (STR "fun") [Typerep.typerep TYPE('a), Typerep.typerep TYPE('b)])

instance ..
end
theory "Quickcheck-Random"

declare [[code drop: rec-term case-term
term-of :: typerep ⇒ - term-of :: term ⇒ - term-of :: String.literal ⇒ -
term-of :: - Predicate.pred ⇒ term term-of :: - Predicate.seq ⇒ term]]

code-printing
constant term-of :: integer ⇒ term ↦ (Eval) HOLogic.mk′-number/ HOLogic.code′-integerT
| constant term-of :: String.literal ⇒ term ⇒ (Eval) HOLogic.mk′-literal

declare [[code drop: term-of :: integer ⇒ -]]

lemma term-of-integer [unfolded typerep-fun-def typerep-num-def typerep-integer-def,
code]:
term-of (i :: integer) =
(if i > 0 then
  App (Const (STR "Num.numeral-class.numeral") (TYPEREPEP(num ⇒ integer)))
  (term-of (num-of-integer i))
else if i = 0 then Const (STR "Groups.zero-class.zero") TYPEREPEP(integer)
else
  App (Const (STR "Groups.uminus-class.uminus") TYPEREPEP(integer ⇒ integer))
  (term-of (− i))
by (rule term-of-anything [THEN meta-eq-to-obj-eq])

code-reserved Eval HOLogic

78.4 Generic reification
ML-file (~~/src/HOL/Tools/reification.ML)

78.5 Diagnostic
definition tracing :: String.literal ⇒ ’a ⇒ ’a where
  [code del]: tracing s x = x
code-printing
constant tracing :: String.literal ⇒ ’a ⇒ ’a ⇒ (Eval) Code′-Evaluation.tracing

hide-const dummy-term valapp
hide-const (open) Const App Abs Free termify valtermify term-of tracing

end

79 A simple counterexample generator performing random testing

theory Quickcheck-Random
imports Random Code-Evaluation Enum
begin

setup (Code-Target.add-derived-target (Quickcheck, [(Code-Runtime.target, I)]))

79.1 Catching Match exceptions

axiomatization catch-match :: 'a => 'a => 'a

code-printing
  constant catch-match -> (Quickcheck) ((-) handle Match => -)

code-reserved Quickcheck Match

79.2 The random class

class random = typerep +
  fixes random :: natural ⇒ Random.seed ⇒ ('a × (unit ⇒ term)) × Random.seed

79.3 Fundamental and numeric types

instantiation bool :: random
begin
  context
    includes state-combinator-syntax
  begin
    definition random i = Random.range 2 o→
      (λk. Pair (if k = 0 then Code-Evaluation.valtermify False else Code-Evaluation.valtermify True))
  end

instance ..
end

end

instantiation itself :: (typerep) random
begin
  definition random-itself :: natural ⇒ Random.seed ⇒ ('a itself × (unit ⇒ term)) × Random.seed
  where random-itself = Pair (Code-Evaluation.valtermify TYPE('a))
  instance ..
end
theory "Quickcheck-Random"

instantiation char :: random
begin

context
  includes state-combinator-syntax
begin

definition
  random = Random.select (Enum.enum :: char list) o→ (λc. Pair (c, λu. Code-Evaluation.term-of c))

instance ..
end

end

instantiation String.literal :: random
begin

definition
  random = Pair (STR "", λu. Code-Evaluation.term-of (STR ""))

instance ..
end

instantiation nat :: random
begin

context
  includes state-combinator-syntax
begin

definition random-nat :: natural ⇒ Random.seed ⇒ (nat × (unit ⇒ Code-Evaluation.term)) × Random.seed
where
  random-nat i = Random.range (i + 1) o→ (λk. Pair (let n = nat-of-natural k
                           in (n, λ_. Code-Evaluation.term-of n)))

instance ..
end

end

instantiation int :: random
begin

instance ..
end

end
context
  includes state-combinator-syntax
begin

definition
  random i = Random.range (2 * i + 1)  o→ (λk. Pair (let j = (if k ≥ i then int (nat-of-natural (k - i)) else - (int (nat-of-natural (i - k)))) in (j, λ-. Code-Evaluation.term-of j)))

instance ..
end

end

instantiation natural :: random
begin
  context
    includes state-combinator-syntax
begin

definition random-natural :: natural ⇒ Random.seed ⇒ (natural × (unit ⇒ Code-Evaluation.term)) × Random.seed
where
  random-natural i = Random.range (i + 1)  o→ (λn. Pair (n, λ-. Code-Evaluation.term-of n))

instance ..
end

end

instantiation integer :: random
begin
  context
    includes state-combinator-syntax
begin

definition random-integer :: natural ⇒ Random.seed ⇒ (integer × (unit ⇒ Code-Evaluation.term)) × Random.seed
where
  random-integer i = Random.range (2 * i + 1)  o→ (λk. Pair (let j = (if k ≥ i then integer-of-natural (k - i) else - (integer-of-natural (i - k))))
in (j, λ.- Code-Evaluation.term-of j)))

instance ..

end

end

79.4 Complex generators

Towards 'a ⇒ 'b

axiomatization random-fun-aux :: typerep ⇒ typerep ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ term) ⇒ (Random.seed ⇒ ('b × (unit ⇒ term)) × Random.seed) ⇒ (Random.seed ⇒ Random.seed × Random.seed) ⇒ Random.seed ⇒ (('a ⇒ 'b) × (unit ⇒ term)) × Random.seed

definition random-fun-lift :: (Random.seed ⇒ ('b × (unit ⇒ term)) × Random.seed) ⇒ Random.seed ⇒ ((('a::typerep ⇒ 'b::typerep) × (unit ⇒ term)) × Random.seed)

where
random-fun-lift f =
random-fun-aux TYPEREP('a) TYPEREP('b) (=) Code-Evaluation.term-of f

Random.split-seed

instantiation fun :: ({equal, term-of}, random) random
begin

definition random-fun :: natural ⇒ Random.seed ⇒ (('a ⇒ 'b) × (unit ⇒ term)) × Random.seed

where random i = random-fun-lift (random i)

instance ..

end

Towards type copies and datatypes

context

includes state-combinator-syntax

begin

definition collapse :: ('a ⇒ ('a ⇒ 'b × 'a) × 'a) ⇒ 'a ⇒ 'b × 'a

where collapse f = (f o→ id)

end

definition beyond :: natural ⇒ natural ⇒ natural

where beyond k l = (if l > k then l else 0)
lemma beyond-zero: beyond k 0 = 0
by (simp add: beyond-def)

context
includes term-syntax
begin

definition [code-unfold]:
valterm-emptyset = Code-Evaluation.valtermify ({}) :: ('a :: typerep) set

definition [code-unfold]:
valtermify-insert x s = Code-Evaluation.valtermify insert {} (x :: ('a :: typerep * -)) {} s

end

instantiation set :: (random) random
begin

context
includes state-combinator-syntax
begin

fun random-aux-set
where
random-aux-set 0 j = collapse (Random.select-weight [(1, Pair valterm-emptyset)])
| random-aux-set (Code-Numeral.Suc i) j =
collapse (Random.select-weight [(1, Pair valterm-emptyset),
(Code-Numeral.Suc i,
random j o→ (%x. random-aux-set i j o→ (%s. Pair (valtermify-insert x s))))])

lemma [code]:
random-aux-set i j =
collapse (Random.select-weight [(1, Pair valterm-emptyset),
(i, random j o→ (%x. random-aux-set (i - 1) j o→ (%s. Pair (valtermify-insert x s)))]))

proof (induct i rule: natural.induct)
case zero
show ?case by (subst select-weight-drop-zero [symmetric])
(simp add: random-aux-set.simps [simplified] less-natural-def)

next
case (Suc i)
show ?case by (simp only: random-aux-set.simps(2) [of i] Suc-natural-minus-one)
qed

definition random-set i = random-aux-set i i
instance ..

end

end

lemma random-aux-rec:
  fixes random-aux :: natural ⇒ 'a
  assumes random-aux 0 = rhs 0
  and \( \forall k. \) random-aux (Code-Numeral.Suc k) = rhs (Code-Numeral.Suc k)
  shows random-aux k = rhs k
  using assms by (rule natural.induct)

79.5 Deriving random generators for datatypes

ML-file ⟨Tools/Quickcheck/quickcheck-common.ML⟩
ML-file ⟨Tools/Quickcheck/random-generators.ML⟩

79.6 Code setup

code-printing
  constant random-fun-aux → (Quickcheck) Random'-Generators.random'-fun
  — With enough criminal energy this can be abused to derive False; for this reason we use a distinguished target Quickcheck not spoiling the regular trusted code generation

code-reserved Quickcheck Random-Generators

hide-const (open) catch-match random collapse beyond random-fun-aux random-fun-lift

hide-fact (open) collapse-def beyond-def random-fun-lift-def

end

80 The Random-Predicate Monad

theory Random-Pred
imports Quickcheck-Random
begin

fun iter' :: 'a itself ⇒ natural ⇒ natural ⇒ Random.seed ⇒ ('a::random) Predicate.pred
  where
    iter' T nrandom sz seed = (if nrandom = 0 then bot-class.bot else
      let ((x, -), seed') = Quickcheck-Random.random sz seed
      in Predicate.Seq (%(u. Predicate.Insert x (iter' T (nrandom - 1) sz seed')))

definition iter :: natural ⇒ natural ⇒ Random.seed ⇒ ('a::random) Predicate.pred
where
iter nrandom sz seed = iter' (TYPE('a)) nrandom sz seed

lemma [code]:
iter nrandom sz seed = (if nrandom = 0 then bot-class.bot else
    let ((x, -), seed') = Quickcheck-Random.random sz seed
    in Predicate.Seq (%u. Predicate.Insert x (iter (nrandom - 1) sz seed')))

unfolding iter-def iter'.simps [of - nrandom] ..

type-synonym 'a random-pred = Random.seed ⇒ ('a Predicate.pred × Random.seed)

definition empty :: 'a random-pred
    where empty = Pair bot

definition single :: 'a => 'a random-pred
    where single x = Pair (Predicate.single x)

definition bind :: 'a random-pred ⇒ ('a ⇒ 'b random-pred) ⇒ 'b random-pred
    where
    bind R f = (λs. let
        (P, s') = R s;
        (s1, s2) = Random.split-seed s'
        in (Predicate.bind P (%a. fst (f a s1)), s2))

definition union :: 'a random-pred ⇒ 'a random-pred ⇒ 'a random-pred
    where
    union R1 R2 = (λs. let
        (P1, s') = R1 s; (P2, s'') = R2 s'
        in (sup-class.sup P1 P2, s''))

definition if-randompred :: bool ⇒ unit random-pred
    where
    if-randompred b = (if b then single () else empty)

definition iterate-upto :: (natural ⇒ 'a) ⇒ natural ⇒ natural ⇒ 'a random-pred
    where
    iterate-upto f n m = Pair (Predicate.iterate-upto f n m)

definition not-randompred :: unit random-pred ⇒ unit random-pred
    where
    not-randompred P = (λs. let
        (P', s') = P s
        in if Predicate.eval P' () then (Orderings.bot, s') else (Predicate.single (), s'))

definition Random :: (Random.seed ⇒ ('a × (unit ⇒ term)) × Random.seed) ⇒ 'a random-pred
    where Random g = scomp g (Pair ◦ (Predicate.single ◦ fst))
THEORY "Random-Sequence"

\[
\text{definition } \text{map} :: (\mathcal{A} \to \mathcal{B}) \Rightarrow \mathcal{A} \text{ random-pred} \Rightarrow \mathcal{B} \text{ random-pred}
\]
\[
\text{where } \text{map } f \text{ P} = \text{bind } \text{P} (\text{single } \circ f)
\]

\[
\text{hide-const (open) } \text{iter}\' \text{ iter empty single bind union if-randompred}
\]
\[
\text{iterate-upto not-randompred Random map}
\]

\[
\text{hide-fact (open) } \text{iter-def empty-def single-def bind-def union-def}
\]
\[
\text{if-randompred-def iterate-upto-def not-randompred-def Random-def map-def}
\]

end

81 Various kind of sequences inside the random monad

theory Random-Sequence
imports Random-Pred
begin

\[
\text{type-synonym } \mathcal{A} \text{ random-dseq} = \text{natural} \Rightarrow \text{natural} \Rightarrow \text{Random.seed} \Rightarrow (\mathcal{A} \text{ Limited-Sequence.dseq} \times \text{Random.seed})
\]

\[
\text{definition empty :: } \mathcal{A} \text{ random-dseq}
\]
\[
\text{where}
\]
\[
\text{empty} = (\%\text{nrandom size. Pair (Limited-Sequence.empty)})
\]

\[
\text{definition single :: } \mathcal{A} \Rightarrow \mathcal{A} \text{ random-dseq}
\]
\[
\text{where}
\]
\[
\text{single } x = (\%\text{nrandom size. Pair (Limited-Sequence.single } x))
\]

\[
\text{definition bind :: } \mathcal{A} \text{ random-dseq} \Rightarrow (\mathcal{A} \Rightarrow \mathcal{B} \text{ random-dseq}) \Rightarrow \mathcal{B} \text{ random-dseq}
\]
\[
\text{where}
\]
\[
\text{bind } R f = (\lambda\text{nrandom size s. let}
\]
\[
(P, s') = R \text{nrandom size s;}
\]
\[
(s1, s2) = \text{Random.split-seed } s'
\]
\[
in (\text{Limited-Sequence.bind } P (\%a. \text{fst } (f a \text{nrandom size } s1)), s2))
\]

\[
\text{definition union :: } \mathcal{A} \text{ random-dseq} \Rightarrow \mathcal{A} \text{ random-dseq} \Rightarrow \mathcal{A} \text{ random-dseq}
\]
\[
\text{where}
\]
\[
\text{union } R1 R2 = (\lambda\text{nrandom size s. let}
\]
\[
(S1, s') = R1 \text{nrandom size s; (S2, s'') = R2 nrandom size s'}
\]
\[
in (\text{Limited-Sequence.union } S1 S2, s''))
\]

\[
\text{definition if-random-dseq :: bool } \Rightarrow \text{unit random-dseq}
\]
\[
\text{where}
\]
\[
\text{if-random-dseq } b = (\text{if } b \text{ then single } () \text{ else empty})
\]
THEORY "Random-Sequence"

**definition** not-random-dseq :: unit random-dseq => unit random-dseq

**where**

not-random-dseq R = (λnrandom size s. let
(S, s') = R nrandom size s
in (Limited-Sequence.not-seq S, s'))

**definition** map :: ('a => 'b) => 'a random-dseq => 'b random-dseq

**where**

map f P = bind P (single o f)

**fun** Random :: (natural => Random.seed => (('a x (unit => term)) x Random.seed)) => 'a random-dseq

**where**

Random g nrandom = (%size. if nrandom <= 0 then (Pair Limited-Sequence.empty)
else (scomp (g size) (%r. scomp (Random g (nrandom - 1) size) (%rs. Pair
(Limited-Sequence.union (Limited-Sequence.single (fst r)) rs)))))

**type-synonym** 'a pos-random-dseq = natural => natural => Random.seed => 'a
Limited-Sequence.pos-dseq

**definition** pos-empty :: 'a pos-random-dseq

**where**

pos-empty = (%nrandom size seed. Limited-Sequence.pos-empty)

**definition** pos-single :: 'a => 'a pos-random-dseq

**where**

pos-single x = (%nrandom size seed. Limited-Sequence.pos-single x)

**definition** pos-bind :: 'a pos-random-dseq => ('a => 'b pos-random-dseq) => 'b
pos-random-dseq

**where**

pos-bind R f = (λnrandom size seed. Limited-Sequence.pos-bind (R nrandom size
seed) (%a. f a nrandom size seed))

**definition** pos-decr-bind :: 'a pos-random-dseq => ('a => 'b pos-random-dseq) =>
'b pos-random-dseq

**where**

pos-decr-bind R f = (λnrandom size seed. Limited-Sequence.pos-decr-bind (R
nrandom size seed) (%a. f a nrandom size seed))

**definition** pos-union :: 'a pos-random-dseq => 'a pos-random-dseq => 'a pos-random-dseq

**where**

pos-union R1 R2 = (λnrandom size seed. Limited-Sequence.pos-union (R1 nrandom
size seed) (R2 nrandom size seed))

**definition** pos-if-random-dseq :: bool => unit pos-random-dseq

**where**
pos-if-random-dseq \( b = (\text{if } b \text{ then } \text{pos-single } () \text{ else } \text{pos-empty}) \)

**definition** pos-iterate-upto :: (natural => 'a) => natural => natural => 'a pos-random-dseq

**where**

\( \text{pos-iterate-upto } f \ n \ m = (\lambda \text{random size seed. Limited-Sequence.pos-iterate-upto } f \ n \ m) \)

**definition** pos-map :: ('a => 'b) => 'a pos-random-dseq => 'b pos-random-dseq

**where**

\( \text{pos-map } f \ P = \text{pos-bind } P \ (\text{pos-single } \circ f) \)

**fun** iter :: (Random.seed => ('a x (unit => term)) x Random.seed) => natural => Random.seed => 'a Lazy-Sequence.lazy-sequence

**where**

\( \text{iter random nrandom seed} = \)

(\( (\text{if } \text{nrandom} = 0 \text{ then Lazy-Sequence.empty else Lazy-Sequence.Lazy-Sequence}
((\%u. \text{let } ((x, -), \text{seed'}) = \text{random seed in Some } (x, \text{iter random } (\text{nrandom} - 1) \text{ seed'}))) \))

**definition** pos-Random :: (natural => Random.seed => ('a x (unit => term)) x Random.seed) => 'a pos-random-dseq

**where**

\( \text{pos-Random } g = (\%nrandom size seed depth. \text{iter } (g \text{ size}) \text{nrandom seed}) \)

**type-synonym** 'a neg-random-dseq = natural => natural => Random.seed => 'a Limited-Sequence.neg-dseq

**definition** neg-empty :: 'a neg-random-dseq

**where**

\( \text{neg-empty} = (\%nrandom size seed. Limited-Sequence.neg-empty) \)

**definition** neg-single :: 'a => 'a neg-random-dseq

**where**

\( \text{neg-single } x = (\%nrandom size seed. Limited-Sequence.neg-single } x) \)

**definition** neg-bind :: 'a neg-random-dseq => ('a => 'b neg-random-dseq) => 'b neg-random-dseq

**where**

\( \text{neg-bind } R \ f = (\lambda \text{nrandom size seed. Limited-Sequence.neg-bind } (R \text{nrandom size seed}) (\%a. \ f \ a \text{nrandom size seed})) \)

**definition** neg-decr-bind :: 'a neg-random-dseq => ('a => 'b neg-random-dseq) => 'b neg-random-dseq

**where**

\( \text{neg-decr-bind } R \ f = (\lambda \text{nrandom size seed. Limited-Sequence.neg-decr-bind } (R \text{nrandom size seed}) (\%a. \ f \ a \text{nrandom size seed})) \)
definition neg-union :: 'a neg-random-dseq => 'a neg-random-dseq => 'a neg-random-dseq
where
  neg-union R1 R2 = (λnrandom size seed. Limited-Sequence.neg-union (R1 nrandom size seed) (R2 nrandom size seed))

definition neg-if-random-dseq :: bool => unit neg-random-dseq
where
  neg-if-random-dseq b = (if b then neg-single () else neg-empty)

definition neg-iterate-upto :: natural => natural => natural => 'a neg-random-dseq
where
  neg-iterate-upto f n m = (λnrandom size seed. Limited-Sequence.neg-iterate-upto f n m)

definition neg-not-random-dseq :: unit pos-random-dseq => unit neg-random-dseq
where
  neg-not-random-dseq R = (λnrandom size seed. Limited-Sequence.neg-not-seq (R nrandom size seed))

definition neg-map :: ('a => 'b) => 'a neg-random-dseq => 'b neg-random-dseq
where
  neg-map f P = neg-bind P (neg-single ◦ f)

definition pos-not-random-dseq :: unit neg-random-dseq => unit pos-random-dseq
where
  pos-not-random-dseq R = (λnrandom size seed. Limited-Sequence.pos-not-seq (R nrandom size seed))

hide-const (open)
empty single bind union if-random-dseq not-random-dseq map Random
pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-random-dseq pos-iterate-upto
pos-not-random-dseq pos-map iter pos-Random
neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-random-dseq neg-iterate-upto
neg-not-random-dseq neg-map

hide-fact (open) empty-def single-def bind-def union-def if-random-dseq-def not-random-dseq-def
map-def Random.simps
pos-iterate-upto-def pos-not-random-dseq-def pos-map-def iter.simps pos-Random-def
neg-empty-def neg-single-def neg-bind-def neg-decr-bind-def neg-union-def neg-if-random-dseq-def
neg-iterate-upto-def neg-not-random-dseq-def neg-map-def

end
A simple counterexample generator performing exhaustive testing

theory Quickcheck-Exhaustive
imports Quickcheck-Random
keywords quickcheck-generator ::thy-decl
begin

82.1 Basic operations for exhaustive generators

definition orelse :: 'a option ⇒ 'a option ⇒ 'a option (infixr orelse 55)
  where [code-unfold]: x orelse y = (case x of Some x' ⇒ Some x' | None ⇒ y)

82.2 Exhaustive generator type classes

class exhaustive = term-of +
  fixes exhaustive :: ('a ⇒ (bool × term list) option) ⇒ natural ⇒ (bool × term list) option

class full-exhaustive = term-of +
  fixes full-exhaustive ::
    ('a × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ natural ⇒ (bool × term list) option

instantiation natural :: full-exhaustive
begin

function full-exhaustive-natural' ::
  (natural × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
  natural ⇒ natural ⇒ (bool × term list) option
  where full-exhaustive-natural' f d i =
    (if d < i then None
     else (f (i, λ-. Code-Evaluation.term-of i)) orelse (full-exhaustive-natural' f d (i + 1)))
  by pat-completeness auto

termination
  by (relation measure (λ(-, d, i). nat-of-natural (d + 1 - i))) (auto simp add: less-natural-def)

definition full-exhaustive f d = full-exhaustive-natural' f d 0

instance ..
end

instantiation natural :: exhaustive
begin
function exhaustive-natural' ::
  (natural ⇒ (bool × term list) option) ⇒ natural ⇒ natural ⇒ (bool × term list) option
where exhaustive-natural' f d i =
  (if d < i then None
   else (f i orelse exhaustive-natural' f d (i + 1)))
by pat-completeness auto

termination
  by (relation measure (λ(-, d, i). nat-of-natural (d + 1 − i))) (auto simp add: less-natural-def)

definition exhaustive f d = exhaustive-natural' f d 0

instance ..
end

instantiation integer :: exhaustive
begin

function exhaustive-integer' ::
  (integer ⇒ (bool × term list) option) ⇒ integer ⇒ integer ⇒ (bool × term list) option
where exhaustive-integer' f d i =
  (if d < i then None else (f i orelse exhaustive-integer' f d (i + 1)))
by pat-completeness auto

termination
  by (relation measure (λ(-, d, i). nat-of-integer (d + 1 − i)))
    (auto simp add: less-integer-def nat-of-integer-def)

definition exhaustive f d = exhaustive-integer' f (integer-of-natural d) (− (integer-of-natural d))

instance ..
end

instantiation integer :: full-exhaustive
begin

function full-exhaustive-integer' ::
  (integer × (unit ⇒ term)) ⇒ (bool × term list) option ⇒
  integer ⇒ integer ⇒ (bool × term list) option
where full-exhaustive-integer' f d i =
  (if d < i then None
   else
     (case f (i, λ-. Code-Evaluation.term-of i) of
Some t ⇒ Some t
| None ⇒ full-exhaustive-integer' f d (i + 1))
by pat-completeness auto

termination
by (relation measure (λ(_, d, i). nat-of-integer (d + 1 − i)))
(auto simp add: less-integer-def nat-of-integer-def)

definition full-exhaustive f d =
full-exhaustive-integer' f (integer-of-natural d) (− (integer-of-natural d))

instance ..
end

instantiation nat :: exhaustive
begin

definition exhaustive f d = exhaustive (λx. f (nat-of-natural x)) d

instance ..
end

instantiation nat :: full-exhaustive
begin

definition full-exhaustive f d =
full-exhaustive (λ(x, xt). f (nat-of-natural x, λ-. Code-Evaluation.term-of (nat-of-natural x))) d

instance ..
end

instantiation int :: exhaustive
begin

function exhaustive-int' ::
(int ⇒ (bool × term list) option) ⇒ int ⇒ int ⇒ (bool × term list) option
where exhaustive-int' f d i =
(if d < i then None else (f i orelse exhaustive-int' f d (i + 1)))
by pat-completeness auto

termination
by (relation measure (λ(_, d, i). nat (d + 1 − i))) auto

definition exhaustive f d =
exhaustive-int' f (int-of-integer (integer-of-natural d))
{− (int-of-integer (integer-of-natural d))}

instance ..

end

instantiation int :: full-exhaustive

begin

function full-exhaustive-int' ::
  (int × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
   int ⇒ int ⇒ (bool × term list) option
where full-exhaustive-int' f d i =
  if d < i then None
  else
    (case f (i, λ - . Code-Evaluation.term-of i) of
        Some t ⇒ Some t
      | None ⇒ full-exhaustive-int' f d (i + 1))
by pat-completeness auto

termination
  by (relation measure (λ(−, d, i). nat (d + 1 − i))) auto

definition full-exhaustive f d =
  full-exhaustive-int' f (int-of-integer (integer-of-natural d))
  {− (int-of-integer (integer-of-natural d))}

instance ..

end

instantiation prod :: (exhaustive, exhaustive) exhaustive

begin

definition exhaustive f d = exhaustive (λx. exhaustive (λy. f ((x, y))) d) d

instance ..

end

context
  includes term-syntax

begin

definition [code-unfold]: valtermify-pair x y =
  Code-Evaluation.valtermify (Pair :: 'a::typerep ⇒ 'b::typerep ⇒ 'a × 'b) {·} x
  {·} y
THEORY "Quickcheck-Exhaustive"

end

instantiation prod :: (full-exhaustive, full-exhaustive) full-exhaustive
begin

definition full-exhaustive f d =
  full-exhaustive (λx. full-exhaustive (λy. f (valtermify-pair x y)) d) d
instance ..
end

instantiation set :: (exhaustive) exhaustive
begin
fun exhaustive-set
where
  exhaustive-set f i =
  (if i = 0 then None else
   f {} orelse
   exhaustive-set
     (λA. f A orelse exhaustive (λx. if x ∈ A then None else f (insert x A)) (i − 1)) (i − 1))
instance ..
end

instantiation set :: (full-exhaustive) full-exhaustive
begin
fun full-exhaustive-set
where
  full-exhaustive-set f i =
  (if i = 0 then None else
   f valterm-emptyset orelse
   full-exhaustive-set
     (λA. f A orelse Quickcheck-Exhaustive.full-exhaustive
      (λx. if fst x ∈ fst A then None else f (valtermify-insert x A)) (i − 1)) (i − 1))
instance ..
end

instantiation fun :: ({equal,exhaustive}, exhaustive) exhaustive
begin
fun exhaustive-fun' ::
    (('a ⇒ 'b) ⇒ (bool × term list) option) ⇒ natural ⇒ natural ⇒ (bool × term list) option
where
  exhaustive-fun' f i d =
  (exhaustive (λb. f (λ-. b)) d) orelse
  (if i > 1 then
   exhaustive-fun'
     (λg. exhaustive (λa. exhaustive (λb. f (g(a := b))) d) d) (i - 1) d else
    None)

definition exhaustive-fun ::
    (('a ⇒ 'b) ⇒ (bool × term list) option) ⇒ natural ⇒ (bool × term list) option
where exhaustive-fun f d = exhaustive-fun' f d d

instance ..

end

definition [code-unfold]:
  valtermify-absdummy =
    (λ(v, t).
     (λ::'a. v,
      λ::unit. Code-Evaluation.Abs (STR "x") (Typerep.typerep TYPE('a::typerep))
      (t ())))

context
    includes term-syntax
begin

definition [code-unfold]:
  valtermify-fun-upd g a b =
    Code-Evaluation.valtermify
    (fun-upd :: ('a::typerep ⇒ 'b::typerep) ⇒ 'a ⇒ 'b ⇒ 'a ⇒ 'b) {·} g {·} a {·} b

end

instantiation fun :: ({equal,full-exhaustive}, full-exhaustive) full-exhaustive
begin

fun full-exhaustive-fun' ::
    (('a ⇒ 'b) × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
    natural ⇒ natural ⇒ (bool × term list) option
where
  full-exhaustive-fun' f i d =
    full-exhaustive (λv. f (valtermify-absdummy v)) d orelse
    (if i > 1 then
     full-exhaustive-fun'

THEORY “Quickcheck-Exhaustive”

\[ \lambda g. \text{full-exhaustive} (\lambda a. \text{full-exhaustive} (\lambda b. f (\text{valtermify-fun-upd} g a b)) d) d) (i - 1) d \text{ else None} \]

definition full-exhaustive-fun ::

\[ (('a \Rightarrow 'b) \times (\text{unit} \Rightarrow \text{term}) \Rightarrow (\text{bool} \times \text{term list}) \Rightarrow \text{natural} \Rightarrow (\text{bool} \times \text{term list}) \Rightarrow \text{option} \Rightarrow \text{option}) \Rightarrow \text{option} \]

where full-exhaustive-fun f d = full-exhaustive-fun' f d d

instance ..

end

82.2.1 A smarter enumeration scheme for functions over finite datatypes

class check-all = enum + term-of +

fixes check-all :: ('a \times (\text{unit} \Rightarrow \text{term}) \Rightarrow (\text{bool} \times \text{term list}) \Rightarrow \text{option}) \Rightarrow (\text{bool} \times \text{term list}) \Rightarrow \text{option}

fixes enum-term-of :: 'a itself \Rightarrow \text{unit} \Rightarrow \text{term list}

fun check-all-n-lists :: ('a::check-all list \times (\text{unit} \Rightarrow \text{term list}) \Rightarrow (\text{bool} \times \text{term list}) \Rightarrow \text{natural} \Rightarrow (\text{bool} \times \text{term list}) \Rightarrow \text{option}) \Rightarrow \text{option}

where check-all-n-lists f n =

\( (\text{if } n = 0 \text{ then } f ([], (\lambda _. [])) \text{ else } \text{check-all} (\lambda(x, xt). \text{check-all-n-lists} (\lambda(xs, xst). f ((x \# xs), (\lambda-. (xt () \# xst ()))))) (n - 1))) \)

context

includes term-syntax

begin

definition |code-unfold|:: termify-fun-upd g a b =

\( \text{Code-Evaluation.termify} \)

\( (\text{fun-upd} :: ('a::typerep \Rightarrow 'b::typerep) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b) <:: g <:: a <:: b) \)

end

definition mk-map-term ::

\( (\text{unit} \Rightarrow \text{typerep}) \Rightarrow (\text{unit} \Rightarrow \text{typerep}) \Rightarrow (\text{unit} \Rightarrow \text{term list}) \Rightarrow (\text{unit} \Rightarrow \text{term list}) \Rightarrow \text{unit} \Rightarrow \text{term} \)

where mk-map-term T1 T2 domm rng =

\( (\lambda-.\) let

\( T1 = T1 (); \)

\( T2 = T2 (); \)
update-term =
  (λg (a, b).
     (Code-Evaluation.Const (STR "fun.fun-upd")
       (Typerep.Typerep (STR "fun") [Typerep.Typerep (STR "fun") [T1, T2],
         Typerep.Typerep (STR "fun") [T1, Typerep.Typerep (STR "fun") [T2, Typerep.Typerep (STR "fun")]]]
         [T1, T2]])))
   g) a) b)
  in
List.foldl update-term
  (Code-Evaluation.Abs (STR "$x")) T1
  (Code-Evaluation.Const (STR "HOL.undefined") T2)) (zip (domm ()))
)

instantiation fun :: {equal,check-all}, check-all check-all begin

definition check-all f =
  (let
    mk-term =
      mk-map-term
        (λ-. Typerep.typerep (TYPE('a)))
        (λ-. Typerep.typerep (TYPE('b')))
        (enum-term-of (TYPE('a')));
    enum = (Enum.enum :: 'a list)
  in
    check-all-n-lists
      (λ(ys, yst). f (the ◦ map-of (zip enum ys), mk-term yst))
      (natural-of-nat (length enum)))

definition enum-term-of-fun :: ('a ⇒ 'b itself ⇒ unit ⇒ term list
  where enum-term-of-fun =
    (λ- -. enum-term-of-a = enum-term-of (TYPE('a));
     mk-term =
       mk-map-term
         (λ-. Typerep.typerep (TYPE('a')))
         (λ-. Typerep.typerep (TYPE('b')))
     enum-term-of-a
     in
     map (λys. mk-term (λ-. ys () )
       (List.n-lists (length (enum-term-of-a ())) (enum-term-of (TYPE('b')) ())))

instance ..
THEORY “Quickcheck-Exhaustive”

end

context
  includes term-syntax
begin

fun check-all-subsets ::
  (((′a::typerep) set × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
  (′a × (unit ⇒ term)) list ⇒ (bool × term list) option
where
check-all-subsets f [] = f valterm-emptyset
| check-all-subsets f (x # xs) =
  check-all-subsets (λs. case f s of Some ts ⇒ Some ts | None ⇒ f (valtermify-insert x s)) xs

definition [code-unfold]:
  term-emptyset = Code-Evaluation.termify ({} :: (′a::typerep) set)

definition [code-unfold]:
  termify-insert x s =
  Code-Evaluation.termify (insert :: (′a::typerep) ⇒ ′a set ⇒ ′a set) <·> x <·> s

definition setify :: (′a::typerep) itself ⇒ term list ⇒ term
where
setify T ts = foldr (termify-insert T) ts (term-emptyset T)

end

instantiation set :: (check-all) check-all
begin

definition check-all-set f =
  check-all-subsets f
  (zip (Enum.enum :: ′a list)
    (map (λa. λu :: unit. a) (Quickcheck-Exhaustive.enum-term-of (TYPE ′a)) ()))))

definition enum-term-of-set :: ′a set itself ⇒ unit ⇒ term list
where enum-term-of-set - - =
  map (setify (TYPE(′a))) (subseqs (Quickcheck-Exhaustive.enum-term-of (TYPE(′a))) ()))

instance ..

end

instantiation unit :: check-all
begin

**definition** check-all \( f \) = \( f \) (Code-Evaluation.valtermify ())

**definition** enum-term-of-unit :: unit itself \( \Rightarrow \) unit \( \Rightarrow \) term list
  **where**
  enum-term-of-unit = \( (\lambda \_ \_. \text{Code-Evaluation.term-of (})) \)

instance ..
end

instantiation bool :: check-all
begin

**definition**
  check-all \( f \) =
  (case \( f \) (Code-Evaluation.valtermify False) of
    Some \( x' \) \( \Rightarrow \) Some \( x' \)
    | None \( \Rightarrow \) \( f \) (Code-Evaluation.valtermify True))

**definition** enum-term-of-bool :: bool itself \( \Rightarrow \) unit \( \Rightarrow \) term list
  **where**
  enum-term-of-bool = \( (\lambda \_ \_. \text{map Code-Evaluation.term-of (Enum.enum :: bool list)}) \)

instance ..
end

context
  includes term-syntax
begin

**definition** [code-unfold]:
  termify-pair \( x \) \( y \) =
  Code-Evaluation.termify (Pair :: \('a::typerep \Rightarrow \'b :: typerep \Rightarrow \'a * \'b\) \( <> \) \( x \) <> \( y \))

end

instantiation prod :: (check-all, check-all) check-all
begin

**definition** check-all \( f \) = check-all (\( \lambda x \). check-all (\( \lambda y \). \( \text{valtermify-pair} \) \( x \) \( y \))))

**definition** enum-term-of-prod :: ('(a * 'b) itself \( \Rightarrow \) unit \( \Rightarrow \) term list
  **where**
  enum-term-of-prod =
  (\( \lambda \_ \_. \text{map} \) \( \lambda (x, y) \). \( \text{termify-pair} \) TYPE('a) TYPE('b) \( x \) \( y \))
THEORY "Quickcheck-Exhaustive"

(List.product (enum-term-of (TYPE('a))) (enum-term-of (TYPE('b))) ()))

instance ..

end

context
  includes term-syntax
begin

definition [code-unfold]: valtermify-Inl x = Code-Evaluation.valtermify (Inl :: 'a::typerep ⇒ 'a + 'b :: typerep) \{\} x

definition [code-unfold]: valtermify-Inr x = Code-Evaluation.valtermify (Inr :: 'b::typerep ⇒ 'a::typerep + 'b) \{\} x

end

instantiation sum :: (check-all, check-all) check-all
begin

definition check-all f = check-all (λa. f (valtermify-Inl a)) orelse check-all (λb. f (valtermify-Inr b))

definition enum-term-of-sum :: ('a + 'b) itself ⇒ unit ⇒ term list
  where enum-term-of-sum =
    (λ· ·).
    let
      T1 = Typerep.tpcrep (TYPE('a));
      T2 = Typerep.tpcrep (TYPE('b))
    in
    map
      (Tycrep.Tycrep (STR "fun") [T1, Tycrep.Tycrep (STR "Sum-Type.sum")
        ((enum-term-of (TYPE('a))) ())) @
      map
      (Code-Evaluation.App (Code-Evaluation.Const (STR "Sum-Type.Inr")
        (Tycrep.Tycrep (STR "fun") [T2, Tycrep.Tycrep (STR "Sum-Type.sum")
          ((enum-term-of (TYPE('b))) ())))

instance ..

end
instantiation char :: check-all
begin

primrec check-all-char':
  (char × (unit ⇒ term)) ⇒ (bool × term list) ⇒ char list ⇒ (bool × term list) option
  where check-all-char' f [] = None
  | check-all-char' f (c ≠ cs) = f (c, λ.-. Code-Evaluation.term-of c)
    orelse check-all-char' f cs

definition check-all-char ::
  (char × (unit ⇒ term)) ⇒ (bool × term list) ⇒ (bool × term list) option
  where check-all f = check-all-char' f Enum.enum

definition enum-term-of-char :: char itself ⇒ unit ⇒ term list
  where
    enum-term-of-char = (λ- -. map Code-Evaluation.term-of (Enum.enum :: char list))

instance ..

end

instantiation option :: (check-all) check-all
begin

definition
  check-all f =
  f (Code-Evaluation.valtermify (None :: 'a option)) orelse
  check-all
    (λ(x, t).
      f
        (Some x,
          λ-.
            Code-Evaluation.App
              (Code-Evaluation.Const (STR "Option.option.Some")
                (Typerep.Typerep (STR "fun")
                  [Typerep.Typerep TYPE('a),
                    Typerep.Typerep (STR "Option.option") [Typerep.Typerep TYPE('a)]]))
            (t ())))

definition enum-term-of-option :: 'a option itself ⇒ unit ⇒ term list
  where
    enum-term-of-option =
      (λ- -. Code-Evaluation.term-of (None :: 'a option) #
        (map
          (Code-Evaluation.App
            (Code-Evaluation.Const (STR "Option.option.Some")
              (Typerep.Typerep (STR "fun"))
              (Typerep.Typerep (STR "fun"))))
THEORY "Quickcheck-Exhaustive"

\[
\text{Typerep.typerep TYPE('a),} \\
\text{Typerep.Typerep (STR "Option.option") [Typerep.Typerep TYPE('a)]]})} \\
(\text{enum-term-of (TYPE('a)) ()})
\]

instance ..
end

instantiation \textit{Enum.finite-1} :: check-all
begin
definition check-all f = f (\text{Code-Evaluation.valtermify Enum.finite-1.a1})
definition enum-term-of-finite-1 :: Enum.finite-1 itself ⇒ unit ⇒ term list
where enum-term-of-finite-1 = (λ -. [\text{Code-Evaluation.term-of Enum.finite-1.a1}])
instance ..
end

instantiation \textit{Enum.finite-2} :: check-all
begin
definition check-all f =
(f (\text{Code-Evaluation.valtermify Enum.finite-2.a1}) orelse
f (\text{Code-Evaluation.valtermify Enum.finite-2.a2}))
definition enum-term-of-finite-2 :: Enum.finite-2 itself ⇒ unit ⇒ term list
where enum-term-of-finite-2 =
(λ -. map \text{Code-Evaluation.term-of} (Enum.enum :: Enum.finite-2 list))
instance ..
end

instantiation \textit{Enum.finite-3} :: check-all
begin
definition check-all f =
(f (\text{Code-Evaluation.valtermify Enum.finite-3.a1}) orelse
f (\text{Code-Evaluation.valtermify Enum.finite-3.a2}) orelse
f (\text{Code-Evaluation.valtermify Enum.finite-3.a3}))
definition enum-term-of-finite-3 :: Enum.finite-3 itself ⇒ unit ⇒ term list
where enum-term-of-finite-3 =
(λ -. map \text{Code-Evaluation.term-of} (Enum.enum :: Enum.finite-3 list))
instance ..
end

instantiation Enum.finite-4 :: check-all
begin

definition check-all f =
    f (Code-Evaluation.valtermify Enum.finite-4.a1) orelse
    f (Code-Evaluation.valtermify Enum.finite-4.a2) orelse
    f (Code-Evaluation.valtermify Enum.finite-4.a3) orelse
    f (Code-Evaluation.valtermify Enum.finite-4.a4)

definition enum-term-of-finite-4 :: Enum.finite-4 itself ⇒ unit ⇒ term list
where enum-term-of-finite-4 =
    (λ- -. map Code-Evaluation.term-of (Enum.enum :: Enum.finite-4 list))

instance ..
end

82.3 Bounded universal quantifiers

class bounded-forall =
    fixes bounded-forall :: ('a ⇒ bool) ⇒ natural ⇒ bool

82.4 Fast exhaustive combinators

class fast-exhaustive = term-of +
    fixes fast-exhaustive :: ('a ⇒ unit) ⇒ natural ⇒ unit

axiomatization throw-Counterexample :: term list ⇒ unit
axiomatization catch-Counterexample :: unit ⇒ term list option

code-printing
    constant throw-Counterexample →
        (Quickcheck) raise (Exhaustive'.Generators.Counterexample -)
    | constant catch-Counterexample →
        (Quickcheck) (((-); NONE) handle Exhaustive'.Generators.Counterexample ts ⇒ SOME ts)

82.5 Continuation passing style functions as plus monad

type-synonym 'a cps = ('a ⇒ term list option) ⇒ term list option

definition cps-empty :: 'a cps
where cps-empty = (λcont. None)
definition cps-single :: 'a ⇒ 'a cps
where  cps-single v = (λcont. cont v)

definition cps-bind :: 'a cps ⇒ ('a ⇒ 'b cps) ⇒ 'b cps
where  cps-bind m f = (λcont. m (λa. (f a) cont))

definition cps-plus :: 'a cps ⇒ 'a cps ⇒ 'a cps
where  cps-plus a b = (λc. case a c of None ⇒ b c | Some x ⇒ Some x)

definition cps-if :: bool ⇒ unit cps
where  cps-if b = (if b then cps-single () else cps-empty)

definition cps-not :: unit cps ⇒ unit cps
where  cps-not n = (λc. case n (λu. Some []) of None ⇒ c () | Some - ⇒ None)

type-synonym 'a pos-bound-cps =
('a ⇒ (bool * term list) option) ⇒ natural ⇒ (bool * term list) option

definition pos-bound-cps-empty :: 'a pos-bound-cps
where  pos-bound-cps-empty = (λcont i. None)

definition pos-bound-cps-single :: 'a ⇒ 'a pos-bound-cps
where  pos-bound-cps-single v = (λcont i. cont v)

definition pos-bound-cps-bind :: 'a pos-bound-cps ⇒ ('a ⇒ 'b pos-bound-cps) ⇒ 'b pos-bound-cps
where  pos-bound-cps-bind m f = (λcont i. if i = 0 then None else (m (λa. (f a) cont i) (i - 1)))

definition pos-bound-cps-plus :: 'a pos-bound-cps ⇒ 'a pos-bound-cps ⇒ 'a pos-bound-cps
where  pos-bound-cps-plus a b = (λc i. case a c i of None ⇒ b c i | Some x ⇒ Some x)

definition pos-bound-cps-if :: bool ⇒ unit pos-bound-cps
where  pos-bound-cps-if b = (if b then pos-bound-cps-single () else pos-bound-cps-empty)

datatype (plugins only: code extraction) (dead 'a) unknown =
Unknown | Known 'a

datatype (plugins only: code extraction) (dead 'a) three-valued =
Unknown-value | Value 'a | No-value

type-synonym 'a neg-bound-cps =
('a unknown ⇒ term list three-valued) ⇒ natural ⇒ term list three-valued

definition neg-bound-cps-empty :: 'a neg-bound-cps
where  neg-bound-cps-empty = (λcont i. No-value)

definition neg-bound-cps-single :: 'a ⇒ 'a neg-bound-cps
where neg-bound-cps-single v = (\cont i. cont (Known v))

definition neg-bound-cps-bind :: 'a neg-bound-cps ⇒ ('a ⇒ 'b neg-bound-cps) ⇒ 'b neg-bound-cps
where neg-bound-cps-bind m f =
  (\cont i.
   if i = 0 then cont Unknown
   else m (\a. case a of Unknown ⇒ cont Unknown | Known a' ⇒ f a' cont i) (i - 1))

definition neg-bound-cps-plus :: 'a neg-bound-cps ⇒ 'a neg-bound-cps ⇒ 'a neg-bound-cps
where neg-bound-cps-plus a b =
  (\c i.
   case a c i of
    No-value ⇒ b c i
    | Value x ⇒ Value x
    | Unknown-value ⇒
      (case b c i of
         No-value ⇒ Unknown-value
         | Value x ⇒ Value x
         | Unknown-value ⇒ Unknown-value))

definition neg-bound-cps-if :: bool ⇒ unit neg-bound-cps
where neg-bound-cps-if b = (if b then neg-bound-cps-single () else neg-bound-cps-empty)

definition neg-bound-cps-not :: unit pos-bound-cps ⇒ unit neg-bound-cps
where neg-bound-cps-not n =
  (\i. case n (\u. Some (True, [])) i of None ⇒ cont c (Known ()) | Some - ⇒ No-value)

definition pos-bound-cps-not :: unit neg-bound-cps ⇒ unit pos-bound-cps
where pos-bound-cps-not n =
  (\i. case n (\u. Value []) i of No-value ⇒ c () | Value - ⇒ None | Unknown-value ⇒ None)

82.6 Defining generators for any first-order data type

axiomatization unknown :: 'a

notation (output) unknown (?)

ML-file ⟨Tools/Quickcheck/exhaustive-generators.ML⟩
declare [[quickcheck-batch-tester = exhaustive]]

82.7 Defining generators for abstract types

ML-file ⟨Tools/Quickcheck/abstract-generators.ML⟩
hide-fact (open) orelse-def
83 A compiler for predicates defined by introduction rules
83.1 Set membership as a generator predicate

Introduce a new constant for membership to allow fine-grained control in code equations.

**definition** contains :: 'a set => 'a => bool
**where** contains A x ↔ x ∈ A

**definition** contains-pred :: 'a set => 'a => unit Predicate.pred
**where** contains-pred A x = (if x ∈ A then Predicate.single () else bot)

**lemma** pred-of-setE:
- **assumes** Predicate.eval (pred-of-set A) x
- **obtains** contains A x
  **using** assms by(simp add: contains-def)

**lemma** pred-of-setI: contains A x ==> Predicate.eval (pred-of-set A) x
**by** (simp add: contains-def)

**lemma** pred-of-set-eq: pred-of-set ≡ λA. Predicate.Pred (contains A)
**by** (simp add: contains-def[abs-def] pred-of-set-def o-def)

**lemma** containsI: x ∈ A ==> contains A x
**by** (simp add: contains-def)

**lemma** containsE: **assumes** contains A x
- **obtains** A' x' where A = A' x = x' x ∈ A
  **using** assms by(simp add: contains-def)

**lemma** contains-predI: contains A x ==> Predicate.eval (contains-pred A x) ()
**by** (simp add: contains-pred-def contains-def)

**lemma** contains-predE:
- **assumes** Predicate.eval (contains-pred A x) y
- **obtains** contains A x
  **using** assms by(simp add: contains-pred-def contains-def split: if-split-asm)

**lemma** contains-pred-eq: contains-pred ≡ λA. Predicate.Pred (λy. contains A x)
**by** (rule eq-reflection)(auto simp add: contains-pred-def fun-eq-iff contains-def intro: pred-eqI)

**lemma** contains-pred-notI:
- ¬ contains A x ==> Predicate.eval (Predicate.not-pred (contains-pred A x)) ()
  **by** (simp add: contains-pred-def contains-def not-pred-eq)

**setup**

let
val Fun = Predicate-Compile-Aux.Fun
val Input = Predicate-Compile-Aux.Input
val Output = Predicate-Compile-Aux.Output

val Bool = Predicate-Compile-Aux.Bool
val io = Fun (Input, Fun (Output, Bool))
val ii = Fun (Input, Fun (Input, Bool))
in
Core-Data.PredData.map (Graph.new-node
(const-name contains),
Core-Data.PredData {
  pos = Position.thread-data (),
  intros = [(NONE, @{thm containsI})],
  elim = SOME @{thm containsE},
  preprocessed = true,
  function-names = [(Predicate-Compile-Aux.Pred,[(io, const-name pred-of-set), (ii, const-name contains-pred)]),
  predfun-data = [
    (io, Core-Data.PredfunData {
      elim = @{thm pred-of-setE}, intro = @{thm pred-of-setI},
      neg-intro = NONE, definition = @{thm pred-of-set-eq}
    }),
    (ii, Core-Data.PredfunData {
      elim = @{thm contains-predE}, intro = @{thm contains-predI},
      neg-intro = SOME @{thm contains-pred-notI}, definition = @{thm contains-pred-eq}
    })],
  needs-random = []})
end

hide-const (open) contains contains-pred
hide-fact (open) pred-of-setE pred-of-setI pred-of-set-eq
containsI containsE contains-predI contains-predE contains-pred-eq contains-pred-notI
end

84 Counterexample generator performing narrowing-based testing

theory Quickcheck-Narrowing
imports Quickcheck-Random
keywords find-unused-assms :: diag
begin

84.1 Counterexample generator

84.1.1 Code generation setup

setup (Code-Target.add-derived-target (Haskell-Quickcheck, [(Code-Haskell.target, I)]))

code-printing
code-module Typerep → (Haskell-Quickcheck) ;
module Typerep(Typerep(..)) where

data Typerep = Typerep String [Typerep]
| for type-constructor typerep constant Typerep.Typerep
| type-constructor typerep → (Haskell-Quickcheck) Typerep.Typerep
| constant Typerep.Typerep → (Haskell-Quickcheck) Typerep.Typerep

code-reserved Haskell-Quickcheck Typerep

code-printing
| type-constructor integer → (Haskell-Quickcheck) Prelude.Int
| constant 0::integer → (Haskell-Quickcheck) !(0/ ::/ Prelude.Int)

setup ( let
val target = Haskell-Quickcheck;
fun print - = Code-Haskell.print-numeral Prelude.Int;
in
Numeral.add-code const-name (Code-Numeral.Pos) I print target
#> Numeral.add-code const-name (Code-Numeral.Neg) (~) print target
end )

84.1.2 Narrowing's deep representation of types and terms

datatype (plugins only: code extraction) narrowing-type =
Narrowing-sum-of-products narrowing-type list list

datatype (plugins only: code extraction) narrowing-term =
Narrowing-variable integer list narrowing-type
| Narrowing-constructor integer narrowing-term list

datatype (plugins only: code extraction) (dead 'a) narrowing-cons =
Narrowing-cons narrowing-type (narrowing-term list ⇒ 'a) list

primrec map-cons :: ('a => 'b) => 'a narrowing-cons => 'b narrowing-cons
where
map-cons f (Narrowing-cons ty cs) = Narrowing-cons ty (map (λc. f o c) cs)

84.1.3 From narrowing's deep representation of terms to HOL.Code-Evaluation's terms

class partial-term-of = typerep +
fixes partial-term-of :: 'a itself ⇒ narrowing-term ⇒ Code-Evaluation.term

lemma partial-term-of-anything: partial-term-of x nt = t
  by (rule eq-reflection) (cases partial-term-of x nt, cases t, simp)
84.1.4 Auxilary functions for Narrowing

consts nth :: 'a list => integer => 'a

code-printing constant nth -> (Haskell-Quickcheck) infixl 9 !!

consts error :: char list => 'a

code-printing constant error -> (Haskell-Quickcheck) error

consts toEnum :: integer => char

code-printing constant toEnum -> (Haskell-Quickcheck) Prelude.toEnum

consts marker :: char

code-printing constant marker -> (Haskell-Quickcheck) '\0'

84.1.5 Narrowing’s basic operations

type-synonym 'a narrowing = integer => 'a narrowing-cons

definition cons :: 'a => 'a narrowing

where
cons a d = (Narrowing-cons (Narrowing-sum-of-products [[[]]]) [(\- a)])

fun conv :: (narrowing-term list => 'a) list => narrowing-term => 'a

where
conv cs (Narrowing-variable p -) = error (marker # map toEnum p)
| conv cs (Narrowing-constructor i xs) = (nth cs i) xs

fun non-empty :: narrowing-type => bool

where
non-empty (Narrowing-sum-of-products ps) = (\ (List.null ps))

definition apply :: ('a => 'b) narrowing => 'a narrowing => 'b narrowing

where
apply f a d = (if d > 0 then
  (case f d of Narrowing-cons (Narrowing-sum-of-products ps) cfs =>
  case a (d - 1) of Narrowing-cons ta cas =>
  let
  shallow = non-empty ta;
  cs = [(\(x # xs) => cf xs (conv cas x)). shallow, cf ← cfs]
  in Narrowing-cons (Narrowing-sum-of-products [ta # p. shallow, p ← ps])
  cs)
else Narrowing-cons (Narrowing-sum-of-products [[]]) [[]])

definition sum :: 'a narrowing => 'a narrowing => 'a narrowing

where
sum a b d =
case a d of Narrowing-cons (Narrowing-sum-of-products ssa) ca ⇒
case b d of Narrowing-cons (Narrowing-sum-of-products ssb) cb ⇒
Narrowing-cons (Narrowing-sum-of-products (ssa @ ssb)) (ca @ cb)

lemma [fundef-cong]:
assumes a d = a' d b d = b' d d = d'
shows sum a b d = sum a' b' d'
using assms unfolding sum-def by (auto split: narrowing-cons.split narrowing-type.split)

lemma [fundef-cong]:
assumes f d = f' d (\land d', 0 \leq d' \land d' < d \implies a d' = a' d')
assumes d = d'
shows apply f a d = apply f' a' d'
proof –
note assms
moreover have 0 < d' \implies 0 \leq d' - 1
by (simp add: less-integer-def less-eq-integer-def)
ultimately show ?thesis
by (auto simp add: apply-def Let-def
split: narrowing-cons.split narrowing-type.split)
qed

84.1.6 Narrowing generator type class

class narrowing =
  fixes narrowing :: integer => 'a narrowing-cons

datatype (plugins only: code extraction) property =
  Universal narrowing-term => property narrowing-term =>
  Code-Evaluation.term
| Existential narrowing-term => property narrowing-term =>
  Code-Evaluation.term
| Property bool

definition exists :: ('a :: {narrowing, partial-term-of} => property) => property
where
  exists f = (case narrowing (100 :: integer) of Narrowing-cons ty cs ⇒ Existential ty (λt. f (conv cs t)) (partial-term-of (TYPE('a))))

definition all :: ('a :: {narrowing, partial-term-of} => property) => property
where
  all f = (case narrowing (100 :: integer) of Narrowing-cons ty cs ⇒ Universal ty (λt. f (conv cs t)) (partial-term-of (TYPE('a))))

84.1.7 class is-testable

The class is-testable ensures that all necessary type instances are generated.
class is-testable
Theory "Quickcheck-Narrowing"

\begin{verbatim}
instance bool :: is-testable ..

instance fun :: (\{term-of, narrowing, partial-term-of\}, is-testable) is-testable ..

definition ensure-testable :: 'a :: is-testable => 'a :: is-testable
where
  ensure-testable f = f

84.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way

datatype (plugins only: code quickcheck-narrowing extraction) (dead 'a, dead 'b)
ffun =
  Constant 'b
| Update 'a 'b ('a, 'b) ffun

primrec eval-ffun :: ('a, 'b) ffun => 'a => 'b
where
  eval-ffun (Constant c) x = c
| eval-ffun (Update x' y f) x = (if x = x' then y else eval-ffun f x)

hide-type (open) ffun
hide-const (open) Constant Update eval-ffun

datatype (plugins only: code quickcheck-narrowing extraction) (dead 'b)
cfun =
  Constant 'b

primrec eval-cfun :: 'b cfun => 'a => 'b
where
  eval-cfun (Constant c) y = c

hide-type (open) cfun
hide-const (open) Constant eval-cfun Abs-cfun Rep-cfun

84.1.9 Setting up the counterexample generator

external-file \langle\~\~\~/src/HOL/Tools/Quickcheck/Narrowing-Engine.hs\rangle
external-file \langle\~\~\~/src/HOL/Tools/Quickcheck/PNF-Narrowing-Engine.hs\rangle
ML-file \langleTools/Quickcheck/narrowing-generators.ML\rangle

definition narrowing-dummy-partial-term-of :: ('a :: partial-term-of) itself =>
narrowing-term => term
where
  narrowing-dummy-partial-term-of = partial-term-of

definition narrowing-dummy-narrowing :: integer => ('a :: narrowing) narrowing-cons
where
  narrowing-dummy-narrowing = narrowing
\end{verbatim}
lemma [code]:
ensure-testable f =
(let
  x = narrowing-dummy-narrowing :: integer => bool narrowing-cons;
  y = narrowing-dummy-partial-term-of :: bool itself => narrowing-term =>
term;
  z = (conv :: - => - => unit) in f)
unfolding Let-def ensure-testable-def ..

84.2 Narrowing for sets

instantiation set :: (narrowing) narrowing
begin

definition narrowing-set = Quickcheck-Narrowing.apply (Quickcheck-Narrowing.cons
set) narrowing

instance ..

end

84.3 Narrowing for integers

definition drawn-from :: 'a list => 'a narrowing-cons
where
drawn-from xs =
  Narrowing-cons (Narrowing-sum-of-products (map (λ-. [])) xs) (map (λx -. x)
xs)

function around-zero :: int => int list
where
  around-zero i = (if i < 0 then [] else (if i = 0 then [0] else around-zero (i - 1)
@ [i, −i]))
by pat-completeness auto
termination by (relation measure nat) auto

declare around-zero.simps [simp del]

lemma length-around-zero:
  assumes i >= 0
  shows length (around-zero i) = 2 * nat i + 1
proof (induct rule: int-ge-induct [OF assms])
case 1
  from 1 show ?case by (simp add: around-zero.simps)
next
case (2 i)
  from 2 show ?case
    by (simp add: around-zero.simps [of i + 1])
qed
instantiation int :: narrowing
begin

definition
  narrowing-int d = (let (u :: - ⇒ - ⇒ unit) = conv; i = int-of-integer d
  in drawn-from (around-zero i))

instance ..
end

declare [[code drop: partial-term-of :: int itself ⇒ -]]

lemma [code]:
  partial-term-of (ty :: int itself) (Narrowing-variable p t) ≡
  Code-Evaluation.Free (STR "\"\") (Typerep.Typerep (STR "Int.int") [])
  partial-term-of (ty :: int itself) (Narrowing-constructor i []) ≡
  (if i mod 2 = 0
    then Code-Evaluation.term-of (− (int-of-integer i) div 2)
    else Code-Evaluation.term-of ((int-of-integer i + 1) div 2))
  by (rule partial-term-of-anything)

instantiation integer :: narrowing
begin

definition
  narrowing-integer d = (let (u :: - ⇒ - ⇒ unit) = conv; i = int-of-integer d
  in drawn-from (map integer-of-int (around-zero i)))

instance ..
end

declare [[code drop: partial-term-of :: integer itself ⇒ -]]

lemma [code]:
  partial-term-of (ty :: integer itself) (Narrowing-variable p t) ≡
  partial-term-of (ty :: integer itself) (Narrowing-constructor i []) ≡
  (if i mod 2 = 0
    then Code-Evaluation.term-of (− i div 2)
    else Code-Evaluation.term-of ((i + 1) div 2))
  by (rule partial-term-of-anything)

code-printing constant Code-Evaluation.term-of :: integer ⇒ term ↦ (Haskell-Quickcheck)

(let { t = Typerep.Typerep Code'-Numeral.integer []);
mkFunT $s \; t = \text{Typerep.Typerep.fun} \; \left[ s, \; t \right];
numT = \text{Typerep.Typerep.Num.num} \; [];
mkBit 0 = \text{Generated'-Code.Const Num.num.Bit0} \; (\text{mkFunT numT numT});
mkBit 1 = \text{Generated'-Code.Const Num.num.Bit1} \; (\text{mkFunT numT numT});
mkNumeral 1 = \text{Generated'-Code.Const Num.num.One} \; numT;
\text{mkNumeral} \; i = \text{let} \; \{ \; q = i \; \text{'Prelude.div'} \; 2; \; r = i \; \text{'Prelude.mod'} \; 2 \; \}\; \text{in}\; \text{Generated'-Code.App} \; (\text{mkBit} \; r) \; (\text{mkNumeral} \; q);
\text{mkNumeral} \; 0 = \text{Generated'-Code.Const Groups.zero'-class.zero} \; t;
\text{mkNumeral} \; 1 = \text{Generated'-Code.Const Groups.one'-class.one} \; t;
\text{mkNumeral} \; i = \text{if} \; i > 0 \; \text{then}
\text{Generated'-Code.App}
(\text{Generated'-Code.Const Num.numeral'-class.numeral} \; (\text{mkFunT numT t}))
(\text{mkNumeral} \; i)
\text{else}
\text{Generated'-Code.App}
(\text{Generated'-Code.Const Groups.uminus'-class.uminus} \; (\text{mkFunT} \; t \; t))
(\text{mkNumber} \; (\; - \; i \;)); \; \text{in} \; \text{mkNumber}

84.4 The find-unused-assms command

ML-file ⟨Tools/Quickcheck/find-unused-assms.ML⟩

84.5 Closing up

hide-type narrowing-type narrowing-term narrowing-cons property
hide-const map-cons nth error toEnum marker empty Narrowing-cons conv non-empty ensure-testable all exists drawn-from around-zero
hide-const (open) Narrowing-variable Narrowing-constructor apply sum cons
hide-fact empty-def cons-def conv.e.simps non-empty.simps apply-def sum-def ensure-testable-def all-def exists-def

end

theory Mirabelle
  imports Sledgehammer Predicate-Compile Presburger
begin

ML-file ⟨Tools/Mirabelle/mirabelle-util.ML⟩
ML-file ⟨Tools/Mirabelle/mirabelle.ML⟩

ML ⟨
signature MIRABELLE-ACTION = sig
  val make-action : Mirabelle.action-context -> string * Mirabelle.action
end⟩

ML-file ⟨Tools/Mirabelle/mirabelle-arith.ML⟩
ML-file ⟨Tools/Mirabelle/mirabelle-order.ML⟩
THEORY “Extraction”

ML-file (Tools/Mirabelle/mirabelle-metis.ML)
ML-file (Tools/Mirabelle/mirabelle-presburger.ML)
ML-file (Tools/Mirabelle/mirabelle-quickcheck.ML)
ML-file (Tools/Mirabelle/mirabelle-sledgehammer-filter.ML)
ML-file (Tools/Mirabelle/mirabelle-sledgehammer.ML)
ML-file (Tools/Mirabelle/mirabelle-try0.ML)

end

85 Program extraction for HOL

theory Extraction
imports Option
begin

85.1 Setup

setup (Extraction.add-types
  [[bool, (]], NONE)]) #>
Extraction.set-preprocessor (fn thy =>>
  Proofterm.rewrite-proof-notypes
  (]], Rewrite-HOL-Proof.elim-cong :: Proof-Rewrite-Rules.rprocs true) o
  Proofterm.rewrite-proof thy
  (Rewrite-HOL-Proof.rews,
   Proof-Rewrite-Rules.rprocs true #> [Proof-Rewrite-Rules.expand-of-class thy]))
  o
   Proof-Rewrite-Rules.elim-vars (curry Const const-name (default)))

lemmas [extraction-expand] =
  meta-spec atomize-eq atomize-all atomize-imp atomize-conj
  allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
  notE' impE' impE iffE imp-cong simp-thms eq-True eq-False
  induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq
  induct-atomize induct-atomize' induct-rulify induct-rulify'
  induct-rulify-fallback induct-trueI
  True-implies-equals implies-True-equals TrueE
  False-implies-equals implies-False-swap

lemmas [extraction-expand-def] =
  HOL.induct-forall-def HOL.induct-implies-def HOL.induct-equal-def
  HOL.induct-conj-def
  HOL.induct-true-def HOL.induct-false-def

datatype (plugins only: code extraction) sumbool = Left | Right

85.2 Type of extracted program

extract-type
THEORY “Extraction”

\[ \text{typeof} \ (\text{Trueprop} \ P) \equiv \text{typeof} \ P \]

\[ \text{typeof} \ P \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \implies \text{typeof} \ Q \equiv \text{Type} \ (\text{TYPE} \ ('Q')) \implies \]
\[ \text{typeof} \ (P \to Q) \equiv \text{Type} \ (\text{TYPE} \ ('Q')) \]

\[ \text{typeof} \ Q \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \implies \text{typeof} \ (P \to Q) \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \]

\[ \text{typeof} \ P \equiv \text{Type} \ (\text{TYPE} \ ('P')) \implies \text{typeof} \ Q \equiv \text{Type} \ (\text{TYPE} \ ('Q')) \implies \]
\[ \text{typeof} \ (P \to Q) \equiv \text{Type} \ (\text{TYPE} \ ('Q')) \]

\[ (\lambda x. \text{typeof} \ (P \ x)) \equiv (\lambda x. \text{Type} \ (\text{TYPE} \ (\text{Null}))) \implies \]
\[ \text{typeof} \ (\forall x. \ P \ x) \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \]

\[ (\lambda x. \text{typeof} \ (P \ x)) \equiv (\lambda x. \text{Type} \ (\text{TYPE} \ ('P'))) \implies \]
\[ \text{typeof} \ (\forall x :: 'a. \ P \ x) \equiv \text{Type} \ (\text{TYPE} \ ('a \Rightarrow 'Q')) \]

\[ (\lambda x. \text{typeof} \ (P \ x)) \equiv (\lambda x. \text{Type} \ (\text{TYPE} \ (\text{Null}))) \implies \]
\[ \text{typeof} \ (\exists x :: 'a. \ P \ x) \equiv \text{Type} \ (\text{TYPE} \ ('a)) \]

\[ (\lambda x. \text{typeof} \ (P \ x)) \equiv (\lambda x. \text{Type} \ (\text{TYPE} \ ('P'))) \implies \]
\[ \text{typeof} \ (\exists x :: 'a. \ P \ x) \equiv \text{Type} \ (\text{TYPE} \ ('a \times 'P')) \]

\[ \text{typeof} \ P \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \implies \text{typeof} \ Q \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \implies \]
\[ \text{typeof} \ (P \lor Q) \equiv \text{Type} \ (\text{TYPE} \ (\text{sumbool})) \]

\[ \text{typeof} \ P \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \implies \text{typeof} \ Q \equiv \text{Type} \ (\text{TYPE} \ ('Q')) \implies \]
\[ \text{typeof} \ (P \lor Q) \equiv \text{Type} \ (\text{TYPE} \ ('Q \ \text{option}')) \]

\[ \text{typeof} \ P \equiv \text{Type} \ (\text{TYPE} \ ('P')) \implies \text{typeof} \ Q \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \implies \]
\[ \text{typeof} \ (P \lor Q) \equiv \text{Type} \ (\text{TYPE} \ ('P \ \text{option}')) \]

\[ \text{typeof} \ P \equiv \text{Type} \ (\text{TYPE} \ ('P')) \implies \text{typeof} \ Q \equiv \text{Type} \ (\text{TYPE} \ ('Q')) \implies \]
\[ \text{typeof} \ (P \lor Q) \equiv \text{Type} \ (\text{TYPE} \ ('P + 'Q')) \]

\[ \text{typeof} \ P \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \implies \text{typeof} \ Q \equiv \text{Type} \ (\text{TYPE} \ ('Q')) \implies \]
\[ \text{typeof} \ (P \lor Q) \equiv \text{Type} \ (\text{TYPE} \ ('Q')) \]

\[ \text{typeof} \ P \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \implies \text{typeof} \ Q \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \implies \]
\[ \text{typeof} \ (P \lor Q) \equiv \text{Type} \ (\text{TYPE} \ ('P')) \]

\[ \text{typeof} \ P \equiv \text{Type} \ (\text{TYPE} \ ('P')) \implies \text{typeof} \ Q \equiv \text{Type} \ (\text{TYPE} \ (\text{Null})) \implies \]
\[ \text{typeof} \ (P \lor Q) \equiv \text{Type} \ (\text{TYPE} \ ('P \times 'Q')) \]

\[ \text{typeof} \ (P = Q) \equiv \text{typeof} \ ((P \to Q) \land (Q \to P)) \]

\[ \text{typeof} \ (x \in P) \equiv \text{typeof} \ P \]
85.3 Realizability

realizability

\[
\begin{align*}
\text{(realizes } t \ (\text{Trueprop } P)) & \equiv (\text{Trueprop} \ (\text{realizes } t \ P)) \\
\text{(typeof } P) & \equiv (\text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (P \rightarrow Q)) \equiv (\text{realizes Null } P \implies \text{realizes } t \ Q) \\
\text{(realizes } t \ (P \rightarrow Q)) & \equiv (\forall x :: P. \text{realizes } x P \implies \text{realizes } t \ Q) \\
\text{(typeof } P) & \equiv (\text{Type} \ (\text{TYPE}'(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (P \rightarrow Q)) \equiv (\forall x :: P. \text{realizes } x P \implies \text{realizes Null } Q) \\
\text{(realizes } t \ (P \rightarrow Q)) & \equiv (\forall x. \text{realizes } x P \implies \text{realizes } (t \ x) \ Q) \\
\text{(\lambda x. typeof } (P x)) & \equiv (\lambda x. \text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (\forall x. \ P x)) \equiv (\forall x. \text{realizes Null } (P x)) \\
\text{(realizes } t \ (\forall x. \ P x)) & \equiv (\forall x. \text{realizes } (t \ x) \ (P x)) \\
\text{(\lambda x. typeof } (P x)) & \equiv (\lambda x. \text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (\exists x. \ P x)) \equiv (\text{realizes Null } (P t)) \\
\text{(realizes } t \ (\exists x. \ P x)) & \equiv (\text{realizes } (\text{snd } t) \ (P (\text{fst } t))) \\
\text{(typeof } P) & \equiv (\text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (P \lor Q)) \equiv \\
& \quad \quad (\text{case } t \ of \ \text{Left} \Rightarrow \text{realizes } \text{Null } P \mid \text{Right} \Rightarrow \text{realizes } \text{Null } Q) \\
\text{typeof } P & \equiv \text{Type} \ (\text{TYPE}(\text{Null})) \implies \\
& \quad (\text{realizes } t \ (P \lor Q)) \equiv \\
& \quad \quad (\text{case } t \ of \ \text{None} \Rightarrow \text{realizes } \text{Null } P \mid \text{Some } q \Rightarrow \text{realizes } q \ Q) \\
\text{typeof } Q & \equiv \text{Type} \ (\text{TYPE}(\text{Null})) \implies \\
& \quad (\text{realizes } t \ (P \lor Q)) \equiv \\
& \quad \quad (\text{case } t \ of \ \text{None} \Rightarrow \text{realizes } \text{Null } Q \mid \text{Some } p \Rightarrow \text{realizes } p \ P) \\
\text{(realizes } t \ (P \lor Q)) & \equiv \\
& \quad (\text{case } t \ of \ \text{Inl } p \Rightarrow \text{realizes } p \ P \mid \text{Inr } q \Rightarrow \text{realizes } q \ Q) \\
\text{typeof } P & \equiv \text{Type} \ (\text{TYPE}(\text{Null})) \implies \\
& \quad (\text{realizes } t \ (P \land Q)) \equiv (\text{realizes } \text{Null } P \land \text{realizes } t \ Q) \\
\text{typeof } Q & \equiv \text{Type} \ (\text{TYPE}(\text{Null})) \implies \\
& \quad (\text{realizes } t \ (P \land Q)) \equiv (\text{realizes } t \ P \land \text{realizes } \text{Null } Q) \\
\text{(realizes } t \ (P \land Q)) & \equiv (\text{realizes } (\text{fst } t) \ P \land \text{realizes } (\text{snd } t) \ Q) \\
\text{typeof } P & \equiv \text{Type} \ (\text{TYPE}(\text{Null})) \implies \\
& \quad (\text{realizes } t \ (\neg P) \equiv \neg \text{realizes } \text{Null } P)
\[
\text{typeof } P \equiv \text{Type} (\text{TYPE}' P) \implies \text{realizes } t (\neg P) \equiv (\forall x : \text{TYPE}' P. \neg \text{realizes } x P)
\]

\[
\text{typeof } (P :: \text{bool}) \equiv \text{Type} (\text{TYPE}(\text{Null})) \implies \\
\text{typeof } Q \equiv \text{Type} (\text{TYPE}(\text{Null})) \implies \\
\text{realizes } t (P = Q) \equiv \text{realizes } \text{Null } P = \text{realizes } \text{Null } Q
\]

\[(\text{realizes } t (P = Q)) \equiv (\text{realizes } t ((P \implies Q) \land (Q \implies P)))\]

### 85.4 Computational content of basic inference rules

**Theorem disjE-realizer:**

- **Assumes** \( r : \text{case } x \text{ of } \text{Inl } p \Rightarrow P p \mid \text{Inr } q \Rightarrow Q q \)
- **And** \( r1 : \bigwedge p. P p \Rightarrow R (f p) \and r2 : \bigwedge q. Q q \Rightarrow R (g q) \)
- **Shows** \( R (\text{case } x \text{ of } \text{Inl } p \Rightarrow f p \mid \text{Inr } q \Rightarrow g q) \)

**Proof (cases x)**

- **Case** `Inl`
  - with \( r \) show ?thesis by simp (rule \( r1 \))
- next
  - **Case** `Inr`
    - with \( r \) show ?thesis by simp (rule \( r2 \))

**QED**

**Theorem disjE-realizer2:**

- **Assumes** \( r : \text{case } x \text{ of } \text{None } \Rightarrow P \mid \text{Some } q \Rightarrow Q q \)
- **And** \( r1 : P \Rightarrow R f \and r2 : \bigwedge q. Q q \Rightarrow R (g q) \)
- **Shows** \( R (\text{case } x \text{ of } \text{None } \Rightarrow f \mid \text{Some } q \Rightarrow g q) \)

**Proof (cases x)**

- **Case** `None`
  - with \( r \) show ?thesis by simp (rule \( r1 \))
- next
  - **Case** `Some`
    - with \( r \) show ?thesis by simp (rule \( r2 \))

**QED**

**Theorem disjE-realizer3:**

- **Assumes** \( r : \text{case } x \text{ of } \text{Left } \Rightarrow P \mid \text{Right } \Rightarrow Q \)
- **And** \( r1 : P \Rightarrow R f \and r2 : Q \Rightarrow R g \)
- **Shows** \( R (\text{case } x \text{ of } \text{Left } \Rightarrow f \mid \text{Right } \Rightarrow g) \)

**Proof (cases x)**

- **Case** `Left`
  - with \( r \) show ?thesis by simp (rule \( r1 \))
- next
  - **Case** `Right`
    - with \( r \) show ?thesis by simp (rule \( r2 \))

**QED**

**Theorem conjI-realizer:**
THEORY "Extraction"

\[ P \Rightarrow Q \Rightarrow P (\text{fst} (p, q)) \land Q (\text{snd} (p, q)) \]

by simp

**theorem** exI-realizer:
\[ P y x \Rightarrow P (\text{snd} (x, y)) (\text{fst} (x, y)) \text{ by simp} \]

**theorem** exE-realizer: \[ P (\text{snd} p) (\text{fst} p) \Rightarrow \]
\((\forall x y. P y x \Rightarrow Q (f x y)) \Rightarrow Q (\text{let} (x, y) = p \text{ in } f x y)\)

by (cases p) (simp add: Let-def)

**theorem** exE-realizer¹: \[ P (\text{snd} p) (\text{fst} p) \Rightarrow \]
\((\forall x y. P y x \Rightarrow Q) \Rightarrow Q \text{ by } \text{(cases } p \text{) simp} \)

**realizers**

**impI** \((P, Q): \lambda pq. pq\)
\[ \lambda(c: -) (d: -) P Q pq (h: -). \text{all}I \cdot \cdot \cdot c \cdot (\lambda x. \text{impI} \cdot \cdot \cdot \cdot (h \cdot x)) \]

**impI** \((P): \text{Null}\)
\[ \lambda(c: -) P Q (h: -). \text{all}I \cdot \cdot \cdot c \cdot (\lambda x. \text{impI} \cdot \cdot \cdot \cdot (h \cdot x)) \]

**impI** \((Q): \lambda q. \lambda(c: -) P Q q. \text{impI} \cdot \cdot \cdot \cdot \cdot \]

**impI** \((P): \text{Null \ impI} \)

**mp** \((P, Q): \lambda pq. pq\)
\[ \lambda(c: -) (d: -) P Q pq (h: -) p. \text{mp} \cdot \cdot \cdot \cdot (\text{spec} \cdot \cdot \cdot p \cdot c \cdot h) \]

**mp** \((P): \text{Null}\)
\[ \lambda(c: -) P Q (h: -) p. \text{mp} \cdot \cdot \cdot \cdot (\text{spec} \cdot \cdot \cdot p \cdot c \cdot h) \]

**mp** \((Q): \lambda q. \lambda(c: -) P Q q. \text{mp} \cdot \cdot \cdot \cdot \]

**mp** \((P): \text{Null \ mp} \)

**allI** \((P): \lambda p. \lambda(c: -) P (d: -) p. \text{all}I \cdot \cdot \cdot d \)

**allI** \((P): \text{Null \ allI} \)

**spec** \((P): \lambda x p. \lambda x P (d: -) p. \text{spec} \cdot \cdot \cdot x \cdot d \)

**spec** \((P): \text{Null \ spec} \)

**exI** \((P): \lambda x p. \lambda x P (d: -) p. \text{exI-realizer} \cdot P \cdot p \cdot x \cdot c \cdot d \)

**exI** \((P): \lambda x P x (c: -) (h: -). h \)

**exE** \((P, Q): \lambda p. \lambda q. \text{let} (x, y) = p \text{ in } pq x y \)
\[ \lambda(c: -) (d: -) P Q (e: -) p (h: -) pq. \text{exE-realizer} \cdot P \cdot p \cdot Q \cdot pq \cdot c \cdot e \cdot d \cdot h \]
THEORY “Extraction”

exE (P): Null
\[ \lambda(c::) \; P \; Q \; (d::) \; p. \; exE\text{-realizer'} \cdot \cdot \cdot \cdot \cdot c \cdot d \]

exE (Q): \lambda x \; pq \; pq \; x
\[ \lambda(c::) \; P \; Q \; (d::) \; x \; (h1::) \; pq \; (h2::) \; h2 \cdot x \cdot h1 \]

exE: Null
\[ \lambda P \; Q \; (c::) \; x \; (h1::) \; (h2::) \; h2 \cdot x \cdot h1 \]

conjI (P, Q): Pair
\[ \lambda(c::) \; (d::) \; P \; Q \; p \; (h::) \; q. \; conjI\text{-realizer} \cdot P \cdot p \cdot Q \cdot q \cdot c \cdot d \cdot h \]

conjI (P): \lambda p. p
\[ \lambda(c::) \; P \; Q \; p. \; conjI \cdot \cdot \cdot \cdot \]

conjI (Q): \lambda q. q
\[ \lambda(c::) \; P \; Q \; (h::) \; q. \; conjI \cdot \cdot \cdot \cdot h \]

conjI: Null conjI

conjunct1 (P, Q): fst
\[ \lambda(c::) \; (d::) \; P \; Q \; pq. \; conjunct1 \cdot \cdot \cdot \cdot \]

conjunct1 (P): \lambda p. p
\[ \lambda(c::) \; P \; Q \; p. \; conjunct1 \cdot \cdot \cdot \cdot \]

conjunct1 (Q): Null
\[ \lambda(c::) \; P \; Q \; q. \; conjunct1 \cdot \cdot \cdot \cdot \]

conjunct1: Null conjunct1

conjunct2 (P, Q): snd
\[ \lambda(c::) \; (d::) \; P \; Q \; pq. \; conjunct2 \cdot \cdot \cdot \cdot \]

conjunct2 (P): Null
\[ \lambda(c::) \; P \; Q \; p. \; conjunct2 \cdot \cdot \cdot \cdot \]

conjunct2 (Q): \lambda p. p
\[ \lambda(c::) \; P \; Q \; p. \; conjunct2 \cdot \cdot \cdot \cdot \]

conjunct2: Null conjunct2

disjI1 (P, Q): Inl
\[ \lambda(c::) \; (d::) \; P \; Q \; p. \; iffD2 \cdot \cdot \cdot \cdot (\text{sum.case-1} \cdot P \cdot \cdot \cdot p \cdot \text{arity-type-bool} \cdot c \cdot d) \]

disjI1 (P): Some
\[ \lambda(c::) \; P \; Q \; p. \; iffD2 \cdot \cdot \cdot \cdot (\text{option.case-2} \cdot \cdot \cdot P \cdot p \cdot \text{arity-type-bool} \cdot c) \]
disj11 (Q): None
\[ \lambda (c: \cdot) P Q. \text{iffD2} \cdot \cdot \cdot (\text{option.case-1} \cdot \cdot \cdot \text{arity-type-bool} \cdot c) \]

disj11: Left
\[ \lambda P Q. \text{iffD2} \cdot \cdot \cdot (\text{sumbool.case-1} \cdot \cdot \cdot \text{arity-type-bool}) \]

disj12 (P, Q): Inr
\[ \lambda (d: \cdot) (c: \cdot) Q P q. \text{iffD2} \cdot \cdot \cdot (\text{sum.case-2} \cdot \cdot Q \cdot q \cdot \text{arity-type-bool} \cdot c \cdot d) \]

disj12 (P): None
\[ \lambda (c: \cdot) Q P. \text{iffD2} \cdot \cdot \cdot (\text{option.case-1} \cdot \cdot \cdot \text{arity-type-bool} \cdot c) \]

disj12 (Q): Some
\[ \lambda (c: \cdot) Q P q. \text{iffD2} \cdot \cdot \cdot (\text{option.case-2} \cdot \cdot Q \cdot q \cdot \text{arity-type-bool} \cdot c) \]

disj12: Right
\[ \lambda Q P. \text{iffD2} \cdot \cdot \cdot (\text{sumbool.case-2} \cdot \cdot \cdot \text{arity-type-bool}) \]

disjE (P, Q, R): \lambda pq pr qr.
\[ \text{(case pq of \text{Inl} p \Rightarrow pr p | \text{Inr} q \Rightarrow qr q)} \]
\[ \lambda (c: \cdot) (d: \cdot) (e: \cdot) P Q R pq (h1: \cdot) pr (h2: \cdot) qr. \]
\[ \text{disjE-realizer} \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot e \cdot h1 \cdot h2 \]

disjE (Q, R): \lambda pq pr qr.
\[ \text{(case pq of \text{None} \Rightarrow pr | \text{Some} q \Rightarrow qr q)} \]
\[ \lambda (c: \cdot) (d: \cdot) P Q R pq (h1: \cdot) pr (h2: \cdot) qr. \]
\[ \text{disjE-realizer2} \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot e \cdot h1 \cdot h2 \]

disjE (P, R): \lambda pq pr qr.
\[ \text{(case pq of \text{None} \Rightarrow qr | \text{Some} p \Rightarrow pr p)} \]
\[ \lambda (c: \cdot) (d: \cdot) P Q R pq (h1: \cdot) pr (h2: \cdot) qr (h3: \cdot). \]
\[ \text{disjE-realizer3} \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot h1 \cdot h3 \cdot h2 \]

disjE (R): \lambda pq pr qr.
\[ \text{(case pq of \text{Left} \Rightarrow pr | \text{Right} \Rightarrow qr)} \]
\[ \lambda (c: \cdot) (d: \cdot) P Q R pq (h1: \cdot) pr (h2: \cdot) qr. \]
\[ \text{disjE-realizer3} \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot c \cdot h1 \cdot h2 \]

disjE (P, Q): Null
\[ \lambda (c: \cdot) (d: \cdot) P Q R pq. \text{disjE-realizer} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot d \cdot \text{arity-type-bool} \]

disjE (Q): Null
\[ \lambda (c: \cdot) P Q R pq. \text{disjE-realizer2} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot \text{arity-type-bool} \]

disjE (P): Null
\[ \lambda (c: \cdot) P Q R pq (h1: \cdot) (h2: \cdot) (h3: \cdot). \]
\[ \text{disjE-realizer2} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot \text{arity-type-bool} \cdot h1 \cdot h3 \cdot h2 \]
\[ \text{disjE: Null} \]
\[ \lambda P \; Q \; R \; pq \; \text{disjE-realizer3} \quad \text{null} \quad (\lambda x. \; R) \quad \text{arity-type-bool} \]

\[ \text{FalseE (P): default} \]
\[ \lambda (c: -) \; P. \; \text{FalseE} \quad - \]

\[ \text{FalseE: Null FalseE} \]

\[ \text{notI (P): Null} \]
\[ \lambda (c: -) \; P \; (h: -). \; \text{allI} \quad - \quad c \quad (\lambda x. \; \text{notI} \quad - \quad (h \cdot x)) \]

\[ \text{notI: Null notI} \]

\[ \text{notE (P, R): \lambda p. \; \text{default} } \]
\[ \lambda (c: -) \; (d: -) \; P \; R \; (h: -) \; p. \; \text{notE} \quad - \quad - \quad (\text{spec} \quad - \quad p \quad c \quad h) \]

\[ \text{notE (P): Null} \]
\[ \lambda (c: -) \; P \; R \; (h: -) \; p. \; \text{notE} \quad - \quad - \quad (\text{spec} \quad - \quad p \quad c \quad h) \]

\[ \text{notE (R): default} \]
\[ \lambda (c: -) \; P \; R. \; \text{notE} \quad - \quad - \quad - \]

\[ \text{notE: Null notE} \]

\[ \text{subst (P): \lambda s \; t \; ps. \; ps} \]
\[ \lambda (c: -) \; s \; t \; P \; (d: -) \; (h: -) \; ps \; . \; \text{subst} \; \cdot \; s \; \cdot \; t \; \cdot \; P \; ps \; \cdot \; d \; \cdot \; h \]

\[ \text{subst: Null subst} \]

\[ \text{iffD1 (P, Q): fst} \]
\[ \lambda (d: -) \; (c: -) \; Q \; P \; pq \; (h: -) \; p. \]
\[ \text{mp} \quad - \quad - \quad - \quad (\text{spec} \quad - \quad p \quad d \quad (\text{conjunct1} \quad - \quad - \quad - \quad h)) \]

\[ \text{iffD1 (P): \lambda p. \; p} \]
\[ \lambda (c: -) \; Q \; P \; p \; (h: -). \; \text{mp} \quad - \quad - \quad - \quad (\text{conjunct1} \quad - \quad - \quad - \quad h) \]

\[ \text{iffD1 (Q): Null} \]
\[ \lambda (c: -) \; Q \; P \; q1 \; (h: -) \; q2. \]
\[ \text{mp} \quad - \quad - \quad - \quad (\text{spec} \quad - \quad q2 \quad c \quad (\text{conjunct1} \quad - \quad - \quad - \quad h)) \]

\[ \text{iffD1: Null iffD1} \]

\[ \text{iffD2 (P, Q): snd} \]
\[ \lambda (d: -) \; (c: -) \; P \; Q \; pq \; (h: -) \; q. \]
\[ \text{mp} \quad - \quad - \quad - \quad (\text{spec} \quad - \quad q \quad d \quad (\text{conjunct2} \quad - \quad - \quad - \quad h)) \]

\[ \text{iffD2 (P): \lambda p. \; p} \]
\[ \lambda (c: -) \; P \; Q \; p \; (h: -). \; \text{mp} \quad - \quad - \quad - \quad (\text{conjunct2} \quad - \quad - \quad - \quad h) \]
theory "Record"
imports Quickcheck-Exhaustive
keywords record :: thy-defn and
print-record :: diag
begin

86 Extensible records with structural subtyping

theory Record
imports Quickcheck-Exhaustive
keywords record :: thy-defn and
print-record :: diag
begin

86.1 Introduction

Records are isomorphic to compound tuple types. To implement efficient records, we make this isomorphism explicit. Consider the record access/update simplification \( \alpha (\beta \text{-update } f \ 	ext{rec}) = \alpha \ 	ext{rec} \) for distinct fields \( \alpha \) and \( \beta \) of some record \( \text{rec} \) with \( n \) fields. There are \( n^2 \) such theorems, which prohibits storage of all of them for large \( n \). The rules can be proved on the fly by case decomposition and simplification in \( O(n) \) time. By creating \( O(n) \) isomorphic-tuple types while defining the record, however, we
can prove the access/update simplification in $O(\log(n)^2)$ time.

The $O(n)$ cost of case decomposition is not because $O(n)$ steps are taken, but rather because the resulting rule must contain $O(n)$ new variables and an $O(n)$ size concrete record construction. To sidestep this cost, we would like to avoid case decomposition in proving access/update theorems.

Record types are defined as isomorphic to tuple types. For instance, a record type with fields `a`, `b`, `c` and `d` might be introduced as isomorphic to `a × (b × (c × d))`. If we balance the tuple tree to `(a × b) × (c × d)` then accessors can be defined by converting to the underlying type then using $O(\log(n))$ fst or snd operations. Updators can be defined similarly, if we introduce a `fst-update` and `snd-update` function. Furthermore, we can prove the access/update theorem in $O(\log(n))$ steps by using simple rewrites on `fst`, `snd`, `fst-update` and `snd-update`.

The catch is that, although $O(\log(n))$ steps were taken, the underlying type we converted to is a tuple tree of size $O(n)$. Processing this term type wastes performance. We avoid this for large $n$ by taking each subtree of size $K$ and defining a new type isomorphic to that tuple subtree. A record can now be defined as isomorphic to a tuple tree of these $O(n/K)$ new types, or, if $n > K*K$, we can repeat the process, until the record can be defined in terms of a tuple tree of complexity less than the constant $K$.

If we prove the access/update theorem on this type with the analogous steps to the tuple tree, we consume $O(\log(n)^2)$ time as the intermediate terms are $O(\log(n))$ in size and the types needed have size bounded by $K$. To enable this analogous traversal, we define the functions seen below: `iso-tuple-fst`, `iso-tuple-snd`, `iso-tuple-fst-update` and `iso-tuple-snd-update`. These functions generalise tuple operations by taking a parameter that encapsulates a tuple isomorphism. The rewrites needed on these functions now need an additional assumption which is that the isomorphism works.

These rewrites are typically used in a structured way. They are here presented as the introduction rule `isomorphic-tuple.intros` rather than as a rewrite rule set. The introduction form is an optimisation, as net matching can be performed at one term location for each step rather than the simplifier searching the term for possible pattern matches. The rule set is used as it is viewed outside the locale, with the locale assumption (that the isomorphism is valid) left as a rule assumption. All rules are structured to aid net matching, using either a point-free form or an encapsulating predicate.

### 86.2 Operators and lemmas for types isomorphic to tuples

datatype (dead 'a, dead 'b, dead 'c) tuple-isomorphism =
  Tuple-Isomorphism 'a ⇒ 'b × 'c × 'b × 'c ⇒ 'a

primrec
THEORY “Record”

repr :: ('a, 'b, 'c) tuple-isomorphism ⇒ 'a ⇒ 'b × 'c where
repr (Tuple-Isomorphism r a) = r

primrec
abst :: ('a, 'b, 'c) tuple-isomorphism ⇒ 'b × 'c ⇒ 'a where
abst (Tuple-Isomorphism r a) = a

definition
iso-tuple-fst :: ('a, 'b, 'c) tuple-isomorphism ⇒ 'b ⇒ 'a where
iso-tuple-fst isom = fst o repr isom

definition
iso-tuple-snd :: ('a, 'b, 'c) tuple-isomorphism ⇒ 'c ⇒ 'a where
iso-tuple-snd isom = snd o repr isom

definition
iso-tuple-fst-update ::
  ('a, 'b, 'c) tuple-isomorphism ⇒ ('b ⇒ 'b) ⇒ ('a ⇒ 'a) where
iso-tuple-fst-update isom f = abst isom ◦ apfst f ◦ repr isom

definition
iso-tuple-snd-update ::
  ('a, 'b, 'c) tuple-isomorphism ⇒ ('c ⇒ 'c) ⇒ ('a ⇒ 'a) where
iso-tuple-snd-update isom f = abst isom ◦ apsnd f ◦ repr isom

definition
iso-tuple-cons ::
  ('a, 'b, 'c) tuple-isomorphism ⇒ 'b ⇒ 'c ⇒ 'a where
iso-tuple-cons isom = curry (abst isom)

86.3 Logical infrastructure for records

definition
iso-tuple-surjective-proof-assist :: 'a ⇒ 'b ⇒ ('a ⇒ 'b) ⇒ bool where
iso-tuple-surjective-proof-assist x y f ←→ f x = y

definition
iso-tuple-update-accessor-cong-assist ::
  (('b ⇒ 'b) ⇒ ('a ⇒ 'a)) ⇒ ('a ⇒ 'b) ⇒ bool where
iso-tuple-update-accessor-cong-assist upd ac ←→
  (∀ f v. upd (λ x. f (ac v)) v = upd f v) ∧ (∀ v. upd id v = v)

definition
iso-tuple-update-accessor-eq-assist ::
  (('b ⇒ 'b) ⇒ ('a ⇒ 'a)) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ ('b ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ bool
where
iso-tuple-update-accessor-eq-assist upd ac v f v' x ←→
  upd f v = v' ∧ ac v = x ∧ iso-tuple-update-accessor-cong-assist upd ac
lemma update-accessor-congruence-foldE:
  assumes uac: iso-tuple-update-accessor-cong-assist upd ac
  and r: r = r' and v: ac r' = v'
  and f: \( \forall v, v' \in v = f v = f' v \)
  shows upd f r = upd f' r'
  using uac r v [symmetric]
  apply (subgoal_tac upd (\( \lambda x. f (ac r') \)) r' = upd (\( \lambda x. f' (ac r') \)) r')
  apply (simp add: iso-tuple-update-accessor-cong-assist-def)
  apply (simp add: f)
  done

lemma update-accessor-congruence-unfoldE:
  iso-tuple-update-accessor-cong-assist upd ac \Rightarrow\n  r = r' \Rightarrow ac r' = v' \Rightarrow (\( \forall v, v' \in v = f v = f' v \)) \Rightarrow\n  upd f r = upd f' r'
  apply (erule (2) update-accessor-congruence-foldE)
  apply simp
  done

lemma iso-tuple-update-accessor-cong-assist-id:
  iso-tuple-update-accessor-cong-assist upd ac \Rightarrow\n  upd id = id
  by rule (simp add: iso-tuple-update-accessor-cong-assist-def)

lemma update-accessor-noopE:
  assumes uac: iso-tuple-update-accessor-cong-assist upd ac
  and ac: f (ac x) = ac x
  shows upd f x = x
  using uac
  by (simp add: ac iso-tuple-update-accessor-cong-assist-id [OF uac, unfolded id-def]
    cong: update-accessor-congruence-unfoldE [OF uac])

lemma update-accessor-noop-compE:
  assumes uac: iso-tuple-update-accessor-cong-assist upd ac
  and ac: f (ac x) = ac x
  shows upd (g o f) x = upd g x
  by (simp add: ac cong: update-accessor-congruence-unfoldE[OF uac])

lemma update-accessor-cong-assist-idI:
  iso-tuple-update-accessor-cong-assist id id
  by (simp add: iso-tuple-update-accessor-cong-assist-def)

lemma update-accessor-cong-assist-triv:
  iso-tuple-update-accessor-cong-assist upd ac \Rightarrow\n  iso-tuple-update-accessor-cong-assist upd ac
  by assumption

lemma update-accessor-accessor-eqE:
  iso-tuple-update-accessor-eq-assist upd ac v f v' x \Rightarrow ac v = x
  by (simp add: iso-tuple-update-accessor-eq-assist-def)
lemma update-accessor-updator-eqE:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies upd f v = v'
by (simp add: iso-tuple-update-accessor-eq-assist-def)

lemma iso-tuple-update-accessor-eq-assist-idI:
v' = f v \implies iso-tuple-update-accessor-eq-assist id id v v'
by (simp add: iso-tuple-update-accessor-eq-assist-def update-accessor-cong-assist-idI)

lemma iso-tuple-update-accessor-eq-assist-triv:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies
iso-tuple-update-accessor-eq-assist upd ac v f v' x
by assumption

lemma iso-tuple-update-accessor-cong-from-eq:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies
iso-tuple-update-accessor-cong-assist upd ac
by (simp add: iso-tuple-update-accessor-eq-assist-def)

lemma iso-tuple-surjective-proof-assistI:
f x = y \implies iso-tuple-surjective-proof-assist x y f
by (simp add: iso-tuple-surjective-proof-assist-def)

lemma iso-tuple-surjective-proof-assist-idE:
iso-tuple-surjective-proof-assist x y id \implies x = y
by (simp add: iso-tuple-surjective-proof-assist-def)

locale isomorphic-tuple =
 fixes isom :: ('a, 'b, 'c) tuple-isomorphism
 assumes repr-inv: \(\forall x.\) abst isom (repr isom x) = x
 and abst-inv: \(\forall y.\) repr isom (abst isom y) = y
begin

lemma repr-inj: repr isom x = repr isom y \iff x = y
by (auto dest: arg-cong [of repr isom x repr isom y abst isom]
simp add: repr-inv)

lemma abst-inj: abst isom x = abst isom y \iff x = y
by (auto dest: arg-cong [of abst isom x abst isom y repr isom]
simp add: abst-inv)

lemmas simps = Let-def repr-inv abst-inv repr-inj abst-inj

lemma iso-tuple-access-update-fst-fst:
f \circ h g = j \circ f \implies
(f \circ iso-tuple-fst isom) \circ (iso-tuple-fst-update isom \circ h) g =
j \circ (f \circ iso-tuple-fst isom)
by (clarsimp simp: iso-tuple-fst-update-def iso-tuple-fst-def simps fun-eq-iff)
lemma iso-tuple-access-update-snd-snd:
\[ f \circ h \circ g = j \circ f = j \circ (f \circ iso-tuple-snd) \]
by (clarsimp simp: iso-tuple-snd-update-def iso-tuple-snd-def simps fun-eq-iff)

lemma iso-tuple-access-update-fst-snd:
\[ (f \circ iso-tuple-fst) \circ (iso-tuple-snd-update) \circ (f \circ iso-tuple-snd) \circ h \circ g = \]
\[ id \circ (f \circ iso-tuple-fst) \circ h \circ g = \]
by (clarsimp simp: iso-tuple-snd-update-def iso-tuple-fst-def simps fun-eq-iff)

lemma iso-tuple-access-update-snd-fst:
\[ (f \circ iso-tuple-snd) \circ (iso-tuple-fst-update) \circ (f \circ iso-tuple-fst) \circ h \circ g = \]
\[ id \circ (f \circ iso-tuple-snd) \circ (iso-tuple-fst-update) \circ (f \circ iso-tuple-fst) \circ h \circ g = \]
by (clarsimp simp: iso-tuple-fst-update-def iso-tuple-snd-def simps fun-eq-iff)

lemma iso-tuple-update-swap-fst-fst:
\[ (iso-tuple-fst-update) \circ (h \circ f) \circ (j \circ h) \circ g = \]
\[ j \circ g \circ (iso-tuple-fst-update) \circ (h \circ f) \circ g = \]
by (clarsimp simp: iso-tuple-fst-update-def simps fun-eq-iff)

lemma iso-tuple-update-swap-snd-snd:
\[ (iso-tuple-snd-update) \circ (h \circ f) \circ (j \circ h) \circ g = \]
\[ j \circ g \circ (iso-tuple-snd-update) \circ (h \circ f) \circ g = \]
by (clarsimp simp: iso-tuple-snd-update-def simps fun-eq-iff)

lemma iso-tuple-update-swap-fst-fst:
\[ (iso-tuple-fst-update) \circ (h \circ f) \circ (j \circ h) \circ g = \]
\[ j \circ g \circ (iso-tuple-fst-update) \circ (h \circ f) \circ g = \]
by (clarsimp simp: iso-tuple-fst-update-def simps fun-eq-iff)

lemma iso-tuple-update-compose-fst-fst:
\[ (iso-tuple-fst-update) \circ (h \circ f) \circ (j \circ g) = \]
\[ k \circ f \circ (iso-tuple-fst-update) \circ (j \circ g) = \]
by (clarsimp simp: iso-tuple-fst-update-def simps fun-eq-iff)
lemma iso-tuple-update-compose-snd-snd:
\[ (f \circ g) \circ (h \circ j) = (f \circ (h \circ j)) \circ g \]
by (clarsimp simp: isom definitions apsnd compose_fun_eq_iff)

lemma iso-tuple-surjective-proof-assist-step:
\[ (f \circ g) \circ h = (f \circ (g \circ h)) \]
by (clarsimp simp: isom definitions)

lemma iso-tuple-fst-update-accessor-cong-assist:
\[ (f \circ g) = (f \circ h) \]
by (clarsimp simp: isom definitions)

lemma iso-tuple-snd-update-accessor-cong-assist:
\[ (f \circ g) = (f \circ h) \]
by (clarsimp simp: isom definitions)

lemma iso-tuple-fst-update-accessor-eq-assist:
\[ (f \circ g) = (f \circ h) \]
by (clarsimp simp: isom definitions)

qed
iso-tuple-fst-update-def iso-tuple-fst-def
iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def simps)
qed

lemma iso-tuple-snd-update-accessor-eq-assist:
  assumes iso-tuple-update-accessor-eq-assist f g b u b' v
  shows iso-tuple-update-accessor-eq-assist
    (iso-tuple-snd-update isom \circ f) \ (g \circ iso-tuple-snd isom)
    (iso-tuple-cons isom a b) u \ (iso-tuple-cons isom a b') v
proof
  from assms have f id = id
  by (auto simp add: iso-tuple-update-accessor-eq-assist-def
       intro: iso-tuple-update-accessor-cong-assist-id)
  with assms show \?thesis
  by (clarsimp simp: iso-tuple-update-accessor-eq-assist-def
       iso-tuple-snd-update-def iso-tuple-snd-def
       iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def simps)
qed

lemma iso-tuple-cons-conj-eqI:
  a = c \land b = d \land P \iff Q
  iso-tuple-cons isom a b = iso-tuple-cons isom c d \land P \iff Q
by (clarsimp simp: iso-tuple-cons-def simps)

lemmas intros =
 iso-tuple-access-update-fst-fst
 iso-tuple-access-update-snd-snd
 iso-tuple-access-update-fst-snd
 iso-tuple-access-update-snd-fst
 iso-tuple-update-swap-fst-fst
 iso-tuple-update-swap-snd-snd
 iso-tuple-update-swap-snd-fst
 iso-tuple-update-swap-fst-snd
 iso-tuple-update-compose-fst-fst
 iso-tuple-update-compose-snd-snd
 iso-tuple-surjective-proof-assist-step
 iso-tuple-fst-update-accessor-eq-assist
 iso-tuple-snd-update-accessor-eq-assist
 iso-tuple-fst-update-accessor-cong-assist
 iso-tuple-snd-update-accessor-cong-assist
 iso-tuple-cons-conj-eqI

end

lemma isomorphic-tuple-intro:
  fixes repr abst
  assumes repr-inj: \land x y. \text{repr} x = \text{repr} y \iff x = y
  and abst-inv: \land z. \text{repr} (\text{abst} z) = z
  and v: v \equiv \text{Tuple-Isomorphism} \text{repr} \text{abst}
shows isomorphic-tuple v

proof
  fix x have repr (abst (repr x)) = repr x
    by (simp add: abst-inv)
  then show Record.abst v (Record.repr v x) = x
    by (simp add: v repr-inj)
next
  fix y
  show Record.repr v (Record.abst v y) = y
    by (simp add: v) (fact abst-inv)
qed

definition
tuple-iso-tuple ≡ Tuple-Isomorphism id id

lemma tuple-iso-tuple:
  isomorphic-tuple tuple-iso-tuple
  by (simp add: isomorphic-tuple-intro [OF - - reflexive] tuple-iso-tuple-def)

lemma refl-conj-eq: Q = R =⇒ P ∧ Q =⇒ P ∧ R
  by simp

lemma iso-tuple-UNIV-I: x ∈ UNIV =⇒ True
  by simp

lemma iso-tuple-True-simp: (True =⇒ PROP P) =⇒ PROP P
  by simp

lemma prop-subst: s = t =⇒ PROP P t =⇒ PROP P s
  by simp

lemma K-record-comp: (λx. c) ◦ f = (λx. c)
  by (simp add: comp-def)

86.4 Concrete record syntax

nonterminal
  ident and
  field-type and
  field-types and
  field and
  fields and
  field-update and
  field-updates

syntax
  -constify :: id => ident (-)
  -constify :: longid => ident (-)
THEORY "GCD"

86.5 Record package

ML-file ⟨Tools/record.ML⟩

hide-const (open) Tuple-Isomorphism repr abst iso-tuple-fst iso-tuple-snd
iso-tuple-fst-update iso-tuple-snd-update iso-tuple-cons
iso-tuple-surjective-proof-assist iso-tuple-update-accessor-cong-assist
iso-tuple-update-accessor-eq-assist tuple-iso-tuple

end

87 Greatest common divisor and least common multiple

theory GCD
  imports Groups-List Code-Numeral
begin

87.1 Abstract bounded quasi semilattices as common foundation

locale bounded-quasi-semilattice = abel-semigroup +
  fixes top :: 'a (⊤) and bot :: 'a (⊥)
  and normalize :: 'a ⇒ 'a
assumes idem-normalize [simp]: \( a \ast a = \text{normalize } a \)
and normalize-left-idem [simp]: \( \text{normalize } a \ast b = a \ast b \)
and normalize-idem [simp]: \( \text{normalize } (a \ast b) = a \ast b \)
and normalize-top [simp]: \( \text{normalize } \top = \top \)
and normalize-bottom [simp]: \( \text{normalize } \bot = \bot \)
and top-left-normalize [simp]: \( \top \ast a = \text{normalize } a \)
and bottom-left-bottom [simp]: \( \bot \ast a = \bot \)

begin

lemma left-idem [simp]:
\( a \ast (a \ast b) = a \ast b \)
using assoc [of a a b], symmetric by simp

lemma right-idem [simp]:
\( (a \ast b) \ast b = a \ast b \)
using left-idem [of b a] by (simp add: ac-simps)

lemma comp-fun-idem: comp-fun-idem f
by standard (simp-all add: fun-eq-iff ac-simps)

interpretation comp-fun-idem f
by (fact comp-fun-idem)

lemma top-right-normalize [simp]:
\( a \ast \top = \text{normalize } a \)
using top-left-normalize [of a] by (simp add: ac-simps)

lemma bottom-right-bottom [simp]:
\( a \ast \bot = \bot \)
using bottom-left-bottom [of a] by (simp add: ac-simps)

lemma normalize-right-idem [simp]:
\( a \ast \text{normalize } b = a \ast b \)
using normalize-left-idem [of b a] by (simp add: ac-simps)

end

locale bounded-quasi-semilattice-set = bounded-quasi-semilattice begin

interpretation comp-fun-idem f
by (fact comp-fun-idem)

definition F :: \\
\( 'a \text{ set } \Rightarrow 'a \)
where
eq-fold: \( F A = (\text{if } \text{finite } A \text{ then } \text{Finite-Set.fold } f \top A \text{ else } \bot ) \)

lemma infinite [simp]:
infinite A \( \Rightarrow \) \( F A = \bot \)
by (simp add: eq-fold)

lemma set-eq-fold [code]:
F (set xs) = fold f xs ⊤
by (simp add: eq-fold fold-set-fold)

lemma empty [simp]:
F {} = ⊤
by (simp add: eq-fold)

lemma insert [simp]:
F (insert a A) = a * F A
by (cases finite A) (simp-all add: eq-fold)

lemma normalize [simp]:
normalize (F A) = F A
by (induct A rule: infinite-finite-induct) simp-all

lemma in-idem:
assumes a ∈ A
shows a * F A = F A
using assms by (induct A rule: infinite-finite-induct)
(auto simp: left-commute [of a])

lemma union:
F (A ∪ B) = F A * F B
by (induct A rule: infinite-finite-induct)
(simp-all add: ac-simps)

lemma remove:
assumes a ∈ A
shows F A = a * F (A - {a})
proof -
from assms obtain B where A = insert a B and a ∉ B
  by (blast dest: mk-disjoint-insert)
with assms show ?thesis by simp
qed

lemma insert-remove:
F (insert a A) = a * F (A - {a})
by (cases a ∈ A) (simp-all add: insert-absorb remove)

lemma subset:
assumes B ⊆ A
shows F B * F A = F A
using assms by (simp add: union [symmetric] Un-absorb1)

end
87.2 Abstract GCD and LCM

class \texttt{gcd} = zero + one + dvd +
fixes \texttt{gcd} :: 'a ⇒ 'a ⇒ 'a 
and \texttt{lem} :: 'a ⇒ 'a ⇒ 'a

class \texttt{Gcd} = \texttt{gcd} +
fixes \texttt{Gcd} :: 'a set ⇒ 'a 
and \texttt{Lcm} :: 'a set ⇒ 'a

syntax
\begin{itemize}
\item \texttt{-GCD1} :: pttrns ⇒ 'b ⇒ 'b \quad ((\texttt{GCD} -./ -) [0, 10])
\item \texttt{-GCD} :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b \quad ((\texttt{GCD} -./ -) [0, 0, 10])
\item \texttt{-LCM1} :: pttrns ⇒ 'b ⇒ 'b \quad ((\texttt{LCM} -./ -) [0, 10])
\item \texttt{-LCM} :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b \quad ((\texttt{LCM} -./ -) [0, 0, 10])
\end{itemize}

translations
\begin{itemize}
\item \texttt{GCD x y. f} \quad \Rightarrow \quad \texttt{GCD x. GCD y. f}
\item \texttt{GCD x. f} \quad \Rightarrow \quad \texttt{CONST Gcd \ (CONST range \ (λx. f))}
\item \texttt{GCD x∈A. f} \quad \Rightarrow \quad \texttt{CONST Gcd \ ((λx. f) · A)}
\item \texttt{LCM x y. f} \quad \Rightarrow \quad \texttt{LCM x. LCM y. f}
\item \texttt{LCM x. f} \quad \Rightarrow \quad \texttt{CONST Lcm \ (CONST range \ (λx. f))}
\item \texttt{LCM x∈A. f} \quad \Rightarrow \quad \texttt{CONST Lcm \ ((λx. f) · A)}
\end{itemize}

class \texttt{semiring-gcd} = normalization-semidom + gcd +
assumes \texttt{gcd-dvd1 [iff]}: \texttt{gcd a b dvd a}
and \texttt{gcd-dvd2 [iff]}: \texttt{gcd a b dvd b}
and \texttt{gcd-greatest} \ (\texttt{c dvd a \implies c dvd b \implies c dvd gcd a b})
and \texttt{normalize-gcd [simp]}; \texttt{normalize \ (gcd a b) = gcd a b}
and \texttt{lcm-gcd}; \texttt{lcm a b = normalize \ (a * b div gcd a b)}

begin

lemma \texttt{gcd-greatest-iff [simp]}: \texttt{a dvd gcd b c \iff a dvd b \land a dvd c}
by (\texttt{blast intro!}; \texttt{gcd-greatest intro; dvd-trans})

lemma \texttt{gcd-dvdI1}: \texttt{a dvd c \implies gcd a b dvd c}
by (\texttt{rule dvd-trans}) (\texttt{rule gcd-dvd1})

lemma \texttt{gcd-dvdI2}: \texttt{b dvd c \implies gcd a b dvd c}
by (\texttt{rule dvd-trans}) (\texttt{rule gcd-dvd2})

lemma \texttt{dvd-gcdD1}: \texttt{a dvd gcd b c \implies a dvd b}
using \texttt{gcd-dvd1 [of b c]} by (\texttt{blast intro!}; \texttt{dvd-trans})

lemma \texttt{dvd-gcdD2}: \texttt{a dvd gcd b c \implies a dvd c}
using \texttt{gcd-dvd2 [of b c]} by (\texttt{blast intro!}; \texttt{dvd-trans})

lemma \texttt{gcd-0-left [simp]}: \texttt{gcd 0 a = normalize a}
by (\texttt{rule associated-eqI}) \texttt{simp-all}
lemma gcd-0-right [simp]: \( \gcd a \ 0 = \text{normalize } a \) 
by (rule associated-eqI) simp-all

lemma gcd-eq-0-iff [simp]: \( \gcd a \ b = 0 \iff a = 0 \land b = 0 \)
(is \( ?P \iff ?Q \))

proof
  assume \( ?P \)
  then have \( 0 \ \text{dvd } \gcd a \ b \)
    by simp
  then have \( 0 \ \text{dvd } a \ \text{and } 0 \ \text{dvd } b \)
    by (blast intro: dvd-trans)+
  then show \( ?Q \)
    by simp
next
  assume \( ?Q \)
  then show \( ?P \)
    by simp
qed

lemma unit-factor-gcd: \( \text{unit-factor } (\gcd a \ b) = (\text{if } a = 0 \land b = 0 \ then \ 0 \ else \ 1) \)

proof (cases \( \gcd a \ b = 0 \))
  case True
  then show \( ?\text{thesis} \) by simp
next
  case False
  have \( \text{unit-factor } (\gcd a \ b) * \text{normalize } (\gcd a \ b) = \gcd a \ b \)
    by (rule unit-factor-mult-normalize)
  then have \( \text{unit-factor } (\gcd a \ b) * \gcd a \ b = \gcd a \ b \)
    by simp
  then have \( \text{unit-factor } (\gcd a \ b) * \gcd a \ b \ \text{div } \gcd a \ b = \gcd a \ b \ \text{div } \gcd a \ b \)
    by simp
  with False show \( ?\text{thesis} \)
    by simp
qed

lemma is-unit-gcd-iff [simp]:
  \( \text{is-unit } (\gcd a \ b) \iff \gcd a \ b = 1 \)

by (cases \( a = 0 \land b = 0 \)) (auto simp: unit-factor-gcd dest: is-unit-unit-factor)

sublocale gcd: abel-semigroup gcd

proof
  fix \( a \ b \ c \)
  show \( \gcd a \ b = \gcd b \ a \)
    by (rule associated-eqI) simp-all
  from gcd-dvd1 have \( \gcd (\gcd a \ b) \ c \ \text{dvd } a \)
    by (rule dvd-trans) simp
  moreover from gcd-dvd1 have \( \gcd (\gcd a \ b) \ c \ \text{dvd } b \)
    by (rule dvd-trans) simp
  ultimately have \( P1: \gcd (\gcd a \ b) \ c \ \text{dvd } \gcd a \ (\gcd b \ c) \)
by (auto intro!: gcd-greatest)

from gcd-dvd2 have gcd a (gcd b c) dvd b
  by (rule dvd-trans) simp

moreover from gcd-dvd2 have gcd a (gcd b c) dvd c
  by (rule dvd-trans) simp

ultimately have P2: gcd a (gcd b c) dvd (gcd a b) c
  by (auto intro!: gcd-greatest)

from P1 P2 show gcd (gcd a b) c = gcd a (gcd b c)
  by (rule associated-eqI) simp-all

qed

sublocale gcd: bounded-quasi-semilattice gcd 0 1 normalize
proof
  show gcd a a = normalize a for a
    proof
      have a dvd gcd a a
        by (rule gcd-greatest) simp-all
      then show ?thesis
        by (auto intro: associated-eqI)
    qed

  show gcd (normalize a) b = gcd a b for a b
    using gcd-dvd1 [of normalize a b]
    by (auto intro: associated-eqI)

  show gcd 1 a = 1 for a
    by (rule associated-eqI) simp-all

qed simp-all

lemma gcd-self: gcd a a = normalize a
  by (fact gcd.idem-normalize)

lemma gcd-left-idem: gcd a (gcd a b) = gcd a b
  by (fact gcd.left-idem)

lemma gcd-right-idem: gcd (gcd a b) b = gcd a b
  by (fact gcd.right-idem)

lemma gcdI:
  assumes c dvd a and c dvd b
  and greatest: (∀d. d dvd a → d dvd b → d dvd c)
  and normalize c = c
  shows c = gcd a b
  by (rule associated-eqI) (auto simp: assms intro: gcd-greatest)

lemma gcd-unique:
  d dvd a ∧ d dvd b ∧ normalize d = d ∧ (∀e. e dvd a ∧ e dvd b → e dvd d) ⟷ d = gcd a b
  by rule (auto intro: gcdI simp: gcd-greatest)

lemma gcd-dvd-prod: gcd a b dvd k * b
using mult-dvd-mono [of 1] by auto

lemma gcd-proj2-if-dvd: b dvd a ⟹ gcd a b = normalize b
  by (rule gcdI [symmetric]) simp-all

lemma gcd-proj1-if-dvd: a dvd b ⟹ gcd a b = normalize a
  by (rule gcdI [symmetric]) simp-all

lemma gcd-proj1-iff: gcd m n = normalize m ⟷ m dvd n
proof
  assume *: gcd m n = normalize m
  show m dvd n
  proof (cases m = 0)
    case True
    with * show ?thesis by simp
  next
    case False
    then have *: c * gcd a b dvd gcd (c * a) (c * b)
      by (auto intro: gcd-greatest)
  moreover from False * have gcd (c * a) (c * b) dvd c * gcd a b
    by (metis div-dvd-iff-mult dvd-mult-left gcd-dvd1 gcd-dvd2 gcd-greatest mult-commute)
  ultimately have normalize (gcd (c * a) (c * b)) = normalize (c * gcd a b)
    by (auto intro: associated-eq1)
  then show ?thesis
    by (simp add: normalize-mult)
next

lemma gcd-mult-left: gcd (c * a) (c * b) = normalize (c * gcd a b)
proof (cases c = 0)
  case True
  then show ?thesis by simp
next
  case False
  then have *: c * gcd a b dvd gcd (c * a) (c * b)
    by (auto intro: gcd-greatest)
  moreover from False * have gcd (c * a) (c * b) dvd c * gcd a b
    by (metis div-dvd-iff-mult dvd-mult-left gcd-dvd1 gcd-dvd2 gcd-greatest mult-commute)
  ultimately have normalize (gcd (c * a) (c * b)) = normalize (c * gcd a b)
    by (auto intro: associated-eq1)
  then show ?thesis
    by (simp add: normalize-mult)
qed

lemma gcd-mult-right: gcd (a * c) (b * c) = normalize (gcd b a * c)
using gcd-mult-left [of c a b] by (simp add: ac-simps)
lemma dvd-lcm1 [iff]: a dvd lcm a b
  by (metis div-mult-swap dvd-mult2 dvd-normalize-iff dvd-refl gcd-dvd2 lcm-gcd)

lemma dvd-lcm2 [iff]: b dvd lcm a b
  by (metis dvd-div-mul dvd-mult dvd-normalize-iff dvd-refl gcd-dvd1 lcm-gcd)

lemma dvd-lcmI1: a dvd b ® a dvd lcm b c
  by (rule dvd-trans) (assumption, blast)

lemma dvd-lcmI2: a dvd c ® a dvd lcm b c
  by (rule dvd-trans) (assumption, blast)

lemma lcm-dvdD1: lcm a b dvd c ® a dvd c
  using dvd-lcm1 [of a b]
  by (blast intro: dvd-trans)

lemma lcm-dvdD2: lcm a b dvd c ® b dvd c
  using dvd-lcm2 [of a b]
  by (blast intro: dvd-trans)

lemma lcm-least:
  assumes a dvd c and b dvd c
  shows lcm a b dvd c
proof (cases c = 0)
  case True
  then show ?thesis by simp
next
  case False
  then have *: is-unit (unit-factor c)
    by simp
  show ?thesis
proof (cases gcd a b = 0)
  case True
  with assms show ?thesis by simp
next
  case False
  have a * b dvd normalize (c * gcd a b)
    using assms by (subst gcd-mult-left [symmetric]) (auto intro!: gcd-greatest simp: mult-ac)
  with False have (a * b div gcd a b) dvd c
    by (subst div-dvd-iff-mult) auto
  thus ?thesis by (simp add: lcm-gcd)
qued

lemma lcm-least-iff [simp]: lcm a b dvd c <-> a dvd c ∧ b dvd c
  by (blast intro!: lcm-least intro: dvd-trans)

lemma normalize-lcm [simp]: normalize (lcm a b) = lcm a b
  by (simp add: lcm-gcd dvd-normalize-div)
lemma lcm-0-left [simp]: lcm 0 a = 0
  by (simp add: lcm-gcd)

lemma lcm-0-right [simp]: lcm a 0 = 0
  by (simp add: lcm-gcd)

lemma lcm-eq-0-iff: lcm a b = 0 ↔ a = 0 ∨ b = 0
  (is ?P ↔ ?Q)
proof
  assume ?P
  then have 0 dvd lcm a b
    by simp
  also have lcm a b dvd (a * b)
    by simp
  finally show ?Q
    by auto
next
  assume ?Q
  then show ?P
    by auto
qed

lemma zero-eq-lcm-iff: 0 = lcm a b ↔ a = 0 ∨ b = 0
  using lcm-eq-0-iff [of a b] by auto

lemma lcm-eq-1-iff [simp]: lcm a b = 1 ↔ is-unit a ∧ is-unit b
  by (auto intro: associated-eqI)

lemma unit-factor-lcm: unit-factor (lcm a b) = (if a = 0 ∨ b = 0 then 0 else 1)
  using lcm-eq-0-iff [of a b] by (cases lcm a b = 0) (auto simp: lcm-gcd)

sublocale lcm: abel-semigroup lcm
proof
  fix a b c
  show lcm a b = lcm b a
    by (simp add: lcm-gcd ac-simps normalize-mult dvd-normalize-div)
  have lcm (lcm a b) c dvd lcm a (lcm b c)
    and lcm a (lcm b c) dvd lcm (lcm a b) c
    by (auto intro: lcm-least
      dvd-trans [of b lcm b c lcm a (lcm b c)]
      dvd-trans [of c lcm b c lcm a (lcm b c)]
      dvd-trans [of a lcm a b lcm (lcm a b) c]
      dvd-trans [of b lcm a b lcm (lcm a b) c])
  then show lcm (lcm a b) c = lcm a (lcm b c)
    by (rule associated-eqI simp-all)
qed

sublocale lcm: bounded-quasi-semilattice lcm 1 0 normalize
proof
d show lcm a a = normalize a for a
proof --
  have lcm a a dvd a 
    by (rule lcm-least) simp-all
  then show ?thesis 
    by (auto intro: associated-eqI)
qed

d show lcm (normalize a) b = lcm a b for a b 
  using dvd-lcm1 [of normalize a b] unfolding normalize-dvd-iff
  by (auto intro: associated-eqI)

d show lcm 1 a = normalize a for a 
  by (rule associated-eqI)

qed simp-all

lemma lcm-self: lcm a a = normalize a
  by (fact lcm.idem-normalize)

lemma lcm-left-idem: lcm a (lcm a b) = lcm a b
  by (fact lcm.left-idem)

lemma lcm-right-idem: lcm (lcm a b) b = lcm a b
  by (fact lcm.right-idem)

lemma gcd-lcm:
  assumes a ≠ 0 and b ≠ 0
  shows gcd a b = normalize (a * b div lcm a b)
proof --
  from assms have [simp]: a * b div gcd a b ≠ 0 
    by (subst dvd-div-eq-0-iff) auto
  let ?u = unit-factor (a * b div gcd a b)
  have gcd a b * normalize (a * b div gcd a b) =
  gcd a b * ((a * b div gcd a b) * (1 div ?u))
    by simp
  also have . . . = a * b div ?u
    by (subst div-unit-dvd-iff [symmetric]) auto
  also have . . . dvd a * b
    by (subst div-unit-dvd-iff) auto
  finally have gcd a b dvd ((a * b) div lcm a b)
    by (intro dvd-mult-imp-div) (auto simp: lcm-gcd)
  moreover have a * b dvd lcm a b dvd a and a * b dvd lcm a b dvd b 
    using assms by (subst div-dvd-iff-mult; simp add: lcm-eq-0-iff mult.commute[of b lcm a b])+
  ultimately have normalize (gcd a b) = normalize (a * b div lcm a b)
    apply --
    apply (rule associated-eqI)
    using assms
    apply (auto simp: div-dvd-iff-mult zero-eq-lcm-iff)
  done
thus \(?\)thesis by simp

qed

lemma lcm-1-left: \(\text{lcm} \ 1 \ a = \text{normalize} \ a\)
by (fact lcm.top-left-normalize)

lemma lcm-1-right: \(\text{lcm} \ a \ 1 = \text{normalize} \ a\)
by (fact lcm.top-right-normalize)

lemma lcm-mult-left: \(\text{lcm} \ (c * a) \ (c * b) = \text{normalize} \ (c * \text{lcm} \ a \ b)\)
proof (cases \(c = 0\))
case True
then show \(?\)thesis by simp
next
case False
then have \(\ast\): \(\text{lcm} \ (c * a) \ (c * b) \ \text{dvd} \ c * \text{lcm} \ a \ b\)
by (auto intro: lcm-least)
moreover have \(\text{lcm} \ a \ b \ \text{dvd} \ text{lcm} \ (c * a) \ (c * b) \ \text{div} \ c\)
by (intro lcm-least) (auto intro: dvd-mult-div simp: mult-ac)

hence \(c * \text{lcm} \ a \ b \ \text{dvd} \ \text{lcm} \ (c * a) \ (c * b)\)
using False by (subst (asm) dvd-div-iff-mult) (auto simp: mult-ac intro: dvd-lcmI1)
ultimately have \(\text{normalize} \ (\text{lcm} \ (c * a) \ (c * b)) = \text{normalize} \ (c * \text{lcm} \ a \ b)\)
then show \(?\)thesis
by (auto intro: associated-eqI)

qed

lemma lcm-mult-right: \(\text{lcm} \ (a * c) \ (b * c) = \text{normalize} \ (\text{lcm} \ b \ a * c)\)
using lcm-mult-left [of c a b] by (simp add: ac-simps)

lemma lcm-mult-unit1: \(\text{is-unit} \ a \ \implies \text{lcm} \ (b * a) \ c = \text{lcm} \ b \ c\)
by (rule associated-eqI) (simp-all add: mult-unit-dvd-iff dvd-lcmI1)

lemma lcm-mult-unit2: \(\text{is-unit} \ a \ \implies \text{lcm} \ b \ (c * a) = \text{lcm} \ b \ c\)
using lcm-mult-unit1 [of a c b] by (simp add: ac-simps)

lemma lcm-div-unit1: 
\(\text{is-unit} \ a \ \implies \text{lcm} \ (b \ \text{div} \ a) \ c = \text{lcm} \ b \ c\)
by (erule is-unitE [of \(-b\)]) (simp add: lcm-mult-unit1)

lemma lcm-div-unit2: \(\text{is-unit} \ a \ \implies \text{lcm} \ b \ (c \ \text{div} \ a) = \text{lcm} \ b \ c\)
by (erule is-unitE [of \(-c\)]) (simp add: lcm-mult-unit2)

lemma normalize-lcm-left: \(\text{lcm} \ (\text{normalize} \ a) \ b = \text{lcm} \ a \ b\)
by (fact lcm.normalize-left-idem)

lemma normalize-lcm-right: \(\text{lcm} \ a \ (\text{normalize} \ b) = \text{lcm} \ a \ b\)
by (fact lcm.normalize-right-idem)
lemma comp-fun-idem-gcd: comp-fun-idem gcd
  by standard (simp-all add: fun-eq-iff ac-simps)

lemma comp-fun-idem-lcm: comp-fun-idem lcm
  by standard (simp-all add: fun-eq-iff ac-simps)

lemma gcd-dvd-antisym: gcd a b dvd gcd c d → gcd c d dvd gcd a b → gcd a b = gcd c d
proof (rule gcdI)
  assume ∗: gcd a b dvd gcd c d
  and ∗∗: gcd c d dvd gcd a b
  have gcd c d dvd c
    by simp
  with ∗ show gcd a b dvd c
    by (rule dvd-trans)
  have gcd c d dvd d
    by simp
  with ∗ show gcd a b dvd d
    by (rule dvd-trans)
  show normalize (gcd a b) = gcd a b
    by simp
  fix l assume l dvd c and l dvd d
  then have l dvd gcd c d
    by (rule gcd-greatest)
  from this and ∗∗ show l dvd gcd a b
    by (rule dvd-trans)
qed

declare unit-factor-lcm [simp]

lemma lcmI:
  assumes a dvd c and b dvd c and \( d \). a dvd d → b dvd d → c dvd d
  and normalize c = c
  shows c = lcm a b
  by (rule associated-eqI) (auto simp: assms intro: lcm-least)

lemma gcd-dvd-lcm [simp]: gcd a b dvd lcm a b
  using gcd-dvd2 by (rule dvd-lcm12)

lemmas lcm-0 = lcm-0-right

lemma lcm-unique:
  a dvd d ∧ b dvd d ∧ normalize d = d ∧ (\( \forall e. a dvd e ∧ b dvd e \) → d dvd e) \( \leftrightarrow \)
  d = lcm a b
  by rule (auto intro: lcmI simp: lcm-least lcm-eq-iff)

lemma lcm-proj1-if-dvd:
  assumes b dvd a shows lcm a b = normalize a
proof —
have normalize (lcm a b) = normalize a
  by (rule associatedI) (use assms in auto)
thus ?thesis by simp
qed

lemma lcm-proj2-if-dvd: a dvd b \implies lcm a b = normalize b
using lcm-proj1-if-dvd [of a b] by (simp add: ac-simps)

lemma lcm-proj1-iff: lcm m n = normalize m \iff n dvd m
proof
  assume \*: lcm m n = normalize m
  show n dvd m
    proof (cases m = 0)
      case True
      then show ?thesis by simp
    next
      case False
      define x y where x = gcd a p and y = p div x
      have p = x \* y by (simp add: x-def y-def)
      moreover have x dvd a by (simp add: x-def)
      moreover from assms have p dvd gcd (b \* a) (b \* p)
        by (intro gcd-greatest) (simp-all add: mult.commute)
    qed
next
  assume n dvd m
  then show lcm m n = normalize m
    by (rule lcm-proj1-if-dvd)
qed

lemma lcm-proj2-iff: lcm m n = normalize n \iff m dvd n
using lcm-proj1-iff [of n m] by (simp add: ac-simps)

lemma gcd-mono: a dvd c \implies b dvd d \implies gcd a b dvd gcd c d
  by (simp add: gcd-dvdI1 gcd-dvdI2)

lemma lcm-mono: a dvd c \implies b dvd d \implies lcm a b dvd lcm c d
  by (simp add: dvd-lcmI1 dvd-lcmI2)

lemma dvd-productE:
  assumes p dvd a \* b
  obtains x y where p = x \* y x dvd a y dvd b
proof (cases a = 0)
  case True
  thus ?thesis by (intro that[of p 1]) simp-all
next
  case False
  define x y where x = gcd a p and y = p div x
  have p = x \* y by (simp add: x-def y-def)
  moreover have x dvd a by (simp add: x-def)
next
hence \( p \text{ dvd } b \ast \text{gcd } a \) \( p \) by (subst (asm) gcd-mult-left) auto
with False have \( y \text{ dvd } b \)
  by (simp add: \( x \)-def \( y \)-def dvd-dvd-iff-mult assms)
ultimately show \( \theta \text{thesis} \) by (rule that)
qed

lemma \( \text{gcd-mult-unit1} \):
assumes \( \text{is-unit } a \)
shows \( \text{gcd } (b \ast a) c = \text{gcd } b c \)
proof
  have \( \text{gcd } (b \ast a) c \text{ dvd } b \)
    using assms dvd-mult-unit-iff by blast
  then show \( \theta \text{thesis} \)
    by (rule gcdI simp-all)
qed

lemma \( \text{gcd-mult-unit2} \): \( \text{is-unit } a \Rightarrow \text{gcd } b (c \ast a) = \text{gcd } b c \)
using \( \text{gcd}. \text{commute} \text{ gcd-mult-unit1} \) by auto

lemma \( \text{gcd-div-unit1} \): \( \text{is-unit } a \Rightarrow \text{gcd } (b \text{ div } a) c = \text{gcd } b c \)
by (erule is-unitE [of - b]) (simp add: gcd-mult-unit1)

lemma \( \text{gcd-div-unit2} \): \( \text{is-unit } a \Rightarrow \text{gcd } b (c \text{ div } a) = \text{gcd } b c \)
by (erule is-unitE [of - c]) (simp add: gcd-mult-unit2)

lemma \( \text{normalize-gcd-left} \): \( \text{gcd } (\text{normalize } a) b = \text{gcd } a b \)
by (fact gcd-normalize-left-idem)

lemma \( \text{normalize-gcd-right} \): \( \text{gcd } a (\text{normalize } b) = \text{gcd } a b \)
by (fact gcd-normalize-right-idem)

lemma \( \text{gcd-add1} \) [simp]: \( \text{gcd } (m + n) n = \text{gcd } m n \)
by (rule gcdI [symmetric]) (simp-all add: dvd-add-left-iff)

lemma \( \text{gcd-add2} \) [simp]: \( \text{gcd } m (m + n) = \text{gcd } m n \)
using \( \text{gcd-add1} \) [of \( m \) \( n \)] by (simp add: ac-simps)

lemma \( \text{gcd-add-mult} \): \( \text{gcd } m (k \ast m + n) = \text{gcd } m n \)
by (rule gcdI [symmetric]) (simp-all add: dvd-add-right-iff)

end

class \( \text{ring-gcd} = \text{comm-ring-1} + \text{semiring-gcd} \)
begin

lemma \( \text{gcd-neg1} \) [simp]: \( \text{gcd } (-a) b = \text{gcd } a b \)
by (rule sym, rule gcdI) (simp-all add: gcd-greatest)

lemma \( \text{gcd-neg2} \) [simp]: \( \text{gcd } a (-b) = \text{gcd } a b \)
by (rule sym, rule gcdI) (simp-all add: gcd-greatest)
lemma gcd-neg-numeral-1 [simp]: gcd (− numeral n) a = gcd (numeral n) a  
  by (fact gcd-neg1)

lemma gcd-neg-numeral-2 [simp]: gcd a (− numeral n) = gcd a (numeral n)  
  by (fact gcd-neg2)

lemma gcd-diff1: gcd (m − n) n = gcd m n  
  by (subst diff-conv-add-uminus, subst gcd-neg2[symmetric], subst gcd-add1, simp)

lemma gcd-diff2: gcd (n − m) n = gcd m n  
  by (subst gcd-neg1[symmetric]) (simp only: minus-diff-eq gcd-diff1)

lemma lcm-neg1 [simp]: lcm (− a) b = lcm a b  
  by (rule sym, rule lcmI) (simp-all add: lcm-least lcm-eq-0-iff)

lemma lcm-neg2 [simp]: lcm a (− b) = lcm a b  
  by (rule sym, rule lcmI) (simp-all add: lcm-least lcm-eq-0-iff)

lemma lcm-neg-numeral-1 [simp]: lcm (− numeral n) a = lcm (numeral n) a  
  by (fact lcm-neg1)

lemma lcm-neg-numeral-2 [simp]: lcm a (− numeral n) = lcm a (numeral n)  
  by (fact lcm-neg2)

end

class semiring-Gcd = semiring-gcd + Gcd +
  assumes Gcd-dvd: a ∈ A =⇒ Gcd A dvd a
  and Gcd-greatest: (∀ b. b ∈ A =⇒ a dvd b) =⇒ a dvd Gcd A
  and normalize-Gcd [simp]: normalize (Gcd A) = Gcd A
  assumes dvd-Lcm: a ∈ A =⇒ a dvd Lcm A
  and Lcm-least: (∀ b. b ∈ A =⇒ b dvd a) =⇒ Lcm A dvd a
  and normalize-Lcm [simp]: normalize (Lcm A) = Lcm A
begin

lemma Lcm-Gcd: Lcm A = Gcd {b. ∀ a ∈ A. a dvd b}  
  by (rule associated-eqI) (auto intro: Gcd-dvd dvd-Lcm Gcd-greatest Lcm-least)

lemma Gcd-Lcm: Gcd A = Lcm {b. ∀ a ∈ A. b dvd a}  
  by (rule associated-eqI) (auto intro: Gcd-dvd dvd-Lcm Gcd-greatest Lcm-least)

lemma Gcd-empty [simp]: Gcd {} = 0  
  by (rule dvd-0-left, rule Gcd-greatest) simp

lemma Lcm-empty [simp]: Lcm {} = 1  
  by (auto intro: associated-eqI Lcm-least)

lemma Gcd-insert [simp]: Gcd (insert a A) = gcd a (Gcd A)
proof
  have \( \text{gcd} (\text{insert } a \; A) \) dvd gcd a (\( \text{Gcd} \; A \))
    by (auto intro: gcd-dvd gcd-greatest)
moreover have gcd a (\( \text{Gcd} \; A \)) dvd \( \text{Gcd} \) (\( \text{insert} \; a \; A \))
proof (rule \( \text{Gcd-greatest} \))
  fix \( b \)
  assume \( b \in \text{insert} \; a \; A \)
  then show \( \text{gcd} \; a \; (\text{Gcd} \; A) \) dvd \( b \)
proof
  assume \( b = a \)
  then show \( \text{thesis} \)
    by simp
next
  assume \( b \in A \)
  then have \( \text{Gcd} \; A \) dvd \( b \)
    by (rule gcd-dvd)
moreover have \( \text{gcd} \; a \; (\text{Gcd} \; A) \) dvd \( \text{Gcd} \; A \)
    by simp
ultimately show \( \text{thesis} \)
    by (blast intro: dvd-trans)
qed
qed
ultimately show \( \text{thesis} \)
  by (auto intro: associated-eqI)
qed

lemma Lcm-insert [simp]: \( \text{Lcm} \) (\( \text{insert} \; a \; A \)) = lcm a (\( \text{Lcm} \; A \))
proof (rule sym)
  have lcm a (\( \text{Lcm} \; A \)) dvd Lcm (\( \text{insert} \; a \; A \))
    by (auto intro: dvd-Lcm Lcm-least)
moreover have Lcm (\( \text{insert} \; a \; A \)) dvd lcm a (\( \text{Lcm} \; A \))
proof (rule Lcm-least)
  fix \( b \)
  assume \( b \in \text{insert} \; a \; A \)
  then show \( b \) dvd lcm a (\( \text{Lcm} \; A \))
proof
  assume \( b = a \)
  then show \( \text{thesis} \) by simp
next
  assume \( b \in A \)
  then have \( b \) dvd \( \text{lcm} \; a \; (\text{Lcm} \; A) \)
    by (rule dvd-Lcm)
moreover have \( \text{Lcm} \; A \) dvd lcm a (\( \text{Lcm} \; A \))
    by simp
ultimately show \( \text{thesis} \)
    by (blast intro: dvd-trans)
qed
qed
ultimately show lcm a (\( \text{Lcm} \; A \)) = \( \text{Lcm} \) (\( \text{insert} \; a \; A \))
by (rule associated-eqI) (simp-all add: lcm-eq-0-iff)

qed

lemma LcmI:
  assumes \( \forall a. a \in A \implies a \mid b \)
  and \( \forall c. (\forall a. a \in A \implies a \mid c) \implies b \mid c \)
  and normalize \( b = b \)
  shows \( b = \text{lcm} A \)
  by (rule associated-eqI) (auto simp: assms dvd-Lcm intro: Lcm-least)

lemma Lcm-subset: \( A \subseteq B \implies \text{lcm} A \mid \text{lcm} B \)
  by (blast intro: Lcm-least dvd-Lcm)

lemma Lcm-Un: \( \text{lcm} (A \cup B) = \text{lcm} (\text{lcm} A) (\text{lcm} B) \)
proof
  have \( \exists d. [\text{lcm} A \mid d; \text{lcm} B \mid d] \implies \text{lcm} (A \cup B) \mid d \)
    by (meson UnE Lcm-least dvd-Lcm dvd-trans)
  then show \(?thesis\)
    by (meson Lcm-subset lcm-unique normalize-Lcm sup.cobounded1 sup.cobounded2)
qed

lemma Gcd-0-iff [simp]: \( \text{Gcd} A = 0 \iff A \subseteq \{0\} \)
(is \(?P \iff \?Q\))
proof
  assume \(?P\)
  show \(?Q\)
    proof
      fix \(a\)
      assume \(a \in A\)
      then have \(\text{Gcd} A \mid a\)
        by (rule Gcd-dvd)
      with \(?P\) have \(a = 0\)
        by simp
      then show \(a \in \{0\}\)
        by simp
    qed
  next
    assume \(?Q\)
    have \(0 \mid \text{Gcd} A\)
    proof (rule Gcd-greatest)
      fix \(a\)
      assume \(a \in A\)
      with \(?Q\) have \(a = 0\)
        by auto
      then show \(0 \mid a\)
        by simp
    qed
    then show \(?P\)
      by simp
lemma Lcm-1-iff [simp]: \(\text{Lcm } A = 1 \iff (\forall a \in A. \text{is-unit } a)\)

proof
  assume \(?P\)
  show \(?Q\)
  proof
    fix \(a\)
    assume \(a \in A\)
    then have \(a \text{ dvd } \text{Lcm } A\) by (rule dvd-Lcm)
    with \(?P\) show \(\text{is-unit } a\)
      by simp
  qed
next
  assume \(?Q\)
  then have \(\text{is-unit } (\text{Lcm } A)\)
    by (blast intro: Lcm-least)
  then have \(\text{normalize } (\text{Lcm } A) = 1\)
    by (rule is-unit-normalize)
  then show \(?P\)
    by simp
qed

lemma unit-factor-Lcm: \(\text{unit-factor } (\text{Lcm } A) = \begin{cases} 0 & \text{if } \text{Lcm } A = 0 \\ 1 & \text{else} \end{cases}\)

proof
  (cases \(\text{Lcm } A = 0\))
  case True
  then show \(?\text{thesis}\)
    by simp
next
  case False
  with \(\text{unit-factor-normalize}\) have \(\text{unit-factor } (\text{normalize } (\text{Lcm } A)) = 1\)
    by blast
  with False show \(?\text{thesis}\)
    by simp
qed

lemma unit-factor-Gcd: \(\text{unit-factor } (\text{Gcd } A) = \begin{cases} 0 & \text{if } \text{Gcd } A = 0 \\ 1 & \text{else} \end{cases}\)

by (simp add: Gcd-Lcm unit-factor-Lcm)

lemma GcdI:
  assumes \(\bigwedge a. a \in A \implies b \text{ dvd } a\)
  and \(\bigwedge c. (\bigwedge a. a \in A \implies c \text{ dvd } a) \implies c \text{ dvd } b\)
  and \(\text{normalize } b = b\)
  shows \(b = \text{Gcd } A\)
by (rule associated-eqI) (auto simp: assms Gcd-dvd intro: Gcd-greatest)

lemma Gcd-eq-1-I:
assumes is-unit a and a ∈ A
shows Gcd A = 1

proof –
  from assms have is-unit (Gcd A)
    by (blast intro: Gcd-dvd dvd-unit-imp-unit)
  then have normalize (Gcd A) = 1
    by (rule is-unit-normalize)
  then show ?thesis
    by simp

qed

lemma Lcm-eq-0-I:
assumes 0 ∈ A
shows Lcm A = 0

proof –
  from assms have 0 dvd Lcm A
    by (rule dvd-Lcm)
  then show ?thesis
    by simp

qed

lemma Gcd-UNIV [simp]: Gcd UNIV = 1
using dvd-refl by (rule Gcd-eq-1-I) simp

lemma Lcm-UNIV [simp]: Lcm UNIV = 0
by (rule Lcm-eq-0-I) simp

lemma Lcm-0-iff:
assumes finite A
shows Lcm A = 0 ↔ 0 ∈ A

proof (cases A = {})
  case True
  then show ?thesis by simp
next
  case False
  with assms show ?thesis
    by (induct A rule: finite-ne-induct) (auto simp: lcm-eq-0-iff)

qed

lemma Gcd-image-normalize [simp]: Gcd (normalize ' A) = Gcd A

proof –
  have Gcd (normalize ' A) dvd a if a ∈ A for a
  proof –
    from that obtain B where A = insert a B
    by blast
    moreover have gcd (normalize a) (Gcd (normalize ' B)) dvd normalize a
      by (rule gcd-dvd1)
    ultimately show Gcd (normalize ' A) dvd a
      by simp
**THEORY “GCD”**

qed

then have \( \text{Gcd} (\text{normalize } \cdot A) \text{ dvd Gcd } A \text{ and Gcd } A \text{ dvd Gcd} (\text{normalize } \cdot A) \)

by (auto intro! \: \text{Gcd-greatest intro} \: \text{Gcd-dvd})

then show \( ?\text{thesis} \)

by (auto intro: \: \text{associated-eqI})

qed

**lemma Gcd-eqI:**

assumes normalize \( a = a \)

assumes \( \forall b. b \in A \implies a \text{ dvd } b \)

and \( \forall c. (\forall b. b \in A \implies c \text{ dvd } b) \implies c \text{ dvd } a \)

shows \( \text{Gcd } A = a \)

using assms by (blast intro: \: \text{associated-eqI} \: \text{Gcd-greatest} \: \text{Gcd-dvd} \: \text{normalize-Gcd})

**lemma dvd-GcdD:** \( x \text{ dvd Gcd } A \implies y \in A \implies x \text{ dvd } y \)

using \( \text{Gcd-dvd} \) \: blast

**lemma dvd-Gcd-iff:** \( x \text{ dvd Gcd } A \iff (\forall y \in A. x \text{ dvd } y) \)

by (blast dest \: \text{dvd-GcdD} intro \: \text{Gcd-greatest})

**lemma Gcd-mult:** \( \text{Gcd } (\text{(*) } c \cdot A) = \text{normalize } (c \cdot \text{Gcd } A) \)

proof (cases \( c = 0 \))

case True

then show \( ?\text{thesis} \) by auto

next

case [simp]: False

have \( \text{Gcd } (\text{(*) } c \cdot A) \text{ div } c \text{ dvd } \text{Gcd } A \)

by (intro \: \text{Gcd-greatest} \: \text{subst div-dvd-iff-mult})

(auto intro! \: \text{Gcd-greatest } \text{Gcd-dvd simp: mult.commute[of } - c\})

then have \( \text{Gcd } (\text{(*) } c \cdot A) \text{ dvd } c \cdot \text{Gcd } A \)

by (subst (asm) \: \text{div-dvd-iff-mult}) (auto intro: \: \text{Gcd-greatest simp: mult-ac})

moreover have \( c \cdot \text{Gcd } A \text{ dvd } \text{Gcd } ((\text{(*) } c \cdot A) \)

by (intro \: \text{Gcd-greatest}) (auto intro: \: \text{mult-dvd-mono} \: \text{Gcd-dvd})

ultimately have \( \text{normalize } (\text{Gcd } (\text{(*) } c \cdot A)) = \text{normalize } (c \cdot \text{Gcd } A) \)

by (rule \: \text{associatedI})

then show \( ?\text{thesis} \) by simp

qed

**lemma Lcm-eqI:**

assumes normalize \( a = a \)

and \( \forall b. b \in A \implies b \text{ dvd } a \)

and \( \forall c. (\forall b. b \in A \implies b \text{ dvd } c) \implies a \text{ dvd } c \)

shows \( \text{Lcm } A = a \)

using assms by (blast intro: \: \text{associated-eqI} \: \text{Lcm-least dvd-Lcm} \: \text{normalize-Lcm})

**lemma Lcm-dvdD:** \( \text{Lcm } A \text{ dvd } x \implies y \in A \implies y \text{ dvd } x \)

using \( \text{dvd-Lcm} \) \: blast

**lemma Lcm-dvd-iff:** \( \text{Lcm } A \text{ dvd } x \iff (\forall y \in A. y \text{ dvd } x) \)
by (blast dest: Lcm-dvdD intro Lcm-least)

lemma Lcm-mult:
assumes $A \neq \{\}$
shows $(\ast) \ c \cdot A = \text{normalize} (c \ast \text{Lcm} \ A)$
proof (cases $c = 0$)
case True
  with assms have $(\ast) \ c \cdot A = \{0\}$
    by auto
  with True show ?thesis by auto
next
case [simp]: False
from assms obtain $x$ where $x \in A$
  by blast
have $c \text{ dvd } c \cdot x$
  by simp
also have $c \ast x \text{ dvd Lcm } ((\ast) \ c \cdot A)$
  by (intro dvd-Lcm) auto
finally have dvd: $c \text{ dvd Lcm } ((\ast) \ c \cdot A)$
  by (auto intro: Lcm-least dvd-mulimpdiv)
ultimately have $c \ast \text{Lcm} \ A \text{ dvd Lcm } ((\ast) \ c \cdot A)$
  by auto
moreover have $\text{Lcm } ((\ast) \ c \cdot A) \text{ dvd } c$
  by (intro Lcm-least dvd-mulimpdiv)
ultimately have $c \ast \text{Lcm} \ A \text{ dvd } c \ast \text{Lcm} \ A$
  by auto
ultimately have $\text{normalize} (c \ast \text{Lcm} \ A) = \text{normalize} (\text{Lcm } ((\ast) \ c \cdot A))$
  by (rule associatedI)
then show ?thesis by simp
qed

lemma Lcm-no-units: $\text{Lcm} \ A = \text{Lcm } (A - \{a. \text{is-unit } a\})$
proof
  have $(A - \{a. \text{is-unit } a\}) \cup \{a \in A. \text{is-unit } a\} = A$
    by blast
  then have $\text{Lcm} \ A = \text{lcm } (\text{Lcm } (A - \{a. \text{is-unit } a\})) (\text{Lcm } \{a \in A. \text{is-unit } a\})$
    by (simp add: Lcm-Un [symmetric])
  also have $\text{Lcm } \{a \in A. \text{is-unit } a\} = 1$
    by simp
  finally show ?thesis
    by simp
qed

lemma Lcm-0-iff': $\text{Lcm} \ A = 0 \iff (\exists l. l \neq 0 \land (\forall a \in A. a \text{ dvd } l))$
by (metis Lcm-least dvd-0-left dvd-Lcm)

lemma Lcm-no-multiple: $(\forall m. m \neq 0 \rightarrow (\exists a \in A. \ a \text{ dvd } m)) \implies \text{Lcm} \ A = 0$
by (auto simp: Lcm-0-iff')
THEORY “GCD”

lemma Lcm-singleton [simp]: Lcm \{a\} = normalize a
  by simp

lemma Lcm-2 [simp]: Lcm \{a, b\} = lcm a b
  by simp

lemma Gcd-1 \(1 \in A \implies Gcd A = 1\)
  by (auto intro: Gcd-eq-1-I)

lemma Gcd-singleton [simp]: Gcd \{a\} = normalize a
  by simp

lemma Gcd-2 [simp]: Gcd \{a, b\} = gcd a b
  by simp

lemma Gcd-mono:
  assumes \(\forall x. x \in A \implies f x \text{ dvd } g x\)
  shows \((GCD x \in A. f x) \text{ dvd } (GCD x \in A. g x)\)
proof (intro Gcd-greatest, safe)
  fix x assume x \in A
  hence \((GCD x \in A. f x) \text{ dvd } f x\)
    by (intro Gcd-dvd) auto
  also have \(f x \text{ dvd } g x\)
    using \(x \in A\) assms by blast
  finally show \((GCD x \in A. f x) \text{ dvd } \ldots .\).
qed

lemma Lcm-mono:
  assumes \(\forall x. x \in A \implies f x \text{ dvd } g x\)
  shows \((LCM x \in A. f x) \text{ dvd } (LCM x \in A. g x)\)
proof (intro Lcm-least, safe)
  fix x assume x \in A
  hence \(f x \text{ dvd } g x\) by (rule assms)
  also have \(g x \text{ dvd } (LCM x \in A. g x)\)
    using \(x \in A\) by (intro dvd-Lcm) auto
  finally show \(f x \text{ dvd } \ldots .\).
qed

end

87.3 An aside: GCD and LCM on finite sets for incomplete gcd rings

context semiring-gcd
begin

sublocale Gcd-fin: bounded-quasi-semilattice-set gcd 0 1 normalize
defines
  Gcd-fin (Gcd_fin) = Gcd-fin.F :: 'a set \Rightarrow 'a ..
abbreviation \texttt{gcd-list} :: 'a list ⇒ 'a
where \texttt{gcd-list} \texttt{xs} \equiv \texttt{Gcd} f in (set \texttt{xs})

sublocale \texttt{Lcm-fin}: bounded-quasi-semilattice-set \texttt{lcm} 1 0 normalize
defines
\texttt{Lcm-fin} (\texttt{Lcm} f in) = \texttt{Lcm-fin}.F ..

abbreviation \texttt{lcm-list} :: 'a list ⇒ 'a
where \texttt{lcm-list} \texttt{xs} \equiv \texttt{Lcm} f in (set \texttt{xs})

lemma \texttt{Gcd-fin-dvd}:
a ∈ A ⇒ \texttt{Gcd} f in A dvd a
by (induct A rule: infinite-finite-induct)
(auto intro: dvd-trans)

lemma \texttt{dvd-Lcm-fin}:
a ∈ A ⇒ a dvd \texttt{Lcm} f in A
by (induct A rule: infinite-finite-induct)
(auto intro: dvd-trans)

lemma \texttt{Gcd-fin-greatest}:
a dvd \texttt{Gcd} f in A if finite A and \( \forall b. b ∈ A \Rightarrow a \) dvd b
using that by (induct A simp-all)

lemma \texttt{Lcm-fin-least}:
\texttt{Lcm} f in A dvd a if finite A and \( \forall b. b ∈ A \Rightarrow b \) dvd a
using that by (induct A simp-all)

lemma \texttt{gcd-list-greatest}:
a dvd \texttt{gcd-list} bs if \( \forall b. b ∈ \text{set} \texttt{bs} \Rightarrow a \) dvd b
by (rule \texttt{Gcd-fin-greatest}) (simp-all add: that)

lemma \texttt{lcm-list-least}:
\texttt{lcm-list} bs dvd a if \( \forall b. b ∈ \text{set} \texttt{bs} \Rightarrow b \) dvd a
by (rule \texttt{Lcm-fin-least}) (simp-all add: that)

lemma \texttt{dvd-Gcd-fin-iff}:
b dvd \texttt{Gcd} f in A if finite A if finite A 
using that by (auto intro: Gcd-fin-greatest Gcd-fin-dvd dvd-trans [of b Gcd f in A])

lemma \texttt{dvd-gcd-list-iff}:
b dvd \texttt{gcd-list} xs if \( \forall a ∈ \text{set} \texttt{xs}. b \) dvd a
by (simp add: dvd-Gcd-fin-iff)

lemma \texttt{Lcm-fin-dvd-iff}:
\texttt{Lcm} f in A dvd b if finite A
using that by (auto intro: Lcm-fin-least dvd-Lcm-fin dvd-trans [of - \texttt{Lcm-fin} A] A)
lemma lcm-list-dvd-iff:
\[ \text{lcm-list } xs \text{ dvd } b \iff (\forall a \in \text{set} \ x s. \ a \text{ dvd } b) \]
by (simp add: Lcm-fin-dvd-iff)

lemma Gcd-fin-mult:
\[ \text{Gcd}_{fin}\ (\text{image} \ (\text{times} \ b) \ A) = \text{normalize} \ (b * \text{Gcd}_{fin} \ A) \text{ if } \text{finite} \ A \]
using that by induction (auto simp: gcd-mult-left)

lemma Lcm-fin-mult:
\[ \text{Lcm}_{fin}\ (\text{image} \ (\text{times} \ b) \ A) = \text{normalize} \ (b * \text{Lcm}_{fin} \ A) \text{ if } A \neq \{\} \]
proof (cases \( b = 0 \))
  case True
  moreover from that have times 0 : A = {0}
  by auto
  ultimately show ?thesis
  by simp
next
  case False
  show ?thesis proof (cases finite A)
    case False
    moreover have inj-on (times b) A
      using \( b \neq 0 \) by (rule inj-on-mult)
    ultimately have infinite (times b : A)
      by (simp add: finite-image-iff)
    with False show ?thesis
    by simp
next
  case True
  then show ?thesis using that
    by (induct A rule: finite-ne-induct) (auto simp: lcm-mult-left)
qed

lemma unit-factor-Gcd-fin:
\[ \text{unit-factor} \ (\text{Gcd}_{fin} \ A) = \text{of-bool} \ (\text{Gcd}_{fin} \ A \neq 0) \]
by (rule normalize-idem-imp-unit-factor-eq) simp

lemma unit-factor-Lcm-fin:
\[ \text{unit-factor} \ (\text{Lcm}_{fin} \ A) = \text{of-bool} \ (\text{Lcm}_{fin} \ A \neq 0) \]
by (rule normalize-idem-imp-unit-factor-eq) simp

lemma is-unit-Gcd-fin-iff [simp]:
\[ \text{is-unit} \ (\text{Gcd}_{fin} \ A) \iff \text{Gcd}_{fin} \ A = 1 \]
by (rule normalize-idem-imp-is-unit-iff) simp

lemma is-unit-Lcm-fin-iff [simp]:
\[ \text{is-unit} \ (\text{Lcm}_{fin} \ A) \iff \text{Lcm}_{fin} \ A = 1 \]
THEORY “GCD”

by (rule normalize-idem-imp-is-unit-iff) simp

lemma Gcd-fin-0-iff:
  \( \text{Gcd}_{\text{fin}} \ A = 0 \iff A \subseteq \{0\} \land \text{finite } A \)
by (induct A rule: infinite-finite-induct) simp-all

lemma Lcm-fin-0-iff:
  \( \text{Lcm}_{\text{fin}} \ A = 0 \iff 0 \in A \) if finite A
using that by (induct A) (auto simp: lcm-eq-0-iff)

lemma Lcm-fin-1-iff:
  \( \text{Lcm}_{\text{fin}} \ A = 1 \iff (\forall a \in A. \text{is-unit } a) \land \text{finite } A \)
by (induct A rule: infinite-finite-induct) simp-all

end

context semiring-Gcd
begin

lemma Gcd-fin-eq-Gcd [simp]:
  \( \text{Gcd}_{\text{fin}} \ A = \text{Gcd } A \) if finite A for A :: 'a set
using that by induct simp-all

lemma Gcd-set-eq-fold [code-unfold]:
  \( \text{Gcd } (\text{set } xs) = \text{fold } \text{gcd } xs \ 0 \)
by (simp add: Gcd-fin.set-eq-fold [symmetric])

lemma Lcm-fin-eq-Lcm [simp]:
  \( \text{Lcm}_{\text{fin}} \ A = \text{Lcm } A \) if finite A for A :: 'a set
using that by induct simp-all

lemma Lcm-set-eq-fold [code-unfold]:
  \( \text{Lcm } (\text{set } xs) = \text{fold } \text{lcm } xs \ 1 \)
by (simp add: Lcm-fin.set-eq-fold [symmetric])

end

87.4 Coprimality

category semiring-gcd
begin

lemma coprime-imp-gcd-eq-1 [simp]:
  \( \text{gcd } a \ b = 1 \) if coprime a b
proof –
  define \( t \ r \ s \) where \( t = \text{gcd } a \ b \) and \( r = a \div t \) and \( s = b \div t \)
then have \( a = t \cdot r \) and \( b = t \cdot s \)
  by simp-all
with that have coprime \( (t \cdot r) \) (\( t \cdot s \)
by simp
then show ?thesis
  by (simp add: t-def)
qed

lemma gcd-eq-1-imp-coprime [dest!]:
coprime a b if gcd a b = 1
proof (rule coprimeI)
  fix c
  assume c dvd a and c dvd b
  then have c dvd gcd a b
    by (rule gcd-greatest)
  with that show is-unit c
    by simp
qed

lemma coprime-iff-gcd-eq-1 [presburger, code]:
coprime a b ←→ gcd a b = 1
by rule (simp-all add: gcd-eq-1-imp-coprime)

lemma is-unit-gcd [simp]:
is-unit (gcd a b) ←→ coprime a b
by (simp add: coprime-iff-gcd-eq-1)

lemma coprime-add-one-left [simp]: coprime (a + 1) a
by (simp add: gcd-eq-1-imp-coprime ac-simps)

lemma coprime-add-one-right [simp]: coprime a (a + 1)
using coprime-add-one-left [of a] by (simp add: ac-simps)

lemma coprime-mult-left-iff [simp]:
coprime (a * b) c ←→ coprime a c ∧ coprime b c
proof
  assume coprime (a * b) c
  with coprime-common-divisor [of a * b c]
  have *: is-unit d if d dvd a * b and d dvd c for d
    using that by blast
  have coprime a c
    by (rule coprimeI, rule *) simp-all
  moreover have coprime b c
    by (rule coprimeI, rule *) simp-all
  ultimately show coprime a c ∧ coprime b c ..
next
  assume coprime a c ∧ coprime b c
  then have coprime a c coprime b c
    by simp-all
  show coprime (a * b) c
    proof (rule coprimeI)
      fix d
THEORY "GCD"

assume \( d \mid a \cdot b \)
then obtain \( r \cdot s \) where \( d = r \cdot s \cdot \) \( r \mid a \cdot s \mid b \)
by (rule dvd-productE)
assume \( d \mid c \)
with \( d \) have \( r \cdot s \mid c \)
by simp
then have \( r \cdot s \mid c \)
by (auto intro: dvd-mult-left dvd-mult-right)
from \( \text{coprime } a \cdot c \) \( \text{r } \mid a \langle r \text{ } \mid d \cdot a \rangle \) \( \text{r } \mid d \cdot c \)
have is-unit \( r \)
by (rule coprime-common-divisor)
moreover from \( \text{coprime } b \cdot c \) \( \text{s } \mid b \langle s \text{ } \mid d \cdot b \rangle \) \( \text{s } \mid d \cdot c \)
have is-unit \( s \)
by (rule coprime-common-divisor)
ultimately show is-unit \( d \)
by (simp add: d is-unit-mult-iff)
qed

lemma coprime-mult-right-iff [simp]:
\( \text{coprime } c \langle a \cdot b \rangle \iff \text{coprime } c \cdot a \langle \text{coprime } c \cdot b \rangle \)
using coprime-mult-left-iff [of a b c]
by (simp add: ac-simps)

lemma coprime-power-left-iff [simp]:
\( \text{coprime } (a \cdot n) \cdot b \iff \text{coprime } a \cdot b \langle \text{coprime } a \cdot b \rangle \cdot n = 0 \)
proof (cases \( n = 0 \))
  case True
  then show ?thesis
  by simp
next
case False
then have \( n > 0 \)
by simp
then show ?thesis
by (induction \( n \) rule: nat-induct-non-zero) simp-all
qed

lemma coprime-power-right-iff [simp]:
\( \text{coprime } a \langle b \cdot n \rangle \iff \text{coprime } a \langle b \cdot n \rangle \cdot n = 0 \)
using coprime-power-left-iff [of b n a]
by (simp add: ac-simps)

lemma prod-coprime-left:
\( \text{coprime } \prod_{i \in A.} f \cdot i \cdot a \cdot \text{if } \bigwedge_{i. \cdot i \in A} \implies \text{coprime } f \cdot i \cdot a \)
using that by (induct A rule: infinite-finite-induct) simp-all

lemma prod-coprime-right:
\( \text{coprime } a \langle \prod_{i \in A.} f \cdot i \rangle \cdot \text{if } \bigwedge_{i. \cdot i \in A} \implies \text{coprime } a \langle f \cdot i \rangle \)
using that prod-coprime-left [of A f a]
by (simp add: ac-simps)
lemma prod-list-coprime-left:
coprime (prod-list xs) a if \( \forall x. \ x \in \text{set} \ \text{xs} \implies \text{coprime} \ \text{x} \ a \)
using that by (induct xs) simp-all

lemma prod-list-coprime-right:
coprime a (prod-list xs) if \( \forall x. \ x \in \text{set} \ \text{xs} \implies \text{coprime} a \ \text{x} \)
using that prod-list-coprime-left [of xs a] by (simp add: ac-simps)

lemma coprime-dvd-mult-left-iff:
a dvd b \ast c \iff a dvd b if coprime a c
proof
assume a dvd b
then show a dvd b \ast c
  by simp
next
assume a dvd b \ast c
show a dvd b
proof (cases b = 0)
case True
  then show ?thesis
    by simp
next
case False
then have unit: is-unit (unit-factor b)
  by simp
from (coprime a c)
have gcd' (b \ast a) (b \ast c) \ast unit-factor b = b
  by (subst gcd-mult-left) (simp add: ac-simps)
moreover from (a dvd b \ast c)
have a dvd gcd (b \ast a) (b \ast c) \ast unit-factor b
  by (simp add: dvd-mult-unit-iff unit)
ultimately show ?thesis
  by simp
qed
eqd

lemma coprime-dvd-mult-right-iff:
a dvd c \ast b \iff a dvd b if coprime a c
using that coprime-dvd-mult-left-iff [of a c b] by (simp add: ac-simps)

lemma divides-mult:
a \ast b dvd c if a dvd c and b dvd c and coprime a b
proof
from (b dvd c) obtain b' where c = b \ast b' ..
with (a dvd c) have a dvd b' \ast b
  by (simp add: ac-simps)
with (coprime a b) have a dvd b'
  by (simp add: coprime-dvd-mult-left-iff)
then obtain a' where b' = a \ast a' ..
with \(c = b \ast b'\) have \(c = (a \ast b) \ast a'\)
by (simp add: ac-sims)
then show \(?\text{thesis} ..\)
qed

lemma div-gcd-coprime:
assumes \(a \neq 0 \lor b \neq 0\)
shows coprime \((a \div \gcd a b) (b \div \gcd a b)\)
proof –
let \(?g = \gcd a b\)
let \(?a' = a \div ?g\)
let \(?b' = b \div ?g\)
let \(?g' = \gcd ?a' ?b'\)
have dvdg: \(?g \div \gcd a b \div a \div ?g \div ?a' \div ?g' \div \gcd a b \div b \div \gcd a b \div b'\)
by simp-all
have dvdg': \(?g' \div \gcd ?a' \div ?a' \div ?g' \div \gcd a b \div b \div \gcd a b \div b'\)
by simp-all
from dvdg dvdg' obtain \(ka \div kb \div ka' \div kb'\) where
\(ka: a = \ ?g * \ ?ka \div b = \ ?g * \ ?kb \div ?a' = \ ?g' * \ ?ka' \div ?b' = \ ?g' * \ ?kb'\)
unfolding dvd-def by blast
from this [symmetric] have \(?g * ?a' = (\ ?g * \ ?g') \div ka' \div ?g * ?b' = (\ ?g * \ ?g') \div \ ?gb'\)
by (simp-all add: mult_assoc mult_left_commute [of \gcd a b])
then have dvdgg': \(?g * ?g' \div \gcd a b \div a \div \gcd a b \div \gcd a b \div b \div \gcd a b \div b'\)
by (auto simp: dvd-mult-cancel [OF dvdgg(1)] dvd-mult-cancel [OF dvdgg(2)]
dvd-def)
have \(?g \neq 0\)
using assms by simp
moreover from gcd-greatest [OF dvdgg'] have \(?g * ?g' \div \gcd a b \div \gcd a b\).
ultimately show \(?\text{thesis} \)
using dvd-times-left-cancel-iff [of \gcd a b - 1]
by simp (simp only: coprime-iff-gcd-eq-1)
qed

lemma gcd-coprime:
assumes \(c: \gcd a b \neq 0\)
and \(a: a = a' \ast \gcd a b\)
and \(b: b = b' \ast \gcd a b\)
shows coprime \(a' \div b'\)
proof –
from \(c\) have \(a \neq 0 \lor b \neq 0\)
by simp
with div-gcd-coprime have coprime \((a \div \gcd a b) (b \div \gcd a b)\).
also from assms have \(a \div \gcd a b = a'\)
using dvd-div-eq-mult gcd-dvd1 by blast
also from assms have \(b \div \gcd a b = b'\)
using dvd-div-eq-mult gcd-dvd1 by blast
finally show \(?\text{thesis} ..\)
qed
lemma \textit{gcd-coprime-exists}:
assumes \( gcd \ a \ b \neq 0 \)
shows \( \exists \ a' \ b': a = a' \ast gcd \ a \ b \land b = b' \ast gcd \ a \ b \land coprime \ a' \ b' \)
proof –
\begin{itemize}
\item have \( coprime \ (a \ div \ gcd \ a \ b) \ (b \ div \ gcd \ a \ b) \)
\item using \( \text{assms div-gcd-coprime by auto} \)
\end{itemize}
then show \( \text{thesis} \)
\begin{itemize}
\item by \( \text{force} \)
\end{itemize}
qed

lemma \textit{pow-divides-pow-iff} \([\text{simp}]\):
\( a ^ n \ dvd \ b ^ n \iff a \ dvd \ b \) if \( n > 0 \)
proof (cases \( gcd \ a \ b = 0 \))
\begin{itemize}
\item case \( \text{True} \)
\item show \( \text{thesis} \)
\begin{itemize}
\item by \( \text{simp} \)
\end{itemize}
\end{itemize}
next
\begin{itemize}
\item case \( \text{False} \)
\item show \( \text{thesis} \)
\begin{itemize}
\item proof
\begin{itemize}
\item let \( ?d = gcd \ a \ b \)
\item from \( \langle n > 0 \rangle \) obtain \( m \) where \( m = Suc \ m \)
\begin{itemize}
\item by \( \text{cases} \ n \) \( \text{simp-all} \)
\end{itemize}
\item from \( \text{False} \) have \( zn: ?d ^ \ n \neq 0 \)
\begin{itemize}
\item by \( \text{rule} \ \text{power-not-zero} \)
\end{itemize}
\item from \( \text{gcd-coprime-exists} \ \langle \text{OF False} \rangle \) obtain \( a' \ b' \) where \( ab': a = a' \ast ?d \ b = b' \ast ?d \ \text{coprime} \ a' \ b' \)
\begin{itemize}
\item by \( \text{blast} \)
\end{itemize}
\item assume \( a ^ \ n \ dvd b ^ \ n \)
\item then have \( (a' \ast ?d) ^ \ n \ dvd (b' \ast ?d) ^ \ n \)
\begin{itemize}
\item by \( \text{simp add: \( \langle 1,2 \rangle \\text{[symmetric]} \rangle \) \)
\end{itemize}
\item then have \( ?d ^ \ n \ast a' ^ \ n \ dvd ?d ^ \ n \ast b' ^ \ n \)
\begin{itemize}
\item by \( \text{simp only: power-mult-distrib ac-simps} \)
\end{itemize}
\item with \( zn \) have \( a' ^ \ n \ dvd b' ^ \ n \)
\begin{itemize}
\item by \( \text{simp} \)
\end{itemize}
\item then have \( a' ^ \ dvd b' ^ \ n \)
\begin{itemize}
\item using \( \text{dvd-trans}[\text{of} \ a' \ a' ^ \ n \ b' \n] \) by \( \text{simp add:} \ m \)
\end{itemize}
\item then have \( a' \ dvd b' ^ \ n * b' \)
\begin{itemize}
\item by \( \text{simp add:} \ m \ \text{ac-simps} \)
\end{itemize}
\item moreover have \( \text{coprime} \ a' \ (b' ^ \ n) \)
\begin{itemize}
\item using \( \text{coprime} \ a' \ b' \) by \( \text{simp} \)
\end{itemize}
\item then have \( a' ^ \ dvd b' \)
\begin{itemize}
\item using \( \text{a' ^ \ dvd b' ^ \ n \ \text{coprime-dvd-mult-left-iff} \ \text{dvd-mult } \text{by blast} \) \)
\end{itemize}
\item then have \( a' \ast ?d \ dvd b' ^ \ * \ ?d \)
\begin{itemize}
\item by \( \text{rule} \ \text{mult-dvd-mono} \) \( \text{simp} \)
\end{itemize}
\item with \( \text{ab'} \langle 1,2 \rangle \) show \( a \ dvd b \)
\begin{itemize}
\item by \( \text{simp} \)
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
}

qed
assume \(a \text{ dvd } b\)

with \(\langle n > 0 \rangle\) show \(a ^ n \text{ dvd } b ^ n\)
by (induction rule: nat-induct-non-zero)
(simp-all add: mult-dvd-mono)
qed

lemma coprime-crossproduct:
fixes \(a \ b \ c \ d\) :: 'a
assumes coprime \(a \ d\) and coprime \(b \ c\)
shows normalize \(a\) * normalize \(c\) = normalize \(b\) * normalize \(d\) \(\longleftrightarrow\)
normalize \(a\) = normalize \(b\) \(\land\) normalize \(c\) = normalize \(d\)
(is \(\text{?lhs} \longleftrightarrow \text{?rhs}\))
proof
assume \(\text{?rhs}\)
then show \(\text{?lhs}\) by simp
next
assume \(\text{?lhs}\)
from \(\langle \text{?lhs}\rangle\) have normalize \(a\) dvd normalize \(b\) * normalize \(d\)
by (auto intro: dvdI dest: sym)
with \(\langle \text{coprime \(a \ d\rangle\)}\) have \(a\) dvd \(b\)
by (simp add: coprime-dvd-mult-left-iff normalize-mult [symmetric])
from \(\langle \text{?lhs}\rangle\) have normalize \(b\) dvd normalize \(a\) * normalize \(c\)
by (auto intro: dvdI dest: sym simp add: mult.commute)
with \(\langle \text{coprime \(b \ c\rangle\)}\) have \(b\) dvd \(a\)
by (simp add: coprime-dvd-mult-left-iff coprime-commute normalize-mult [symmetric])
from \(\langle \text{?lhs}\rangle\) have normalize \(d\) dvd normalize \(c\) * normalize \(a\)
by (auto intro: dvdI dest: sym simp add: mult.commute)
with \(\langle \text{coprime \(a \ d\rangle\)}\) have \(d\) dvd \(c\)
by (simp add: coprime-dvd-mult-left-iff coprime-commute normalize-mult [symmetric])
from \(\langle \text{a dvd b \(\land\) b dvd a\rangle\)} have normalize \(a\) = normalize \(b\)
by (rule associatedI)
moreover from \(\langle \text{c dvd d \(\land\) d dvd c}\rangle\) have normalize \(c\) = normalize \(d\)
by (rule associatedI)
ultimately show \(\text{?rhs}\) ..
qed

lemma gcd-mult-left-left-cancel:
gcd (\(c \ast a\)) \(b\) = gcd \(a\) \(b\) if \(\text{coprime \(a \ c\)}\)
proof –
have coprime (gcd \(b\) \(\langle a \ast c\rangle\)) \(c\)
by (rule coprimeI) (auto intro: that coprime-common-divisor)
then have gcd \(b\) \(\langle a \ast c\rangle\) dvd \(a\)
using coprime-dvd-mult-left-iff [of gcd \(b\) \(\langle a \ast c\rangle\) \(c\) \(a\)]
by simp
then show \( \text{thesis} \)
by (auto intro: associated-eqI simp add: ac-simps)

qed

lemma gcd-mult-left-right-cancel:
gcd \((a \cdot c) \cdot b = gcd \ a \cdot b \if \text{coprime } b \ c \)
using that gcd-mult-left-left-cancel [of \(b \ c \ a\)]
by (simp add: ac-simps)

lemma gcd-mult-right-left-cancel:
gcd a \((c \cdot b) = gcd \ a \cdot b \if \text{coprime } a \ c \)
using that gcd-mult-right-right-cancel [of \(a \ c \ b\)]
by (simp add: ac-simps)

lemma gcd-mult-right-right-cancel:
gcd a \((b \cdot c) = gcd \ a \cdot b \if \text{coprime } a \ c \)
using that gcd-mult-right-left-cancel [of \(a \ c \ b\)]
by (simp add: ac-simps)

lemma gcd-exp-weak:
gcd \((x^n) \cdot (y^n) = normalize \,(gcd \ a \cdot b \uparrow n)\)
proof (cases a = 0 \land b = 0 \lor n = 0)
case True
then show \( \text{thesis} \)
by (cases n) simp-all
next
case False
then have coprime \((a \div gcd \ a \cdot b) \cdot (b \ div gcd \ a \cdot b) \land n > 0 \)
by (auto intro: div-gcd-coprime)
then have coprime \((a \ div gcd \ a \cdot b) \uparrow n) \cdot ((b \ div gcd \ a \cdot b) \uparrow n)\)
by simp
then have \(1 = gcd \ ((a \ div gcd \ a \cdot b) \uparrow n) \cdot ((b \ div gcd \ a \cdot b) \uparrow n)\)
by simp
then have normalize \((gcd \ a \cdot b \uparrow n) = normalize \,(gcd \ a \cdot b \uparrow n \ast \ldots)\)
by simp
also have \(a \ div gcd \ a \cdot b \uparrow n \ast (a \ div gcd \ a \cdot b) \uparrow n) \cdot (b \ div gcd \ a \cdot b) \uparrow n)\)
by (rule gcd-mult-left [symmetric])
also have \((gcd \ a \cdot b) \uparrow n \ast (a \ div gcd \ a \cdot b) \uparrow n = a \uparrow n)\)
by (simp add: ac-simps div-power dvd-power-same)
also have \((gcd \ a \cdot b) \uparrow n \ast (b \ div gcd \ a \cdot b) \uparrow n = b \uparrow n)\)
by (simp add: ac-simps div-power dvd-power-same)
finally show \( \text{thesis} \ by simp \)
qed

lemma division-decomp:
assumes \(a \ dvd b \ast c\)
says \( \exists b' \ c'. a = b' \ast c' \land b' \ dvd b \land c' \ dvd c\)
proof (cases \(gcd \ a \cdot b = 0\))
case True
then have \( a = 0 \land b = 0 \)
  by simp
then have \( a = 0 \lor c \land 0 \text{ dvd } b \land c \text{ dvd } c \)
  by simp
then show \( \text{thesis by blast} \)
next
case False
let \(?d = \gcd a b\)
from \( \text{gcd-coprime-exists [OF False]} \)
  obtain \( a', b' \) where \( ab' \)
  by blast
from \( ab'(1) \) have \( a' \text{ dvd } a \ldots \)
with \( \text{assms have } a' \text{ dvd } b \times c \)
  by (simp add: dvd-trans [of a' a b * c])
from \( \text{assms } ab'(1,2) \) have \( a' \times ?d \text{ dvd } (b' \times ?d) \times c \)
  by simp
then have \( ?d \times a' \text{ dvd } (b' \times c) \)
  by (simp add: mult-ac)
with \( \langle ?d \neq 0 \cdot \) have \( a' \text{ dvd } b' \times c \)
  by simp
then have \( a' \text{ dvd } c \)
  using \( \langle \text{coprime } a' b' \rangle \) by (simp add: coprime-dvd-mult-right-iff)
with \( ab'(1) \) have \( a = ?d \times a' \land ?d \text{ dvd } b \land a' \text{ dvd } c \)
  by (simp add: ac-simps)
then show \( \text{thesis by blast} \)
qed

lemma lcm-coprime: \( \text{coprime } a b \implies \text{lcm } a b = \text{normalize } (a \times b) \)
  by (subst lcm-gcd) simp

end

class ring_gcd
begin

lemma coprime-minus-left-iff [simp]:
  \( \text{coprime } (a - b) \leftrightarrow \text{coprime } a b \)
  by (rule; rule coprimeI) (auto intro: coprime-common-divisor)

lemma coprime-minus-right-iff [simp]:
  \( \text{coprime } a (b - a) \leftrightarrow \text{coprime } a b \)
  using coprime-minus-left-iff [of b a] by (simp add: ac-simps)

lemma coprime-diff-one-left [simp]: \( \text{coprime } (a - 1) a \)
  using coprime-add-one-right [of a - 1] by simp

lemma coprime-diff-one-right [simp]: \( \text{coprime } a (a - 1) \)
  using coprime-diff-one-left [of a] by (simp add: ac-simps)
theory "GCD"

end

custom semiring-Gcd

begin

lemma Lcm-coprime:
  assumes finite A
  and A ≠ {} and ∃ a b. a ∈ A ⇒ b ∈ A ⇒ a ≠ b ⇒ coprime a b
  shows Lcm A = normalize (∏ A)
  using assms
  proof (induct rule: finite-ne-induct)
    case singleton
    then show ?case by simp
  next
    case (insert a A)
    have Lcm (insert a A) = lcm a (Lcm A)
    by simp
    also from insert have Lcm A = normalize (∏ A)
    by blast
    also have lcm a ... = lcm a (∏ A)
    by (cases ∏ A = 0) (simp-all add: lcm-div-unit2)
    also from insert have coprime a (∏ A)
    by (subst coprime-commute, intro prod-coprime-left) auto
    with insert have lcm a (∏ A) = normalize (∏ (insert a A))
    by (simp add: lcm-coprime)
    finally show ?case .
  qed

lemma Lcm-coprime’:
  card A ≠ 0 ⇒ (∀ a b. a ∈ A ⇒ b ∈ A ⇒ a ≠ b ⇒ coprime a b) ⇒
  Lcm A = normalize (∏ A)
  by (rule Lcm-coprime) (simp-all add: card-eq-0-iff)

end

And some consequences: cancellation modulo m

lemma mult-mod-cancel-right:
  fixes m :: 'a::{euclidean_ring-cancel, semiring_gcd}
  assumes eq: (a * n) mod m = (b * n) mod m and coprime m n
  shows a mod m = b mod m
  proof
    have m dvd a * n − b * n
      using eq mod-eq-dvd-iff by blast
    then have m dvd a − b
      by (metis coprime m n coprime-dvd-mult-left-iff left-diff-distrib’)
    then show ?thesis
using mod-eq-dvd-iff by blast
qed

lemma mult-mod-cancel-left:
  fixes m :: 'a::{euclidean_ring_cancel,semiring_gcd}
  assumes (n * a) mod m = (n * b) mod m and coprime m n
  shows a mod m = b mod m
  by (metis assms mult.commute mult-mod-cancel-right)

87.5 GCD and LCM for multiplicative normalisation functions

class semiring_gcd_mult_normalize = semiring_gcd + normalization_semidom_multiplicative
begin

lemma mult-gcd-left: c * gcd a b = unit_factor c * gcd (c * a) (c * b)
  by (simp add: gcd_mult_left normalize_mult mult.assoc [symmetric])

lemma mult-gcd-right: gcd a b * c = gcd (a * c) (b * c) * unit_factor c
  using mult-gcd-left [of c a b] by (simp add: ac-simps)

lemma gcd-mult-distrib': normalize c * gcd a b = gcd (c * a) (c * b)
  by (subst gcd_mult_left) (simp_all add: normalize_mult)

lemma gcd-mult-distrib: k * gcd a b = gcd (k * a) (k * b) * unit_factor k
proof-
  have normalize k * gcd a b = gcd (k * a) (k * b)
    by (simp add: gcd_mult_left)
  then have normalize k * gcd a b = gcd (k * a) (k * b) * unit_factor k
    by simp
  then show thesis
    by simp
qed

lemma gcd-mult-lcm [simp]: gcd a b * lcm a b = normalize a * normalize b
  by (simp add: lcm_gcd normalize_mult dvd_normalize_div)

lemma lcm-mult-gcd [simp]: lcm a b * gcd a b = normalize a * normalize b
  using gcd_mult_lcm [of a b] by (simp add: ac-simps)

lemma mult-lcm-left: c * lcm a b = unit_factor c * lcm (c * a) (c * b)
  by (simp add: lcm_mult_left mult.assoc [symmetric] normalize_mult)

lemma mult-lcm-right: lcm a b * c = lcm (a * c) (b * c) * unit_factor c
  using mult_lcm_left [of c a b] by (simp add: ac-simps)
lemma lcm-gcd-prod: lcm a b * gcd a b = normalize (a * b)
by (simp add: lcm-gcd dvd-normalize-div normalize-mult)

lemma lcm-mult-distrib': normalize c * lcm a b = lcm (c * a) (c * b)
by (subst lcm-mult-left) (simp add: normalize-mult)

lemma lcm-mult-distrib: k * lcm a b = lcm (k * a) (k * b) * unit-factor k
proof -
  have normalize k * lcm a b = lcm (k * a) (k * b)
    by (simp add: lcm-mult-distrib')
  then have normalize k * lcm a b * unit-factor k = lcm (k * a) (k * b) * unit-factor k
    by simp
  then have normalize k * unit-factor k * lcm a b = lcm (k * a) (k * b) * unit-factor k
    by (simp only: ac-simps)
  then show ?thesis by simp
qed

lemma coprime-crossproduct':
  fixes a b c d
  assumes b ≠ 0
  assumes unit-factors: unit-factor b = unit-factor d
  assumes coprime: coprime a b coprime c d
  shows a * d = b * c ←→ a = c ∧ b = d
proof safe
  assume eq: a * d = b * c
  hence normalize a * normalize d = normalize c * normalize b
    by (simp only: normalize-mult [symmetric] mult-ac)
  with coprime have normalize b = normalize d
    by (subst (asm) coprime-crossproduct) simp-all
  from this and unit-factors show b = d
    by (rule normalize-unit-factor-eqI)
  from eq have a * d = c * d by (simp only: ‹b = d› mult-ac)
  with ‹b ≠ 0›, ‹b = d› show a = c by simp
qed (simp-all add: mult-ac)

lemma gcd-exp [simp]:
  gcd (a ^ n) (b ^ n) = gcd a b ^ n
  using gcd-exp-weak[of a n b] by (simp add: normalize-power)
end

87.6 GCD and LCM on nat and int

instantiation nat :: gcd
begin
fun gcd-nat :: nat ⇒ nat ⇒ nat
where gcd-nat x y = (if y = 0 then x else gcd y (x mod y))
definition lcm-nat :: nat ⇒ nat ⇒ nat
where lcm-nat x y = x * y div (gcd x y)

instance ..
end

instantiation int :: gcd
begin
definition gcd-int :: int ⇒ int ⇒ int
where gcd-int x y = int (gcd (nat |x|) (nat |y|))
definition lcm-int :: int ⇒ int ⇒ int
where lcm-int x y = int (lcm (nat |x|) (nat |y|))

instance ..
end

lemma gcd-int-int-eq [simp]:
    gcd (int m) (int n) = int (gcd m n)
by (simp add: gcd-int-def)

lemma gcd-nat-abs-left-eq [simp]:
    gcd (nat |k|) n = nat (gcd k (int n))
by (simp add: gcd-int-def)

lemma gcd-nat-abs-right-eq [simp]:
    gcd n (nat |k|) = nat (gcd (int n) k)
by (simp add: gcd-int-def)

lemma abs-gcd-int [simp]:
    |gcd x y| = gcd x y
for x y :: int
by (simp only: gcd-int-def)

lemma gcd-abs1-int [simp]:
    gcd |x| y = gcd x y
for x y :: int
by (simp only: gcd-int-def) simp

lemma gcd-abs2-int [simp]:
    gcd x |y| = gcd x y
for x y :: int
by (simp only: gcd-int-def) simp

lemma lcm-int-int-eq [simp]:
lcm (int m) (int n) = int (lcm m n)
by (simp add: lcm-int-def)

lemma lcm-nat-abs-left-eq [simp]:
lcm (nat |k|) n = nat (lcm k (int n))
by (simp add: lcm-int-def)

lemma lcm-nat-abs-right-eq [simp]:
lcm n (nat |k|) = nat (lcm (int n) k)
by (simp add: lcm-int-def)

lemma lcm-abs1-int [simp]:
lcm |x| y = lcm x y
for x y :: int
by (simp only: lcm-int-def) simp

lemma lcm-abs2-int [simp]:
lcm x |y| = lcm x y
for x y :: int
by (simp only: lcm-int-def) simp

lemma abs-lcm-int [simp]: |lcm i j| = lcm i j
for i j :: int
by (simp only: lcm-int-def)

lemma gcd-nat-induct [case-names base step]:
fixes m n :: nat
assumes ⋀m. P m 0
and ⋀m n. 0 < n ===> P n (m mod n) ===> P m n
shows P m n
proof (induction m n rule: gcd-nat.induct)
case (1 x y)
then show ?case
using assms neq0-conv by blast
qed

lemma gcd-neg1-int [simp]: gcd (− x) y = gcd x y
for x y :: int
by (simp only: gcd-int-def) simp

lemma gcd-neg2-int [simp]: gcd x (− y) = gcd x y
for x y :: int
by (simp only: gcd-int-def) simp

lemma gcd-cases-int:
fixes x y :: int
assumes \( x \geq 0 \implies y \geq 0 \implies P (gcd x y) \)
and \( x \geq 0 \implies y \leq 0 \implies P (gcd x (-y)) \)
and \( x \leq 0 \implies y \geq 0 \implies P (gcd (-x) y) \)
and \( x \leq 0 \implies y \leq 0 \implies P (gcd (-x) (-y)) \)
shows \( P (gcd x y) \)
using assms by auto arith

lemma gcd-ge-0-int \[simp\]: \( gcd (x::int) y \geq 0 \)
for \( x y :: \text{int} \)
by (simp add: gcd-int-def)

lemma lcm-neg1-int: \( lcm (-x) y = lcm x y \)
for \( x y :: \text{int} \)
by (simp only: lcm-int-def simp)

lemma lcm-neg2-int: \( lcm x (-y) = lcm x y \)
for \( x y :: \text{int} \)
by (simp only: lcm-int-def simp)

lemma lcm-cases-int:
fixes \( x y :: \text{int} \)
assumes \( x \geq 0 \implies y \geq 0 \implies P (lcm x y) \)
and \( x \geq 0 \implies y \leq 0 \implies P (lcm x (-y)) \)
and \( x \leq 0 \implies y \geq 0 \implies P (lcm (-x) y) \)
and \( x \leq 0 \implies y \leq 0 \implies P (lcm (-x) (-y)) \)
shows \( P (lcm x y) \)
using assms by (auto simp: lcm-neg1-int lcm-neg2-int) arith

lemma lcm-ge-0-int \[simp\]: \( lcm x y \geq 0 \)
for \( x y :: \text{int} \)
by (simp only: lcm-int-def)

lemma gcd-0-nat: \( gcd x 0 = x \)
for \( x :: \text{nat} \)
by simp

lemma gcd-0-int \[simp\]: \( gcd x 0 = \lfloor x \rfloor \)
for \( x :: \text{int} \)
by (auto simp: gcd-int-def)

lemma gcd-0-left-nat: \( gcd 0 x = x \)
for \( x :: \text{nat} \)
by simp

lemma gcd-0-left-int \[simp\]: \( gcd 0 x = \lfloor x \rfloor \)
for \( x :: \text{int} \)
by (auto simp: gcd-int-def)

lemma gcd-red-nat: \( gcd x y = gcd y (x \mod y) \)
for \( x, y :: \text{nat} \)
by (cases \( y = 0 \)) auto

Weaker, but useful for the simplifier.

lemma gcd-non-0-nat: \( y \neq 0 \implies \gcd x y = \gcd y (x \mod y) \)
for \( x, y :: \text{nat} \)
by simp

lemma gcd-1-nat [simp]: \( \gcd m 1 = 1 \)
for \( m :: \text{nat} \)
by simp

lemma gcd-Suc-0 [simp]: \( \gcd m (\text{Suc} 0) = \text{Suc} 0 \)
for \( m :: \text{nat} \)
by simp

lemma gcd-1-int [simp]: \( \gcd m 1 = 1 \)
for \( m :: \text{int} \)
by (simp add: gcd-int-def)

lemma gcd-idem-nat: \( \gcd x x = x \)
for \( x :: \text{nat} \)
by simp

lemma gcd-idem-int: \( \gcd x x = |x| \)
for \( x :: \text{int} \)
by (auto simp: gcd-int-def)

declare gcd-nat.simps [simp del]

\( \gcd m n \) divides \( m \) and \( n \). The conjunctions don’t seem provable separately.

instance nat :: semiring-gcd
proof
  fix \( m, n :: \text{nat} \)
  show \( \gcd m n \text{ dvd } m \) and \( \gcd m n \text{ dvd } n \)
  proof (induct \( m, n \))
    case (step \( m, n \))
    then have \( \gcd n (m \mod n) \text{ dvd } m \)
    by (metis dvd-mod-imp-dvd)
    with step show \( \gcd m n \text{ dvd } m \)
    by (simp add: gcd-non-0-nat)
  qed (simp-all add: gcd-0-nat gcd-non-0-nat)

next
  fix \( m, n, k :: \text{nat} \)
  assume \( k \text{ dvd } m \) and \( k \text{ dvd } n \)
  then show \( k \text{ dvd } \gcd m n \)
  by (induct \( m, n \)) (simp-all add: gcd-non-0-nat dvd-mod gcd-0-nat)

qed (simp-all add: lcm-nat-def)
instance int :: ring-gcd
proof
fix k l r :: int
show \[\text{simp}]: \text{gcd } k \text{ l dvd } k \text{ gcd } k \text{ l dvd } l\]
using gcd-dvd1 [of \text{nat} |k| \text{nat} |l|]
gcd-dvd2 [of \text{nat} |k| \text{nat} |l|]
by simp-all
show lcm k l = normalize (k * l div gcd k l)
using lcm-gcd [of \text{nat} |k| \text{nat} |l|]
by (simp add: nat-eq-iff of-nat-div abs-mult abs-div)
assume r dvd k r dvd l
then show r dvd gcd k l
using gcd-greatest [of \text{nat} |r| \text{nat} |k| \text{nat} |l|]
by simp
qed simp

lemma gcd-le1-nat [simp]: a ≠ 0 ⟹ gcd a b ≤ a
for a b :: \text{nat}
by (rule dvd-imp-le) auto

lemma gcd-le2-nat [simp]: b ≠ 0 ⟹ gcd a b ≤ b
for a b :: \text{nat}
by (rule dvd-imp-le) auto

lemma gcd-le1-int [simp]: a > 0 ⟹ gcd a b ≤ a
for a b :: \text{int}
by (rule zdvd-imp-le) auto

lemma gcd-le2-int [simp]: b > 0 ⟹ gcd a b ≤ b
for a b :: \text{int}
by (rule zdvd-imp-le) auto

lemma gcd-pos-nat [simp]: gcd m n > 0 ⟷ m ≠ 0 ∨ n ≠ 0
for m n :: \text{nat}
using gcd-eq-0-iff [of m n] by arith

lemma gcd-pos-int [simp]: gcd m n > 0 ⟷ m ≠ 0 ∨ n ≠ 0
for m n :: \text{int}
using gcd-eq-0-iff [of m n] gcd-ge-0-int [of m n] by arith

lemma gcd-unique-nat: d dvd a ∧ d dvd b ∧ (∀e. e dvd a ∧ e dvd b ⟹ e dvd d)
⟷ d = gcd a b
for d a :: \text{nat}
using gcd-unique by fastforce

lemma gcd-unique-int:
\[d ≥ 0 ∧ d dvd a ∧ d dvd b ∧ (∀e. e dvd a ∧ e dvd b ⟹ e dvd d) ⟷ d = gcd a b\]
for \( d a :: \text{int} \)
using \( \text{zdvd-antisym-nonneg} \) by auto

interpretation \( \text{gcd-nat} \):
semilattice-neutr-order \( \text{gcd} 0 :: \text{nat} \) Rings.dvd \( \lambda m n. m \text{ dvd } n \wedge m \neq n \)
by standard (auto simp: \( \text{gcd-unique-nat} \) [symmetric] intro: dvd-antisym dvd-trans)

lemma \( \text{gcd-proj1-if-dvd-int} \) [simp]: \( x \text{ dvd } y \Rightarrow \text{gcd} x y = |x| \)
for \( x y :: \text{int} \)
by (metis \( \text{abs-dvd-iff gcd-0-left-int gcd-unique-int} \))

lemma \( \text{gcd-proj2-if-dvd-int} \) [simp]: \( y \text{ dvd } x \Rightarrow \text{gcd} x y = |y| \)
for \( x y :: \text{int} \)
by (metis \( \text{gcd-proj1-if-dvd-int gcd. commute} \))

Multiplication laws.

lemma \( \text{gcd-mult-distrib-nat} \): \( k \ast \text{gcd} m n = \text{gcd} (k \ast m) (k \ast n) \)
for \( k m n :: \text{nat} \)
— [1, page 27]
by (simp add: \( \text{gcd-mult-left} \))

lemma \( \text{gcd-mult-distrib-int} \): \( |k| \ast \text{gcd} m n = \text{gcd} (k \ast m) (k \ast n) \)
for \( k m n :: \text{int} \)
by (simp add: \( \text{gcd-mult-left abs-mult} \))

Addition laws.

lemma \( \text{gcd-diff1-nat} \): \( m \geq n \Rightarrow \text{gcd} (m - n) n = \text{gcd} m n \)
for \( m n :: \text{nat} \)
by (subst \( \text{gcd-add1} \) [symmetric]) auto

lemma \( \text{gcd-diff2-nat} \): \( n \geq m \Rightarrow \text{gcd} (n - m) n = \text{gcd} m n \)
for \( m n :: \text{nat} \)
by (metis \( \text{gcd-commute gcd-add2 gcd-diff1-nat le-add-diff-inverse2} \))

lemma \( \text{gcd-non-0-int} \):
fixes \( x y :: \text{int} \)
assumes \( y > 0 \)
shows \( \text{gcd} x y = \text{gcd} y (x \text{ mod } y) \)
proof (cases \( x \text{ mod } y = 0 \))
case False
then have \( \text{neg: } x \text{ mod } y = y - (-x) \text{ mod } y \)
by (simp add: \( \text{zmod-zminus1-eq-if} \))

have \( \text{xy: } 0 \leq x \text{ mod } y \)
by (simp add: \( \text{assms} \))
show \( \text{thesis} \)
proof (cases \( x < 0 \))
case True
have \( \text{nat: } (-x \text{ mod } y) \leq \text{nat } y \)
by (simp add: \( \text{assms dual-order.order-iff-strict} \))
moreover have \( 	ext{gcd} \ (\text{nat} \ (- \ x)) \ (\text{nat} \ y) = \text{gcd} \ (\text{nat} \ (- \ x \ mod \ y)) \ (\text{nat} \ y) \)

using True assms gcd-non-0-nat nat-mod-distrib by auto
ultimately have \( \text{gcd} \ (\text{nat} \ (- \ x)) \ (\text{nat} \ y) = \text{gcd} \ (\text{nat} \ y) \ (\text{nat} \ (x \ mod \ y)) \)

using assms
by (simp add: neg nat-diff-distrib′) (metis gcd.commute gcd-diff2-nat)

with \( \text{assms} \) ‹0 ≤ x mod y› show ?thesis
by (simp add: True dual-order.order-iff-strict gcd-int-def)

next

with \( \text{assms} \) \( xy \)
have \( \text{gcd} \ (\text{nat} \ x) \ (\text{nat} \ y) = \text{gcd} \ (\text{nat} \ y) \ (\text{nat} \ (x \ mod \ nat \ y)) \)

using gcd-red-nat by blast

with False assms show ?thesis
by (simp add: gcd-int-def nat-mod-distrib)

qed

qed (use assms in auto)

lemma gcd-red-int: \( \text{gcd} \ x \ y = \text{gcd} \ y \ (x \ mod \ y) \)
for \( x \ y :: \text{int} \)
proof (cases y 0::int rule: linorder-cases)
  case less
  with gcd-non-0-int [of −y −x] show ?thesis
  by auto
next
  case greater
  with gcd-non-0-int [of y x] show ?thesis
  by auto
qed auto

lemma finite-divisors-nat [simp]:
  fixes \( m :: \text{nat} \)
  assumes \( m > 0 \)
  shows finite \{d. d dvd m\}

proof −
  from assms have \{d. d dvd m\} ⊆ \{d. d ≤ m\}
  by (auto dest: dvd-imp-le)
  then show ?thesis
  using finite-Collect-le-nat by (rule finite-subset)
qed

lemma finite-divisors-int [simp]:
  fixes \( i :: \text{int} \)
  assumes \( i ≠ 0 \)
  shows finite \{d. d dvd i\}

proof −
  have \{d. |d| ≤ |i|\} = \{-|i|..|i|\}

by (auto simp: abs-if)
then have finite {d. |d| ≤ |i|}
  by simp

from finite-subset [OF - this] show ?thesis
  using assms by (simp add: dvd-imp-le-int subset-iff)
qed

lemma Max-divisors-self-nat [simp]:
n ≠ 0 ⇒ Max {d::nat. d dvd n} = n
by (fastforce intro: antisym Max-le-iff[THEN iffD2] simp: dvd-imp-le)

lemma Max-divisors-self-int [simp]:
assumes n ≠ 0 shows Max {d::int. d dvd n} = |n|
proof (rule antisym)
  show Max {d. d dvd n} ≤ |n|
    using assms by (auto intro: abs-le-D1 dvd-imp-le-int intro: Max-le-iff[THEN iffD2])
  qed (simp add: assms)

lemma gcd-is-Max-divisors-nat:
fixes m n :: nat
assumes n > 0 shows gcd m n = Max {d. d dvd m ∧ d dvd n}
proof (rule Max-eqI[THEN sym], simp-all)
  show finite {d. d dvd m ∧ d dvd n}
    by (simp add: ‹n > 0›)
  show ∀y. y dvd m ∧ y dvd n ⇒ y ≤ gcd m n
    by (simp add: ‹n > 0› dvd-imp-le)
qed

lemma gcd-is-Max-divisors-int:
fixes m n :: int
assumes n ≠ 0 shows gcd m n = Max {d. d dvd m ∧ d dvd n}
proof (rule Max-eqI[THEN sym], simp-all)
  show finite {d. d dvd m ∧ d dvd n}
    by (simp add: ‹n ≠ 0›)
  show ∀y. y dvd m ∧ y dvd n ⇒ y ≤ gcd m n
    by (simp add: ‹n ≠ 0› zdvd-imp-le)
qed

lemma gcd-code-int [code]: gcd k l = |if l = 0 then k else gcd l (|k| mod |l|)|
for k l :: int
using gcd-red-int [of |k| |l|] by simp

lemma coprime-Suc-left-nat [simp]:
coprime (Suc n) n
using coprime-add-one-left [of n] by simp

lemma coprime-Suc-right-nat [simp]:
coprime n (Suc n)
using coprime-Suc-left-nat [of n] by (simp add: ac-simps)
lemma coprime-diff-one-left-nat [simp]:
coprime (n − 1) n if n > 0 for n :: nat
using that coprime-Suc-right-nat [of n − 1] by simp

lemma coprime-diff-one-right-nat [simp]:
coprime n (n − 1) if n > 0 for n :: nat
using that coprime-diff-one-left-nat [of n] by (simp add: ac-simps)

lemma coprime-crossproduct-nat:
  fixes a b c d :: nat
  assumes coprime a d and coprime b c
  shows a * c = b * d ←→ a = b ∧ c = d
  using assms coprime-crossproduct [of a d b c]
  by simp

lemma coprime-crossproduct-int:
  fixes a b c d :: int
  assumes coprime a d and coprime b c
  shows |a| * |c| = |b| * |d| ←→ |a| = |b| ∧ |c| = |d|
  using assms coprime-crossproduct [of a d b c]
  by simp

87.7 Bezout’s theorem

Function bezw returns a pair of witnesses to Bezout’s theorem – see the theorems that follow the definition.

fun bezw :: nat ⇒ nat ⇒ int * int
where bezw x y =
  (if y = 0 then (1, 0)
  else (snd (bezw y (x mod y)),
      fst (bezw y (x mod y)) − snd (bezw y (x mod y)) * int(x div y)))

lemma bezw-0 [simp]: bezw x 0 = (1, 0)
  by simp

lemma bezw-non-0:
y > 0 ⇒ bezw x y =
  (snd (bezw y (x mod y)), fst (bezw y (x mod y)) − snd (bezw y (x mod y)) * int(x div y))
  by simp

declare bezw.simps [simp del]

lemma bezw-aux: int (gcd x y) = fst (bezw x y) * int x + snd (bezw x y) * int y
proof (induct x y rule: gcd-nat-induct)
case (step m n)
  then have fst (bezw m n) * int m + snd (bezw m n) * int n − int (gcd m n)
    = int m * snd (bezw n (m mod n)) −
THEORY “GCD”

\[
\begin{align*}
\text{by (simp add: bezw-non-0 gcd-non-0-nat field-simps)}
\end{align*}
\]
also have \ldots = int m * snd (bezw n (m mod n)) - (int (m mod n) + int (n * (m div n))) * snd (bezw n (m mod n))
\begin{align*}
\text{by (simp add: distrib-right)}
\end{align*}
also have \ldots = 0
\begin{align*}
\text{by (metis cancel-comm-monoid-add-class.diff-cancel mod-mult-div-eq of-nat-add)}
\end{align*}
finally show \?case
\begin{align*}
\text{by simp}
\end{align*}
qed auto

lemma bezout-int: \(\exists u v. u * x + v * y = \gcd x y\)
for \(x y :: \int\)
proof
have aux: \(x \geq 0 \implies y \geq 0 \implies \exists u v. u * x + v * y = \gcd x y\) for \(x y :: \int\)
\begin{align*}
\text{apply (rule_tac x = \text{fst} (bezw (nat x) (nat y)) in exI)}
\end{align*}
\begin{align*}
\text{apply (rule_tac x = \text{snd} (bezw (nat x) (nat y)) in exI)}
\end{align*}
\begin{align*}
\text{by (simp add: bezw-aux gcd-int-def)}
\end{align*}
consider \(x \geq 0\)
\(y \geq 0\)
\(x \geq 0\)
\(y \leq 0\)
\(x \leq 0\)
\(y \geq 0\)
\(x \leq 0\)
\(y \leq 0\)
\begin{align*}
\text{using linear by blast}
\end{align*}
then show \?thesis
proof cases
\begin{align*}
\text{case 1}
\end{align*}
then show \?thesis by (rule aux)
next
\begin{align*}
\text{case 2}
\end{align*}
then show \?thesis
\begin{align*}
\text{using aux [of x \text{ - } y]}
\end{align*}
\begin{align*}
\text{by (metis gcd-neg2-int mult.commute mult-minus-right neg-0-le-iff-le)}
\end{align*}
next
\begin{align*}
\text{case 3}
\end{align*}
then show \?thesis
\begin{align*}
\text{using aux [of \text{ - } x \text{ - } y]}
\end{align*}
\begin{align*}
\text{by (metis gcd.commute gcd-neg2-int mult.commute mult-minus-right neg-0-le-iff-le)}
\end{align*}
next
\begin{align*}
\text{case 4}
\end{align*}
then show \?thesis
\begin{align*}
\text{using aux [of \text{ - } x \text{ - } y]}
\end{align*}
\begin{align*}
\text{by (metis diff-0 diff-ge-0-iff-ge gcd-neg1-int gcd-neg2-int mult.commute mult-minus-right)}
\end{align*}
qed

\text{Versions of Bezout for} \text{ nat, by Amine Chaieb.}

lemma Euclid-induct [case-names swap zero add]:
fixes \(P :: \text{ nat \Rightarrow nat \Rightarrow bool}\)
assumes \(c: \bigwedge a b. P a b \leftrightarrow P b a\)
and \(z: \bigwedge a. P a 0\)
and add: \( \forall a \ b. \ P \ a \ b \rightarrow P \ a \ (a + b) \)

shows \( P \ a \ b \)

proof (induct \( a + b \) arbitrary: \( a \ b \) rule: less-induct)

case less

consider (eq) \( a = b \) | (lt) \( a < b \ a + b - a < a + b \) | \( b = 0 \) | \( b + a - b < a + b \)

by arith

show ?case

proof (cases \( a \ b \) rule: linorder-cases)

case equal

with add [rule-format, OF \( z \) [rule-format, of \( a \)]] show ?thesis by simp

next

case lt: less

then consider \( a = 0 \) | \( a + b - a < a + b \) by arith

then show ?thesis

proof cases

case 1

with \( z \ c \) show ?thesis by blast

next

case 2

also have \( \ast: \ a + b - a = a + (b - a) \) using lt by arith

finally have \( a + (b - a) < a + b \).

then have \( P \ a \ (a + (b - a)) \) by (rule add [rule-format, OF less])

then show ?thesis by (simp add: \( \ast \)[symmetric])

qed

next

case gt: greater

then consider \( b = 0 \) | \( b + a - b < a + b \) by arith

then show ?thesis

proof cases

case 1

with \( z \ c \) show ?thesis by blast

next

case 2

also have \( \ast: \ b + a - b = b + (a - b) \) using gt by arith

finally have \( b + (a - b) < a + b \).

then have \( P \ b \ (b + (a - b)) \) by (rule add [rule-format, OF less])

then have \( P \ b \ a \) by (simp add: \( \ast \)[symmetric])

with \( c \) show ?thesis by blast

qed

qed

lemma bezout-lemma-nat:

fixes \( d :: \mathbb{N} \)

shows \( \exists d \ \text{dvd} \ a; \ d \ \text{dvd} \ b; \ a \times x = b \times y + d \ \vee \ b \times x = a \times y + d \)

\( \rightarrow \exists x \ y. \ d \ \text{dvd} \ a \ \wedge \ d \ \text{dvd} \ a + b \ \wedge (a \times x = (a + b) \times y + d \ \vee (a + b) \times x = a \times y + d) \)

apply auto

apply (metis add-mult-distrib2 left-add-mult-distrib)
apply (rule-tac x=x in ex1)
by (metis add-mult-distrib2 mult.commute add.assoc)

lemma bezout-add-nat:
\[ \exists (d::nat) x y. d dvd a \land d dvd b \land (a \ast x = b \ast y + d \lor b \ast x = a \ast y + d) \]
proof (induct a b rule: Euclid-induct)
case (swap a b)
then show ?case
  by blast
next
case (zero a)
then show ?case
  by fastforce
next
case (add a b)
then show ?case
  by (meson bezout-lemma-nat)
qed

lemma bezout1-nat: \[ \exists (d::nat) x y. d dvd a \land d dvd b \land (a \ast x - b \ast y = d \lor b \ast x - a \ast y = d) \]
using bezout-add-nat[of a b]
by (metis add-diff-cancel-left')

lemma bezout-add-strong-nat:
fixes a b :: nat
assumes a: a \neq 0
shows \[ \exists d x y. d dvd a \land d dvd b \land a \ast x = b \ast y + d \]
proof -
consider d x y where d dvd a d dvd b a \ast x = b \ast y + d
| d x y where d dvd a d dvd b b \ast x = a \ast y + d
  using bezout-add-nat[of a b] by blast
then show ?thesis
proof cases
  case 1
  then show ?thesis by blast
next
case H: 2
show ?thesis
proof (cases b = 0)
case True
  with H show ?thesis by simp
next
case False
then have by: b > 0 by simp
  with dvd-imp-le[OF H(2)] consider d = b \mid d < b
  by atomize-elim auto
then show ?thesis
proof cases
  case 1
with a H show ?thesis
  by (metis Suc-pred.add.commute mult.commute mult-Suc-right neq0-conv)
next
  case 2
  show ?thesis
  proof (cases x = 0)
    case True
    with a H show ?thesis by simp
  next
    case x0: False
    then have xp: x > 0 by simp
    from d < b have d ≤ b − 1 by simp
    then have d * b ≤ b * (b − 1) by simp
    with xp mult-mono[of 1 x d * b b * (b − 1)]
    have dble: d * b ≤ x * b * (b − 1) using bp by simp
    from H(3) have d + (b − 1) * (b * x) = d + (b − 1) * (a * y + d)
      by simp
    then have a * ((b − 1) * y) + d * (b − 1 + 1) = d + x * b * (b − 1)
      by algebra
    then have a * ((b − 1) * y) = d + x * b * (b − 1) − d * b
      using bp by simp
    then have a * ((b − 1) * y) = d + (x * b * (b − 1) − d * b)
      by (simp only: diff-add-assoc[of dble, of d, symmetric])
    then have a * ((b − 1) * y) = b * (x * (b − 1) − d) + d
      by (simp only: diff-mult-distrib2 ac-simps)
    with H(1,2) show ?thesis
      by blast
    qed
  qed
  qed
  qed

lemma bezout-nat:
  fixes a :: nat
  assumes a: a ≠ 0
  shows ∃x y. a * x = b * y + gcd a b
proof
  obtain d x y where d: d dvd a d dvd b and eq: a * x = b * y + d
    using bezout-add-strong-nat [OF a, of b] by blast
  from d have d dvd gcd a b
    by simp
  then obtain k where k: gcd a b = d * k
    unfolding dvd-def by blast
  from eq have a * x * k = (b * y + d) * k
    by auto
  then have a * (x * k) = b * (y * k) + gcd a b
by (algebra add: \( k \))
then show \( \text{thesis} \)
by blast
qed

87.8 LCM properties on \( \text{nat} \) and \( \text{int} \)

**lemma lcm-altdef-int** [code]: \( \text{lcm} \ a \ b = |a| \ast |b| \ div \ \text{gcd} \ a \ b \)

_for\ a \ b :: \text{int} _by (simp add: abs-mult lcm-gcd abs-div)

**lemma prod-gcd-lcm-nat**: \( m \ast n = \text{gcd} \ m \ n \ast \text{lcm} \ m \ n \)
_for\ m \ n :: \text{nat} _by (simp add: lcm-gcd)

**lemma prod-gcd-lcm-int**: \( |m| \ast |n| = \text{gcd} \ m \ n \ast \text{lcm} \ m \ n \)
_for\ m \ n :: \text{int} _by (simp add: lcm-gcd abs-div abs-mult)

**lemma lcm-pos-nat**: \( m > 0 \Rightarrow n > 0 \Rightarrow \text{lcm} \ m \ n > 0 \)
_for\ m \ n :: \text{nat} _using lcm-eq-0-iff [of m n] by auto

**lemma lcm-pos-int**: \( m \neq 0 \Rightarrow n \neq 0 \Rightarrow \text{lcm} \ m \ n > 0 \)
_for\ m \ n :: \text{int} _by (simp add: less-le lcm-eq-0-iff)

**lemma dvd-pos-nat**: \( n > 0 \Rightarrow m \ dvd \ n \Rightarrow m > 0 \)
_for\ m \ n :: \text{nat} _by auto

**lemma lcm-unique-nat**:
\( a \ dvd \ d \land b \ dvd \ d \land (\forall e. \ a \ dvd \ e \land b \ dvd \ e \arrow d \ dvd \ e) \arrow d = \text{lcm} \ a \ b \)
_for\ a \ b \ d :: \text{nat} _by (auto intro: dvd-antisym lcm-least)

**lemma lcm-unique-int**:
\( d \geq 0 \land a \ dvd \ d \land b \ dvd \ d \land (\forall e. \ a \ dvd \ e \land b \ dvd \ e \arrow d \ dvd \ e) \arrow d = \text{lcm} \ a \ b \)
_for\ a \ b \ d :: \text{int} _using lcm-least zdvd-antisym-nonneg by auto

**lemma lcm-proj2-if-dvd-nat** [simp]: \( x \ dvd \ y \arrow \text{lcm} \ x \ y = y \)
_for\ x \ y :: \text{nat} _by (simp add: lcm-proj2-if-dvd)

**lemma lcm-proj2-if-dvd-int** [simp]: \( x \ dvd \ y \arrow \text{lcm} \ x \ y = |y| \)
_for\ x \ y :: \text{int} _by (simp add: lcm-proj2-if-dvd)
lemma lcm-proj1-if-dvd-nat [simp]: \( x \) dvd \( y \) \( \Rightarrow \) \( \text{lcm} \ y \ x = y \)
  for \( x \ y :: \text{nat} \)
  by (subst \text{lcm}\_\text{commute}) (erule lcm-proj2-if-dvd-nat)

lemma lcm-proj1-if-dvd-int [simp]: \( x \) dvd \( y \) \( \Rightarrow \) \( \text{lcm} \ y \ x = \lvert y \rvert \)
  for \( x \ y :: \text{int} \)
  by (subst \text{lcm}\_\text{commute}) (erule lcm-proj2-if-dvd-int)

lemma lcm-proj1-iff-nat [simp]: \( \text{lcm} \ m \ n = m \iff \) \( n \) dvd \( m \)
  for \( m \ n :: \text{nat} \)
  by (metis lcm-proj1-if-dvd-nat lcm-unique-nat)

lemma lcm-proj2-iff-nat [simp]: \( \text{lcm} \ m \ n = n \iff \) \( m \) dvd \( n \)
  for \( m \ n :: \text{nat} \)
  by (metis lcm-proj2-if-dvd-nat lcm-unique-nat)

lemma lcm-proj1-iff-int [simp]: \( \text{lcm} \ m \ n = \lvert m \rvert \iff \) \( n \) dvd \( m \)
  for \( m \ n :: \text{int} \)
  by (metis dvd-abs-iff lcm-proj1-if-dvd-int lcm-unique-int)

lemma lcm-proj2-iff-int [simp]: \( \text{lcm} \ m \ n = \lvert n \rvert \iff \) \( m \) dvd \( n \)
  for \( m \ n :: \text{int} \)
  by (metis dvd-abs-iff lcm-proj2-if-dvd-int lcm-unique-int)

lemma lcm-1-iff-nat [simp]: \( \text{lcm} \ m \ n = \text{Suc} \ 0 \iff \) \( m = \text{Suc} \ 0 \land n = \text{Suc} \ 0 \)
  for \( m \ n :: \text{nat} \)
  using lcm-eq-1-iff [of \( m \ n \)] by simp

lemma lcm-1-iff-int [simp]: \( \text{lcm} \ m \ n = 1 \iff \) \( m = 1 \lor m = -1 \) \( \land \) \( n = 1 \lor n = -1 \)
  for \( m \ n :: \text{int} \)
  by auto

87.9 The complete divisibility lattice on nat and int

Lifting \( \text{gcd} \) and \( \text{lcm} \) to sets (\( \text{Gcd} / \text{Lcm} \)). \( \text{Gcd} \) is defined via \( \text{Lcm} \) to facilitate the proof that we have a complete lattice.

instantiation nat :: semiring-Gcd

begin

interpretation semilattice-neutr-set \text{lcm} :: nat
  by standard simp-all

definition \( \text{Lcm} \ M = (\text{if finite} \ M \ \text{then} \ F \ M \ \text{else} \ 0) \) for \( M :: \text{nat set} \)

lemma Lcm-nat-empty: \( \text{Lcm} \ \{\} = (1 :: \text{nat}) \)
  by (simp add: \text{Lcm}-nat-def del: \text{One}-nat-def)
lemma Lcm-nat-insert: \( \text{lcm} (\text{insert } n M) = \text{lcm} n (\text{lcm } M) \) for \( n :: \text{nat} \)
by (cases finite M) (auto simp: Lcm-nat-def simp del: One-nat-def)

lemma Lcm-nat-infinite: \( \text{infinite } M \implies \text{lcm } M = 0 \) for \( M :: \text{nat set} \)
by (simp add: Lcm-nat-def)

lemma dvd-Lcm-nat [simp]:
fixes \( M :: \text{nat set} \)
assumes \( m \in M \)
shows \( m \text{ dvd } \text{lcm } M \)
proof –
from assms have \( \text{insert } m M = M \)
by auto
moreover have \( m \text{ dvd } \text{lcm } (\text{insert } m M) \)
by (simp add: Lcm-nat-insert)
ultimately show \( \text{?thesis} \)
by simp
qed

lemma Lcm-dvd-nat [simp]:
fixes \( M :: \text{nat set} \)
assumes \( \forall m \in M. m \text{ dvd } n \)
shows \( \text{lcm } M \text{ dvd } n \)
proof (cases \( n > 0 \))
case False
then show \( \text{?thesis} \) by simp
next
case True
then have \( \text{finite } \{d. d \text{ dvd } n\} \)
by (rule finite-divisors-nat)
moreover have \( M \subseteq \{d. d \text{ dvd } n\} \)
using assms by fast
ultimately have \( \text{finite } M \)
by (rule rev-finite-subset)
then show \( \text{?thesis} \)
using assms by (induct M) (simp-all add: Lcm-nat-empty Lcm-nat-insert)
qed

definition \( \text{Gcd } M = \text{lcm } \{d. \forall m \in M. d \text{ dvd } m\} \) for \( M :: \text{nat set} \)

instance
proof
fix \( N :: \text{nat set} \)
fix \( n :: \text{nat} \)
show \( \text{Gcd } N \text{ dvd } n \) if \( n \in N \)
using that by (induct N rule: infinite-finite-induct) (auto simp: Gcd-nat-def)
show \( n \text{ dvd } \text{Gcd } N \) if \( \forall m. m \in N \implies n \text{ dvd } m \)
using that by (induct N rule: infinite-finite-induct) (auto simp: Gcd-nat-def)
show \( n \text{ dvd } \text{lcm } N \) if \( n \in N \)
using that by (induct N rule: infinite-finite-induct) auto
show Lcm N dvd n if \( \forall m, m \in N \implies m \text{ dvd } n \)
using that by (induct N rule: infinite-finite-induct) auto
show normalize (Gcd N) = Gcd N and normalize (Lcm N) = Lcm N
by simp-all
qed
end

lemma Gcd-nat-eq-one: \( 1 \in N \implies \text{Gcd } N = 1 \)
for N :: nat set
by (rule Gcd-eq-1-I) auto

instance nat :: semiring-gcd-mult-normalize
by intro-classes (auto simp: unit-factor-nat-def)

Alternative characterizations of Gcd:

lemma Gcd-eq-Max:
fixes M :: nat set
assumes finite (M::nat set) and M \( \neq \) {} and 0 \( \notin \) M
shows Gcd M = Max (\( \bigcap m \in M. \) \( \{ d. d \text{ dvd } m \} \))
proof (rule antisym)
from assms obtain m where m \( \in \) M and m > 0
by auto
from \( \langle m > 0 \rangle \) have finite \( \{ d. d \text{ dvd } m \} \)
by (blast intro: finite-divisors-nat)
with \( \langle m \in M \rangle \) have fin: finite (\( \bigcap m \in M. \) \( \{ d. d \text{ dvd } m \} \))
by blast
from fin show Gcd M \( \leq \) Max (\( \bigcap m \in M. \) \( \{ d. d \text{ dvd } m \} \))
by (auto intro: Max-ge Gcd-dvd)
from fin show Max (\( \bigcap m \in M. \) \( \{ d. d \text{ dvd } m \} \)) \( \leq \) Gcd M
proof (rule Max.boundedI, simp-all)
show (\( \bigcap m \in M. \) \( \{ d. d \text{ dvd } m \} \)) \( \neq \) {}
by auto
show \( \forall a. \forall x \in M. a \text{ dvd } x \implies a \leq \text{Gcd } M \)
by (meson Gcd-dvd Gcd-greatest \( \langle 0 < m \rangle \) \( \langle m \in M \rangle \) dvd-imp-le dvd-pos-nat)
qed

lemma Gcd-remove0-nat: finite M \( \implies \) Gcd M = Gcd (M - \{0\})
for M :: nat set
proof (induct pred: finite)
case (insert x M)
then show \?case
by (simp add: insert-Diff-if)
qed auto

lemma Lcm-in-lcm-closed-set-nat:
fixes M :: nat set

assumes finite $M \neq \{\} \land m, n. [m \in M; n \in M] \Longrightarrow lcm m n \in M$

shows $Lcm M \in M$

using assms

proof (induction $M$ rule: finite-linorder-min-induct)

case (insert $x$ $M$)

then have $\land m, n. [m \in M; n \in M] \Longrightarrow lcm m n \in M$

by (metis dvd-lcm1 gr0I insert-iff lcm-pos-nat nat-dvd-not-less)

with insert show ?case

by simp (metis Lcm-nat-empty One-nat-def dvd-1-left dvd-lcm2)

qed auto

lemma Lcm-eq-Max-nat:

fixes $M :: \text{nat set}$

assumes $M :: \text{finite } M \neq \{\} \land 0 \notin M \land lcm :: \land m, n. [m \in M; n \in M] \Longrightarrow lcm m n \in M$

shows $Lcm M = \text{Max } M$

proof (rule antisym)

show $Lcm M \leq \text{Max } M$

by (simp add: Lcm-in-lcm-closed-set-nat \text{finite } M \land M \neq \{\}$)

show $\text{Max } M \leq \text{Lcm } M$

by (meson Lcm-0-iff Max-in M dvd-Lcm dvd-imp-le le-0-eq not-le)

qed

lemma mult-inj-if-coprime-nat:

inj-on $f A \Longrightarrow \text{inj-on } g B \Longrightarrow (\land a, b. [a \in A; b \in B] \Longrightarrow \text{coprime } (f a) (g b)) \Longrightarrow \text{inj-on } (\lambda (a, b). f a * g b) (A \times B)$

for $f :: 'a \Rightarrow \text{nat}$ and $g :: 'b \Rightarrow \text{nat}$

by (auto simp: inj-on-def coprime-crossproduct-nat simp del: One-nat-def)

87.9.1 Setwise GCD and LCM for integers

instantiation int :: Gcd

begin

definition Gcd-int :: int set $\Rightarrow$ int

where $\text{Gcd } K = \text{int } (GCD k \in K. (\text{nat } \circ \text{abs} ) k)$

definition Lcm-int :: int set $\Rightarrow$ int

where $\text{Lcm } K = \text{int } (LCM k \in K. (\text{nat } \circ \text{abs} ) k)$

instance..

end

lemma Gcd-int-eq [simp]:

$(GCD n \in N. \text{int } n) = \text{int } (\text{Gcd } N)$

by (simp add: Gcd-int-def image-image)

lemma Gcd-nat-abs-eq [simp]:
(\text{GCD} \ k \in K. \ \text{nat} \ |k|) = \text{nat} \ (\text{gcd} \ K)
\text{by} \ \text{(simp add: gcd-int-def)}

\textbf{lemma} abs-Gcd-eq \ [\text{simp}]:
|Gcd \ K| = \text{Gcd} \ K \ \text{for} \ K :: \text{int} \ \text{set}
\text{by} \ \text{(simp only: gcd-int-def)}

\textbf{lemma} Gcd-int-greater-eq-0 \ [\text{simp}]:
Gcd \ K \geq 0
\text{for} \ K :: \text{int} \ \text{set}
\text{using} \ \text{abs-ge-zero} \ [\text{of} \ \text{Gcd} \ K] \ \text{by} \ \text{simp}

\textbf{lemma} Gcd-abs-eq \ [\text{simp}]:
(\text{GCD} \ k \in K. \ |k|) = \text{Gcd} \ K
\text{for} \ K :: \text{int} \ \text{set}
\text{by} \ \text{(simp only: gcd-int-def image-image) simp}

\textbf{lemma} Lcm-int-eq \ [\text{simp}]:
(\text{LCM} \ n \in N. \ \text{int} \ n) = \text{int} \ (\text{lcm} \ N)
\text{by} \ \text{(simp add: lcm-int-def image-image)}

\textbf{lemma} Lcm-nat-abs-eq \ [\text{simp}]:
(\text{LCM} \ k \in K. \ \text{nat} \ |k|) = \text{nat} \ (\text{lcm} \ K)
\text{by} \ \text{(simp add: lcm-int-def)}

\textbf{lemma} abs-Lcm-eq \ [\text{simp}]:
|Lcm \ K| = \text{Lcm} \ K \ \text{for} \ K :: \text{int} \ \text{set}
\text{by} \ \text{(simp only: lcm-int-def) simp}

\textbf{lemma} Lcm-int-greater-eq-0 \ [\text{simp}]:
Lcm \ K \geq 0
\text{for} \ K :: \text{int} \ \text{set}
\text{using} \ \text{abs-ge-zero} \ [\text{of} \ \text{Lcm} \ K] \ \text{by} \ \text{simp}

\textbf{lemma} Lcm-abs-eq \ [\text{simp}]:
(\text{LCM} \ k \in K. \ |k|) = \text{Lcm} \ K
\text{for} \ K :: \text{int} \ \text{set}
\text{by} \ \text{(simp only: lcm-int-def image-image) simp}

\textbf{instance} \ \text{int} :: \text{semiring-Gcd}
\textbf{proof}
\textbf{fix} \ K :: \text{int} \ \text{set} \ \text{and} \ k :: \text{int}
\textbf{show} \ \text{Gcd} \ K \ \text{dvd} \ k \ \text{and} \ k \ \text{dvd} \ \text{lcm} \ K \ \text{if} \ k \in K
\text{using} \ \text{that} \ \text{Gcd-dvd} \ [\text{of} \ \text{nat} \ |k| \ (\text{nat} \ o \ \text{abs}) \ \cdot \ K]
\ \text{dvd-Lcm} \ [\text{of} \ \text{nat} \ |k| \ (\text{nat} \ o \ \text{abs}) \ \cdot \ K]
\ \text{by} \ \text{(simp-all add: comp-def)}
\textbf{show} \ k \ \text{dvd} \ \text{Gcd} \ K \ \text{if} \ \bigwedge \ l. \ l \in K \implies k \ \text{dvd} \ l
\textbf{proof} \ −
\textbf{have} \ \text{nat} \ |k| \ \text{dvd} \ (\text{GCD} \ k \in K. \ \text{nat} \ |k|)
by (rule Gcd-greatest) (use that in auto)
then show thesis by simp
qed
show \( Lcm K \) dvd \( k \) if \( \forall l. l \in K \implies l \) dvd \( k \)
proof
have \( (LCM k \in K. \text{nat } \vert k\vert) \) dvd \( \text{nat } \vert k\vert \)
by (rule Lcm-least) (use that in auto)
then show thesis by simp
qed
qed (simp-all add: sgn-mult)

instance int :: semiring-gcd-mult-normalize
by intro-classes (auto simp: sgn-mult)

87.10 GCD and LCM on integer

instantiation integer :: gcd
begin

context
includes integer.lifting
begin

lift-definition gcd-integer :: integer \mapsto integer \mapsto integer is gcd.

lift-definition lcm-integer :: integer \mapsto integer \mapsto integer is lcm.

end

instance ..

end

lifting-update integer.lifting
lifting-forget integer.lifting

context
includes integer.lifting
begin

lemma gcd-code-integer [code]: \( \text{gcd } k \) \( l \) = \( \text{if } l = (0::\text{integer}) \, \text{then } k \, \text{else } \text{gcd } l \, (\vert k \vert \) \mod \( \vert l \vert \))\)
by transfer (fact gcd-code-int)

lemma lcm-code-integer [code]: \( \text{lcm } a \) \( b \) = \( \vert a \vert \) \* \( \vert b \vert \) \div \text{gcd } a \) \( b \)
for \( a \) \( b :: \text{integer} \)
by transfer (fact lcm-altdef-int)

end
THEORY "GCD"

code-printing
constant gcd :: integer ⇒ integer

(OCaml) \( \text{fun } k \ l \Rightarrow \text{if } Z.\text{equal } k \ Z.\text{zero} \text{ then } Z.\text{abs } k \ \text{else } Z.\text{gcd } k \ l \) \n
(Haskell) Prelude.gcd

(Scala) -.gcd((-)')

— There is no gcd operation in the SML standard library, so no code setup for SML.

Some code equations

lemmas Lcm-nat-set-eq-fold [code] = Lcm-set-eq-fold [where ?'a = nat]
lemmas Lcm-int-set-eq-fold [code] = Lcm-set-eq-fold [where ?'a = int]

Fact aliases.

lemma lcm-0-iff-nat [simp]; lcm m n = 0 \iff m = 0 \lor n = 0
for m n :: nat
by (fact lcm-eq-0-iff)

lemma lcm-0-iff-int [simp]; lcm m n = 0 \iff m = 0 \lor n = 0
for m n :: int
by (fact lcm-eq-0-iff)

lemma dvd-lcm-I1-nat [simp]; k dvd m \Longrightarrow k dvd lcm m n
for k m n :: nat
by (fact dvd-lcmI1)

lemma dvd-lcm-I2-nat [simp]; k dvd n \Longrightarrow k dvd lcm m n
for k m n :: nat
by (fact dvd-lcmI2)

lemma dvd-lcm-I1-int [simp]; i dvd m \Longrightarrow i dvd lcm m n
for i m n :: int
by (fact dvd-lcmI1)

lemma dvd-lcm-I2-int [simp]; i dvd n \Longrightarrow i dvd lcm m n
for i m n :: int
by (fact dvd-lcmI2)

lemmas Gcd-dvd-int [simp] = Gcd-dvd [where ?'a = int]
lemmas Gcd-greatest-nat [simp] = Gcd-greatest [where ?'a = nat]
lemmas Gcd-greatest-int [simp] = Gcd-greatest [where ?'a = int]

lemma dvd-Lcm-int [simp]; m \in M \Longrightarrow m dvd Lcm M
for M :: int set
by (fact dvd-Lcm)
lemma gcd-neg-numeral-1-int [simp]: gcd (─ numeral n :: int) x = gcd (numeral n) x
  by (fact gcd-neg1-int)

lemma gcd-neg-numeral-2-int [simp]: gcd x (─ numeral n :: int) = gcd x (numeral n)
  by (fact gcd-neg2-int)

lemma gcd-proj1-if-dvd-nat [simp]: x dvd y =⇒ gcd x y = x for x y :: nat
  by (fact gcd-nat.absorb1)

lemma gcd-proj2-if-dvd-nat [simp]: y dvd x =⇒ gcd x y = y for x y :: nat
  by (fact gcd-nat.absorb2)

lemma Gcd-in:
  fixes A :: nat set
  assumes ⋀ a b. a ∈ A =⇒ b ∈ A =⇒ gcd a b ∈ A
  assumes A ≠ {} shows Gcd A ∈ A
  proof (cases A = {0})
    case False
    with assms obtain x where x ∈ A x > 0
      by auto
    thus Gcd A ∈ A
    proof (induction x rule: less-induct)
      case (less x)
      show ?case
      proof (cases x = Gcd A)
        case False
        have ∃ y ∈ A. ¬x dvd y
          using False less.prems by (metis Gcd-dvd Gcd-greatest-nat gcd-nat.asym)
        then obtain y where y ∈ A ¬x dvd y
          by blast
        have gcd x y ∈ A
          by (rule assms(1)) (use x ∈ A y in auto)
        moreover have gcd x y < x
          using ⟨x > 0⟩ y by (metis gcd-dvd1 gcd-dvd2 nat-dvd-not-less nat-neq-iff)
        moreover have gcd x y > 0
          using ⟨x > 0⟩ by auto
        ultimately show ?thesis using less.IH by blast
      qed
      qed auto
    qed
  qed

lemma bezout-gcd-nat':
  fixes a b :: nat
shows $\exists \, x \cdot b \cdot y \leq a \cdot x \land a \cdot x - b \cdot y = \gcd a \cdot b \lor a \cdot y \leq b \cdot x \land b \cdot x
$ using \texttt{bezout-nat[of a b]}
b by (metis add-diff-cancel-left \texttt{diff-zero \texttt{gcd.commute gcd-0-nat}}
\texttt{le-addSame-cancel1 mult.right-neutral zero-le})

lemmas \texttt{Lcm-eq-0-I-nat [simp]} = \texttt{Lcm-eq-0-I [where \texttt{?\texttt{a} = nat]}}
lemmas \texttt{Lcm-0-iff-nat [simp]} = \texttt{Lcm-0-iff [where \texttt{?\texttt{a} = nat]}}
lemmas \texttt{Lcm-least-int [simp]} = \texttt{Lcm-least [where \texttt{?\texttt{a} = int]}}

87.11 Characteristic of a semiring

definition (in \texttt{semiring-1}) \texttt{semiring-char :: \texttt{'a itself \Rightarrow nat}}
\texttt{where semiring-char - = Gcd \{n. of-nat n = (0 :: 'a)\}}

class \texttt{semiring-1}
begin

context
fixes \texttt{CHAR :: nat}
defines \texttt{CHAR \equiv semiring-char (Pure.type :: 'a itself)}
begin

lemma \texttt{of-nat-CHAR [simp]: of-nat CHAR = (0 :: 'a)}
proof –
have \texttt{CHAR \in \{n. of-nat n = (0::'a)\}}
unfolding \texttt{CHAR-def semiring-char-def}
proof (rule \texttt{Gcd-in, clarify})
fix \texttt{a b :: nat}
assume \texttt{*: of-nat a = (0 :: 'a) of-nat b = (0 :: 'a)}
show \texttt{of-nat (gcd a b) = (0 :: 'a)}
proof (cases \texttt{a = 0})
case False
with \texttt{bezout-nat} obtain \texttt{x y where a \cdot x = b \cdot y + \texttt{gcd a b}}
by blast
hence \texttt{of-nat (a \cdot x) = (of-nat (b \cdot y + \texttt{gcd a b}) :: 'a)}
by (rule \texttt{arg-cong})
thus \texttt{of-nat (gcd a b) = (0 :: 'a)}
using \texttt{* by simp}
qed (use \texttt{* in auto})
next
have \texttt{of-nat 0 = (0 :: 'a)}
by simp
thus \texttt{\{n. of-nat n = (0 :: 'a)\} \neq \{\}}
by blast
qed
thus \texttt{?thesis}
by simp
qed
lemma of-nat-eq-0-iff-char-dvd: of-nat n = (0 :: 'a) ⟷ CHAR dvd n
proof
  assume of-nat n = (0 :: 'a)
  thus CHAR dvd n
    unfolding CHAR-def semiring-char-def by (intro Gcd-dvd) auto
next
  assume CHAR dvd n
  then obtain m where n = CHAR * m
    by auto
  thus of-nat n = (0 :: 'a)
    by simp
qed

lemma CHAR-eqI:
  assumes of-nat c = (0 :: 'a)
  assumes \( \forall x. \) of-nat x = (0 :: 'a) ⟷ c dvd x
  shows CHAR = c
  using assms by (intro dvd-antisym) (auto simp: of-nat-eq-0-iff-char-dvd)

lemma CHAR-eq0-iff: CHAR = 0 ⟷ (\forall n>0. of-nat n ≠ (0::'a))
  by (auto simp: of-nat-eq-0-iff-iff-char-dvd)

lemma CHAR-pos-ifff: CHAR > 0 ⟷ (\exists n>0. of-nat n = (0::'a))
  using CHAR-eq0-iff neq0_conv by blast

lemma CHAR-eq-posI:
  assumes c > 0 of-nat c = (0 :: 'a) \( \forall x > 0 \) ⟷ x < c ⟷ of-nat x ≠ (0 :: 'a)
  shows CHAR = c
proof (rule antisym)
  from assms have CHAR > 0
    by (auto simp: CHAR-pos-ifff)
  from assms(3)[OF this] show CHAR ≥ c
    by force
next
  have CHAR dvd c
    using assms by (auto simp: of-nat-eq-0-iff-char-dvd)
  thus CHAR ≤ c
    using c > 0 by (intro dvd-imp-le) auto
qed

end

end

lemma (in semiring-char-0) CHAR-eq-0 [simp]: semiring-char (Pure.type :: 'a itself) = 0
  by (simp add: CHAR-eq0-iff)
syntax -type-char :: type => nat ((1CHAR/(1('.'))))

translations CHAR('t) => CONST semiring-char (CONST Pure.type :: 't itself)

print-translation :
let
  fun char-type-tr' ctxt [Const (const-syntax (Pure.type), Type (-, [T]))] =
    Syntax.const syntax-const (-type-char) $ Syntax-Phases.term-of-typ ctxt T
  in ((const-syntax (semiring-char), char-type-tr') end

lemma CHAR-not-1 [simp]: CHAR('a :: {semiring-1, zero-neq-one}) ≠ Suc 0
  by (metis One-nat-def of-nat-1 of-nat-CHAR zero-neq-one)

lemma (in idom) CHAR-not-1' [simp]: CHAR('a) ≠ Suc 0
  using local.of-nat-CHAR by fastforce

lemma (in ring-1) minus-CHAR-2:
  assumes CHAR('a) = 2
  shows    (x :: 'a) = x
proof -
  have x + x = 2 * x
    by (simp add: mult-2)
  also have 2 = (0 :: 'a)
    using assms local.of-nat-CHAR by auto
  finally show ?thesis
    by (simp add: add-eq-0-iff2)
qed

lemma (in ring-1) minus-CHAR-2:
  assumes CHAR('a) = 2
  shows    (x - y :: 'a) = x + y
proof -
  have x - y = x + (-y)
    by simp
  also have -y = y
    by (rule minus-CHAR-2) fact
  finally show ?thesis
qed

lemma (in semiring-1-cancel) of-nat-eq-iff-char-dvd:
  assumes m < n
  shows  of-nat m = (of-nat n :: 'a) <-> CHAR('a) dvd (n - m)
proof
  assume *: of-nat m = (of-nat n :: 'a)
THEORY "GCD"

have of-nat n = (of-nat m + of-nat (n - m) :: 'a)
  using assms by (metis le-add-diff-inverse local.of-nat-add nless-le)

hence of-nat (n - m) = (0 :: 'a)
  by (simp add: *)

thus CHAR('a) dvd (n - m)
  by (simp add: of-nat-eq-0-iff-char-dvd)

next

assume CHAR('a) dvd (n - m)

hence of-nat (n - m) = (0 :: 'a)
  by (simp add: of-nat-eq-0-iff-char-dvd)

hence of-nat m = (of-nat m + of-nat (n - m) :: 'a)
  by simp

also have ... = of-nat n
  using assms by (metis le-add-diff-inverse local.of-nat-add nless-le)

finally show of-nat m = (of-nat n :: 'a).

qed

lemma (in ring-1) of-int-eq-0-iff-char-dvd:
(of-int n = (0 :: 'a)) = (int CHAR('a) dvd n)

proof (cases n ≥ 0)

  case True
  hence (of-int n = (0 :: 'a)) ←→ (of-nat (nat n)) = (0 :: 'a)
    by auto

  also have ... ←→ CHAR('a) dvd nat n
    by (subst of-nat-eq-0-iff-char-dvd) auto

  also have ... ←→ int CHAR('a) dvd n
    using True by presburger

  finally show ?thesis.

next

  case False
  hence (of-int n = (0 :: 'a)) ←→ -(of-nat (nat (-n))) = (0 :: 'a)
    by auto

  also have ... ←→ CHAR('a) dvd nat (-n)
    by (auto simp: of-nat-eq-0-iff-char-dvd)

  also have ... ←→ int CHAR('a) dvd n
    using False dvd-nat-abs-iff[of CHAR('a) n] by simp

  finally show ?thesis.

qed

lemma (in semiring-1-cancel) finite-imp-CHAR-pos:
assumes finite (UNIV :: 'a set)
shows CHAR('a) > 0

proof –

  have ∃n∈UNIV. infinite {m ∈ UNIV. of-nat m = (of-nat n :: 'a)}
    proof (rule pigeonhole-infinite)
      show infinite (UNIV :: nat set)
        by simp
      show finite (range (of-nat :: nat ⇒ 'a))
        by (rule finite-subset[OF - assms]) auto
    qed

  thus show ... by auto

qed
qed
then obtain $n :: \text{nat}$ where
\[
\infinite \{ m \in \text{UNIV}. \text{of-nat} \ m = (\text{of-nat} \ n :: 'a) \}\]
by blast
hence $n \subseteq \{ n \}$
by (intro notI) (use finite-subset in blast)
then obtain $m$ where $m \neq n$ of-nat $m = (\text{of-nat} \ n :: 'a)$
by blast
thus ?thesis
proof (induction $m \ n$ rule: linorder-wlog)
\[\begin{align*}
\text{case (le $m \ n$)} \quad
\text{hence} \quad \text{CHAR}('a) \ dvd (n - m) \\
\hspace{1em} \text{using} \quad \text{of-nat-eq-iff-char-dvd}\[\text{of m n}] \hspace{0.5em} \text{by auto} \\
\text{thus} \quad ?\text{thesis} \\
\hspace{1em} \text{using le} \hspace{0.5em} \text{by (intro Nat.gr0I) auto}
\end{align*}\]
qed (simp-all add: eq-commute)
qed

end

88 Nitpick: Yet Another Counterexample Generator for Isabelle/HOL

theory Nitpick
imports Record GCD
keywords
\text{nitpick :: diag and}
\text{nitpick-params :: thy-decl}
begin

datatype (plugins only: extraction) (dead 'a, dead 'b) fun-box = FunBox 'a \Rightarrow 'b
datatype (plugins only: extraction) (dead 'a, dead 'b) pair-box = PairBox 'a 'b
datatype (plugins only: extraction) (dead 'a) word = Word 'a set

typedecl bisim-iterator
typedecl unsigned-bit
typedecl signed-bit

consts
\text{unknown :: 'a}
\text{is-unknown :: 'a \Rightarrow bool}
\text{bisim :: bisim-iterator \Rightarrow 'a \Rightarrow 'a \Rightarrow bool}
\text{bisim-iterator-max :: bisim-iterator}
\text{Quot :: 'a \Rightarrow 'b}
\text{safe-The :: ('a \Rightarrow bool) \Rightarrow 'a}

Alternative definitions.
lemma \text{Ex1-unfold[nitpick-unfold]}: \text{Ex1 \ P \equiv \exists x. \{ x. \ P \ x \} = \{ x \}}
apply (rule eq-reflection)
apply (simp add: Ex1-def set-eq-iff)
apply (rule iffI)
apply (erule exE)
apply (erule conjE)
apply (rule-tac x = x in exI)
apply (rule allI)
apply (rename-tac y)
apply (erule-tac x = y in allE)
by auto

lemma rtrancl-unfold [nitpick-unfold]: r∗ ≡ (r +) =
by (simp only: rtrancl-trancl-refl)

lemma rtranclp-unfold [nitpick-unfold]: rtranclp r a b ≡ (a = b ∨ tranclp r a b)
by (rule eq-reflection) (auto dest: rtranclpD)

lemma tranclp-unfold [nitpick-unfold]: tranclp r a b ≡ (a, b) ∈ trancl { (x, y). r x y }
by (simp add: trancl-def)

lemma [nitpick-simp]: of-nat n = (if n = 0 then 0 else 1 + of-nat (n − 1))
by (cases n) auto

definition prod :: 'a set ⇒ 'b set ⇒ ('a × 'b) set where
prod A B = {(a, b). a ∈ A ∧ b ∈ B}

definition refl' :: ('a × 'a) set ⇒ bool where
refl' r ≡ ∀ x. (x, x) ∈ r

definition wf' :: ('a × 'a) set ⇒ bool where
wf' r ≡ acyclic r ∧ (finite r ∨ unknown)

definition card' :: 'a set ⇒ nat where
card' A ≡ if finite A then length (SOME xs. set xs = A ∧ distinct xs) else 0

definition sum' :: ('a ⇒ 'b::comm-monoid-add) ⇒ 'a set ⇒ 'b where
sum' f A ≡ if finite A then sum-list (map f (SOME xs. set xs = A ∧ distinct xs))
else 0

inductive fold-graph' :: ('a ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b ⇒ bool where
fold-graph' f z {} z | [x ∈ A; fold-graph' f z (A − {x}) y] ⇒ fold-graph' f z A (f x y)

The following lemmas are not strictly necessary but they help the specialize optimization.

lemma The-psimp[nitpick-psimp]: P = (=) x ⇒ The P = x
by auto
lemma Eps-psimp[nitpick-psimp]:
\[ P \ x; \ \neg \ P \ y; \ Eps \ P = y \] \implies Eps \ P = x
apply (cases \ P \ (Eps \ P))
apply auto
apply (erule contrapos-np)
by (rule someI)

lemma case-unit-unfold[nitpick-unfold]:
case-unit \ x \ u \equiv x
apply (subgoal-tac \ u = ())
apply (simp only: unit_case)
by simp

declare unit_case[nitpick-simp del]

lemma case-nat-unfold[nitpick-unfold]:
case-nat \ x \ f \ n \equiv if \ n = 0 \ then \ x \ else \ f \ (n - 1)
apply (rule eq-reflection)
by (cases \ n) auto

declare nat_case[nitpick-simp del]

lemma size-list-simp[nitpick-simp]:
size-list \ f \ xs = (if \ xs = [] \ then \ 0 \ else \ Suc \ (f \ (hd \ xs) + size-list \ f \ (tl \ xs)))
size \ xs = (if \ xs = [] \ then \ 0 \ else \ Suc \ (size \ (tl \ xs)))
by (cases \ xs) auto

Auxiliary definitions used to provide an alternative representation for rat and real.

fun nat-gcd :: \ nat \Rightarrow \ nat \Rightarrow \ nat \ where
nat-gcd \ x \ y = (if \ y = 0 \ then \ x \ else \ nat-gcd \ y \ (x \ mod \ y))

declare nat-gcd.simps [simp del]

definition nat-lcm :: \ nat \Rightarrow \ nat \Rightarrow \ nat \ where
nat-lcm \ x \ y = \ x * \ y \ div \ (nat-gcd \ x \ y)

lemma gcd-eq-nitpick-gcd [nitpick-unfold]:
gcd \ x \ y = Nitpick.nat-gcd \ x \ y
by (induct \ x \ y \ rule: nat-gcd.induct)
(simp add: gcd-nat.simps Nitpick.nat-gcd.simps)

lemma lcm-eq-nitpick-lcm [nitpick-unfold]:
lcm \ x \ y = Nitpick.nat-lcm \ x \ y
by (simp only: lcm-nat-def Nitpick.nat-lcm-def gcd-eq-nitpick-gcd)

definition Frac :: \ int \times \ int \Rightarrow \ bool \ where
Frac \equiv \lambda(a, b). \ b > 0 \land \ coprime \ a \ b
consts
Abs-Frac :: int × int ⇒ 'a
Rep-Frac :: 'a ⇒ int × int

definition zero-frac :: 'a where
zero-frac ≡ Abs-Frac (0, 1)
definition one-frac :: 'a where
one-frac ≡ Abs-Frac (1, 1)
definition num :: 'a ⇒ int where
num ≡ fst ◦ Rep-Frac
definition denom :: 'a ⇒ int where
denom ≡ snd ◦ Rep-Frac

function norm-frac :: int ⇒ int ⇒ int × int where
norm-frac a b =
(if b < 0 then norm-frac (− a) (− b)
else if a = 0 ∨ b = 0 then (0, 1)
else let c = gcd a b in (a div c, b div c))
by pat-completeness auto
termination by (relation measure (λ(−, b). if b < 0 then 1 else 0)) auto

declare norm-frac.simps[simp del]
definition frac :: int ⇒ int ⇒ 'a where
frac a b ≡ Abs-Frac (norm-frac a b)
definition plus-frac :: 'a ⇒ 'a ⇒ 'a where
[ nitpick-simp]: plus-frac q r = (let d = lcm (denom q) (denom r) in
frac (num q * (d div denom q) + num r * (d div denom r)) d)
definition times-frac :: 'a ⇒ 'a ⇒ 'a where
[ nitpick-simp]: times-frac q r = frac (num q * num r) (denom q * denom r)
definition uminus-frac :: 'a ⇒ 'a where
uminus-frac q ≡ Abs-Frac (− num q, denom q)
definition number-of-frac :: int ⇒ 'a where
number-of-frac n ≡ Abs-Frac (n, 1)
definition inverse-frac :: 'a ⇒ 'a where
inverse-frac q ≡ frac (denom q) (num q)
definition less-frac :: 'a ⇒ 'a ⇒ bool where
[ nitpick-simp]: less-frac q r ←→ num (plus-frac q (uminus-frac r)) < 0
definition less-eq-frac :: 'a ⇒ 'a ⇒ bool where
THEORY "Nitpick"

[nitpick-simp]: less-eq-frac q r \iff num (plus-frac q (uminus-frac r)) \leq 0

**definition** of-frac :: 'a \Rightarrow 'b::{inverse,ring-1} where
of-frac q \equiv of-int (num q) / of-int (denom q)

**axiomatization** wf-wfrec :: ('a \times 'a) set \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b

**definition** wf-wfrec' :: ('a \times 'a) set \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b where
[nitpick-simp]: wf-wfrec' R F x = F (cut (wf-wfrec R F) R) x

**definition** wfrec' :: ('a \times 'a) set \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b where
wfrec' R F x \equiv if wf R then wf-wfrec' R F x else THE y. wf-rec R (\lambda f. F (cut f R x)) x y

ML-file ⟨Tools/Nitpick/kodkod.ML⟩
ML-file ⟨Tools/Nitpick/kodkod-sat.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-util.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-hol.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-mono.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-preproc.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-scope.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-peephole.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-rep.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-nat.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-kodkod.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-model.ML⟩
ML-file ⟨Tools/Nitpick/nitpick.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-commands.ML⟩
ML-file ⟨Tools/Nitpick/nitpick-tests.ML⟩

setup ⟨
Nitpick-HOL.register-ersatz-global
[(const-name (card), const-name (card')), (const-name (sum), const-name (sum')), (const-name (fold-graph), const-name (fold-graph')), (const-abbrev (wf), const-name (wf')), (const-name (wf-wfrec), const-name (wf-wfrec')), (const-name (wfrec), const-name (wfrec'))]
⟩

hide-const (open) unknown is-unknown bisim bisim-iterator-max Quot safe-The FunBox PairBox Word prod refl' wf' card' sum' fold-graph' nat-gcd nat-lcm Frac Abs-Frac Rep-Frac zero-frac one-frac num denom norm-frac frac plus-frac times-frac uminus-frac number-of-frac inverse-frac less-frac less-eq-frac of-frac wf-wfrec wf-wfrec wfrec'

hide-type (open) bisim-iterator fun-box pair-box unsigned-bit signed-bit word
hide-fact (open) ExI-unfold rtrancl-unfold rtranclp-unfold tranclp-unfold prod-def refl'-def wf'-def
card'-def sum'-def The-psimp Eps-psimp case-unit-unfold case-nat-unfold
size-list-simp nat-lcm-def frac-def zero-frac-def one-frac-def
num-def denom-def frac-def plus-frac-def times-frac-def uminus-frac-def
number-of-frac-def inverse-frac-def less-frac-def less-eq-frac-def of-frac-def wf-wfrec'-def
wfrec'-def
derf

end

theory Nunchaku
imports Nitpick
keywords
  nunchaku :: diag and
  nunchaku-params :: thy-decl
begin
consts unreachable :: 'a

definition The-unsafe :: ('a ⇒ bool) ⇒ 'a where
  The-unsafe = The

definition rmember :: 'a set ⇒ 'a ⇒ bool where
  rmember A x ⇔ x ∈ A

ML-file ⟨Tools/Nunchaku/nunchaku-util.ML⟩
ML-file ⟨Tools/Nunchaku/nunchaku-collect.ML⟩
ML-file ⟨Tools/Nunchaku/nunchaku-problem.ML⟩
ML-file ⟨Tools/Nunchaku/nunchaku-translate.ML⟩
ML-file ⟨Tools/Nunchaku/nunchaku-model.ML⟩
ML-file ⟨Tools/Nunchaku/nunchaku-reconstruct.ML⟩
ML-file ⟨Tools/Nunchaku/nunchaku-display.ML⟩
ML-file ⟨Tools/Nunchaku/nunchaku-tool.ML⟩
ML-file ⟨Tools/Nunchaku/nunchaku.ML⟩
ML-file ⟨Tools/Nunchaku/nunchaku-commands.ML⟩

hide-const (open) unreachable The-unsafe rmember
end

89 Greatest Fixpoint (Codatatype) Operation on Bounded Natural Functors

theory BNF-Greatest-Fixpoint
imports BNF-Fixpoint-Base String
keywords
codatatype :: thy-defn and
primcorecursive :: thy-goal-defn and primcorec :: thy-defn

begin

alias proj = Equiv-Relations.proj

lemma one-pointE: \[ \forall x. s = x \Longrightarrow P \] \Longrightarrow P by simp

lemma obj-sumE: \[ \forall x. s = Inl x \longrightarrow P; \forall x. s = Inr x \longrightarrow P \] \Longrightarrow P by (cases s) auto

lemma not-TrueE: \[ \neg True = \Longrightarrow P \] by (erule notE, rule TrueI)

lemma neq-eq-eq-contradict: \[ t \neq u; s = t; s = u \] \Longrightarrow P by fast

lemma converse-Times: \((A \times B)^{-1} = B \times A\) by fast

lemma equiv-proj:
  assumes e: equiv A R and m: z \in R
  shows (proj R \circ fst) z = (proj R \circ snd) z
proof -
  from m have z: (fst z, snd z) \in R by auto
  with e have \[ \forall x. (fst z, x) \in R \Longrightarrow (snd z, x) \in R \wedge (snd z, x) \in R \Longrightarrow (fst z, x) \in R \] unfolding equiv-def sym-def trans-def by blast+
  then show ?thesis unfolding proj-def[abs-def] by auto
qed

definition image2 where image2 A f g = \{(f a, g a) \mid a. a \in A\}

lemma Id-on-Gr: Id-on A = Gr A id
  unfolding Id-on-def Gr-def by auto

lemma image2-eqv: \[ b = f x; c = g x; x \in A \] \Longrightarrow (b, c) \in image2 A f g
  unfolding image2-def by auto

lemma IdD: \((a, b) \in Id \Longrightarrow a = b\)
  by auto

lemma image2-Gr: image2 A f g = (Gr A f)^{-1} O (Gr A g)
  unfolding image2-def Gr-def by auto

lemma GrD1: \((x, fx) \in Gr A f \Longrightarrow x \in A\)
  unfolding Gr-def by simp
lemma GrD2: \((x, fx) \in Gr A f \implies f x = fx\)
  unfolding Gr-def by simp

lemma Gr-incl: \(Gr A f \subseteq A \times B \iff f \upharpoonright A \subseteq B\)
  unfolding Gr-def by auto

lemma subset-Collect-iff: \(B \subseteq A \implies (\{x \in A. P x\}) = (\forall x \in B. P x)\)
  unfolding by blast

lemma subset-CollectI: \(B \subseteq A \implies (\forall x \in B. Q x \implies P x) \implies (\{x \in B. Q x\} \subseteq \{x \in A. P x\})\)
  by blast

lemma in-rel-Collect-case-prod-eq: \(in-rel (Collect (case-prod X)) = X\)
  unfolding fun-eq-iff by auto

lemma Collect-case-prod-in-rel-leI: \(X \subseteq Y \implies (X \subseteq Collect (case-prod (in-rel Y)))\)
  by auto

lemma Collect-case-prod-in-rel-leE: \(X \subseteq Collect (case-prod (in-rel Y)) \implies (X \subseteq Y \implies R) \implies R\)
  by force

lemma conversep-in-rel: \((in-rel R)^{-1} = in-rel (R^{-1})\)
  unfolding fun-eq-iff by auto

lemma relcompp-in-rel: \(in-rel R O O in-rel S = in-rel (R O S)\)
  unfolding fun-eq-iff by auto

lemma in-rel-Gr: \(in-rel (Gr A f) = Grp A f\)
  unfolding Gr-def Grp-def fun-eq-iff by auto

definition relImage where
\[relImage R f \equiv \{(f a1, f a2) \mid a1 a2. (a1,a2) \in R\}\]

definition relInvImage where
\[relInvImage A R f \equiv \{(a1, a2) \mid a1 a2. a1 \in A \land a2 \in A \land (f a1, f a2) \in R\}\]

lemma relImage-Gr: \([R \subseteq A \times A] \implies relImage R f = (Gr A f)^{-1} O R O Gr A f\)
  unfolding relImage-def Gr-def relcomp-def by auto

lemma relInvImage-Gr: \([R \subseteq B \times B] \implies relInvImage A R f = Gr A f O R O (Gr A f)^{-1}\)
  unfolding Gr-def relcomp-def image-def relInvImage-def by auto

lemma relImage-mono:
\[ R_1 \subseteq R_2 \implies \text{relImage} \ R_1 \ f \subseteq \text{relImage} \ R_2 \ f \]

*unfolding* \text{relImage-def} \ by \ \text{auto}

**Lemma** \text{relInvImage-mono}:
\[ R_1 \subseteq R_2 \implies \text{relInvImage} \ A \ R_1 \ f \subseteq \text{relInvImage} \ A \ R_2 \ f \]
*unfolding* \text{relInvImage-def} \ by \ \text{auto}

**Lemma** \text{relInvImage-Id-on}:
\[(\forall a_1 \ a_2. \ f \ a_1 = f \ a_2 \iff a_1 = a_2) \implies \text{relInvImage} \ A \ (\text{Id-on} \ B) \ f \subseteq \text{Id} \]
*unfolding* \text{relInvImage-def} \text{Id-on-def} \ by \ \text{auto}

**Lemma** \text{relInvImage-UNIV-relImage}:
\[ R \subseteq \text{relInvImage} \ \text{UNIV} \ (\text{relImage} \ R \ f) \ f \]
*unfolding* \text{relInvImage-def} \text{relImage-def} \ by \ \text{auto}

**Lemma** \text{relImage-proj}:
\begin{itemize}
\item \text{assumes} \equiv \ A \ R
\item \text{shows} \text{relImage} \ R \ (\text{proj} \ R) \subseteq \text{Id-on} \ (A \ / \ R)
\item *unfolding* \text{relImage-def} \text{Id-on-def}
\item *using* \text{proj-iff} [OF \ \text{assms}] \text{equiv-class-eq-iff} [OF \ \text{assms}]
\item *by* (\text{auto simp: proj-preserves})
\end{itemize}

**Lemma** \text{relImage-relInvImage}:
\begin{itemize}
\item \text{assumes} \ R \subseteq f \ A \times f \ A
\item \text{shows} \text{relImage} \ (\text{relInvImage} \ A \ R \ f) \ f = R
\item *using* \text{assms} \ *unfolding* \text{relImage-def} \text{relInvImage-def} \ by \ \text{fast}
\end{itemize}

**Lemma** \text{subst-Pair}:
\[ P \ x \ y \implies a = (x, y) \implies P (\text{fst} \ a) \ (\text{snd} \ a) \]
*by* \text{simp}

**Lemma** \text{fst-diag-id}:
\[(\text{fst} \circ (\lambda x. \ (x, x))) \ z = \text{id} \ z \ by \ text{simp}
\]

**Lemma** \text{snd-diag-id}:
\[(\text{snd} \circ (\lambda x. \ (x, x))) \ z = \text{id} \ z \ by \ text{simp}
\]

**Lemma** \text{fst-diag-fst}:
\[ (\text{fst} \circ (\lambda x. \ (x, x))) \circ \text{fst} = \text{fst} \ by \ \text{auto}
\]

**Lemma** \text{snd-diag-fst}:
\[ (\text{snd} \circ (\lambda x. \ (x, x))) \circ \text{fst} = \text{fst} \ by \ \text{auto}
\]

**Lemma** \text{fst-diag-snd}:
\[ (\text{fst} \circ (\lambda x. \ (x, x))) \circ \text{snd} = \text{snd} \ by \ \text{auto}
\]

**Lemma** \text{snd-diag-snd}:
\[ (\text{snd} \circ (\lambda x. \ (x, x))) \circ \text{snd} = \text{snd} \ by \ \text{auto}
\]

**Definition** \text{Succ} \ where \ \text{Succ} \ Kl \ kl = \{ k . \ kl @ [k] \in Kl \}

**Definition** \text{Shift} \ where \ \text{Shift} \ Kl \ k = \{ kl . k \# kl \in Kl \}

**Definition** \text{shift} \ where \ \text{shift} \ lab \ k = (\lambda kl. \ lab \ (k \# kl))

**Lemma** \text{empty-Shift}:
\[ [[]] \in Kl; \ k \in \text{Succ} \ Kl \ [[]] \implies [[]] \in \text{Shift} \ Kl \ k\]
*unfolding* \text{Shift-def} \text{Succ-def} \ by \ \text{simp}

**Lemma** \text{SuccD}:
\[ k \in \text{Succ} \ Kl \ kl \implies kl @ [k] \in Kl\]
*unfolding* \text{Succ-def} \ by \ \text{simp}

**Lemmas** \text{SuccE} = \text{SuccD[elim-format]}
lemma \textit{SuccI}: \( kl @ [k] \in Kl \implies k \in \text{Succ } Kl \)

unfolding \textit{Succ-def} by simp

lemma \textit{ShiftD}: \( kl \in \text{Shift } Kl \) \( k \implies k \# kl \in Kl \)

unfolding \textit{Shift-def} by simp

lemma \textit{Succ-Shift}: \( \text{Succ } (\text{Shift } Kl \ k) \) \( kl = \text{Succ } Kl \) \( k \# kl \)

unfolding \textit{Succ-def} \textit{Shift-def} by auto

lemma \textit{length-Cons}: \( \text{length } (x \# xs) = \text{Suc } (\text{length } xs) \)

by simp

lemma \textit{length-append-singleton}: \( \text{length } (xs @ [x]) = \text{Suc } (\text{length } xs) \)

by simp

\begin{align*}
\text{definition } & \text{toCard-pred } A \ r \ f \equiv \text{inj-on } f \ A \land f \ ' A \subseteq \text{Field } r \land \text{Card-order } r \\
\text{definition } & \text{toCard } A \ r \equiv \text{SOME } f . \text{toCard-pred } A \ r \ f \\
\text{lemma } & \text{ex-toCard-pred: }[[A] \leq o \ r ; \text{Card-order } r] \implies \exists f . \text{toCard-pred } A \ r \ f \\
& \text{unfolding } \text{toCard-pred-def} \\
& \text{using } \text{card-of-ordLeq} [\text{of } A \text{ Field } r] \\
& \text{ordLeq-ordIso-trans} [\text{OF - card-of-unique[of Field } r \ r], \text{of } |A|] \\
& \text{by blast} \\
\text{lemma } & \text{toCard-pred-toCard: }[[A] \leq o \ r ; \text{Card-order } r] \implies \text{toCard-pred } A \ r \ (\text{toCard } A \ r) \\
& \text{unfolding } \text{toCard-def } \text{using } \text{someI-ex} [\text{OF - ex-toCard-pred}] \\
\text{lemma } & \text{toCard-inj: }[[A] \leq o \ r ; \text{Card-order } r ; x \in A ; y \in A] \implies \text{toCard } A \ r \ x = \text{toCard } A \ r \ y \iff x = y \\
& \text{using } \text{toCard-pred-toCard } \text{unfolding } \text{inj-on-def } \text{toCard-pred-def } \text{by blast} \\
\text{definition } & \text{fromCard } A \ r \ k \equiv \text{SOME } b . \ b \in A \land \text{toCard } A \ r \ b = k \\
\text{lemma } & \text{fromCard-toCard: }[[A] \leq o \ r ; \text{Card-order } r ; b \in A] \implies \text{fromCard } A \ r \ (\text{toCard } A \ r \ b) = b \\
& \text{unfolding } \text{fromCard-def } \text{by } \text{(rule some-equality)} \ (\text{auto simp add: toCard-inj}) \\
\text{lemma } & \text{Inl-Field-csum: } a \in \text{Field } r \implies \text{Inl } a \in \text{Field } (r + c s) \\
& \text{unfolding } \text{Field-card-of csum-def } \text{by } \text{auto} \\
\text{lemma } & \text{Inr-Field-csum: } a \in \text{Field } s \implies \text{Inr } a \in \text{Field } (r + c s) \\
& \text{unfolding } \text{Field-card-of csum-def } \text{by } \text{auto} \\
\text{lemma } & \text{rec-nat-0-imp: } f = \text{rec-nat } f1 \ (\lambda n \text{ rec. } f2 n \text{ rec}) \implies f \ 0 = f1 \\
& \text{by } \text{auto}
lemma rec-nat-Suc-imp: \( f = \text{rec-nat} f_1 \ (\lambda n \ \text{rec}. \ f_2 \ n \ \text{rec}) \implies f \ (\text{Suc} \ n) = f_2 \ n \ (f \ n) \)
by auto

lemma rec-list-Nil-imp: \( f = \text{rec-list} f_1 \ (\lambda x \ xs \ \text{rec}. \ f_2 \ x \ xs \ \text{rec}) \implies f \ [] = f_1 \)
by auto

lemma rec-list-Cons-imp: \( f = \text{rec-list} f_1 \ (\lambda x \ xs \ \text{rec}. \ f_2 \ x \ xs \ \text{rec}) \implies f \ (x \# xs) = f_2 \ x \ xs \ (f \ xs) \)
by auto

lemma not-arg-cong-Inr: \( x \neq y \implies \text{Inr} \ x \neq \text{Inr} \ y \)
by simp

definition image2p where
\( \text{image2p} \ f \ g \ R = (\lambda x \ y. \ \exists x' \ y'. \ R \ x' \ y' \land f \ x' = x \land g \ y' = y) \)

lemma unfolding image2pI: \( R \ x \ y \implies \text{image2p} \ f \ g \ R \ (f \ x) \ (g \ y) \)
unfolding image2p-def by blast

lemma unfolding image2pE: \( \forall \ x \ y. \ f \ x \ y \implies \forall \ x \ y. \ f \ x \ y \implies g \ y \)
unfolding image2p-def by blast

lemma rel-fun-iff-geq-image2p: \( \text{rel-fun} \ R \ S \ f \ g \ = \ (\text{image2p} \ f \ g \ R \leq S) \)
unfolding rel-fun-def image2p-def by auto

lemma rel-fun-image2p: \( \text{rel-fun} \ R \ (\text{image2p} \ f \ g \ R) \ f \ g \)
unfolding rel-fun-def image2p-def by auto

89.1 Equivalence relations, quotients, and Hilbert’s choice

lemma equiv-Eps-in:
\( [\equiv A \ r; \ X \in A//r] \implies \text{Eps} \ (\lambda x. \ x \in X) \in X \)
apply (rule someI2-ex)
using in-quotient-imp-non-empty by blast

lemma equiv-Eps-preserves:
assumes ECH: \( \equiv A \ r \) and \( X: X \in A//r \)
shows \( \text{Eps} \ (\lambda x. \ x \in X) \in A \)
apply (rule in_mono_rule_format)
using assms apply (rule in-quotient-imp-subset)
by (rule equiv-Eps-in) (rule assms)+

lemma proj-Eps:
assumes equiv A r and X \in A//r
shows proj r \ (Eps (\lambda x. \ x \in X)) = X
unfolding proj-def
proof auto
  fix x assume x: x ∈ X
  thus (Eps (λx. x ∈ X), x) ∈ r using assms equiv-Eps-in in-quotient-imp-in-rel by fast
next
  fix x assume (Eps (λx. x ∈ X), x) ∈ r
  thus x ∈ X using in-quotient-imp-closed[OF assms equiv-Eps-in[OF assms]] by fast
qed

definition univ where univ f X == f (Eps (λx. x ∈ X))

lemma univ-commute:
  assumes ECH: equiv A r and RES: f respects r and x: x ∈ A
  shows (univ f) (proj r x) = f x
proof (unfold univ-def)
  have prj: proj r x ∈ A//r using proj-preserves by fast
  hence Eps (λy. y ∈ proj r x) ∈ A using ECH equiv-Eps-preserves by fast
  moreover have proj r (Eps (λy. y ∈ proj r x)) = proj r x using ECH proj-Eps by fast
  ultimately have (x, Eps (λy. y ∈ proj r x)) ∈ r using ECH proj-iff by fast
  hence f (Eps (λy. y ∈ proj r x)) = f x using RES unfolding congruent-def by fastforce
thus f (Eps (λy. y ∈ proj r x)) = f x using RES unfolding congruent-def by fastforce
qed

lemma univ-preserves:
  assumes ECH: equiv A r and RES: f respects r and PRES: ∀x ∈ A. f x ∈ B
  shows ∀X ∈ A//r. univ f X ∈ B
proof
  fix X assume X: X ∈ A//r
  then obtain x where x: x ∈ A and X: X = proj r x using ECH proj-image[of r A] by blast
  hence univ f X = f x using ECH RES univ-commute by fastforce
  thus univ f X ∈ B using x PRES by simp
qed

lemma card-suc-ordLess-imp-ordLeq:
  assumes ORD: Card-order r Card-order r' card-order r'
  and LESS: r < card-suc r'
  shows r ≤ o card-suc r'
proof
  have Card-order (card-suc r') by (rule Card-order-card-suc[OF ORD(3)])
  then have cardSuc r ≤ o card-suc r' using cardSuc-least ORD LESS by blast
  then have cardSuc r ≤ o cardSuc r' using cardSuc-ordIso-card-suc ordIso-symmetric ordLeq-ordIso-trans ORD(3) by blast
  then show ?thesis using cardSuc-mono-ordLeq ORD by blast
qed

lemma natLeq-ordLess-cinfinite: [ Cinfinite r; card-order r ] ⇒ natLeq < o card-suc
r using natLeq-ordLeq-cinfinite card-suc-greater ordLeq-ordLess-trans by blast

corollary natLeq-ordLess-cinfinite': [Cinfinite r'; card-order r'; r ≡ card-suc r']
⇒ natLeq < o r
using natLeq-ordLess-cinfinite by blast

ML-file ⟨Tools/BNF/bnf-gfp-util.ML⟩
ML-file ⟨Tools/BNF/bnf-gfp-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-gfp.ML⟩
ML-file ⟨Tools/BNF/bnf-gfp-rec-sugar-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-gfp-rec-sugar.ML⟩

end

90 Filters on predicates

theory Filter
imports Set-Interval Lifting-Set
begin

90.1 Filters

This definition also allows non-proper filters.

locale is-filter =
  fixes F :: ('a ⇒ bool) ⇒ bool
  assumes True: F (λx. True)
  assumes conj: F (λx. P x) =⇒ F (λx. Q x) =⇒ F (λx. P x ∧ Q x)
  assumes mono: ∀ x. P x =⇒ Q x =⇒ F (λx. P x) =⇒ F (λx. Q x)

typedef 'a filter = {F :: ('a ⇒ bool) ⇒ bool. is-filter F}
proof
  show (λx. True) ∈ filter by (auto intro: is-filter.intro)
qed

lemma is-filter-Rep-filter: is-filter (Rep-filter F)
  using Rep-filter [of F] by simp

lemma Abs-filter-inverse':
  assumes is-filter F shows Rep-filter (Abs-filter F) = F
  using assms by (simp add: Abs-filter-inverse)

90.1.1 Eventually

definition eventually :: ('a ⇒ bool) ⇒ 'a filter ⇒ bool
  where eventually P F ≡ Rep-filter F P

syntax
-eventually :: pattrn => 'a filter => bool => bool  ((3∀ F - in -/ -) [0, 0, 10] 10)
translations
∀ F x in F. P == CONST eventually (λx. P) F

lemma eventually-Abs-filter:
  assumes is-filter F
  shows eventually P (Abs-filter F) = F P
  unfolding eventually-def using assms by (simp add: Abs-filter-inverse)

lemma filter-eq-iff:
  shows F = F' ←→ (∀ P. eventually P F = eventually P F')
  unfolding Rep-filter-inject [symmetric] fun-eq-iff eventually-def ..

lemma eventually-True [simp]: eventually (λx. True) F
  unfolding eventually-def by (rule is-filter.True [OF is-filter-Rep-filter])

lemma always-eventually: ∀ x. P x → eventually P F
proof –
  assume ∀ x. P x hence P = (λx. True) by (simp add: ext)
  thus eventually P F by simp
qed

lemma eventuallyI: (∀ x. P x) → eventually P F
  by (auto intro: always-eventually)

lemma filter-eqI: (∀ P. eventually P F <-> eventually P G) → F = G
  by (auto simp: filter-eq-iff)

lemma eventually-mono:
  [eventually P F; ∀ x. P x → Q x] → eventually Q F
  unfolding eventually-def
  by (blast intro: is-filter.mono [OF is-filter-Rep-filter])

lemma eventually-conj:
  assumes P: eventually (λx. P x) F
  assumes Q: eventually (λx. Q x) F
  shows eventually (λx. P x ∧ Q x) F
  using assms unfolding eventually-def
  by (rule is-filter.conj [OF is-filter-Rep-filter])

lemma eventually-mp:
  assumes eventually (λx. P x → Q x) F
  assumes eventually (λx. P x) F
  shows eventually (λx. Q x) F
proof –
  have eventually (λx. (P x → Q x) ∧ P x) F
    using assms by (rule eventually-conj)
  then show ?thesis
lemma eventually-rev-mp:
  assumes eventually (λx. P x) F
  assumes eventually (λx. P x → Q x) F
  shows eventually (λx. Q x) F
using assms(2) assms(1) by (rule eventually-mp)

lemma eventually-conj-iff:
  eventually (λx. P x ∧ Q x) F ←→ eventually P F ∧ eventually Q F
by (auto intro: eventually-conj elim: eventually-rev-mp)

lemma eventually-elim2:
  assumes eventually (λi. P i) F
  assumes eventually (λi. Q i) F
  assumes ∀i. P i ⇒ Q i ⇒ R i
  shows eventually (λi. R i) F
using assms by (auto elim!: eventually-rev-mp)

lemma eventually-cong:
  assumes eventually P F and ∀x. P x =⇒ Q x =⇒ R x
  shows eventually Q F =⇒ eventually R F
using eventually-ball-finite[of UNIV P] assms by simp

lemma eventually-ex: (∀fx in F. ∃y. P x y) ←→ (∃Y. ∀fx in F. P x (Y x))
proof
  assume ∀fx in F. ∃y. P x y
  then have ∀fx in F. P x (SOME y. P x y)
    by (auto intro: someI-ex eventually-mono)
  then show ∃Y. ∀fx in F. P x (Y x)
    by auto
qed (auto intro: eventually-mono)
lemma not-eventually-impI: \( \text{eventually } P F \implies \neg \text{eventually } Q F \implies \neg \text{eventually } (\lambda x. P x \rightarrow Q x) F \)
   by (auto intro: eventually-mp)

lemma not-eventuallyD: \( \neg \text{eventually } P F \implies \exists x. \neg P x \)
   by (metis always-eventually)

lemma eventually-subst:
   assumes \( \text{eventually } (\lambda n. P n = Q n) F \)
   shows \( \text{eventually } P F = \text{eventually } Q F \) (is \(?L = ?R\))
   proof –
      from assms have \( \text{eventually } (\lambda x. P x \rightarrow Q x) F \)
      and \( \text{eventually } (\lambda x. Q x \rightarrow P x) F \)
      by (auto elim: eventually-mp)
      then show \( ?\text{thesis} \) by (auto elim: eventually-elim2)
   qed

90.2 Frequently as dual to eventually

definition frequently :: ('a ⇒ bool) ⇒ 'a filter ⇒ bool
   where frequently P F ≡ \( \neg \text{eventually } (\lambda x. \neg P x) F \)

syntax
   - frequently :: pattrn ⇒ 'a filter ⇒ bool ⇒ bool \(((\exists F \cdot \in -/ -) \ 0, 0, 10)\ 10)\
translations
   \( \exists F x \cdot P \) in F. P \( \equiv \) CONST frequently (\( \lambda x. P \) F)

lemma not-frequently-False [simp]: \( \neg (\exists F x \cdot \in F. \ False) \)
   by (simp add: frequently-def)

lemma frequently-ex: \( \exists F x \cdot \in F. P x \implies \exists x. P x \)
   by (auto simp: frequently-def dest: not-eventuallyD)

lemma frequentlyE: assumes frequently P F obtains x where P x
   using frequently-ex[OF assms] by auto

lemma frequently-mp:
   assumes ev: \( \forall F x \in F. P x \rightarrow Q x \) and P: \( \exists F x \in F. P x \) shows \( \exists F x \in F. Q x \)
   proof –
      from ev have \( \text{eventually } (\lambda x. \neg Q x \rightarrow \neg P x) F \)
      by (rule eventually-rev-mp) (auto intro: always-eventually)
      from eventually-mp[OF this] P show \( ?\text{thesis} \)
      by (auto simp: frequently-def)
   qed

lemma frequently-rev-mp:
   assumes \( \exists F x \cdot \in F. P x \)
assumes $\forall F \ x \ in \ F. \ P \ x \ \rightarrow \ Q \ x$
shows $\exists F \ x \ in \ F. \ Q \ x$
using `assms(2)` `assms(1)` by (rule `frequently-mp`)

lemma `frequently-mono`: $(\forall x. \ P \ x \ \rightarrow \ Q \ x) \ \Rightarrow \ \forall F \ P \ F \ \Rightarrow \ \forall F \ Q \ F$
using `frequently-mp[of P Q]` by (simp add: `always-eventually`)

lemma `frequently-elim1`: $\exists F \ x \ in \ F. \ P \ x \ \Rightarrow \ \forall i. \ P \ i \ \Rightarrow \ Q \ i \ \Rightarrow \ \exists F \ x \ in \ F. \ Q \ x$
by (metis `frequently-mono`)

lemma `frequently-disj-iff`: $(\exists F \ x \ in \ F. \ P \ x \ \lor \ Q \ x) \ \iff \ (\exists F \ x \ in \ F. \ P \ x) \ \lor \ (\exists F \ x \ in \ F. \ Q \ x)$
by (simp add: `frequently-def` `eventually-conj-iff`)

lemma `frequently-disj`: $\exists F \ x \ in \ F. \ P \ x \ \Rightarrow \ \exists F \ x \ in \ F. \ Q \ x \ \Rightarrow \ \exists F \ x \ in \ F. \ P \ x \ \lor \ Q \ x$
by (simp add: `frequently-disj-iff`)

lemma `frequently-bex-finite-distrib`: assumes `finite A` shows $(\exists F \ x \ in \ F. \ \exists y \in A. \ P \ x \ y) \ \iff \ (\exists y \in A. \ \exists F \ x \ in \ F. \ P \ x \ y)$
using `assms` by induction (auto simp: `frequently-def`)

lemma `frequently-bex-finite`: `finite A` $\Rightarrow \ \exists F \ x \ in \ F. \ \exists y \in A. \ P \ x \ y \ \Rightarrow \ \exists y \in A. \ \exists F \ x \ in \ F. \ P \ x \ y$
by (simp add: `frequently-bex-finite-distrib`)

lemma `frequently-all`: $(\exists F \ x \ in \ F. \ \forall y. \ P \ x \ y) \ \iff \ (\forall Y \exists F \ x \ in \ F. \ P \ x \ (Y \ x))$
using `eventually-ex[of \ \lambda y \ x. \ \neg \ P \ x \ y \ F]` by (simp add: `frequently-def`)

lemma
shows `not-eventually`: $\neg \ \forall x. \ P \ x \ \Rightarrow \ \exists F \ x \ in \ F. \ \neg \ P \ x$
and `not-frequently`: $\neg \ \forall F \ x \ in \ F. \ P \ x \ \Rightarrow \ \forall F \ x \ in \ F. \ \neg \ P \ x$
by (auto simp: `frequently-def`)

lemma `frequently-imp-iff`: $(\exists F \ x \ in \ F. \ P \ x \ \Rightarrow \ Q \ x) \ \iff \ (eventually P F \ \Rightarrow \ frequently Q F)$
unfolding `imp-conv-disj` `frequently-disj-iff` `not-eventually[ symmetric]` ..

lemma `frequently-eventually-conj`: assumes `\exists F \ x \ in \ F. \ P \ x` assumes `\forall F \ x \ in \ F. \ Q \ x`
shows `\exists F \ x \ in \ F. \ Q \ x \ \& \ P \ x` using `assms` `eventually-elim2` by (force simp add: `frequently-def`)

lemma `frequently-cong`: assumes `ev` `eventually P F` and `QR`: $(\ \exists x. \ P \ x \ \Rightarrow \ Q \ x \ \iff \ R \ x$
shows frequently Q F ✭→ frequently R F

unfolding frequently-def

using QR by (auto intro!: eventually-cong [OF ev])

lemma frequently-eventually-frequently:
frequently P F ✭→ eventually Q F ✭→ frequently (λx. P x ∧ Q x) F

using frequently-cong [of Q F P λ x. P x ∧ Q x] by meson

lemma eventually-frequently-const-simps:
(∃ F x in F. P x ∧ C) ✭→ (∃ F x in F. P x) ∧ C
(∃ F x in F. C ∧ P x) ✭→ C ∧ (∃ F x in F. P x)
(∀ F x in F. P x ∨ C) ✭→ (∀ F x in F. P x) ∨ C
(∀ F x in F. C ∨ P x) ✭→ C ∨ (∀ F x in F. P x)
(∀ F x in F. P x → C) ✭→ (∃ F x in F. P x → C)
(∀ F x in F. C → P x) ✭→ (C → (∃ F x in F. P x))

by (cases C; simp add: not-frequently)+

lemmas eventually-frequently-simps =
eventually-frequently-const-simps
not-eventually
eventually-conj-iff
eventually-ball-finite-distrib
eventually-ex
not-frequently
frequently-disj-iff
frequently-bex-finite-distrib
frequently-all
frequently-imp-iff

ML

fun eventually-elim-tac facts =
  CONTEXT-SUBGOAL (fn (goal, i) => fn (ctxt, st) =>
  let
    val mp-facts = facts RL @{thms eventually-rev-mp}
    val rule =
      @{thm eventuallyI}
      |> fold (fn mp-fact => fn th => th RS mp-fact) mp-facts
      |> funpow (length facts) (fn th => @{thm impI} RS th)
    val cases-prop =
      Thm.prop_of (Rule-Cases.internalize-params (rule RS Goal.init (Thm.cterm_of ctxt goal))
    val cases = Rule-Cases.make-common ctxt cases-prop [(elim, []), []])
  in CONTEXT-CASES cases (resolve-tac ctxt [rule] i) (ctxt, st) end

method-setup eventually-elim =
  Scan.succeed (fn - => CONTEXT-METHOD (fn facts => eventually-elim-tac facts 1))

elimination of eventually quantifiers
90.2.1 Finer-than relation

$F \leq F'$ means that filter $F$ is finer than filter $F'$.

**instantiation** filter :: (type) complete-lattice

**definition** le-filter-def:

$F \leq F' \iff (\forall P. \text{eventually } P F' \rightarrow \text{eventually } P F)$

**definition**

$(F :: \text{'a filter}) < F' \iff F \leq F' \land \neg F' \leq F$

**definition**

$\text{top} = \text{Abs-filter} (\lambda P. \forall x. P x)$

**definition**

$\text{bot} = \text{Abs-filter} (\lambda P. \text{True})$

**definition**

$\text{sup } F F' = \text{Abs-filter} (\lambda P. \text{eventually } P F \land \text{eventually } P F')$

**definition**

$\text{inf } F F' = \text{Abs-filter}

(\lambda P. \exists Q R. \text{eventually } Q F \land \text{eventually } R F' \land (\forall x. Q x \land R x \rightarrow P x))$

**definition**

$\text{Sup } S = \text{Abs-filter} (\lambda P. \forall F \in S. \text{eventually } P F)$

**definition**

$\text{Inf } S = \text{Sup } \{F :: \text{'a filter}. \forall F' \in S. F \leq F'\}$

**lemma** eventually-top [simp]: eventually $P$ $\text{top} \iff (\forall x. P x)$

**unfolding** top-filter-def

**by** (rule eventually-Abs-filter, rule is-filter.intro, auto)

**lemma** eventually-bot [simp]: eventually $P$ $\text{bot}$

**unfolding** bot-filter-def

**by** (subst eventually-Abs-filter, rule is-filter.intro, auto)

**lemma** eventually-sup:

eventually $P$ $(\text{sup } F F') \iff \text{eventually } P F \land \text{eventually } P F'$

**unfolding** sup-filter-def

**by** (rule eventually-Abs-filter, rule is-filter.intro)

(auto elim!: eventually-rev-mp)

**lemma** eventually-inf:

eventually $P$ $(\text{inf } F F') \iff 

(\exists Q R. \text{eventually } Q F \land \text{eventually } R F' \land (\forall x. Q x \land R x \rightarrow P x))$

**unfolding** inf-filter-def
apply (rule eventually-Abs-filter [OF is-filter.intro])
apply (blast intro: eventually-True)
apply (force elim!: eventually-conj)+
done

lemma eventually-Sup:
eventually P (Sup S) \iff (\forall F \in S. \ eventually P F)
unfolding Sup-filter-def
apply (rule eventually-Abs-filter [OF is-filter.intro])
apply (auto intro: eventually-conj elim!: eventually-rev-mp)
done

instance proof
fix F F' F'' :: 'a filter and S :: 'a filter set
\{ show F < F' \iff F \leq F' \land \neg F' \leq F 
by (rule less-filter-def) \}
\{ show F \leq F' 
unfolding le-filter-def by simp \}
\{ assume F \leq F' and F' \leq F'' thus F \leq F'' 
unfolding le-filter-def by simp \}
\{ assume F \leq F' and F' \leq F thus F = F' 
unfolding le-filter-def filter-eq-iff by fast \}
\{ show \inf F F' \leq F and \inf F F' \leq F' 
unfolding le-filter-def eventually-inf by (auto intro: eventually-True) \}
\{ assume F \leq F' and F \leq F'' thus F \leq \inf F' F'' 
unfolding le-filter-def eventually-inf 
by (auto intro: eventually-mono [OF eventually-conj]) \}
\{ show F \leq \sup F F' and F' \leq \sup F F'' 
unfolding le-filter-def eventually-sup by simp-all \}
\{ assume F \leq F'' and F' \leq F'' thus \sup F F' \leq F'' 
unfolding le-filter-def eventually-sup by simp \}
\{ assume F'' \in S thus \inf S \leq F'' 
unfolding le-filter-def Inf-filter-def eventually-Sup Ball-def by simp \}
\{ assume \\land F', F' \in S \Rightarrow F \leq F' thus F \leq \inf S 
unfolding le-filter-def Inf-filter-def eventually-Sup Ball-def by simp \}
\{ assume F \in S thus F \leq \sup S 
unfolding le-filter-def eventually-Sup by simp \}
\{ assume \\land F. F \in S \Rightarrow F \leq F' thus \sup S \leq F' 
unfolding le-filter-def eventually-Sup by simp \}
\{ show \inf \{\} = (top::'a filter) 
by (auto simp: top-filter-def Inf-filter-def Sup-filter-def) 
(metis (full-types) top-filter-def always-eventually eventually-top) \}
\{ show \sup \{\} = (bot::'a filter) 
by (auto simp: bot-filter-def Sup-filter-def) \}
qued

end

instance filter :: (type) distrib-lattice
proof

fix F G H :: 'a filter

show sup F (inf G H) = inf (sup F G) (sup F H)

proof (rule order.antisym)

show inf (sup F G) (sup F H) ≤ sup F (inf G H)

unfolding le-filter-def eventually-sup

proof safe

fix P assume 1: eventually P F and 2: eventually P (inf G H)

from 2 obtain Q R

where QR: eventually Q G eventually R H ⋀ x. Q x ⋱ R x ⋱ P x

by (auto simp: eventually-inf)

define Q' where Q' = (λx. Q x ∨ P x)
define R' where R' = (λx. R x ∨ P x)

from 1 have eventually Q' F

by (elim eventually-mono) (auto simp: Q' -def)

moreover from 1 have eventually R' F

by (elim eventually-mono) (auto simp: R' -def)

moreover from QR(1) have eventually Q' G

by (elim eventually-mono) (auto simp: Q' -def)

moreover from QR(2) have eventually R' H

by (elim eventually-mono) (auto simp: R' -def)

moreover from QR have P x if Q' x R' x for x

using that by (auto simp: Q' -def R' -def)

ultimately show eventually P (inf (sup F G) (sup F H))

by (auto simp: eventually-inf eventually-sup)

qed

qed (auto intro: inf.coboundedI1 inf.coboundedI2)

qed

lemma filter-leD:
F ≤ F' ⋱ eventually P F' ⋱ eventually P F

unfolding le-filter-def by simp

lemma filter-leI:
(⋀P. eventually P F' ⋱ eventually P F) ⋱ F ≤ F'

unfolding le-filter-def by simp

lemma eventually-False:
eventually (λx. False) F ⋭ F = bot

unfolding filter-eq-iff by (auto elim: eventually-rev-mp)

lemma eventually-frequently: F ≠ bot ⋱ eventually P F ⋱ frequently P F

using eventually-conf[of P F λx. ¬ P x]

by (auto simp add: frequently-def eventually-False)

lemma eventually-frequentlyE:
assumes eventually P F

assumes eventually (λx. ¬ P x ∨ Q x) F F≠bot
shows frequently Q F
proof –
  have eventually Q F
    using eventually-conj[OF assms(1,2),simplified] by (auto elim: eventually-mono)
then show ?thesis using eventually-frequently[OF F≠bot] by auto
qed

lemma eventually-const-iff: eventually (λx. P) F ←→ P ∨ F = bot
  by (cases P) (auto simp: eventually-False)

lemma eventually-const[simp]: F ≠ bot ⇒ eventually (λx. P) F ←→ P
  by (simp add: eventually-const-iff)

lemma frequently-const-iff: frequently (λx. P) F ←→ P ∧ F ≠ bot
  by (simp add: frequently-def eventually-const-iff)

lemma frequently-const[simp]: F ≠ bot ⇒ frequently (λx. P) F ←→ P
  by (simp add: frequently-const-iff)

lemma eventually-happens: eventually P net ⇒ net = bot ∨ (∃x. P x)
  by (metis frequentlyE eventually-frequently)

lemma eventually-happens':
  assumes F ≠ bot eventually P F
  shows ∃x. P x
  using assms eventually-frequently frequentlyE by blast

abbreviation (input) trivial-limit :: 'a filter ⇒ bool
  where trivial-limit F ≡ F = bot

lemma trivial-limit-def: trivial-limit F ←→ eventually (λx. False) F
  by (rule eventually-False [symmetric])

lemma False-imp-not-eventually: (∀x. ¬ P x) ⇒ ¬ trivial-limit net ⇒ ¬ eventually (λx. P x) net
  by (simp add: eventually-False)

lemma trivial-limit-eventually: trivial-limit net ⇒ eventually P net
  by simp

lemma trivial-limit-eq: trivial-limit net ←→ (∀P. eventually P net)
  by (simp add: filter-eq-iff)

lemma eventually-Inf: eventually P (Inf B) ←→ (∃X⊆B. finite X ∧ eventually P (Inf X))
proof –
  let ?F = λP. ∃X⊆B. finite X ∧ eventually P (Inf X)
  have eventually-F: eventually P (Abs-filter ?F) ←→ ?F P for P
proof (rule eventually-Abs-filter is-filter.intro)+
  show \(?F (\lambda x. \text{True})\)
  by (rule exI[of - {}]) (simp add: le-fun-def)
next
  fix P Q
  assume \(?F P \ ?F Q\)
  then obtain X Y where
    \(X \subseteq B\) finite \(X\) eventually \(P\) \((\prod X)\)
    \(Y \subseteq B\) finite \(Y\) eventually \(Q\) \((\prod Y)\) by blast
  then show \(?F (\lambda x. \text{P x \& Q x})\)
  by (intro exI[of - \(X \cup Y\)]) (auto simp: Inf-union-distrib eventually-inf)
next
  fix P Q
  assume \(?F P\)
  then obtain X where
    \(X \subseteq B\) finite \(X\) eventually \(P\) \((d X)\)
  by blast
  moreover assume \(\forall x. P x \rightarrow Q x\)
  ultimately show \(?F Q\)
  by (intro exI[of - X]) (auto elim: eventually-mono)
qed

have Inf B = Abs-filter \(?F\)
proof (intro antisym Inf-greatest)
  show Inf B \(\leq\) Abs-filter \(?F\)
  by (auto simp: le-filter-def eventually-F dest: Inf-superset-mono)
next
  fix F assume F \(\in\) B then show Abs-filter \(?F \leq F\)
  by (auto simp add: le-filter-def eventually-F intro: exI[of - \{F\}])
qed

lemma eventually-INF: eventually \(P\) \((\prod b \in B. \text{F b})\) \(\iff\) \((\exists X \subseteq B. \text{finite X \& eventually P} \ (\prod X))\)
unfolding eventually-Inf[of P \(\cdot\)\(B\)]
by (metis finite-imageI image-mono finite-subset-image)

lemma Inf-filter-not-bot:
  fixes B :: 'a filter set
  shows \((\forall X. X \subseteq B \rightarrow \text{finite X} \rightarrow \text{Inf X} \neq \text{bot}) \rightarrow \text{Inf B} \neq \text{bot}\)
unfolding trivial-limit-def eventually-Inf[of - B]
  bot-bool-def [symmetric] bot-fun-def [symmetric] bot-unique by simp

lemma INF-filter-not-bot:
  fixes F :: 'i \Rightarrow 'a filter
  shows \((\forall X. X \subseteq B \rightarrow \text{finite X} \rightarrow \prod b \in X. \text{F b} \neq \text{bot}) \rightarrow \prod b \in B. \text{F b} \neq \text{bot}\)
unfolding trivial-limit-def eventually-INF[of - B]
THEORY "Filter"

bot-bool-def [symmetric] bot-fun-def [symmetric] bot-unique by simp

lemma eventually-inf-base:
assumes B \neq \{\} and base: \( \forall F. G. F \in B \implies G \in B \implies \exists x \in B. x \leq \inf F \ G \)
shows eventually P (Inf B) \iff (\exists b \in B. eventually P b)
proof (subst eventually-inf, safe)
fix X assume finite X X \subseteq B
then have \( \exists b \in B. \forall x \in X. b \leq x \)
proof induct
  case empty then show \(?case
    using \( \{\} \) by auto
next
  case \( \{\text{insert} \ \ x \ X\} \)
  then obtain b where \( b \in B \ \land x \in X \implies b \leq x \)
  by auto
  with \( \text{insert} \ \ x \ X \subseteq B \) base[of b x] show \(?case
    by (auto intro: order-trans)
qed
then show eventually P (Inf X) \implies Bex B (eventually P)
by (intro bexI[of - b]) (auto simp: le-finite)
qed (auto intro!: exI[of - \{x\} for x])

lemma eventually-INF-base:
B \neq \{\} \implies (\forall a. a \in B \implies b \in B \implies \exists x \in B. F \ x \leq \inf (F a) (F b)) \implies
eventually P (\inf b \in B. F b) \iff (\exists b \in B. eventually P (F b))
by (subst eventually-INF-base) auto

lemma eventually-INF1: \( i \in I \implies \text{eventually} \ P (F i) \implies \text{eventually} \ P (\prod i \in I. \ F i) \)
using filter-leD[OF INF-lower]

lemma eventually-INF-finite:
assumes finite A
shows eventually P (\prod x \in A. F x) \iff
(\exists Q. (\forall x \in A. eventually (Q x) (F x)) \land (\forall y. (\forall x \in A. Q x y) \rightarrow P y))
using assms
proof (induction arbitrary: P rule: finite-induct)
case \( \text{insert} \ a \ A \ P \)
from insert.hyps have \[\text{simp}: \ x \neq a \ \text{if} \ x \in A \ \text{for} \ x\]
using that by auto
have eventually P (\prod x \in insert a A. F x) \iff
(\exists Q R S. eventually Q (F a) \land ((\forall x \in A. eventually (S x) (F x)) \land
(\forall y. (\forall x \in A. S x y) \rightarrow R y)) \land (\forall x. \ Q x \land R x \rightarrow P x)))
unfolding ex-simps by (simp add: eventually-inf insert.IH)
also have \ldots \iff (\exists Q. (\forall x \in \text{insert} \ a \ A. \text{eventually} (Q x) (F x)) \land
(\forall y. (\forall x \in \text{insert} \ a \ A. \ Q x y) \rightarrow P y))
proof (safe, goal-cases)
case \((1 \ Q \ R \ S)\)
thus ?case using 1 by (intro exI[of - S(a := Q)]) auto
next
case \((2 \ Q)\)
show ?case
  by (rule exI[of - Q a], rule exI[of - \(\lambda y. \forall x \in A. \ Q x \ y\)],
      rule exI[of - Q(a := (\_. True))]) (use 2 in auto)
qed
finally show ?case .
qed auto

lemma eventually-le-le:
  fixes \(\cdot a \Rightarrow \cdot b::\text{preorder}\)
  assumes eventually \((\lambda x. \ P x \leq \ Q x)\) \(\ F\)
  assumes eventually \((\lambda x. \ Q x \leq \ R x)\) \(\ F\)
  shows eventually \((\lambda x. \ P x \leq \ R x)\) \(\ F\)
using assms by eventually-elim (rule order-trans)

90.2.2 Map function for filters

definition filtermap :: \((\cdot a \Rightarrow \cdot b)::\text{preorder}\)
where filtermap \(\ f \ F\) = Abs-filter \((\lambda P. \text{eventually} \ (\lambda x. \ P \ (f \ x)) \ F)\)

lemma eventually-filtermap:
  eventually \(\ P\) (filtermap \(\ f \ F\)) = eventually \((\lambda x. \ P \ (f \ x)) \ F\)
unfolding filtermap-def
apply (rule eventually-Abs-filter [OF is-filter.intro])
apply (auto elim!: eventually-rev-mp)
done

lemma eventually-comp-filtermap:
  eventually \((P \circ f)\) \(\ F\) \iff eventually \(P\) (filtermap \(f \ F\))
unfolding comp-def using eventually-filtermap by auto

lemma filtermap-compose: filtermap \((f \circ g)\) \(\ F\) = filtermap \(f\) (filtermap \(g \ F\))
unfolding filter-eq-iff by (simp add: eventually-filtermap)

lemma filtermap-ident: filtermap \((\lambda x. \ x)\) \(\ F\)
by (simp add: filter-eq-iff eventually-filtermap)

lemma filtermap-compose:
  filtermap \(f\) (filtermap \(g \ F\)) = filtermap \((\lambda x. \ f \ (g \ x))\) \(\ F\)
by (simp add: filter-eq-iff eventually-filtermap)

lemma filtermap-mono: \(F \leq F'\) \implies filtermap \(f \ F\) \(\leq filtermap \(f \ F'\)
unfolding le-filter-def eventually-filtermap by simp

lemma filtermap-bot [simp]: filtermap \(f \ bot\) = bot
by (simp add: filter-eq-iff eventually-filtermap)
lemma filtermap-bot-iff: filtermap f F = bot ↔ F = bot
  by (simp add: trivial-limit-def eventually-filtermap)

lemma filtermap-sup: filtermap f (sup F1 F2) = sup (filtermap f F1) (filtermap f F2)
  by (simp add: filter-eq-iff eventually-filtermap eventually-sup)

lemma filtermap-SUP: filtermap f (⨆ b ∈ B. F b) = (⨆ b ∈ B. filtermap f (F b))
  by (simp add: filter-eq-iff eventually-Sup eventually-filtermap)

lemma filtermap-inf: filtermap f (inf F1 F2) ≤ inf (filtermap f F1) (filtermap f F2)
  by (intro inf-greatest filtermap-mono inf-sup-ord)

lemma filtermap-INF: filtermap f (⨇ b ∈ B. F b) ≤ (⨇ b ∈ B. filtermap f (F b))
  by (rule INF-greatest, rule filtermap-mono, erule INF-lower)

lemma frequently-filtermap:
  frequently P (filtermap f F) = frequently (λx. P (f x)) F
  by (simp add: frequently-def eventually-filtermap)

90.2.3 Contravariant map function for filters

definition filtercomap :: ('a ⇒ 'b) ⇒ 'b filter ⇒ 'a filter
  where filtercomap f F = Abs-filter (λP. ∃ Q. eventually Q F ∧ (∀x. Q (f x) −→ P x))

lemma eventually-filtercomap:
  eventually P (filtercomap f F) ←→ (∃ Q. eventually Q F ∧ (∀x. Q (f x) −→ P x))
  unfolding filtercomap-def
proof (intro eventually-Abs-filter, unfold-locales, goal-cases)
  case 1
  show ?case by (auto intro!: exI[of - λ. True])
next
  case (2 P Q)
  then obtain P' Q' where P'Q':
    eventually P' F ∨ x. P' (f x) −→ P x
    eventually Q' F ∨ x. Q' (f x) −→ Q x
    by (elim exE conjE)
  show ?case
  by (rule exI[of - λx. P' x ∧ Q' x]) (use P'Q' in auto intro!: eventually-conj)
next
  case (3 P Q)
  thus ?case by blast
qed

lemma filtercomap-ident: filtercomap (λx. x) F = F
  by (auto simp: filter-eq-iff eventually-filtercomap elim!: eventually-mono)
lemma filtercomap-filtercomap: filtercomap f (filtercomap g F) = filtercomap (λx. g (f x)) F
    unfolding filter-eq-iff by (auto simp: eventually-filtercomap)

lemma filtercomap-mono: F ≤ F′ → filtercomap f F ≤ filtercomap f F′
    by (auto simp: eventually-filtercomap le-filter-def)

lemma filtercomap-bot [simp]: filtercomap f bot = bot
    by (auto simp: filter-eq-iff eventually-filtercomap)

lemma filtercomap-top [simp]: filtercomap f top = top
    by (auto simp: filter-eq-iff eventually-filtercomap)

lemma filtercomap-inf: filtercomap f (inf F1 F2) = inf (filtercomap f F1) (filtercomap f F2)
    unfolding filter-eq-iff
proof safe
fix P
  assume eventually P (filtercomap f (F1 ⊓ F2))
  then obtain Q R S where *
    eventually Q F1 eventually R F2 \x. Q x \implies R x \implies S x \implies P x
  unfolding eventually-filtercomap eventually-inf by blast
from * have eventually (λx. Q (f x)) (filtercomap f F1)
    eventually (λx. R (f x)) (filtercomap f F2)
    by (auto simp: eventually-filtercomap)
with * show eventually P (filtercomap f F1 ⊓ filtercomap f F2)
  unfolding eventually-inf by blast
next
fix P
  assume eventually P (inf (filtercomap f F1) (filtercomap f F2))
  then obtain Q R S where *:
    eventually Q F1 eventually R F2 \x. Q (f x) \implies R (f x) \implies S (f x) \implies P x
  unfolding eventually-filtercomap eventually-inf by blast
from * have eventually (λx. Q x ∧ R x) (F1 ⊓ F2) by (auto simp: eventually-inf)
with * show eventually P (filtercomap f (F1 ⊓ F2))
  by (auto simp: eventually-filtercomap)
qed

lemma filtercomap-sup: filtercomap f (sup F1 F2) ≥ sup (filtercomap f F1) (filtercomap f F2)
    by (intro sup-least filtercomap-mono inf-sup-ord)

lemma filtercomap-INF: filtercomap f (⨅ b∈B. F b) = (⨅ b∈B. filtercomap f (F b))
proof —
have \( \ast \): \( \text{filtercomap} \ f \ (\bigwedge_{b \in B} F \ b) = (\bigwedge_{b \in B} \text{filtercomap} \ f \ (F \ b)) \) if \( \text{finite} \ B \) for \( B \)

using that by induction (simp-all add: filtercomap-inf)

show \(?\text{thesis}\) unfolding \( \text{filter-eq-iff}\)

proof

fix \( P \)

have \( \text{eventually} \ P \ (\bigwedge_{b \in B} \text{filtercomap} \ f \ (F \ b)) \iff \ (\exists X. (X \subseteq B \land \text{finite} X) \land \text{eventually} \ P \ (\bigwedge_{b \in X} \text{filtercomap} \ f \ (F \ b))) \)

by (subst eventually-\(\text{INF}\)) blast

also have \( \ldots \iff (\exists X. (X \subseteq B \land \text{finite} X) \land \text{eventually} \ P \ (\text{filtercomap} \ f \ (\bigwedge_{b \in X} F \ b))) \)

by (rule \( \text{ex-cong} \)) (simp add: \( \ast \))

also have \( \ldots \iff \text{eventually} \ P \ (\text{filtercomap} \ f \ (\bigwedge_{b \in B} F \ b)) \)

unfolding eventually-filtercomap by (subst eventually-\(\text{INF}\)) blast

finally show \( \text{eventually} \ P \ (\text{filtercomap} \ f \ (\bigwedge_{b \in B} F \ b)) = \text{eventually} \ P \ (\bigwedge_{b \in B} \text{filtercomap} \ f \ (F \ b)) \).

qed

lemma \( \text{filtercomap-SUP} \):

\( \text{filtercomap} \ f \ (\bigcup_{b \in B} F \ b) \geq (\bigcup_{b \in B} \text{filtercomap} \ f \ (F \ b)) \)

by (intro SUP-least filtercomap-mono SUP-upper)

lemma \( \text{filtermap-le-iff-le-filtercomap} \):

\( \text{filtermap} \ f \ F \leq G \iff F \leq \text{filtercomap} \ f \ G \)

unfolding le-filter-def eventually-filtercomap eventually-filtercomap

using eventually-monotone by auto

lemma \( \text{filtercomap-neq-bot} \):

assumes \( \forall P. \text{eventually} \ P \ F \Rightarrow \exists x. P \ (f \ x) \)

shows \( \text{filtercomap} \ f \ F \neq \bot \)

using \( \text{assms} \) by (auto simp: trivial-limit-def eventually-filtercomap)

lemma \( \text{filtercomap-neq-bot-surj} \):

assumes \( F \neq \bot \) and \( \text{surj} \ f \)

shows \( \text{filtercomap} \ f \ F \neq \bot \)

proof (rule filtercomap-neq-bot)

fix \( P \) assume \( \ast \): \( \text{eventually} \ P \ F \)

show \( \exists x. P \ (f \ x) \)

proof (rule \( \text{constr} \))

assume \( \ast \ast \): \( \neg (\exists x. P \ (f \ x)) \)

from \( \ast \) have \( \text{eventually} \ (\lambda -. \text{False}) \ F \)

proof \( \text{eventually-elim} \)

case \( (\text{elim} \ x) \)

from \( \text{surj} \ f \) obtain \( y \) where \( x = f \ y \) by auto

with \( \text{elim and} \ \ast \ast \) show \( \text{False} \) by auto

qed

with \( \text{assms} \) show \( \text{False} \) by (simp add: trivial-limit-def)

qed

qed
lemma eventually-filtercomapI [intro]:
  assumes eventually P F
  shows eventually (λx. P (f x)) (filtercomap f F)
  using assms by (auto simp: eventually-filtercomap)

lemma filtermap-filtercomap: filtermap f (filtercomap f F) ≤ F
  by (auto simp: le-filter_def eventually-filtermap eventually-filtercomap)

lemma filtercomap-filtermap: filtercomap f (filtermap f F) ≥ F
  unfolding le-filter-def eventually-filtermap eventually-filtercomap
  by (auto elim!: eventually-mono)

90.2.4 Standard filters

definition principal :: 'a set => 'a filter where
  principal S = Abs-filter (λP. ∀ x∈S. P x)

lemma eventually-principal: eventually P (principal S) <-> (∀ x∈S. P x)
  unfolding principal-def
  by (rule eventually-Abs-filter, rule is-filter.intro) auto

lemma eventually-inf-principal: eventually P (inf F (principal s)) <-> eventually (λx. x ∈ s --> P x) F
  unfolding eventually-inf eventually-principal
  by (auto elim: eventually-mono)

lemma principal-UNIV[simp]: principal UNIV = top
  by (auto simp: filter-eq-iff eventually-principal)

lemma principal-empty[simp]: principal {} = bot
  by (auto simp: filter-eq-iff eventually-principal)

lemma principal-eq-bot-iff: principal X = bot <-> X = {}
  by (auto simp add: filter-eq-iff eventually-principal)

lemma principal-le-iff[iff]: principal A ≤ principal B <-> A ⊆ B
  by (auto simp: le-filter_def eventually-principal)

lemma le-principal: F ≤ principal A <-> eventually (λx. x ∈ A) F
  unfolding le-filter-def eventually-principal
  by (force elim: eventually-mono)

lemma principal-inject[iff]: principal A = principal B <-> A = B
  unfolding eq-iff by simp

lemma sup-principal[simp]: sup (principal A) (principal B) = principal (A ∪ B)
  unfolding filter-eq-iff eventually-sup eventually-principal by auto

lemma inf-principal[simp]: inf (principal A) (principal B) = principal (A ∩ B)
unfolding filter-eq-iff eventually-inf eventually-principal
by (auto intro: exI[of - λx. x ∈ A] exI[of - λx. x ∈ B])

lemma SUP-principal[simp]: (∧ i∈I. principal (A i)) = principal (∪ i∈I. A i)
unfolding filter-eq-iff eventually-Sup by (auto simp: eventually-principal)

lemma INF-principal-finite: finite X =⇒ (∩ x∈X. principal (f x)) = principal (∪ x∈X. f x)
by (induct X rule: finite-induct) auto

lemma filtermap-principal[simp]: filtermap f (principal A) = principal (f ' A)
unfolding filter-eq-iff eventually-filtermap eventually-principal by simp

lemma filtercomap-principal[simp]: filtercomap f (principal A) = principal (f −' A)
unfolding filter-eq-iff eventually-filtercomap eventually-principal by fast

90.2.5 Order filters

definition at-top :: ('a::order) filter
where at-top = (∩ k. principal {k ..})

lemma at-top-sub: at-top = (∩ k∈{c::'a::linorder..}. principal {k ..})
by (auto intro: INF-eq max cobounded1 max cobounded2 simp: at-top-def)

lemma eventually-at-top-linorder: eventually P at-top ←→ (∃ N::'a::linorder. ∀ n≥N. P n)
unfolding at-top-def by (subst eventually-INF-base) (auto simp: eventually-principal intro: max cobounded1 max cobounded2)

lemma eventually-filtercomap-at-top-linorder:
  eventually P (filtercomap f at-top) ←→ (∃ N::'a::linorder. ∀ x. f x ≥ N =⇒ P x)
by (auto simp: eventually-filtercomap eventually-at-top-linorder)

lemma eventually-at-top-linorderI:
  fixes c::'a::linorder
  assumes ∃ x. c ≤ x =⇒ P x
  shows eventually P at-top
using assms by (auto simp: eventually-at-top-linorder)

lemma eventually-ge-at-top [simp]:
eventually (λx. (c::'a::linorder) ≤ x) at-top
unfolding eventually-at-top-linorder by auto

lemma eventually-at-top-dense: eventually P at-top ←→ (∃ N::'a::{no-top, linorder}. ∀ n>N. P n)
proof −
have eventually P (∩ k. principal {k <..}) ←→ (∃ N::'a. ∀ n>N. P n)

THEORY “Filter”
by (subst eventually-INF-base) (auto simp: eventually-principal intro: max.cobounded1 max.cobounded2)
also have \((\bigcap k. \text{principal} \{k::'a <..\}) = \text{at-top}\)
  unfolding at-top-def
  by (intro INF-eq) (auto intro: less-imp-le simp: Ici-subset-Ioi-iff gt-ex)
finally show \(?thesis\).
qed

lemma eventually-filtercomap-at-top-dense:
  \(\text{eventually } P (\text{filtercomap } f \text{ at-top}) \iff (\exists N::'a::\{\text{no-top, linorder}\}. \forall x. f x > N \rightarrow P x)\)
  by (auto simp: eventually-filtercomap eventually-at-top-dense)

lemma eventually-at-top-not-equal [simp]:
  \(\text{eventually } (\lambda x::'a::\{\text{no-top, linorder}\}. x \neq c) \text{ at-top}\)
  unfolding eventually-at-top-dense by auto

lemma eventually-gt-at-top [simp]:
  \(\text{eventually } (\lambda x. (c::::\{\text{no-top, linorder}\}) < x) \text{ at-top}\)
  unfolding eventually-at-top-dense by auto

lemma eventually-all-ge-at-top:
  assumes \(\text{eventually } P \text{ (at-top :: ('a :: linorder) filter)}\)
  shows \(\text{eventually } (\lambda x. \forall y \geq x. P y) \text{ at-top}\)
proof
  from assms obtain \(x\) where \(\\forall y. y \geq x \implies P y\) by (auto simp: eventually-at-top-linorder)
  hence \(\forall z \geq y. P z \text{ if } y \geq x \text{ for } y\) using that by simp
  thus \(?thesis\) by (auto simp: eventually-at-top-linorder)
qed

definition at-bot :: ('a::order) filter
  where at-bot = (\[k\in{..}. k\])

lemma at-bot-sub: at-bot = (\[k\in{..}. 'a::linorder\}). principal \{.. k\})
  by (auto intro!: INF-eq min.cobounded1 min.cobounded2 simp: at-bot-def)

lemma eventually-at-bot-linorder:
  fixes \(P :: 'a::linorder \Rightarrow \text{bool}\) shows \(\text{eventually } P \text{ at-bot} \iff (\exists N. \forall n \leq N. P n)\)
  unfolding at-bot-def
  by (subst eventually-INF-base) (auto simp: eventually-principal intro: min.cobounded1 min.cobounded2)

lemma eventually-filtercomap-at-bot-linorder:
  \(\text{eventually } P (\text{filtercomap } f \text{ at-bot}) \iff (\exists N::'a::linorder. \forall x. f x \leq N \rightarrow P x)\)
  by (auto simp: eventually-filtercomap eventually-at-bot-linorder)

lemma eventually-le-at-bot [simp]:
  \(\text{eventually } (\lambda x. x \leq (c::::\{\text{linorder}\})) \text{ at-bot}\)
```
THEORY "Filter"

unfolding eventually-at-bot-linorder by auto

lemma eventually-at-bot-dense: eventually P at-bot ⟷ (∃ N::'a::{no-bot, linorder}. ∀ n<N. P n)
proof -
  have eventually P (∩ k. principal {..< k}) ⟷ (∃ N::'a. ∀ n<N. P n)
    by (subst eventually-INF-base) (auto simp: eventually-principal intro: min.cobounded1 min.cobounded2)
  also have (∩ k. principal {..< k::'a}) = at-bot
    unfolding at-bot-def
    by (intro INF-eq) (auto intro: less-imp-le simp: Iic-subset-Iio-Iff lt-ex)
finally show ?thesis .
qed

lemma eventually-filtercomap-at-bot-dense:
  eventually P (filtercomap f at-bot) ⟷ (∃ N::'a::{no-bot, linorder}. ∀ x. f x < N ⟹ P x)
  by (auto simp: eventually-filtercomap eventually-at-bot-dense)

lemma eventually-at-bot-not-equal [simp]: eventually (λx::'a::{no-bot, linorder}. x ≠ c) at-bot
  unfolding eventually-at-bot-dense by auto

lemma eventually-gt-at-bot [simp]: eventually (λx::'a::{no-bot, dense-linorder}. x < c) at-bot
  unfolding eventually-at-bot-dense by auto

lemma trivial-limit-at-bot-linorder [simp]: ¬ trivial-limit (at-bot ::'a::linorder filter)
  unfolding trivial-limit-def
  by (metis eventually-at-bot-linorder order-refl)

lemma trivial-limit-at-top-linorder [simp]: ¬ trivial-limit (at-top ::'a::linorder filter)
  unfolding trivial-limit-def
  by (metis eventually-at-top-linorder order-refl)

90.3 Sequentially
abbreviation sequentially :: nat filter
  where sequentially ≡ at-top

lemma eventually-sequentially:
  eventually P sequentially ⟷ (∃ N. ∀ n≥N. P n)
  by (rule eventually-at-top-linorder)

lemma sequentially-bot [simp, intro]: sequentially ≠ bot
  unfolding filter-eq-Iff eventually-sequentially by auto
```
lemmas trivial-limit-sequentially = sequentially-bot

lemma eventually-False-sequentially [simp]:
~ eventually (\lambda n. False) sequentially
by (simp add: eventually-False)

lemma le-sequentially:
F ≤ sequentially ↔ (∀ N. eventually (\lambda n. N ≤ n) F)
by (simp add: at-top-def le-INF-iff le-principal)

lemma eventually-sequentiallyI [intro?]:
assumes \( \bigwedge x. c ≤ x \Rightarrow P x \)
sows eventually P sequentially
using assms by (auto simp: eventually-sequentially)

lemma eventually-sequentially-Suc [simp]: eventually (\lambda i. P (Suc i)) sequentially
↔ eventually P sequentially
unfolding eventually-sequentially by (metis Suc-le-D Suc-le-mono le-Suc-eq)

lemma eventually-sequentially-seg [simp]: eventually (\lambda n. P (n + k)) sequentially
↔ eventually P sequentially
using eventually-sequentially-Suc[of \( \lambda n. P (n + k) \) for k] by (induction k) auto

lemma filtermap-sequentually-ne-bot: filtermap f sequentially ≠ bot
by (simp add: filtermap-bot-iff)

90.4 Increasing finite subsets

definition finite-subsets-at-top where
finite-subsets-at-top A = (\Pi X ∈ {X. finite X ∧ X ⊆ A}. principal {Y. finite Y ∧ X ⊆ Y ∧ Y ⊆ A})

lemma eventually-finite-subsets-at-top:
eventually P (finite-subsets-at-top A) ↔
(∃ X. finite X ∧ X ⊆ A ∧ (∀ Y. finite Y ∧ X ⊆ Y ∧ Y ⊆ A → P Y))
unfolding finite-subsets-at-top-def
proof (subst eventually-INF-base, goal-cases)
show \{X. finite X ∧ X ⊆ A \} ≠ {} by auto
next
case (2 B C)
thus ?case by (intro bexI[of - B ∪ C]) auto
qed (simp-all add: eventually-principal)

lemma eventually-finite-subsets-at-top-weakI [intro]:
assumes \( \bigwedge X. finite X \Rightarrow X ⊆ A \Rightarrow P X \)
sows eventually P (finite-subsets-at-top A)
proof
have eventually (\lambda X. finite X ∧ X ⊆ A) (finite-subsets-at-top A)
by (auto simp: eventually-finite-subsets-at-top)
thus \( ?\text{thesis} \) by eventually-elimp (use assms in auto)

qed

\begin{lemma}
finite-subsets-at-top-neq-bot [simp]: finite-subsets-at-top \( A \neq \text{bot} \)
\begin{proof}
- have \( \neg \text{eventually} \ (\lambda x. \text{False}) \ (\text{finite-subsets-at-top} \ A) \)
  by (auto simp: eventually-finite-subsets-at-top)
  thus \( ?\text{thesis} \) by auto
\end{proof}
\end{lemma}

\begin{lemma}
filtermap-image-finite-subsets-at-top:
assumes inj-on f A
shows \( \text{filtermap} \ (\lambda x. f \ x) \ (\text{finite-subsets-at-top} \ A) = \text{finite-subsets-at-top} \ (f \ A) \)
\begin{proof}
(safe, goal-cases)
  case (1 P)
  then obtain \( X \) where \( X \) (finite X X \subseteq A \land Y. \text{finite} Y \Longrightarrow X \subseteq Y \Longrightarrow Y \subseteq A \Longrightarrow P (f \ Y) \)
    unfolding eventually-finite-subsets-at-top by force
  show \( ?\text{case} \)
    unfolding eventually-finite-subsets-at-top eventually-filtermap
    proof (rule exI[of - f \ ' X], intro conjI allI impI, goal-cases)
      case (3 Y)
      with assms and X(1,2) have \( P (f \ (f \ ' Y \cap A)) \) using X(1,2)
        by (intro X(3) finite-vimage-IntI) auto
    also have \( f \ (f \ ' Y \cap A) = Y \) using assms 3 by blast
    finally show \( ?\text{case} \).
    qed (insert assms X(1,2), auto intro!: finite-vimage-IntI)
  next
  case (2 P)
  then obtain \( X \) where \( X \) (finite X X \subseteq f \ ' A \land Y. \text{finite} Y \Longrightarrow X \subseteq Y \Longrightarrow Y \subseteq f \ ' A \Longrightarrow P Y \)
    unfolding eventually-finite-subsets-at-top by force
  show \( ?\text{case} \)
    unfolding eventually-finite-subsets-at-top eventually-filtermap
    proof (rule exI[of - f \ ' X \cap A], intro conjI allI impI, goal-cases)
      case (3 Y)
      with X(1,2) and assms show \( ?\text{case} \) by (intro X(3)) force+
    qed (insert assms X(1), auto intro!: finite-vimage-IntI)
  qed
\end{proof}
\end{lemma}

\begin{lemma}
eventually-finite-subsets-at-top-finite:
assumes finite A
shows eventually P (finite-subsets-at-top A) \iff P A
\begin{proof}
unfolding eventually-finite-subsets-at-top using assms by force
\end{proof}
\end{lemma}

\begin{lemma}
finite-subsets-at-top-finite: finite A \Longrightarrow finite-subsets-at-top A = \text{principal} \ \{A\}
by (auto simp: filter-eq-iff eventually-finite-subsets-at-top-finite eventually-principal)
90.5 The cofinite filter

definition cofinite = Abs-filter (λP. finite {x. ¬ P x})

abbreviation Inf-many :: ('a ⇒ bool) ⇒ bool (binder ∃∞ 10)
where Inf-many P ≡ frequently P cofinite

abbreviation Alm-all :: ('a ⇒ bool) ⇒ bool (binder ∀∞ 10)
where Alm-all P ≡ eventually P cofinite

notation (ASCII)
Inf-many (binder INFM 10) and
Alm-all (binder MOST 10)

lemma eventually-cofinite: eventually P cofinite ↔ finite {x. ¬ P x}
  unfolding cofinite-def
proof (rule eventually-Abs-filter, rule is-filter.intro)
  fix P Q :: 'a ⇒ bool assume finite {x. ¬ P x} finite {x. ¬ Q x}
  from finite-UnI[OF this] show finite {x. ¬ (P x ∧ Q x)}
  by (rule rev-finite-subset) auto
next
  fix P Q :: 'a ⇒ bool assume P: finite {x. ¬ P x} and *: ∀x. P x → Q x
  from * show finite {x. ¬ Q x}
  by (intro finite-subset[OF - P]) auto
qed simp

lemma frequently-cofinite: frequently P cofinite ↔ ¬ finite {x. P x}
  by (simp add: frequently-def eventually-cofinite)

lemma cofinite-bot[simp]: cofinite = (bot::'a filter) ↔ finite (UNIV :: 'a set)
  unfolding trivial-limit-def eventually-cofinite by simp

lemma cofinite-eq-sequentially: cofinite = sequentially
  unfolding filter-eq-iff eventually-sequentially eventually-cofinite
proof safe
  fix P :: nat ⇒ bool assume [simp]: finite {x. ¬ P x}
  show ∃N. ∀n≥N. P n
  proof cases
    assume {x. ¬ P x} ≠ {} then show ?thesis
    by (intro exI[of - Suc (Max {x. ¬ P x})]) (auto simp: Suc-le-eq)
  qed auto
next
  fix P :: nat ⇒ bool and N :: nat assume ∀n≥N. P n
  then have {x. ¬ P x} ⊆ {..< N}
  by (auto simp: not-le)
  then show finite {x. ¬ P x}
  by (blast intro: finite-subset)
qed
90.5.1 Product of filters

**definition prod-filter :: 'a filter ⇒ 'b filter ⇒ ('a × 'b) filter (infixr × F 80)**

where

\[
\text{prod-filter } F \ G = (\{ (P, Q) \in \{ (P, Q) \in \text{eventually } P \ F \wedge \text{eventually } Q \ G \} \wedge \text{principal } \{ (x, y). P \ x \wedge Q \ y \})
\]

**lemma eventually-prod-filter: eventually P (F × F G) ←→ (∃ Pf Pg. eventually Pf F ∧ eventually Pg G ∧ (∀ x y. Pf x → Pg y → P (x, y)))**

**proof (subst eventually-INF-base, goal-cases)**

**case 2**

moreover have eventually Pf F ⇒ eventually Qf F ⇒ eventually Pg G ⇒ eventually Qg G ⇒

∃ P Q. eventually P F ∧ eventually Q G ∧ Collect P × Collect Q ⊆ Collect Pf × Collect Pg ∩ Collect Qf × Collect Qg

for Pf Pg Qf Qg

by (intro conjI exI[of - inf Pf Qf] exI[of - inf Pg Qg])

(auto simp: inf-fun-def eventually-conj)

ultimately show ?case

by auto

qed (auto simp: eventually-principal intro: eventually-True)

**lemma eventually-prod1:**

assumes B ≠ bot

shows (∀ F (x, y) in A × F B. P x) ←→ (∀ F x in A. P x)

**proof safe**

fix R Q

assume *: (∀ F x in A. R x ∀ F x in B. Q x ∨ x y. R x → Q y → P x)

with ⟨B ≠ bot⟩ obtain y where Q y by (auto dest: eventually-happens)

with * show eventually P A

by (force elim: eventually-mono)

next

assume eventually P A

then show ∃ Pf Pg. eventually Pf A ∧ eventually Pg B ∧ (∀ x y. Pf x → Pg y → P x)

by (intro exI[of - P] exI[of - λx. True]) auto

qed

**lemma eventually-prod2:**

assumes A ≠ bot

shows (∀ F (x, y) in A × F B. P y) ←→ (∀ F y in B. P y)

**proof safe**

fix R Q

assume *: (∀ F x in A. R x ∀ F x in B. Q x ∨ x y. R x → Q y → P y)

with ⟨A ≠ bot⟩ obtain x where R x by (auto dest: eventually-happens)
with * show eventually \(PB\)
   by (force elim: eventually-mono)
next
assume eventually \(PB\)
then show \(\exists PfPg.\) eventually \(PfA \land \) eventually \(PgB \land (\forall x y. Pf x \rightarrow Pg y \rightarrow P y)\)
   by (intro exI[of - P] exI[of \(\lambda x.\) True]) auto
qed

lemma INF-filter-bot-base:
  fixes \(F::\)'a \Rightarrow '\b filter
  assumes *: \(\forall i j.\) \(i \in I \implies j \in I \implies \exists k \in I.\) \(F k \leq F i \cap F j\)
  shows \((\prod i \in I.\) \(F i) = \bot \iff (\exists i \in I.\) \(F i = \bot)\)
proof (cases \(\exists i \in I.\) \(F i = \bot\))
  case True
  then have \((\prod i \in I.\) \(F i) \leq \bot\)
  by (auto intro: INF-lower2)
  with True show ?thesis
  by (auto simp: bot-unique)
next
  case False
  moreover have \((\prod i \in I.\) \(F i) \neq \bot\)
  proof (cases \(I = \{\}\))
    case True
    then show ?thesis
    by (auto simp add: filter-eq-iff)
  next
    case False': False
    show ?thesis
    proof (rule INF-filter-not-bot)
      fix \(J\)
      assume finite \(J J \subseteq I\)
      then have \(\exists k \in I.\) \(F k \leq (\prod i \in J.\) \(F i)\)
      proof (induct \(J\))
        case empty
        then show ?case
        using \(I \neq \{\}\) by auto
      next
        case (insert i J)
        then obtain \(k\) where \(k \in I F k \leq (\prod i \in J.\) \(F i)\) by auto
        with insert *[of i k] show ?case
        by auto
      qed
      with False show \((\prod i \in J.\) \(F i) \neq \bot\)
      by (auto simp: bot-unique)
    qed
  qed
ultimately show ?thesis
  by auto
qed

**lemma** Collect-empty-eq-bot: \( \text{Collect } P = \{ \} \iff P = \bot \)

by auto

**lemma** prod-filter-eq-bot: \( A \times F B = \bot \iff A = \bot \lor B = \bot \)

unfolding trivial-limit-def

**proof**

assume \( \forall F \ x \in A \times F B. \ False \)

then obtain \( Pf Pg \) where \( Pf \): eventually \( (\lambda x. Pf x) A \) and \( Pg \): eventually \( (\lambda y. Pg y) B \)

and \( \ast : \forall x y. Pf x \longrightarrow Pg y \longrightarrow False \)

unfolding eventually-prod-filter by fast

from \( \ast \) have \( (\forall x. \neg Pf x) \lor (\forall y. \neg Pg y) \) by fast

with \( Pf Pg \) show \( (\forall F \ x \in A. \ False) \lor (\forall F \ x \in B. \ False) \) by auto

next

assume \( (\forall F \ x \in A. \ False) \lor (\forall F \ x \in B. \ False) \)

then show \( \forall F \ x \in A \times F B. \ False \)

unfolding eventually-prod-filter by (force intro: eventually-True)

qed

**lemma** prod-filter-mono: \( F \leq F' \implies G \leq G' \implies F \times F G \leq F' \times F G' \)

by (auto simp: le-filter-def eventually-prod-filter)

**lemma** prod-filter-mono-iff:

assumes nAB: \( A \neq \bot B \neq \bot \)

shows \( A \times F B \leq C \times F D \iff A \leq C \land B \leq D \)

**proof**
safe

assume \( \ast : A \times F B \leq C \times F D \)

with assms have \( A \times F B \neq \bot \)

by (auto simp: bot-unique prod-filter-eq-bot)

with \( \ast \) have \( C \times F D \neq \bot \)

by (auto simp: bot-unique)

then have nCD: \( C \neq \bot D \neq \bot \)

by (auto simp: prod-filter-eq-bot)

show \( A \leq C \)

**proof** (rule filter-leI)

fix \( P \) assume eventually \( P C \) with \( \ast \)[THEN filter-leD, of \( \lambda(x, y). \ P x \)] show eventually \( P A \)

using nAB nCD by (simp add: eventually-prod1 eventually-prod2)

qed

show \( B \leq D \)

**proof** (rule filter-leI)

fix \( P \) assume eventually \( P D \) with \( \ast \)[THEN filter-leD, of \( \lambda(x, y). \ P y \)] show eventually \( P B \)

using nAB nCD by (simp add: eventually-prod1 eventually-prod2)

qed
THEORY "Filter"

qed (intro prod-filter-mono)

lemma eventually-prod-same: eventually P (F × F F) ←→
(∃ Q. eventually Q F ∧ (∀ x y. Q x → Q y → P (x, y)))
unfolding eventually-prod-filter by (blast intro: eventually-conj)

lemma eventually-prod-sequentially:
eventually P (sequentially × F sequentially) ←→
(∃ N. ∀ m ≥ N. ∀ n ≥ N. P (n, m))
unfolding eventually-prod-same eventually-sequentially by auto

lemma principal-prod-principal: principal A × F principal B = principal (A × B)
unfolding filter-eq-iff eventually-prod-filter eventually-principal
by (fast intro: exI[of - λ x. x ∈ A] exI[of - λ x. x ∈ B])

lemma le-prod-filterI: filtermap fst F ≤ A =⇒ filtermap snd F ≤ B =⇒ F ≤ A × F B
unfolding le-filter-def eventually-filtermap eventually-prod-filter
by (force elim: eventually-elim2)

lemma filtermap-fst-prod-filter: filtermap fst (A × F B) ≤ A
unfolding le-filter-def eventually-filtermap eventually-prod-filter
by (force intro: eventually-True)

lemma filtermap-snd-prod-filter: filtermap snd (A × F B) ≤ B
unfolding le-filter-def eventually-filtermap eventually-prod-filter
by (force intro: eventually-True)

lemma prod-filter-INF:
assumes I ≠ {} and J ≠ {}
shows (⨆ i∈I. A i × F j∈J. B j) = (⨆ i∈I. ⨆ j∈J. A i × F B j)
proof (rule antisym)
from ⟨I ≠ {}⟩ obtain i where i ∈ I by auto
from ⟨J ≠ {}⟩ obtain j where j ∈ J by auto
show (⨆ i∈I. ⨆ j∈J. A i × F B j) ≤ (⨆ i∈I. ⨆ j∈J. A i × F B j)
by (fast intro: le-prod-filter1 INF-greatest INF-lower2
order-trans[OF filtermap-INF] ⊢ i ∈ I, j ∈ J,
filtermap-fst-prod-filter filtermap-snd-prod-filter)
show (⨆ i∈I. A i × F ⨆ j∈J. B j) ≤ (⨆ i∈I. ⨆ j∈J. A i × F B j)
by (intro INF-greatest prod-filter-mono INF-lower)
qed

lemma filtermap-Pair: filtermap (λx. (f x, g x)) F ≤ filtermap f F × F filtermap g F
by (rule le-prod-filterI, simp-all add: filtermap-filtermap)

lemma eventually-prodI: eventually P F =⇒ eventually Q G =⇒ eventually (λx. P (fst x) ∧ Q (snd x)) (F × F G)
unfolding eventually-prod-filter by auto

lemma prod-filter-INF1: \( I \neq \{ \} \implies (\prod i \in I. A \ i) \times_F B = (\prod i \in I. A \ i \times_F B) \)
using prod-filter-INF[of I \{ B \} A \ \lambda x. x] by simp

lemma prod-filter-INF2: \( J \neq \{ \} \implies A \times_F (\prod i \in J. B \ i) = (\prod i \in J. A \times_F B \ i) \)
using prod-filter-INF[of \{ A \} J \ \lambda x. x B] by simp

lemma prod-filtermap1: prod-filter (filtermap f F) G = filtermap (apfst f) (prod-filter F G)
unfolding filter-eq-iff eventually-filtermap eventually-prod-filter
apply safe
subgoal by auto
subgoal for \( P Q R \) by (rule exI[where \( x=\lambda y. \exists x. y = f x \land Q x \)]) (auto intro: eventually-mono)
done

lemma prod-filtermap2: prod-filter F (filtermap g G) = filtermap (apsnd g) (prod-filter F G)
unfolding filter-eq-iff eventually-filtermap eventually-prod-filter
apply safe
subgoal by auto
subgoal for \( P Q R \) by (auto intro: exI[where \( x=\lambda y. \exists x. y = g x \land R x \)])
done

lemma prod-filter-assoc:
prod-filter (prod-filter F G) H = filtermap (\( \lambda (x, y, z). ((x, y), z) \)) (prod-filter F (prod-filter G H))
apply (clarsimp simp add: filter-eq-iff eventually-filtermap eventually-prod-filter; safe)
subgoal for \( P Q R S T \) by (auto 4 4 intro: exI[where \( x=\lambda (a, b). T a \land S b \)])
subgoal for \( P Q R S T \) by (auto 4 3 intro: exI[where \( x=\lambda (a, b). Q a \land S b \)])
done

lemma prod-filter-principal-singleton: prod-filter (principal \( \{ x \} \)) F = filtermap (Pair x) F
by (fastforce simp add: filter-eq-iff eventually-prod-filter eventually-principal eventually-filtermap elim: eventually-mono intro: exI[where \( x=\lambda a. a = x \)])

lemma prod-filter-principal-singleton2: prod-filter F (principal \( \{ x \} \)) = filtermap (\( \lambda a. (a, x) \)) F
by (fastforce simp add: filter-eq-iff eventually-prod-filter eventually-principal eventually-filtermap elim: eventually-mono intro: exI[where \( x=\lambda a. a = x \)])

lemma prod-filter-commute: prod-filter F G = filtermap prod.swap (prod-filter G F)
by (auto simp add: filter-eq-iff eventually-prod-filter eventually-filtermap)
90.6 Limits

definition filterlim :: ('a ⇒ 'b) ⇒ 'b filter ⇒ 'a filter ⇒ bool where
  filterlim f F2 F1 ←→ filtermap f F1 ≤ F2

syntax
  -LIM :: pttrns ⇒ 'a ⇒ 'b ⇒ 'a ⇒ bool ((3LIM (-)/ (-)/ (-) :> (-)) [1000, 10, 0, 10] 10)

translations
  LIM x F1. f :> F2 == CONST filterlim (λx. f) F2 F1

lemma filterlim-filtercomapI: filterlim f F G =⇒ filterlim (λx. f (g x)) F (filtercomap g G)
  unfolding filterlim-def
  by (metis order-trans filtermap-filtercomap filtermap-filtermap filtermap-mono)

lemma filterlim-top [simp]: filterlim f top F
  by (simp add: filterlim-def)

lemma filterlim-iff: (LIM x F1. f x :> F2) ←→ (∀ P. eventually P F2 −→ eventually (λx. P (f x)) F1)
  unfolding filterlim-def le-filter-def eventually-filtermap ..

lemma filterlim-compose: filterlim g F3 F2 =⇒ filterlim f F2 F1 =⇒ filterlim (λx. g (f x)) F3 F1
  unfolding filterlim-def filtermap-filtermap[symmetric] by (metis filtermap-mono order-trans)

lemma filterlim-mono: filterlim f F2 F1 =⇒ F2 ≤ F2' =⇒ F1' ≤ F1 =⇒ filterlim f F2' F1'
  unfolding filterlim-def by (metis filtermap-mono order-trans)

lemma filterlim-ident: LIM x F. x :> F
  by (simp add: filterlim-def filtermap-ident)

lemma filterlim-cong: F1 = F1' =⇒ F2 = F2' =⇒ eventually (λx. f x = g x) F2 =⇒ filterlim f F1 F2 = filterlim g F1' F2'
  by (auto simp: filterlim-def le-filter-def eventually-filtermap elim: eventually-elim2)

lemma filterlim-mono-eventually:
  assumes filterlim f F G and ord: F ≤ F' G' ≤ G
  assumes eq: eventually (λx. f x = g x) G'
  shows filterlim f' F' G'
proof −
  have filterlim f F' G'
    by (simp add: filterlim-mono[OF - ord] assms)
  then show ?thesis
by (rule filterlim-cong[OF refl refl eq, THEN iffD1])

qed

lemma filtermap-mono-strong: inj f \implies filtermap f F \leq filtermap f G \iff F \leq G
apply (safe intro: filtermap-mono)
apply (auto simp: le-filter_def eventually-filtermap)
apply (erule tac x=\lambda x. P (inv f x) in allE)
apply auto
done

lemma eventually-compose-filterlim:
assumes eventually P F filterlim f F G
shows eventually (\lambda x. P (f x)) G
using assms by (simp add: filterlim_iff)

lemma filtermap-eq-strong: inj f \implies filtermap f F = filtermap f G \iff F = G
by (simp add: filtermap-mono-strong eq_iff)

lemma filtermap-fun-inverse:
assumes g: filterlim g F G
assumes f: filterlim f G F
assumes ev: eventually (\lambda x. f (g x) = x) G
shows filtermap f F = G
proof (rule antisym)
show filtermap f F \leq G
using f unfolding filterlim_def.
have G = filtermap f (filtermap g G)
using ev by (auto elim: eventually-elim2 simp: filter-eq_iff eventually-filtermap)
also have \ldots \leq filtermap f F
using g by (intro filtermap-mono) (simp add: filterlim-def)
finally show G \leq filtermap f F.
qed

lemma filterlim-principal:
(LIM x F. f x \bdash principal S) \iff (eventually (\lambda x. f x \in S) F)
unfolding filterlim-def eventually-filtermap le-principal ..

lemma filterlim-filtercomap [intro]: filterlim f F (filtercomap f F)
unfolding filterlim-def by (rule filterlim-filtercomap)

lemma filterlim-inf:
(LIM x F1. f x \bdash inf F2 F3) \iff ((LIM x F1. f x \bdash F2) \land (LIM x F1. f x \bdash F3))
unfolding filterlim-def by simp

lemma filterlim-INF:
(LIM x F. f x \bdash (\prod b \in B. G b)) \iff (\forall b \in B. LIM x F. f x \bdash G b)
unfolding filterlim-def le-INF iff ..
THEORY "Filter"

lemma filterlim-INF-INF: 
(∃ m. m ∈ J ⇒ ∃ i ∈ I. filtermap f (F i) ≤ G m) ⇒ LIM x (∩ i ∈ I. F i). f x := (∩ j ∈ J. G j)
  unfolding filterlim-def by (rule order-trans[OF filtermap-INF INF-mono])

lemma filterlim-INF': x ∈ A ⇒ filterlim f F (G x) ⇒ filterlim f F (∩ x ∈ A. G x)
  unfolding filterlim-def by (rule order-trans[OF filtermap-INF INF-mono])

lemma filterlim-filtercomap-iff: filterlim f (filtercomap g G) F ←→ filterlim (g ∘ f) G F
  by (simp add: filterlim-def filtermap-le-iff-le-filtercomap filtercomap-filtercomap)

lemma filterlim-iff-le-filtercomap: filterlim f F G ←→ G ≤ filtercomap f F
  by (simp add: filterlim-def filtermap-le-iff-le-filtercomap)

lemma filterlim-base: 
(∀ m x. m ∈ J ⇒ i m ∈ I) ⇒ (∀ m x. m ∈ J ⇒ x ∈ F (i m) ⇒ f x ∈ G m)
  LIM x (∩ i ∈ I. principal (F i)). f x := (∩ j ∈ J. principal (G j))
  by (force intro!: filterlim-INF-INF simp: image-subset-iff)

lemma filterlim-base-iff: 
assumes I ≠ {} and chain: ∃ i j. i ∈ I ⇒ j ∈ I ⇒ F i ⊆ F j ∨ F j ⊆ F i
shows (LIM x (∩ i ∈ I. principal (F i)). f x := (∩ j ∈ J. principal (G j)) ←→
  (∀ j ∈ J. ∃ i ∈ I. ∃ x ∈ F i. f x ∈ G j)
  unfolding filterlim-INF filterlim-principal
  proof (subst eventually-INF-base)
  fix i j assume i ∈ I j ∈ I
  with chain[OF this] show ∃ x ∈ I. principal (F x) ≤ inf (principal (F i)) (principal (F j))
    by auto
  qed (auto simp: eventually-principal i ≠ {})
THEORY "Filter"

by (simp add: filterlim-iff eventually-sequentially)

lemma filterlim-if:
  \[ \text{LIM} x \infty F (\text{principal} \{ x. \, P x \}). \, f x :> G \implies \]
  \[ \text{LIM} x \infty F (\text{principal} \{ x. \neg P x \}). \, g x :> G \implies \]
  \[ \text{LIM} x \infty F. \, \text{if} \, P \, x \, \text{then} \, f \, x \, \text{else} \, g \, x :> G \]
unfolding filterlim-iff eventually-inf-principal by (auto simp: eventually-conj-iff)

lemma filterlim-Pair:
  \[ \text{LIM} x \infty F. \, f x :> G \implies \]
  \[ \text{LIM} x \infty F. \, g x :> H \implies \]
  \[ \text{LIM} x \infty F. \, \text{if} \, P \, x \, \text{then} \, f \, x \, \text{else} \, g \, x :> G \times F H \]
unfolding filterlim-def by (rule order-trans[OF filtermap-Pair prod-filter-mono])

90.7 Limits to at-top and at-bot

lemma filterlim-at-top:
  fixes \( f :: 'a \Rightarrow ('b::linorder) \)
  shows \( \text{LIM} x \infty F. \, f x :> at-top \iff \)
  \( \forall Z. \, \text{eventually} \, (\lambda x. \, Z \leq f x) \, F \)
by (auto simp: filterlim-iff eventually-at-top-linorder elim!: eventually-mono)

lemma filterlim-at-top-mono:
  \[ \text{LIM} x \infty F. \, f x :> at-top \implies \]
  \[ \text{eventually} \, (\lambda x. \, f x \leq (g x :: 'a::linorder)) \, F \implies \]
  \[ \text{LIM} x \infty F. \, g x :> at-top \]
by (auto simp: filterlim-at-top elim: eventually-elim2 intro: order-trans)

lemma filterlim-at-top-dense:
  fixes \( f :: 'a \Rightarrow ('b::unbounded-dense-linorder) \)
  shows \( \forall Z. \, \text{eventually} \, (\lambda x. \, Z \leq f x) \, F \)
by (metis eventually-mono[of F] eventually-at-top linorder_less_imp_le filterlim-at-top[of f F] filterlim-iff[of at-top F])

lemma filterlim-at-top-ge:
  fixes \( f :: 'a \Rightarrow ('b::linorder) \) \text{ and } c :: 'b \)
  shows \( \forall Z\geq c. \, \text{eventually} \, (\lambda x. \, Z \leq f x) \, F \)
unfolding at-top-sub[of c] filterlim-INF by (auto simp add: filterlim-principal)

lemma filterlim-at-top-at-top:
  fixes \( f :: 'a::linorder \Rightarrow 'b::linorder \)
  assumes mono: \( \forall x y. \, Q x \implies Q y \implies x \leq y \implies f x \leq f y \)
  assumes bij: \( \forall x. \, P x \implies f (g x) = x \land x. \, P x \implies Q (g x) \)
  assumes Q: \text{ eventually } Q \text{ at-top }
  assumes P: \text{ eventually } P \text{ at-top }
  shows \text{ filterlim} \, f \, \text{ at-top at-top }
proof -
  from P obtain \( x \) \text{ where } \( \forall y. \, x \leq y \implies P y \)
  unfolding eventually-at-top-linorder by auto
show \( ? \)thesis
proof (intro filterlim-at-top-ge[THEN iffD2] allI impI)
  fix \( z \) assume \( x \leq z \)
with \( z \) have \( P \, z \) by auto
have eventually \((\lambda x. \, g \, z \leq x)\) at-top
by (rule eventually-ge-at-top)
with \( Q \) show eventually \((\lambda x. \, z \leq f \, x)\) at-top
by eventually-elim (metis mono bij \( P \, z \))
qed
qed

lemma filterlim-at-top-gt:
fixes \( f :: \, 'a \Rightarrow ('b::unbounded-dense-linorder) \) and \( c :: \, 'b \)
shows \( (\text{LIM} \, x \, F, \, f \, x :> \text{at-top}) \iff (\forall \, Z>c. \text{eventually} \,(\lambda x. \, Z \leq f \, x) \, F) \)
by (metis filterlim-at-top order-less-le-trans gt-ex filterlim-at-top-ge)

lemma filterlim-at-bot:
fixes \( f :: \, 'a \Rightarrow ('b::linorder) \)
shows \( (\text{LIM} \, x \, F, \, f \, x :> \text{at-bot}) \iff (\forall \, Z. \text{eventually} \,(\lambda x. \, f \, x \leq Z) \, F) \)
by (auto simp: filterlim-iff eventually-at-bot-linorder elim!: eventually-mono)

lemma filterlim-at-bot-dense:
fixes \( f :: \, 'a \Rightarrow ('b::{dense-linorder, no-bot}) \)
shows \( (\text{LIM} \, x \, F, \, f \, x :> \text{at-bot}) \iff (\forall \, Z. \text{eventually} \,(\lambda x. \, f \, x < Z) \, F) \)
proof (auto simp add: filterlim-at-bot[of \( f \, F \)])
    fix \( Z :: \, 'b \)
    from lt-ex [of \( Z \)] obtain \( Z' \) where \( 1: \, Z' < Z \)
    assume \( \forall \, Z. \text{eventually} \,(\lambda x. \, f \, x \leq Z) \, F \)
    hence eventually \((\lambda x. \, f \, x \leq Z') \, F \) by auto
    thus eventually \((\lambda x. \, f \, x < Z) \, F \)
    by (rule eventually-mono) (use \( 1 \) in auto)
next
    fix \( Z :: \, 'b \)
    show \( \forall \, Z. \text{eventually} \,(\lambda x. \, f \, x < Z) \, F \implies \text{eventually} \,(\lambda x. \, f \, x \leq Z) \, F \)
    by (drule spec [of \( - \, Z \), erule eventually-mono, auto simp add: less-imp-le])
qed

lemma filterlim-at-bot-le:
fixes \( f :: \, 'a \Rightarrow ('b::linorder) \) and \( c :: \, 'b \)
shows \( (\text{LIM} \, x \, F, \, f \, x :> \text{at-bot}) \iff (\forall \, Z \leq c. \text{eventually} \,(\lambda x. \, Z \geq f \, x) \, F) \)
unfolding filterlim-at-bot
proof safe
    fix \( Z \)
    assume \( \ast: \, \forall \, Z \leq c. \text{eventually} \,(\lambda x. \, Z \geq f \, x) \, F \)
    with \( \ast[\text{THEN spec, of min Z c}] \) show \( \text{eventually} \,(\lambda x. \, Z \geq f \, x) \, F \)
    by (auto elim!: eventually-mono)
qed simp

lemma filterlim-at-bot-lt:
fixes \( f :: \, 'a \Rightarrow ('b::unbounded-dense-linorder) \) and \( c :: \, 'b \)
shows \( (\text{LIM} \, x \, F, \, f \, x :> \text{at-bot}) \iff (\forall \, Z < c. \text{eventually} \,(\lambda x. \, Z \geq f \, x) \, F) \)
by (metis filterlim-at-bot filterlim-at-bot-le lt-ex order-le-less-trans)
lemma filterlim-at-top-div-const-nat:
  assumes c > 0
  shows filterlim (λx::nat. x div c) at-top at-top
unfolding filterlim-at-top
proof
  fix C :: nat
  have ∗: n div c ≥ C if n ≥ C * c for n
    using assms that by (metis div-le-mono div-mult-self-is-m)
  have eventually (λn. n ≥ C * c) at-top
    by (rule eventually-ge-at-top)
  thus eventually (λn. n div c ≥ C) at-top
    by eventually-elim (use ∗ in auto)
qed

lemma filterlim-finite-subsets-at-top:
  filterlim f (finite-subsets-at-top A) F ←→
  (∀ X. finite X ∧ X ⊆ A → eventually (λy. finite (f y) ∧ X ⊆ f y ∧ f y ⊆ A)) F
(is ?lhs = ?rhs)
proof
  assume ?lhs
  thus ?rhs unfolding filterlim-def le-filter-def eventually-finite-subsets-at-top
proof (safe, goal-cases)
  case (1 P X)
  hence ∗: (∀ F x in F. P (f x)) if eventually P (finite-subsets-at-top A) for P
    using that by (auto simp: filterlim-def le-filter-def eventually-filtermap)
  have ∀ F. Y in finite-subsets-at-top A. finite Y ∧ X ⊆ Y ∧ Y ⊆ A
    using / unfolding eventually-finite-subsets-at-top by force
  thus ?case by (intro ∗) auto
qed
next
  assume rhs: ?rhs
  show ?rhs unfolding filterlim-def le-filter-def eventually-finite-subsets-at-top
proof (safe, goal-cases)
  case (1 P X)
  with rhs have ∀ F. Y in F. finite (f y) ∧ X ⊆ f y ∧ f y ⊆ A by auto
  thus eventually P (filtermap f F) unfolding eventually-filtermap
    by eventually-elim (insert 1, auto)
qed
qed

lemma filterlim-atMost-at-top:
  filterlim (λn. {..n}) (finite-subsets-at-top (UNIV :: nat set)) at-top
unfolding filterlim-finite-subsets-at-top
proof (safe, goal-cases)
  case (1 X)
  then obtain n where X ⊆ {..n} by (auto simp: finite-nat-set-iff-bounded-le)
  show ?case using eventually-ge-at-top[of n]
    by eventually-elim (insert n, auto)
qed

lemma filterlim-lessThan-at-top:
  filterlim (λn. {..<n}) (finite-subsets-at-top (UNIV :: nat set)) at-top
unfolding filterlim-finite-subsets-at-top
proof (safe, goal-cases)
case (1 X)
  then obtain n where n : X ⊆ {..<n} by (auto simp: finite-nat-set-iff-bounded)
  show ?case using eventually-ge-at-top[of n]
    by eventually-elim (insert n, auto)
qued

lemma filterlim-minus-const-nat-at-top:
  filterlim (λn. n - c) sequentially sequentially
unfolding filterlim-at-top
proof
  fix a :: nat
  show eventually (λn. n - c ≥ a) at-top
    using eventually-ge-at-top[of a + c]
    by eventually-elim auto
qued

lemma filterlim-add-const-nat-at-top:
  filterlim (λn. n + c) sequentially sequentially
unfolding filterlim-at-top
proof
  fix a :: nat
  show eventually (λn. n + c ≥ a) at-top
    using eventually-ge-at-top[of a]
    by eventually-elim auto
qued

90.8 Setup 'a filter for lifting and transfer

lemma filtermap-id [simp, id-simps]: filtermap id = id
  by (simp add: fun_eq_iff id_def filtermap-ident)

lemma filtermap-id' [simp]: filtermap (λx. x) = (λF. F)
  by (filtermap-id unfolding id_def).

context includes lifting-syntax
begin

definition map-filter-on :: 'a set ⇒ ('a ⇒ 'b) ⇒ 'a filter ⇒ 'b filter where
  map-filter-on X f F = Abs-filter (ΛP. eventually (Λx. P (f x) ∧ x ∈ X) F)

lemma is-filter-map-filter-on:
  is-filter (ΛP. ∀ x in F. P (f x) ∧ x ∈ X) ⇐⇒ eventually (Λx. x ∈ X) F
proof (rule iff; unfold-locales)
  show ∀ F x in F. True ∧ x ∈ X if eventually (Λx. x ∈ X) F using that by simp
  show ∀ F x in F. (P (f x) ∧ Q (f x)) ∧ x ∈ X if ∀ F x in F. P (f x) ∧ x ∈ X

\begin{verbatim}

∀ F x in F. Q (f x) ∧ x ∈ X for P Q
  using eventually-conj[OF that] by(auto simp add: conj-ac cong: conj-cong)
show ∀ F x in F. Q (f x) ∧ x ∈ X if ∀ x. P x → Q x ∀ F x in F. P (f x) ∧ x ∈ X for P Q
  using that(2) by(rule eventually-mono)(use that(1) in auto)
show eventually (λx. x ∈ X) F if is-filter (λP. ∀ F x in F. P (f x) ∧ x ∈ X)
  using is-filter.True[OF that] by simp
qed

lemma eventually-map-filter-on: eventually P (map-filter-on X f F) = (∀ F x in F. P (f x) ∧ x ∈ X)
  if eventually (λx. x ∈ X) F
  by(simp add: is-filter-map-filter-on map-filter-on-def eventually-Abs-filter that)

lemma map-filter-on-UNIV: map-filter-on UNIV = filtermap
  by(simp add: map-filter-on-def filtermap-def fun-eq-iff)

lemma map-filter-on-comp: map-filter-on X f (map-filter-on Y g F) = map-filter-on Y (f o g) F
  if g ' Y ⊆ X and eventually (λx. x ∈ Y) F
unfolding map-filter-on-def using that(1)
  by(auto simp add: eventually-Abs-filter that(2) is-filter-map-filter-on intro: arg-cong[where f=Abs-filter] arg-cong2[where f=eventually])

inductive rel-filter :: ('a ⇒ 'b ⇒ bool) ⇒ 'a filter ⇒ 'b filter ⇒ bool for R F G
where
  rel-filter R F G if eventually (case_prod R) Z map-filter-on {(x, y). R x y} fst Z
  = F map-filter-on {(x, y). R x y} snd Z = G

lemma rel-filter-eq [relator-eq]: rel-filter (=) (=)
  proof(intro ext iff)+
  show F = G if rel-filter (=) F G for F G using that
    by cases(clarsimp simp add: filter-eq-iff eventually-map-filter-on split-def cong: rev-conj-cong)
  show rel-filter (=) F G if F = G for F G unfolding (=) F G
    proof
      let ?Z = map-filter-on UNIV (λx. (x, x)) G
      have [simp]: range (λx. (x, x)) ⊆ {(x, y). x = y} by auto
      show map-filter-on {(x, y). x = y} fst ?Z = G and map-filter-on {(x, y). x = y} snd ?Z = G
        by(simp-add: map-filter-on-comp)(simp-add: map-filter-on-UNIV o-def)
      show ∀ F (x, y) in ?Z. x = y by(simp add: eventually-map-filter-on)
    qed
  qed

lemma rel-filter-mono [relator-mono]: rel-filter A ≤ rel-filter B if le: A ≤ B
  proof(clarsimp elim!: rel-filter.cases)
    show rel-filter B (map-filter-on {(x, y). A x y} fst Z) (map-filter-on {(x, y). A x y} snd Z)
  qed

THEORY "Filter"

\end{verbatim}
(is rel-filter - \?X \?Y) if \( \forall_F (x, y) \) in Z. A x y for Z

proof
let \(?Z = \text{map-filter-on} \{(x, y). A x y\} \text{id} Z\)
show \( \forall_F (x, y) \in ?Z. B x y \text{ using } \text{le that}\)
by(simp add: eventually-map-filter-on le-fun-def split-def conj-commute cong: conj-cong)

have [simp]: \(\{(x, y). A x y\} \subseteq \{(x, y). B x y\}\) using le by auto
show map-filter-on \{(x, y). B x y\} \(\text{fst } ?Z = ?X\) map-filter-on \{(x, y). B x y\}
\(\text{snd } ?Z = ?Y\)
using le that by(simp-all add: le-fun-def map-filter-on-comp)

qed

lemma rel-filter-conversep: rel-filter A\(^{-1-1}\) = (rel-filter A)\(^{-1-1}\)

proof
(safe intro; ext elim!: rel-filter.cases)
show *: rel-filter A (map-filter-on \{(x, y). A\(^{-1-1}\) x y\} \text{snd} Z\) (map-filter-on \{(x, y). A\(^{-1-1}\) x y\} \text{fst} Z\)
(is rel-filter - \?X \?Y) if \( \forall_F (x, y) \) in Z. A\(^{-1-1}\) x y for A

proof
let \(?Z = \text{map-filter-on} \{(x, y). A y x\} \text{prod.swap} Z\)
show \( \forall_F (x, y) \in ?Z. A x y \text{ using } \text{by(simp add: eventually-map-filter-on)}\)

have [simp]: prod.swap \(\{(x, y). A x y\} \subseteq \{(x, y). A y x\}\) by auto
show map-filter-on \{(x, y). A x y\} \(\text{fst } ?Z = ?X\) map-filter-on \{(x, y). A x y\}
\(\text{snd } ?Z = ?Y\)
using that by(simp-all add: map-filter-on-comp o-def)

qed

show rel-filter A\(^{-1-1}\) (map-filter-on \{(x, y). A x y\} \text{snd} Z\) (map-filter-on \{(x, y). A x y\} \text{fst} Z\)
if \( \forall_F (x, y) \) in Z. A x y for Z using *[of A\(^{-1-1}\) Z] that by simp

qed

lemma rel-filter-distr [relator-distr]:
rel-filter A OO rel-filter B = rel-filter (A OO B)

proof
(safe intro; ext elim!: rel-filter.cases)
let \(?AB = \{(x, y). (A OO B) x y\}\)
show (rel-filter A OO rel-filter B)
(map-filter-on \{(x, y). (A OO B) x y\} \text{fst} Z\) (map-filter-on \{(x, y). (A OO B) x y\} \text{snd} Z\)
(is (-. OO -. \?F \?H) if \( \forall_F (x, y) \) in Z. (A OO B) x y for Z

proof
let \(?G = \text{map-filter-on } ?AB (\lambda(x, y). \text{SOME} z. A x z \land B z y)\) Z
show rel-filter A \?F \?G

proof
let \(?Z = \text{map-filter-on } ?AB (\lambda(x, y). \text{SOME} z. A x z \land B z y)) \) Z
show \( \forall_F (x, y) \in ?Z. A x y \text{ using that}\)
by(auto simp add: eventually-map-filter-on split-def elim!: eventually-mono intro: someI2)

have [simp]: (\lambda. (\text{fst} p, \text{SOME} z. A (\text{fst} p) z \land B z (\text{snd} p))) \subseteq \{p. (A OO B) (\text{fst} p) (\text{snd} p)\}

by(auto intro: someI2)
show map-filter-on \{(x, y). A \times y\} \text{fst} ?Z = ?F \text{map-filter-on} \{(x, y). A \times y\}

\text{snd} ?Z = ?G

using that \text{by}(\text{simp-all add: map-filter-on-comp split-def o-def})

qed

show rel-filter B ?G ?H

proof

let ?Z = map-filter-on ?AB (λ(x, y). (SOME z. A \times z \land B \times y, y)) Z

show \forall F (x, y) \in ?Z. B \times y \text{ using that}

\text{by}(\text{auto simp add: eventually-map-filter-on split-def elim!: eventually-mono intro: someI2})

have \{simp\}: (λp. (SOME z. A \times p \land B \times (\text{snd} p), \text{snd} p)) \cdot \{p. \text{A OO B} \times p \times \text{snd} p\} \subseteq \{p. B \times p \times \text{snd} p\} \text{ by(auto intro: someI2)}

show map-filter-on \{(x, y). B \times y\} \text{fst} ?Z = ?G \text{map-filter-on} \{(x, y). B \times y\}

\text{snd} ?Z = ?H

using that \text{by}(\text{simp-all add: map-filter-on-comp split-def o-def})

qed

qed

fix F G

assume F: \forall F (x, y) \in F. A \times y \text{ and } G: \forall F (x, y) \in G. B \times y

and eq: map-filter-on \{(x, y). B \times y\} \text{fst} G = \text{map-filter-on} \{(x, y). A \times y\} \text{snd}

F (is \text{\texttt{?Y2 = \texttt{?Y1}}})

let ?X = map-filter-on \{(x, y). A \times y\} \text{fst} F

and ?Z = (map-filter-on \{(x, y). B \times y\} \text{snd} G)

have step: \exists P' \subseteq P. \exists Q' \subseteq Q. \text{eventually } P' \times F \land \text{eventually } Q' \times G \land \{y. \exists x. P'

(x, y)\} = \{y. \exists z. Q' (y, z)\}

if P: \text{eventually } P \times F \text{ and } Q: \text{eventually } Q \times G \text{ for } P \times Q

proof

let ?P = λ(x, y). P (x, y) \land A \times y \text{ and } ?Q = λ(y, z). Q (y, z) \land B \times y

define P' where P' = λ(x, y). ?P (x, y) \land (∃z. ?Q (y, z))

define Q' where Q' = λ(y, z). ?Q (y, z) \land (∃x. ?P (x, y))

have P' \subseteq P \times Q' \subseteq Q \{y. \exists x. P' (x, y)\} = \{y. ∃z. Q' (y, z)\}

\text{by(auto simp add: P'-def Q'-def})

moreover

from P \times Q \times G \text{ have } P': \text{eventually } ?P \times F \text{ and } Q': \text{eventually } ?Q \times G

\text{by(simp-all add: eventually-conj-iff split-def)}

from P' \times F \text{ have } \forall F y \in ?Y1. \exists x. P (x, y) \land A \times y

\text{by(auto simp add: eventually-map-filter-on elim!: eventually-mono)}

from this[folded eq] obtain Q'' \text{ where } Q'' : \text{eventually } Q'' \times G

and Q''\times P: \{y. ∃z. Q'' (y, z)\} \subseteq \{y. ∃x. ?P (x, y)\}

\text{using } G \text{ by(fastforce simp add: eventually-map-filter-on)}

have eventually (inf Q'' ?Q) \times G \text{ using } Q'' \times Q' \text{ by(auto intro: eventually-conj simp add: inf-fun-def)}

then have eventually Q' \times G \text{ using } Q''\times P \text{ by(auto elim!: eventually-mono simp add: Q'-def})

moreover

from Q' \times G \text{ have } \forall F \times y \in ?Y2. ∃z. Q (y, z) \land B \times y

\text{by(auto simp add: eventually-map-filter-on elim!: eventually-mono)}

from this[unfolded eq] obtain P'' where P'': \text{eventually } P'' \times F
and $P''Q$: \{y, \exists z. P''(x, y)\} \subseteq \{y, \exists z. ?Q(y, z)\}

using $F$ by (fastforce simp add: eventually-map-filter-on)

have eventually (inf $P'' ?P$) $F$ using $P'' P'$ by (auto intro: eventually-conj simp add: inf-fun-def)

then have eventually $P' F$ using $P''Q$ by (auto elim!: eventually-mono simp add: $P'$-def)

ultimately show $?thesis$ by blast
qed

show rel-filter $(A OO B) ?X ?Z$
proof
  let $?Y = \lambda Y. \exists X Z. \text{eventually } ?X \land \text{eventually } ?Z \land (\lambda(x, z). X x \land Z z \land (A OO B) x z) \leq Y$

  have $Y$: is-filter $?Y$
  proof
    show $?Y(\lambda x. \text{True})$ by (auto simp add: le-fun-def intro: eventually-True)
    show $?Y(\lambda x. P x \land Q x)$ if $?Y P ?Y Q$ for $P Q$ using that
      apply clarify
      apply (intro ex1 conjI; (elim eventually-rev-mp; fold imp-conjL; intro always-eventually allI; rule imp-refl)?)
      apply auto
      done
    show $?Y Q$ if $?Y P \forall x. P x \longrightarrow Q x$ for $P Q$ using that by blast
  qed

define $Y$ where $Y = \text{Abs-filter } ?Y$

have eventually-$Y$: eventually $P Y \iff ?Y P$ for $P$
  using eventually-Abs-filter[OF $Y$, of $P$] by (simp add: $Y$-def)

show $YY: \forall F(x, y) \in Y. (A OO B) x y$ using $F G$

  by (auto simp add: eventually-$Y$ eventually-map-filter-on eventually-conj-iff intro!: eventually-True)

  have $?Y(\lambda(x, z). P x \land (A OO B) x z) \iff (\forall F(x, y) \in F. P x \land A x y)$

(is $？rhs = ？lhs$) for $P$

proof
  show $？lhs$ if $？rhs$ using $G F$ that
    by (auto int 3 intro: ex1[where $x=\lambda x. \text{True}$] simp add: eventually-map-filter-on split-def)

  assume $？lhs$

then obtain $X Z$ where $\forall F(x, y) \in F. X x \land A x y$
  and $\forall F(x, y) \in G. Z y \land B x y$
  and $(\lambda(x, z). X x \land Z z \land (A OO B) x z) \leq (\lambda(x, z). P x \land (A OO B) x z)$

using $F G$ by (auto simp add: eventually-map-filter-on split-def)

from step[OF this(1, 2)] this(3)

show $？rhs$ by (clarsimp elim!: eventually-rev-mp simp add: le-fun-def)(fastforce intro: always-eventually)
qed

then show map-filter-on $?AB\text{ fst } Y = ?X$
  by (simp add: filter-eq-iff $YY$ eventually-map-filter-on)(simp add: eventually-$Y$

eventually-map-filter-on $F G$; simp add: split-def)
have \( \forall Y \lambda(x, z). \, P \, z \land (A \, O \, O \, B) \, x \, z \) \( \iff \) \( (\forall F \, (x, y) \text{ in } G. \, P \, y \land B \, x \, y) \)
(is \( \text{?lhs = ?rhs} \) for \( P \))

proof
  show \( \text{?lhs if ?rhs using } G \, F \) that
    by(auto 4 3 intro: ext[where \( x=\lambda\). True] simp add: eventually-map-filter-on split-def)
  assume \( \text{?lhs} \)
  then obtain \( X \, Z \) where \( \forall F \, (x, y) \text{ in } F. \, X \, x \land A \, x \, y \)
  and \( \forall F \, (x, y) \text{ in } G. \, Z \, y \land B \, x \, y \)
  and \( (\lambda(x, z). \, X \, x \land Z \, z \land (A \, O \, O \, B) \, x \, z) \leq (\lambda(x, z). \, P \, z \land (A \, O \, O \, B) \, x \, z) \)
  using \( F \, G \) by(auto simp add: eventually-map-filter-on split-def)
  from step[OF this(1, 2)] this(3)
  show \( \text{?rhs by(clarsimp elim!: eventually-rev-mp simp add: le-fun-def)(fastforce intro: always-eventually) \)
  qed
then show \( \text{map-filter-on } ?AB \, \text{snd } Y = ?Z \)
  qed

lemma filtermap-parametric: \((A \Rightarrow B) \Rightarrow \Rightarrow \text{rel-filter } A \Rightarrow \Rightarrow \text{rel-filter } B) \text{ filtermap filtermap}
proof(intro rel-funI; crule rel-filter.cases; hypsubst)
  fix \( f \, g \, Z \)
  assume \( fg: (A \Rightarrow B) \, f \, g \) and \( Z: \forall F \, (x, y) \text{ in } Z. \, A \, x \, y \)
  have rel-filter \( B \) (map-filter-on \( \{(x, y). \, A \, x \, y\} \) \( (\text{f } \text{fst}) \, Z \) (\text{map-filter-on } \{(x, y). \, A \, x \, y\} \) \( (g \, \text{snd}) \, Z \))
  (is \( \text{rel-filter - ?F ?G} \))
  proof
    let \( \forall Z = \text{map-filter-on } \{(x, y). \, A \, x \, y\} \) \( (\text{map-prod } f \, g) \, Z \)
    show \( \forall F \, (x, y) \text{ in } ?Z. \, B \, x \, y \) using \( fg \, Z \)
      by(auto simp add: eventually-map-filter-on split-def elim!: eventually-monotonic rel-funD)
    have \( \text{simp: } \text{map-prod } f \, g = \{ p. \, A \, (\text{fst } p) \, (\text{snd } p) \} \subseteq \{ p. \, B \, (\text{fst } p) \, (\text{snd } p) \} \)
      using \( fg \) by(auto dest: rel-funD)
    show \( \text{map-filter-on } \{(x, y). \, B \, x \, y\} \) \( (\text{fst \ ?Z = ?F \ map-filter-on } \{(x, y). \, B \, x \, y\}) \)
      using \( Z \) by(auto simp add: map-filter-on-compress split-def)
  qed
thus rel-filter \( B \) \( \text{filtermap } (\text{map-filter-on } \{(x, y). \, A \, x \, y\} \) \( (\text{fst \ Z}) \) \( (\text{filtermap \ g}) \) \( \text{map-filter-on } \{(x, y). \, A \, x \, y\} \) \( (\text{snd \ Z})) \)
  using \( Z \) by(simp add: map-filter-on-UNIV[symmetric] map-filter-on-compress)
  qed

lemma rel-filter-Grp: \( \text{rel-filter } (\text{Grp } UNIV \, f) = \text{Grp } UNIV \) \( \text{filtermap } f \)
proof(intro antisym predicate2I; elim GrpE; hypsubst)(i)
  rule GrpI[OF UNIV-I]
  fix \( F \, G \)
  assume rel-filter \( (\text{Grp } UNIV \, f) \) \( F \, G \)
hence \( \text{rel-filter} \, (=) \, (\text{filtermap} \, f \, F) \, (\text{filtermap} \, \text{id} \, G) \)
by (rule filtermap-parametric THEN rel-funD, THEN rel-funD, rotated) (simp add: Grp-def rel-fun-def)
thus \( \text{filtermap} \, f \, F = G \) by (simp add: rel-filter-eq)

next
fix \( F :: 'a \) filter
have \( \text{rel-filter} \, (=) \, F \, F \)
by (simp add: rel-filter-eq)

hence \( \text{rel-filter} \, (\text{Grp} \, \text{UNIV} \, f) \, (\text{filtermap} \, \text{id} \, F) \, (\text{filtermap} \, f \, F) \)
by (rule filtermap-parametric THEN rel-funD, THEN rel-funD, rotated) (simp add: Grp-def rel-fun-def)
thus \( \text{rel-filter} \, (\text{Grp} \, \text{UNIV} \, f) \, F \, (\text{filtermap} \, f \, F) \) by simp
qed

lemma Quotient-filter [quot-map]:
Quotient \( R \) Abs Rep T \( \Longrightarrow \) Quotient (rel-filter \( R \)) (filtermap Abs) (filtermap Rep) (rel-filter \( T \))
unfolding Quotient-alt-def5 rel-filter-eq [symmetric] rel-filter-Grp [symmetric]

lemma left-total-rel-filter [transfer-rule]: left-total \( A \) \( \Longrightarrow \) left-total (rel-filter \( A \))
by (rule rel-filter-mono)

lemma right-total-rel-filter [transfer-rule]: right-total \( A \) \( \Longrightarrow \) right-total (rel-filter \( A \))
using left-total-rel-filter [of \( A^{-1} \)] by (simp add: rel-filter-conversep)

lemma bi-total-rel-filter [transfer-rule]: bi-total \( A \) \( \Longrightarrow \) bi-total (rel-filter \( A \))
unfolding bi-total-alt-def by (simp add: left-total-rel-filter right-total-rel-filter)

lemma left-unique-rel-filter [transfer-rule]: left-unique \( A \) \( \Longrightarrow \) left-unique (rel-filter \( A \))
by (rule rel-filter-mono)

lemma right-unique-rel-filter [transfer-rule]:
right-unique \( A \) \( \Longrightarrow \) right-unique (rel-filter \( A \))
using left-unique-rel-filter [of \( A^{-1} \)] by (simp add: rel-filter-conversep)

lemma bi-unique-rel-filter [transfer-rule]: bi-unique \( A \) \( \Longrightarrow \) bi-unique (rel-filter \( A \))
by (simp add: bi-unique-alt-def left-unique-rel-filter right-unique-rel-filter)

lemma eventually-parametric [transfer-rule]:
\((A \Longrightarrow (=)) \Longrightarrow \text{rel-filter} \, A \Longrightarrow (=)\) eventually eventually
by (auto 4 intro: rel-funI elim!: rel-filter.cases simp add: eventually-map-filter-on dest: rel-funD intro: always-eventually elim!: eventually-rev-mp)
lemma frequently-parametric [transfer-rule]: \((A \Longrightarrow (\_)) \Longrightarrow \) rel-filter A
===\Longrightarrow (\_)) frequently frequently
  unfolding frequently-def [abs-def] by transfer-prover

lemma is-filter-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-total A
  assumes [transfer-rule]: bi-unique A
  shows \(((A \Longrightarrow (\_)) \Longrightarrow (\_)) \Longrightarrow (\_)) is-filter is-filter
  unfolding is-filter-def by transfer-prover

lemma top-filter-parametric [transfer-rule]: rel-filter A top top if bi-total A
proof
  let \(?Z = principal \{(x, y). A x y\}\)
  show \(\forall F (x, y) in \?Z. A x y by (simp add: eventually-principal)\)
  show map-filter-on \{(x, y). A x y\} fst \?Z = top map-filter-on \{(x, y). A x y\}
  snd \?Z = top
    using that by(auto simp add: filter-eq-iff eventually-map-filter-on eventually-principal bi-total-def)
qed

lemma bot-filter-parametric [transfer-rule]: rel-filter A bot bot
proof
  show \(\forall F (x, y) in bot. A x y by simp\)
  show map-filter-on \{(x, y). A x y\} fst bot = bot map-filter-on \{(x, y). A x y\}
  snd bot = bot
    by(simp-all add: filter-eq-iff eventually-map-filter-on)
qed

lemma principal-parametric [transfer-rule]: (rel-set A === rel-filter A) principal
proof(rule rel-funI rel-filter.intros+)
  fix S S'
  assume *: rel-set A S S'
  define SS' where SS' = S \times S' \cap \{(x, y). A x y\}
  have SS': SS' \subseteq \{(x, y). A x y\} and [simp]: S = fst ' SS' S' = snd ' SS'
    using * by(auto 4 3 dest: rel-setD1 rel-setD2 intro: rev-image-eqI simp add: SS'-def)
  let \(?Z = principal SS'\)
  show \(\forall F (x, y) in \?Z. A x y using SS' by(auto simp add: eventually-principal)\)
  then show map-filter-on \{(x, y). A x y\} fst \?Z = principal S
    and map-filter-on \{(x, y). A x y\} snd \?Z = principal S'
      by(auto simp add: filter-eq-iff eventually-map-filter-on eventually-principal)
qed

lemma sup-filter-parametric [transfer-rule]:
  (rel-filter A === rel-filter A) sup sup
proof(intro rel-funI; elim rel-filter.cases; hypsubst)
  show rel-filter A
    (map-filter-on \{(x, y). A x y\} fst FG \sqcup map-filter-on \{(x, y). A x y\} fst FG')
THEORY "Filter"

(map-filter-on {x, y}. A x y) snd FG \sqcup map-filter-on {x, y}. A x y) snd FG'
(is rel-filter - (sup ?F ?G) (sup ?F' ?G'))
if \forall (x, y) in FG. A x y \forall (x, y) in FG'. A x y for FG FG'
proof
let ?Z = sup FG FG'
show \forall (x, y) in ?Z. A x y by(simp add: eventually-sup that)
then show map-filter-on {x, y}. A x y) fst ?Z = sup ?F ?G
and map-filter-on {x, y}. A x y) snd ?Z = sup ?F' ?G'
by(simp-all add: filter-eq-iff eventually-map-filter-on eventually-sup)
qed
qed

lemma Sup-filter-parametric [transfer-rule]: (rel-set (rel-filter A) ===> rel-filter A) Sup
proof (rule rel-funI)
fix S S'
define SS' where SS' = S \times S' \cap \{(F, G). rel-filter A F G\}
assume rel-set (rel-filter A) S S'
then have SS': SS' \subseteq \{(F, G). rel-filter A F G\} and [simp]: S = fst ' SS' S' = snd ' SS'
by(auto 4 3 dest: rel-setD1 rel-setD2 intro: rev-image-eqI simp add: SS'-def)
from SS' obtain Z where Z: \forall (F, G). (F, G) \in SS' \implies
(id F = map-filter-on {x, y}. A x y) \land
id G = map-filter-on {x, y}. A x y) snd (Z F G)
unfolding rel-filter.simps by atomize-elim((rule choice allI)+; auto)
have id: eventually P F = eventually P (id F) eventually Q G = eventually Q (id G)
if (F, G) \in SS' for P Q F G by simp-all
show rel-filter A (Sup S) (Sup S')
proof
let ?Z = \bigcup (F, G)\in SS'. Z F G
show \forall (x, y) in ?Z. A x y using Z by(auto simp add: eventually-Sup)
show map-filter-on {x, y}. A x y) fst ?Z = Sup S map-filter-on {x, y}. A x y) snd ?Z = Sup S'
unfolding filter-eq-iff
by(auto 4 4 simp add: id eventually-Sup eventually-map-filter-on *[simplified eventually-Sup] simp del: id-apply dest: Z)
qed
qed

context
fixes A :: 'a \Rightarrow 'b \Rightarrow bool
assumes [transfer-rule]: bi-unique A
begin
lemma le-filter-parametric [transfer-rule]:
(rel-filter A ===> rel-filter A ===> (=)) (\leq) (\leq)
unfolding le-filter-def[abs-def] by transfer-prover
lemma less-filter-parametric [transfer-rule]:
  (rel-filter A ===> rel-filter A ===> (=)) (<) (<)
unfolding less-filter-def [abs-def] by transfer-prover

context
  assumes [transfer-rule]: bi-total A
begin

lemma Inf-filter-parametric [transfer-rule]:
  (rel-set (rel-filter A) ===> rel-filter A) Inf Inf
unfolding Inf-filter-def [abs-def] by transfer-prover

lemma inf-filter-parametric [transfer-rule]:
  (rel-filter A ===> rel-filter A ===> rel-filter A) inf inf
proof (intro rel-funI+)
  fix F F' G G'
  assume [transfer-rule]: rel-filter A F F' rel-filter A G G'
  have rel-filter A (Inf {F, G}) (Inf {F', G'}) by transfer-prover
  thus rel-filter A (inf F G) (inf F' G') by simp
qed

end

context
  includes lifting-syntax
begin

lemma prod-filter-parametric [transfer-rule]:
  (rel-filter R ===> rel-filter S ===> rel-filter (rel-prod R S)) prod-filter prod-filter
proof (intro rel-funI; elim rel-filter_cases; hypsubst)
  fix F G
  assume F: ∀ F (x, y) in F. R x y and G: ∀ F (x, y) in G. S x y
  show rel-filter (rel-prod R S)
    (map-filter-on {(x, y), R x y} fst F × F map-filter-on {(x, y), S x y} fst G)
    (map-filter-on {(x, y), R x y} snd F × F map-filter-on {(x, y), S x y} snd G)
    (is rel-filter ?RS ?F ?G)
proof
  let ?Z = filtermap (λ((a, b), (a', b')). ((a, a'), (b, b'))) (prod-filter F G)
  show ∀ F (x, y) in ?Z. rel-prod R S x y using F G
    by (auto simp add: eventually-filtermap split-beta eventually-prod-filter)
  show map-filter-on {(x, y), ?RS x y} fst ?Z = ?F using F G
    apply (clarsimp simp add: filter-eq-iff eventually-map-filter-on *)
    apply (simp add: eventually-filtermap split-beta eventually-prod-filter)
apply (subst eventually-map-filter-on; simp)+
apply (rule iffI; clarsimp)
subgoal for P P' P''
  apply (rule exI [where x=λ a. ∃ b. P' (a, b) ∧ R a b]; rule conjI)
subgoal by (fastforce elim: eventually-rev-mp eventually-mono)
  done
subgoal by fastforce
done
show map-filter-on \{(x, y). ?RS x y\} snd ?Z = ?G
using F G
apply (clarsimp simp add: filter-eq iff eventually-map-filter-on *)
apply (simp add: eventually-filtermap split-beta eventually-prod-filter)
apply (subst eventually-map-filter-on; simp)+
apply (rule iffI; clarsimp)
subgoal for P P' P''
  apply (rule exI [where x=λ b. ∃ a. P' (a, b) ∧ R a b]; rule conjI)
subgoal by (fastforce elim: eventually-rev-mp eventually-mono)
  done
subgoal by fastforce
done
qed
qed
end

Code generation for filters

definition abstract-filter :: (unit ⇒ 'a filter) ⇒ 'a filter
  where [simp]: abstract-filter f = f ()

code-datatype principal abstract-filter

hide-const (open) abstract-filter

declare [[code drop: filterlim prod-filter filtermap eventually
  inf :: - filter ⇒ - sup :: - filter ⇒ - less-eq :: - filter ⇒ -
  Abs-filter]]

declare filterlim-principal [code]
declare principal-prod-principal [code]
declare filtermap-principal [code]
declare filtercomap-principal [code]
declare eventually-principal [code]
declare inf-principal [code]
declare sup-principal [code]
declare principal-le-iff [code]

lemma Rep-filter-iff-eventually [simp, code]:
  Rep-filter F P ⟷ eventually P F
  by (simp add: eventually-def)

lemma bot-eq-principal-empty [code]:
  bot = principal {}
  by simp

lemma top-eq-principal-UNIV [code]:
  top = principal UNIV
  by simp

instantiate filter :: (equal) equal
begin
  definition equal-filter :: 'a filter ⇒ 'a filter ⇒ bool
    where equal-filter F F' ⟷ F = F'

  lemma equal-filter [code]:
    HOL.equal (principal A) (principal B) ⟷ A = B
    by (simp add: equal-filter-def)

  instance
    by standard (simp add: equal-filter-def)
end

end

91 Conditionally-complete Lattices

theory Conditionally-Complete-Lattices
imports Finite-Set Lattices-Big Set-Interval
begin
locale preordering-bdd = preordering
begin
  definition bdd :: 'a set ⇒ bool
    where unfold: bdd A ⟷ (∃ M. ∀ x ∈ A. x ≤ M)

  lemma empty [simp, intro]:
    bdd {}
    by (simp add: unfold)

  lemma I [intro]:

THEORY “Conditionally-Complete-Lattices”

\[ \langle \text{bdd } A \rangle \text{ if } \bigwedge x. x \in A \implies x \leq M \]

\text{using that by (auto simp add: unfold)}

\text{lemma } E:
\text{assumes } \langle \text{bdd } A \rangle
\text{obtains } M \text{ where } \bigwedge x. x \in A \implies x \leq M
\text{using assms that by (auto simp add: unfold)}

\text{lemma } I2:
\langle \text{bdd } (f : A) \rangle \text{ if } \bigwedge x. x \in A \implies f x \leq M
\text{using that by (auto simp add: unfold)}

\text{lemma } mono:
\langle \text{bdd } A \rangle \text{ if } \langle \text{bdd } B \rangle \langle A \subseteq B \rangle
\text{using that by (auto simp add: unfold)}

\text{lemma } Int1 [simp]:
\langle \text{bdd } (A \cap B) \rangle \text{ if } \langle \text{bdd } A \rangle
\text{using mono that by auto}

\text{lemma } Int2 [simp]:
\langle \text{bdd } (A \cap B) \rangle \text{ if } \langle \text{bdd } B \rangle
\text{using mono that by auto}

\text{end}

91.1 Preorders

c\text{ontext preorder}
\text{begin}

\text{sublocale } bdd-above: preordering-bdd \langle (\leq) \rangle \langle (\lt) \rangle
\text{defines } bdd-above-primitive-def: bdd-above = bdd-above.bdd ..

\text{sublocale } bdd-below: preordering-bdd \langle (\geq) \rangle \langle (\gt) \rangle
\text{defines } bdd-below-primitive-def: bdd-below = bdd-below.bdd ..

\text{lemma } bdd-above-def: \langle bdd-above A \leftrightarrow (\exists M. \forall x \in A. x \leq M) \rangle
\text{by (fact bdd-above.unfold)}

\text{lemma } bdd-below-def: \langle bdd-below A \leftrightarrow (\exists M. \forall x \in A. M \leq x) \rangle
\text{by (fact bdd-below.unfold)}

\text{lemma } bdd-aboveI: (\bigwedge x. x \in A \implies x \leq M) \implies bdd-above A
\text{by (fact bdd-above.I)}

\text{lemma } bdd-belowI: (\bigwedge x. x \in A \implies m \leq x) \implies bdd-below A
\text{by (fact bdd-below.I)}
THEORY "Conditionally-Complete-Lattices"

lemma bdd-aboveI2: \((\forall x. x \in A \Rightarrow f x \leq M) \Rightarrow bdd-above (f' A)\)
by (fact bdd-above.I2)

lemma bdd-belowI2: \((\forall x. x \in A \Rightarrow m \leq f x) \Rightarrow bdd-below (f' A)\)
by (fact bdd-below.I2)

lemma bdd-above-empty: bdd-above {}
by (fact bdd-above.empty)

lemma bdd-below-empty: bdd-below {}
by (fact bdd-below.empty)

lemma bdd-above-mono: bdd-above B \Rightarrow A \subseteq B \Rightarrow bdd-above A
by (fact bdd-above.mono)

lemma bdd-below-mono: bdd-below B \Rightarrow A \subseteq B \Rightarrow bdd-below A
by (fact bdd-below.mono)

lemma bdd-above-Int1: bdd-above A \Rightarrow bdd-above (A \cap B)
by (fact bdd-above.Int1)

lemma bdd-above-Int2: bdd-above B \Rightarrow bdd-above (A \cap B)
by (fact bdd-above.Int2)

lemma bdd-below-Int1: bdd-below A \Rightarrow bdd-below (A \cap B)
by (fact bdd-below.Int1)

lemma bdd-below-Int2: bdd-below B \Rightarrow bdd-below (A \cap B)
by (fact bdd-below.Int2)

lemma bdd-above-Ioo [simp, intro]: bdd-above \{ a..<b \}
by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Ico [simp, intro]: bdd-above \{ a..b \}
by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Iio [simp, intro]: bdd-above \{ ..b \}
by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Iic [simp, intro]: bdd-above \{ a..b \}
by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Iocc [simp, intro]: bdd-above \{ a..<b \}
by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Ioc [simp, intro]: bdd-above \{ a..<b \}
by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Ico [simp, intro]: bdd-above \{ a..b \}
by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Icc [simp, intro]: bdd-above \{ ..b \}
by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Iic [simp, intro]: bdd-above \{ a..b \}
by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Ioo [simp, intro]: bdd-above \{ a..<b \}
THEORY “Conditionally-Complete-Lattices”

by (auto simp add: bdd-below-def intro: exI[of - a] less-imp-le)

lemma bdd-below-Ioc [simp, intro]: bdd-below {a <..< b}
  by (auto simp add: bdd-below-def intro: exI[of - a] less-imp-le)

lemma bdd-below-Ioi [simp, intro]: bdd-below {a <..}
  by (auto simp add: bdd-below-def intro: exI[of - a] less-imp-le)

lemma bdd-below-Ico [simp, intro]: bdd-below {a ..< b}
  by (auto simp add: bdd-below-def intro: exI[of - a] less-imp-le)

lemma bdd-below-Icc [simp, intro]: bdd-below {a .. b}
  by (auto simp add: bdd-below-def intro: exI[of - a] less-imp-le)

end

context order-top
begin

lemma bdd-above-top [simp, intro!]: bdd-above A
  by (rule bdd-aboveI[of - top]) simp

end

context order-bot
begin

lemma bdd-below-bot [simp, intro!]: bdd-below A
  by (rule bdd-belowI[of - bot]) simp

end

lemma bdd-above-image-mono: mono f \Rightarrow bdd-above A \Rightarrow bdd-above (f:A)
  by (auto simp: bdd-above-def mono-def)

lemma bdd-below-image-mono: mono f \Rightarrow bdd-below A \Rightarrow bdd-below (f:A)
  by (auto simp: bdd-below-def mono-def)

lemma bdd-above-image-antimono: antimono f \Rightarrow bdd-below A \Rightarrow bdd-above (f:A)
  by (auto simp: bdd-above-def bdd-below-def antimono-def)

lemma bdd-below-image-antimono: antimono f \Rightarrow bdd-above A \Rightarrow bdd-below (f:A)
  by (auto simp: bdd-above-def bdd-below-def antimono-def)
THEORY "Conditionally-Complete-Lattices"

lemma fixes X :: 'a::ordered-ab-group-add set
  shows bdd-above-uminus[simp]: bdd-above (uminus ' X) ←→ bdd-below X
  and bdd-below-uminus[simp]: bdd-below (uminus ' X) ←→ bdd-above X
using bdd-above-image-antimono[of uminus X] bdd-below-image-antimono[of uminus 'X]
  using bdd-below-image-antimono[of uminus X] bdd-above-image-antimono[of uminus 'X]
  by (auto simp: antimono-def image-image)

91.2 Lattices

context lattice begin

lemma bdd-above-insert [simp]: bdd-above (insert a A) = bdd-above A
  by (auto simp: bdd-above-def intro: le-supI2 sup-ge1)

lemma bdd-below-insert [simp]: bdd-below (insert a A) = bdd-below A
  by (auto simp: bdd-below-def intro: le-infI2 inf-le1)

lemma bdd-finite [simp]:
  assumes finite A shows bdd-above-finite: bdd-above A and bdd-below-finite: bdd-below A
  using assms by (induct rule: finite-induct, auto)

lemma bdd-above-Un [simp]: bdd-above (A ∪ B) = (bdd-above A ∧ bdd-above B)
proof
  assume bdd-above (A ∪ B)
  thus bdd-above A ∧ bdd-above B unfolding bdd-above-def by auto
next
  assume bdd-above A ∧ bdd-above B
  then obtain a b where ∀x∈A. x ≤ a ∀x∈B. x ≤ b unfolding bdd-above-def by auto
  hence ∀x ∈ A ∪ B. x ≤ sup a b by (auto intro: Un-iff le-supI1 le-supI2)
  thus bdd-above (A ∪ B) unfolding bdd-above-def ..
qed

lemma bdd-below-Un [simp]: bdd-below (A ∪ B) = (bdd-below A ∧ bdd-below B)
proof
  assume bdd-below (A ∪ B)
  thus bdd-below A ∧ bdd-below B unfolding bdd-below-def by auto
next
  assume bdd-below A ∧ bdd-below B
  then obtain a b where ∀x∈A. a ≤ x ∀x∈B. b ≤ x unfolding bdd-below-def by auto
  hence ∀x ∈ A ∪ B. inf a b ≤ x by (auto intro: Un-iff le-infI1 le-infI2)
  thus bdd-below (A ∪ B) unfolding bdd-below-def ..
qed
lemma `bdd-above-image-sup` [simp]:
`bdd-above (λ x. sup (f x) (g x)) `A` ←→ bdd-above (f `A`) ∧ bdd-above (g `A`)` by (auto simp: bdd-above-def intro: le-supI1 le-supI2)

lemma `bdd-below-image-inf` [simp]:
`bdd-below (λ x. inf (f x) (g x)) `A` ←→ bdd-below (f `A`) ∧ bdd-below (g `A`)` by (auto simp: bdd-below-def intro: le-infI1 le-infI2)

lemma `bdd-below-UN` [simp]:
`finite I =⇒ bdd-below (∪ i ∈ I. A i) = (∀ i ∈ I. bdd-below (A i))` by (induction I rule: finite induct) auto

lemma `bdd-above-UN` [simp]:
`finite I =⇒ bdd-above (∪ i ∈ I. A i) = (∀ i ∈ I. bdd-above (A i))` by (induction I rule: finite induct) auto

end

To avoid name classes with the `complete-lattice`-class we prefix `Sup` and `Inf` in theorem names with `c`.

### 91.3 Conditionally complete lattices

class `conditionally-complete-lattice` = `lattice` + `Sup` + `Inf` +
assumes `cInf-lower`: `x ∈ X =⇒ bdd-below X =⇒ Inf X ≤ x`
and `cInf-greatest`: `X ≠ {} =⇒ (∀ x ∈ X =⇒ z ≤ x) =⇒ z ≤ Inf X`
assumes `cSup-upper`: `x ∈ X =⇒ bdd-above X =⇒ x ≤ Sup X`
and `cSup-least`: `X ≠ {} =⇒ (∀ x ∈ X =⇒ x ≤ z) =⇒ Sup X ≤ z`

begin

lemma `cSup-upper2`: `x ∈ X =⇒ y ≤ x =⇒ bdd-above X =⇒ y ≤ Sup X` by (metis `cSup-upper` order-trans)

lemma `cInf-lower2`: `x ∈ X =⇒ x ≤ y =⇒ bdd-below X =⇒ Inf X ≤ y` by (metis `cInf-lower` order-trans)

lemma `cSup-mono`: `B ≠ {} =⇒ bdd-above A =⇒ (∀ b ∈ B =⇒ ∃ a ∈ A. b ≤ a) =⇒ Sup B ≤ Sup A`
by (metis `cSup-least` `cSup-upper2`)

lemma `cInf-mono`: `B ≠ {} =⇒ bdd-below A =⇒ (∀ b ∈ B =⇒ ∃ a ∈ A. a ≤ b) =⇒ Inf A ≤ Inf B`
by (metis `cInf-greatest` `cInf-lower2`)

lemma `cSup-subset-mono`: `A ≠ {} =⇒ bdd-above B =⇒ A ⊆ B =⇒ Sup A ≤ Sup B`
by (metis `cSup-least` `cSup-upper` subsetD)
THEORY “Conditionally-Complete-Lattices”

lemma cInf-superset-mono: \(A \neq \{\} \implies \text{bdd-below } B \implies A \subseteq B \implies \text{Inf } B \leq \text{Inf } A\)
by (metis cInf-greatest cInf-lower subsetD)

lemma cSup-eq-maximum: \(z \in X \implies (\forall x \in X \implies x \leq z) \implies \text{Sup } X = z\)
by (intro order.antisym cSup-upper \{of z X\} cSup-least \{of X z\}) auto

lemma cInf-eq-minimum: \(z \in X \implies (\forall x \in X \implies z \leq x) \implies \text{Inf } X = z\)
by (intro order.antisym cInf-lower \{of z X\} cInf-greatest \{of X z\}) auto

lemma cSup-le-iff: \(S \neq \{\} \implies \text{bdd-above } S \implies \text{Sup } S \leq a \iff (\forall x \in S. x \leq a)\)
by (metis order-trans cSup-upper cSup-least)

lemma cInf-le-iff: \(S \neq \{\} \implies \text{bdd-below } S \implies a \leq \text{Inf } S \iff (\forall x \in S. a \leq x)\)
by (metis order-trans cInf-lower cInf-greatest)

lemma cSup-eq-non-empty:
assumes 1: \(X \neq \{\}\)
assumes 2: \(\forall x. x \in X \implies x \leq a\)
assumes 3: \(\forall y. (\forall x. x \in X \implies x \leq y) \implies a \leq y\)
shows \(\text{Sup } X = a\)
by (intro 3 1 order.antisym cSup-least) (auto intro: 2 1 cSup-upper)

lemma cInf-eq-non-empty:
assumes 1: \(X \neq \{\}\)
assumes 2: \(\forall x. x \in X \implies a \leq x\)
assumes 3: \(\forall y. (\forall x. x \in X \implies y \leq x) \implies y \leq a\)
shows \(\text{Inf } X = a\)
by (intro 3 1 order.antisym cInf-greatest) (auto intro: 2 1 cInf-lower)

lemma cInf-cSup: \(S \neq \{\} \implies \text{bdd-below } S \implies \text{Inf } S = \text{Sup } \{x. \forall s \in S. x \leq s\}\)
by (rule cInf-eq-non-empty) (auto intro!: cSup-upper cSup-least simp: bdd-below-def)

lemma cSup-cInf: \(S \neq \{\} \implies \text{bdd-above } S \implies \text{Sup } S = \text{Inf } \{x. \forall s \in S. s \leq x\}\)
by (rule cSup-eq-non-empty) (auto intro!: cInf-lower cInf-greatest simp: bdd-above-def)

lemma cSup-insert: \(X \neq \{\} \implies \text{bdd-above } X \implies \text{Sup } (\text{insert } a X) = \text{sup } a (\text{Sup } X)\)
by (intro cSup-eq-non-empty) (auto intro: le-supI2 cSup-upper cSup-least)

lemma cInf-insert: \(X \neq \{\} \implies \text{bdd-below } X \implies \text{Inf } (\text{insert } a X) = \text{inf } a (\text{Inf } X)\)
by (intro cInf-eq-non-empty) (auto intro: le-infI2 cInf-lower cInf-greatest)

lemma cSup-singleton [simp]: \(\text{Sup } \{x\} = x\)
by (intro cSup-eq-maximum) auto

lemma cInf-singleton [simp]: \(\text{Inf } \{x\} = x\)
by (intro cInf-eq-minimum) auto
lemma cSup-insert-If: bdd-above X \implies \Sup (\insert a X) = (if X = {} then a else sup a (\Sup X))
  using cSup-insert[of X] by simp

lemma cInf-insert-If: bdd-below X \implies \Inf (\insert a X) = (if X = {} then a else inf a (\Inf X))
  using cInf-insert[of X] by simp

lemma le-cSup-finite: finite X \implies x \in X \implies x \leq \Sup X
proof (induct X arbitrary: x rule: finite-induct)
  case (insert x X y) then show ?case
  by (cases X = {}) (auto simp: cSup-insert intro: le-supI2)
qed simp

lemma cInf-le-finite: finite X \implies x \in X \implies \Inf X \leq x
proof (induct X arbitrary: x rule: finite-induct)
  case (insert x X y) then show ?case
  by (cases X = {}) (auto simp: cInf-insert intro: le-infI2)
qed simp

lemma cSup-eq-Sup-finite: finite X \implies X \neq {} \implies \Sup X = \Sup-finite X
by (induct X rule: finite-ne-induct) (simp-all add: cSup-insert)

lemma cInf-eq-Inf-finite: finite X \implies X \neq {} \implies \Inf X = \Inf-finite X
by (induct X rule: finite-ne-induct) (simp-all add: cInf-insert)

lemma cSup-atMost[simp]: \Sup \{..x\} = x
by (auto intro!: cSup-eq-maximum)

lemma cSup-greaterThanAtMost[simp]: y < x \implies \Sup \{y..<\} = x
by (auto intro!: cSup-eq-maximum)

lemma cSup-atLeastAtMost[simp]: y \leq x \implies \Sup \{y..x\} = x
by (auto intro!: cSup-eq-maximum)

lemma cInf-atLeast[simp]: \Inf \{..\} = x
by (auto intro!: cInf-eq-minimum)

lemma cInf-atLeastLessThan[simp]: y < x \implies \Inf \{y..<\} = y
by (auto intro!: cInf-eq-minimum)

lemma cInf-atLeastAtMost[simp]: y \leq x \implies \Inf \{y..x\} = y
by (auto intro!: cInf-eq-minimum)

lemma cINF-lower: bdd-below (\fun A) \implies x \in A \implies \bigcap (\fun A) \leq f x
using cINF-lower[of - \fun A] by simp

lemma cINF-greatest: A \neq {} \implies (\forall x. x \in A \implies m \leq f x) \implies m \leq \bigcap (\fun A)
using cINF-greatest[of \fun A] by auto
lemma cSUP-upper: $x \in A \implies \text{bdd-above } (f' A) \implies f x \leq \bigsqcup (f' A)$

using cSup-upper [of $f' A$] by simp

lemma cSUP-least: $A \neq \{\} \implies (\forall x. x \in A \implies f x \leq M) \implies \bigsqcup (f' A) \leq M$

using cSup-le-iff [of $f' A$] by auto

lemma cINF-lower2: $\text{bdd-below } (f' A) \implies x \in A \implies f x \leq u \implies \bigcap (f' A) \leq u$

by (auto intro: cINF-lower order-trans)

lemma cSUP-upper2: $\text{bdd-above } (f' A) \implies x \in A \implies u \leq f x \implies u \leq \bigsqcup (f' A)$

by (auto intro: cSUP-upper order-trans)

lemma cSUP-const [simp]: $A \neq \{\} \implies (\bigcup x \in A. c) = c$

by (intro order.antisym cSUP-least) (auto intro: cSUP-upper)

lemma cINF-lower [simp]: $A \neq \{\} \implies (\bigcap x \in A. c) = c$

by (intro order.antisym cINF-greatest) (auto intro: cINF-lower)

lemma le-cINF-iff: $A \neq \{\} \implies \text{bdd-below } (f' A) \implies u \leq \bigcap (f' A) \iff (\forall x \in A. u \leq f x)$

by (metis cINF-greatest cINF-lower order-trans)

lemma cSUP-le-iff: $A \neq \{\} \implies \text{bdd-above } (f' A) \implies \bigcup (f' A) \leq u \iff (\forall x \in A. f x \leq u)$

by (metis cINF-greatest cINF-lower order-trans)

lemma less-cINF-D: $\text{bdd-below } (f' A) \implies y < (\bigcap i \in A. f i) \implies i \in A \implies y < f i$

by (metis cINF-lower less-le-trans)

lemma cSUP-lessD: $\text{bdd-above } (f' A) \implies (\bigcup i \in A. f i) < y \implies i \in A \implies f i < y$

by (metis cSUP-upper le-less-trans)

lemma cINF-insert: $A \neq \{\} \implies \text{bdd-below } (f' A) \implies \bigcap (f' \text{ insert } a A) = \inf (f\ a) \ (\bigcap (f' A))$

by (simp add: cInf-insert)

lemma cSUP-insert: $A \neq \{\} \implies \text{bdd-above } (f' A) \implies \bigcup (f' \text{ insert } a A) = \sup (f\ a) \ (\bigcup (f' A))$

by (simp add: cSup-insert)

lemma cINF-mono: $B \neq \{\} \implies \text{bdd-below } (f' A) \implies (\forall m. m \in B \implies \exists n \in A. f n \leq g m) \implies \bigcap (f' A) \leq \bigcap (g' B)$

using cInf-mono [of $g' B$ $f' A$] by auto

lemma cSUP-mono: $A \neq \{\} \implies \text{bdd-above } (g' B) \implies (\forall n. n \in A \implies \exists m \in B. f n \leq g m) \implies \bigcup (f' A) \leq \bigcup (g' B)$

using cSup-mono [of $f' A$ $g' B$] by auto
lemma cINF-superset-mono: $A \neq \{\} \Rightarrow bdd\text{-}below (g \cdot B) \Rightarrow A \subseteq B \Rightarrow (\bigwedge x. x \in B \Rightarrow g x \leq f x) \Rightarrow \bigcap(g \cdot B) \leq \bigcap(f \cdot A)$
  by (rule cINF-mono) auto

lemma cSUP-subset-mono:
  \[ [A \neq \{\}; bdd\text{-}above (g \cdot B); A \subseteq B; \bigwedge x. x \in A \Rightarrow f x \leq g x] \Rightarrow \bigcup (f \cdot A) \leq \bigcup (g \cdot B) \]
  by (rule cSUP-mono) auto

lemma less-eq-cInf-inter: bdd\text{-}below $A \Rightarrow bdd\text{-}below B \Rightarrow A \cap B \neq \{\} \Rightarrow \inf \ (\inf A) (\inf B) \leq \inf (A \cap B)$
  by (metis cINF-superset-mono lattice-class.inf-ord(1) le-infI1)

lemma cSup-inter-less-eq: bdd\text{-}above $A \Rightarrow bdd\text{-}above B \Rightarrow A \cap B \neq \{\} \Rightarrow \sup (A \cap B) \leq \sup (\sup A) (\sup B)$
  by (metis cSUP-subset-mono lattice-class.inf-ord(1) le-supI1)

lemma cInf-union-distrib: $A \neq \{\} \Rightarrow bdd\text{-}below A \Rightarrow B \neq \{\} \Rightarrow bdd\text{-}below B \Rightarrow \inf (A \cup B) = \inf (\inf A) (\inf B)$
  by (intro order.antisym le-infI cInf-greatest cInf-lower) (auto intro: le-infI1 le-infI2 cInf-lower)

lemma cINF-union: $A \neq \{\} \Rightarrow bdd\text{-}below (f \cdot A) \Rightarrow B \neq \{\} \Rightarrow bdd\text{-}below (f \cdot B) \Rightarrow \bigcap(f \cdot (A \cup B)) = \bigcap(f \cdot A) \cap \bigcap(f \cdot B)$
  using cinf-union-distrib [of f \cdot A f \cdot B] by (simp add: image-Un)

lemma cSup-union-distrib: $A \neq \{\} \Rightarrow bdd\text{-}above A \Rightarrow B \neq \{\} \Rightarrow bdd\text{-}above B \Rightarrow Sup (A \cup B) = sup (Sup A) (Sup B)$
  by (intro order.antisym le-supI1 cSup-least cSup-upper) (auto intro: le-supI1 le-supI2 cSup-upper)

lemma cSUP-union: $A \neq \{\} \Rightarrow bdd\text{-}above (f \cdot A) \Rightarrow B \neq \{\} \Rightarrow bdd\text{-}above (f \cdot B) \Rightarrow \bigcup(f \cdot (A \cup B)) = \bigcup(f \cdot A) \cup \bigcup(f \cdot B)$
  using cSup-union-distrib [of f \cdot A f \cdot B] by (simp add: image-Un)

lemma cINF-inf-distrib: $A \neq \{\} \Rightarrow bdd\text{-}below (f \cdot A) \Rightarrow bdd\text{-}below (g \cdot A) \Rightarrow \bigcap(f \cdot A) \cap \bigcap(g \cdot A) = (\bigcap f \cdot a \cdot A = (\bigcap f \cdot a \cdot A) (g a))$
  by (intro order.antisym le-infI cINF-greatest cINF-lower2)
  (auto intro: le-infI1 le-infI2 cINF-greatest cINF-lower le-infI)

lemma SUP-sup-distrib: $A \neq \{\} \Rightarrow bdd\text{-}above (f \cdot A) \Rightarrow bdd\text{-}above (g \cdot A) \Rightarrow \bigcup(f \cdot A) \cup \bigcup(g \cdot A) = (\bigcup a \cdot A \cdot sup (f a \cdot a) (g a))$
  by (intro order.antisym le-supI1 cSUP-least cSUP-upper2)
  (auto intro: le-supI1 le-supI2 cSUP-least cSUP-upper le-supI)

lemma cInf-le-cSup:
  $A \neq \{\} \Rightarrow bdd\text{-}above A \Rightarrow bdd\text{-}below A \Rightarrow \inf A \leq \sup A$
  by (auto intro!: cSup-upper2[of SOME a. a \in A] intro: someI cInf-lower)
context
  fixes f :: 'a ⇒ 'b::conditionally-complete-lattice
  assumes mono f
begin

lemma mono-cInf: \[ \[ \text{bdd-below } A ; A \neq \{\} \] \] =⇒ \( f (\text{Inf } A) \leq (\text{INF } x \in A. f x) \)
  by (simp add: 'mono f' conditionally-complete-lattice-class.cINF-greatest cInf-lower monoD)

lemma mono-cSup: \[ \[ \text{bdd-above } A ; A \neq \{\} \] \] =⇒ \( (\text{SUP } x \in A. f x) \leq f (\text{Sup } A) \)
  by (simp add: 'mono f' conditionally-complete-lattice-class.cSUP-least cSup-upper monoD)

lemma mono-cINF: \[ \[ \text{bdd-below } (A' \cdot I); I \neq \{\} \] \] =⇒ \( f (\text{INF } i \in I. A i) \leq (\text{INF } x \in I. f (A x)) \)
  by (simp add: 'mono f' conditionally-complete-lattice-class.cINF-greatest cINF-lower monoD)

lemma mono-cSUP: \[ \[ \text{bdd-above } (A' \cdot I); I \neq \{\} \] \] =⇒ \( (\text{SUP } x \in I. f (A x)) \leq f (\text{SUP } i \in I. A i) \)
  by (simp add: 'mono f' conditionally-complete-lattice-class.cSUP-least cSUP-upper monoD)

end

end

The special case of well-orderings

lemma wellorder-InfI:
  fixes k :: 'a::{wellorder,conditionally-complete-lattice}
  assumes k ∈ A shows Inf A ∈ A
  using wellorder-class.LeastI \[ \[ \lambda x. x \in A \rightarrow k \] \]
  by (simp add: Least-le assms cInf-eq-minimum)

lemma wellorder-Inf-le1:
  fixes k :: 'a::{wellorder,conditionally-complete-lattice}
  assumes k ∈ A shows Inf A ≤ k
  by (meson Least-le assms bdd-below.I cInf-lower)

91.4 Complete lattices

instance complete-lattice ⊆ conditionally-complete-lattice
  by standard (auto intro: Sup-upper Sup-least Inf-lower Inf-greatest)

lemma cSup-eq:
  fixes a :: 'a :: {conditionally-complete-lattice, no-bot}
  assumes upper: \( \text{\( \forall x. \ x \in X \implies x \leq a \) } \)
  assumes least: \( \text{\( \forall y. \ (\forall x. \ x \in X \implies x \leq y) \implies a \leq y \) } \)
  shows Sup X = a
proof cases
  assume \( X = \{ \} \) with \( \ll-ex[\text{of } a] \) least show \( ?\text{thesis} \) by (auto simp: less-le-not-le)
qed (intro cSup-eq-non-empty assms)

lemma cSup-unique:
  fixes \( b :: 'a :: \{ \text{conditionally-complete-lattice, no-bot} \} \)
  assumes \( \forall c. (\forall x \in s. x \leq c) \iff b \leq c \)
  shows \( Sup s = b \) by (meson assms cSup-eq order_refl)

lemma cInf-eq:
  fixes \( a :: 'a :: \{ \text{conditionally-complete-lattice, no-top} \} \)
  assumes upper: \( \forall x. x \in X = \Rightarrow a \leq x \)
  assumes least: \( \forall y. (\forall x \in X \Rightarrow y \leq x) \Rightarrow y \leq a \)
  shows \( Inf X = a \) proof cases
    assume \( X = \{ \} \) with \( gt-ex[\text{of } a] \) least show \( ?\text{thesis} \) by (auto simp: less-le-not-le)
  qed (intro cInf-eq-non-empty assms)

lemma cInf-unique:
  fixes \( b :: 'a :: \{ \text{conditionally-complete-lattice, no-top} \} \)
  assumes \( \forall c. (\forall x \in s. x \geq c) \iff b \geq c \)
  shows \( Inf s = b \) by (meson assms cInf-eq order_refl)

class conditionally-complete-linorder = conditionally-complete-lattice + linorder

begin

lemma less-cSup-iff:
  \( X \neq \{ \} \Rightarrow \text{bdd-above } X \Rightarrow y < Sup X \iff (\exists x \in X. y < x) \)
  by (rule iffI) (metis cSup-least not-le assms that)

lemma cInf-less-iff: \( X \neq \{ \} \Rightarrow \text{bdd-below } X \Rightarrow Inf X < y \iff (\exists x \in X. x < y) \)
  by (rule iffI) (metis cInf-greatest not-less assms least)

lemma cINF-less-iff: \( A \neq \{ \} \Rightarrow \text{bdd-below } (f'\cdot A) \Rightarrow (\prod i \in A. f i) < a \iff (\exists x \in A. f x < a) \)
  using cInf-less-iff[of f'\cdot A] by auto

lemma less-cSUP-iff: \( A \neq \{ \} \Rightarrow \text{bdd-above } (f'\cdot A) \Rightarrow a < (\bigcup i \in A. f i) \iff (\exists x \in A. a < f x) \)
  using less-cSup-iff[of f'\cdot A] by auto

lemma less-cSupE:
  assumes \( y < Sup X X \neq \{ \} \) obtains \( x \) where \( x \in X y < x \)
  by (metis cSup-least assms not-le that)

lemma less-cSupD:
  \( X \neq \{ \} \Rightarrow z < Sup X \Rightarrow \exists x \in X. z < x \)
THEORY "Conditionally-Complete-Lattices"

lemma cInf-lessD:
  \( X \neq \{ \} \implies \inf X < z \implies \exists x \in X. x < z \)
by (metis cInf-less-iff not-le-imp-less bdd-below-def)

lemma complete-interval:
  assumes a < b and \( P a \) and \( \neg P b \)
  shows \( \exists c. a \leq c \land c < d \implies P c \) \land
  \( \forall d. (\forall x. a \leq x \land x < d \implies P x) \implies d \leq c \)
proof (rule exI [where x = Sup \( \{ d. \forall x. a \leq x \land x < d \implies P x \} \)], safe)
  show a \leq Sup \( \{ d. \forall c. a \leq c \land c < d \implies P c \} \)
    by (rule cSup-upper auto simp add: bdd-above-def)
  (metis \( a < b \) \( \neg P b \) linear less-le)
next
  show Sup \( \{ d. \forall c. a \leq c \land c < d \implies P c \} \) \leq b
    by (rule cSup-least)
    (use \( a < b \) \( \neg P b \) in auto simp add: less-le-not-le)
next
  fix x
  assume x: a \leq x and lt: x < Sup \( \{ d. \forall c. a \leq c \land c < d \implies P c \} \)
  show P x
    by (rule less-cSupE [OF lt]) (use less-le-not-le x in auto)
next
  fix d
  assume 0: \( \forall x. a \leq x \land x < d \implies P x \)
  then have d \in \( \{ d. \forall c. a \leq c \land c < d \implies P c \} \)
    by auto
  moreover have bdd-above \( \{ d. \forall c. a \leq c \land c < d \implies P c \} \)
    unfolding bdd-above-def using \( a < b \) \( \neg P b \) linear
    by (simp add: less-le) blast
  ultimately show d \leq Sup \( \{ d. \forall c. a \leq c \land c < d \implies P c \} \)
    by (auto simp: cSup-upper)
qed

91.5 Instances

instance complete-linorder < conditionally-complete-linorder ..

lemma cSup-eq-Max: finite \( X::'a::conditionally-complete-linorder\) set \( \implies X \neq \{ \} \implies \sup X = \max X \)
  using cSup-eq-Sup-fin[of X] by (simp add: Sup-fin-Max)

lemma cInf-eq-Min: finite \( X::'a::conditionally-complete-linorder\) set \( \implies X \neq \{ \} \implies \inf X = \min X \)
  using cInf-eq-Inf-fin[of X] by (simp add: Inf-fin-Min)
lemma \text{cSup-lessThan} [simp]: \( \text{Sup} \{ \ldots \text{x} :: \langle a::\{\text{conditionally-complete-linorder, no-bot, dense-linorder}\}\} = \text{x} \)
  by (auto intro!: \text{cSup-eq-non-empty} intro: dense-le)

lemma \text{cSup-greaterThanLessThan} [simp]: \( y < \text{x} \implies \text{Sup} \{ y..\text{x} :: \langle a::\{\text{conditionally-complete-linorder, dense-linorder}\}\} = \text{x} \)
  by (auto intro!: \text{cSup-eq-non-empty} intro: dense-le-bounded)

lemma \text{cSup-atLeastLessThan} [simp]: \( y < \text{x} \implies \text{Sup} \{ y..\langle x :: \langle a::\{\text{conditionally-complete-linorder, dense-linorder}\}\}} = \text{x} \)
  by (auto intro!: \text{cSup-eq-non-empty} intro: dense-le-bounded)

lemma \text{cInf-greaterThan} [simp]: \( \text{Inf} \{ x :: \langle a::\{\text{conditionally-complete-linorder, no-top, dense-linorder}\}\} < \ldots \} = \text{x} \)
  by (auto intro!: \text{cInf-eq-non-empty} intro: dense-ge)

lemma \text{cInf-greaterThanAtMost} [simp]: \( y < \text{x} \implies \text{Inf} \{ y..<x :: \langle a::\{\text{conditionally-complete-linorder, dense-linorder}\}\} = \text{y} \)
  by (auto intro!: \text{cInf-eq-non-empty} intro: dense-ge-bounded)

lemma \text{cInf-greaterThanLessThan} [simp]: \( y < \text{x} \implies \text{Inf} \{ y..<\langle x :: \langle a::\{\text{conditionally-complete-linorder, dense-linorder}\}\}} = \text{y} \)
  by (auto intro!: \text{cInf-eq-non-empty} intro: dense-ge-bounded)

lemma \text{Sup-inverse-eq-inverse-Inf}:
  \[ \text{fixes} \ f :: \langle b \Rightarrow \langle a::\{\text{conditionally-complete-linorder, linordered-field}\}\} \]
  \[ \text{assumes} \ \text{bdd-above} \ (\text{range} \ f) \ L > 0 \ \text{and} \ \text{geL}: \forall \text{x}. \ f \text{x} \geq L \]
  \[ \text{shows} \ (\text{SUP} \ \text{x}. \ 1 / f \text{x}) = 1 / (\text{INF} \ \text{x}. f \text{x}) \]

proof (rule antisym)
  have bdd-f: bdd-below (range f)
    by (meson assms bdd-belowI2)
  have Inf (range f) \geq L
    by (simp add: cINF-greatest geL)
  have bdd-invF: bdd-above (range (\(\text{\lambda} \text{x}. \ 1 / f \text{x}\}))
    (rule bdd-aboveI2)
  show \(\forall \text{x}. \ 1 / f \text{x} \leq 1/L \)
    using assms by (auto simp: divide-simps)
qed

moreover have le-inverse-Inf: \( 1 / f \text{x} \leq 1 / (\text{INF} \ \text{range} \ f) \) \text{ for } \text{x}
  proof =
    have Inf (range f) \leq f \text{x}
      by (simp add: bdd-f cINF-lower)
    then show \(\forall \text{thesis} \)
      using assms \(\langle \text{L} \leq \text{Inf} \ (\text{range} \ f) \rangle \)
        by (auto simp: divide-simps)
  qed

ultimately show \(\ast: (\text{SUP} \ \text{x}. \ 1 / f \text{x}) \leq 1 / (\text{INF} \ (\text{range} \ f)) \)
  by (auto simp: cSup-le-iff cINF-lower)

have \(1 / (\text{SUP} \ \text{x}. \ 1 / f \text{x}) \leq f \text{y} \) \text{ for } \text{y}
proof (cases (SUP x. 1 / f x) < 0)
  case True
    with assms show ?thesis
      by (meson less-asym’ order-trans linorder-not-le zero-le-divide-1-iff)
next
  case False
    have 1 / f y ≤ (SUP x. 1 / f x)
      by (simp add: bdd-invf cSup-upper)
    with False assms show ?thesis
      by (metis (no-types) div-by-1 divide-divide-eq-right dual-order.strict-trans1 inverse-eq-divide inverse-le-imp-le mult.left-neutral)
qed

then have 1 / (SUP x. 1 / f x) ≤ Inf (range f)
  using bdd-f by (simp add: le-cInf-iff)
moreover have (SUP x. 1 / f x) > 0
  using assms cSUP-upper [OF - bdd-invf] by (meson UNIV-I less-le-trans zero-less-divide-1-iff)
ultimately show 1 / Inf (range f) ≤ (SUP t. 1 / f t)
  using L ≤ Inf (range f) [L>0] by (auto simp: field-simps)
qed

lemma Inf-inverse-eq-inverse-Sup:
  fixes f :: 'b ⇒ 'a::(conditionally-complete-linorder,linordered-field)
  assumes bdd-above (range f) L > 0 and geL: ∃x. f x ≥ L
  shows (INF x. 1 / f x) = 1 / (SUP x. f x)
proof -
  obtain M where M>0 and M: ∃x. f x ≤ M
    by (meson assms cSup-upper dual-order.strict-trans1 rangeI)
  have bdd: bdd-above (range (inverse o f))
    using assms le-imp-inverse-le by (auto simp: bdd-above-def)
  have f x > 0 for x
    using (L>0) geL order-less-le-trans by blast
  then have [simp]: 1 / inverse(f x) = f x 1 / M ≤ 1 / f x for x
    using M [M>0] by (auto simp: divide-simps)
  show ?thesis
    using Sup-inverse-eq-inverse-Inf [OF bdd, of inverse M] [M>0]
    by (simp add: inverse-eq-divide)
qed

lemma Inf-insert-finite:
  fixes S :: 'a::conditionally-complete-linorder set
  shows finite S ⇒ Inf (insert x S) = (if S = {} then x else min x (Inf S))
  by (simp add: cInf-eq-Min)

lemma Sup-insert-finite:
  fixes S :: 'a::conditionally-complete-linorder set
  shows finite S ⇒ Sup (insert x S) = (if S = {} then x else max x (Sup S))
  by (simp add: cSup-insert sup-max)
THEORY “Conditionally-Complete-Lattices” 1839

lemma finite-imp-less-Inf:
  fixes a :: 'a::conditionally-complete-linorder
  shows \[ \{ \text{finite } X; x \in X; \forall x. x \in X \Rightarrow a < x \} \Rightarrow a < \text{Inf } X \]
  by (induction X rule: finite-induct) (simp-all add: cInf-eq-Min Inf-insert-finite)

lemma finite-less-Inf-iff:
  fixes a :: 'a::conditionally-complete-linorder
  shows \[ \{ \text{finite } X; X \neq \{\} \} \Rightarrow a < \text{Inf } X \leftrightarrow (\forall x \in X. a < x) \]
  by (auto simp: cInf-eq-Min)

lemma finite-imp-Sup-less:
  fixes a :: 'a::conditionally-complete-linorder
  shows \[ \{ \text{finite } X; x \in X; \forall x. x \in X \Rightarrow a > x \} \Rightarrow a > \text{Sup } X \]
  by (induction X rule: finite-induct) (simp-all add: cSup-eq-Max Sup-insert-finite)

lemma finite-Sup-less-iff:
  fixes a :: 'a::conditionally-complete-linorder
  shows \[ \{ \text{finite } X; X \neq \{\} \} \Rightarrow a > \text{Sup } X \leftrightarrow (\forall x \in X. a > x) \]
  by (auto simp: cSup-eq-Max)

class linear-continuum = conditionally-complete-linorder + dense-linorder +
  assumes UNIV-not-singleton: \exists a b::'a. a \neq b
begin

lemma ex-gt-or-lt: \exists b. a < b \lor b < a
  by (metis UNIV-not-singleton neq-iff)
end

context
  fixes f::'a \Rightarrow 'b::{conditionally-complete-linorder, ordered-ab-group-add}
begin

lemma bdd-above-uminus-image: bdd-above ((\lambda x. - f x) ' A) \leftrightarrow bdd-below (f ' A)
  by (metis bdd-above-uminus image-image)

lemma bdd-below-uminus-image: bdd-below ((\lambda x. - f x) ' A) \leftrightarrow bdd-above (f ' A)
  by (metis bdd-below-uminus image-image)

lemma uminus-cSUP:
  assumes bdd-above (f ' A) A \neq \{\}
  shows - (SUP x\in A. f x) = (INF x\in A. - f x)
proof (rule antisym)
  show (INF x\in A. - f x) \leq - (SUP (f ' A)
    by (metis cINF-lower cSUP-least bdd-below-uminus-image assms le-minus-iff)
have \(*\): \(\forall x. x \in A \implies f x \leq \text{Sup } (f \cdot A)\)
by (simp add: assms cSup-upper)
then show \(-\text{Sup } (f \cdot A) \leq (\text{INF } x \in A. - f x)\)
by (simp add: assms cINF-greatest)
qed

end

context
  fixes \(f::'a \Rightarrow 'b::\{conditionally-complete-linorder, ordered-ab-group-add\}\)
begin

lemma uminus-cINF:
  assumes bdd-below \((f \cdot A)\) \(A \neq \{\}\)
  shows \(- (\text{INF } x \in A. f x) = (\text{SUP } x \in A. - f x)\)
by (metis (mono-tags, lifting) INF-cong uminus-cSUP assms bdd-above-uminus-image minus-equation-iff)

lemma Sup-add-eq:
  assumes bdd-above \((f \cdot A)\) \(A \neq \{\}\)
  shows \((\text{SUP } x \in A. a + f x)\) = \(a + (\text{SUP } x \in A. f x)\) (is \(?L=?R\))
proof (rule antisym)
  have bdd: bdd-above \(((\lambda x. a + f x) \cdot A)\)
  by (simp add: assms bdd-above-image-mono image-image mono-add)
  with assms have \(\forall x. x \in A = \implies f x \leq (SUP x \in A. a + f x) - a\)
  by (simp add: bdd cSup-upper le-diff-eq)
  with \(A \neq \{\}\) have \(\bigcup (f \cdot A) \leq (\bigcup x \in A. a + f x) - a\)
  by (simp add: cSUP-least)
  then show \(?R \leq ?L\)
  by (metis add.commute le-diff-eq)
qed

lemma Inf-add-eq: — you don’t get a shorter proof by duality
  assumes bdd-below \((f \cdot A)\) \(A \neq \{\}\)
  shows \((\text{INF } x \in A. a + f x)\) = \(a + (\text{INF } x \in A. f x)\) (is \(?L=?R\))
proof (rule antisym)
  show \(?R \leq ?L\)
  using assms mono-add mono-cINF by blast
  have bdd: bdd-below \(((\lambda x. a + f x) \cdot A)\)
  by (metis add-left-mono assms(1) bdd-below.E bdd-below.I2 imageI)
  with assms have \(\forall x. x \in A = \implies f x \geq (INF x \in A. a + f x) - a\)
  by (simp add: cInf-lower diff-le-eq)
  with \(A \neq \{\}\) have \(\bigcap x \in A. a + f x) - a \leq \bigcap (f \cdot A)\)
  by (simp add: cINF-greatest)
  with assms show \(?L \leq ?R\)
  by (metis add.commute diff-le-eq)
qed
end

instantiation nat :: conditionally-complete-linorder
begin

definition Sup (X::nat set) = (if X={} then 0 else Max X)
definition Inf (X::nat set) = (LEAST n. n ∈ X)

lemma bdd-above-nat: bdd-above X ⟷ finite (X::nat set)
proof
assume bdd-above X
then obtain z where X ⊆ .. z
  by (auto simp: bdd-above-def)
then show finite X
  by (rule finite-subset simp)
qed simp

instance proof
  fix x :: nat
  fix X :: nat set
  show Inf X ≤ x if x ∈ X bdd-below X
    using that by (simp add: Inf-nat-def Least-le)
  show x ≤ Inf X if X ≠ {} ∧ y. y ∈ X ⟹ x ≤ y
    using that unfolding Inf-nat-def ex-in-conv[ symmetric] by (rule LeastI2-ex)
  show x ≤ Sup X if x ∈ X bdd-above X
    using that by (auto simp add: Sup-nat-def bdd-above-nat)
  show Sup X ≤ x if X ≠ {} ∧ y. y ∈ X ⟹ y ≤ x
    proof
      from that have bdd-above X
        by (auto simp: bdd-above-def)
      with that show ?thesis
        by (simp add: Sup-nat-def bdd-above-nat)
    qed
  qed

end

lemma Inf-nat-def1:
  fixes K :: nat set
  assumes K ≠ {}
  shows Inf K ∈ K
  by (auto simp add: Min-def Inf-nat-def) (meson LeastI assms bot.extremum-unique subsetI)

lemma Sup-nat-empty [simp]: Sup {} = (0::nat)
  by (auto simp add: Sup-nat-def)
THEORY “Conditionally-Complete-Lattices”

instantiation int :: conditionally-complete-linorder
begin

definition Sup (X::int set) = (THE x. x ∈ X ∧ (∀ y∈X. y ≤ x))
definition Inf (X::int set) = −(Sup (uminus ' X))

instance
proof
{ fix x :: int and X :: int set assume X ≠ {} bdd-above X
  then obtain x y where X ⊆ {...} x ∈ X
    by (auto simp: bdd-above-def)
  then have *: finite (X ∩ {...}) X ∩ {...} ≠ {} and x ≤ y
    by (auto simp: subset-eq)
  have ∃ x∈X. (∀ y∈X. y ≤ x)
    proof
      { fix z assume z ∈ X
        have z ≤ Max (X ∩ {...})
          proof cases
            assume x ≤ z with * (1) show ?thesis
              by (auto intro!: Max-ge)
          next
            assume ¬ x ≤ z
            then have z < x by simp
            also have x ≤ Max (X ∩ {...})
              using * by (intro Max-ge) auto
            finally show ?thesis by simp
          qed }
        note le = this
        with Max-in[OF *] show: Max (X ∩ {...}) ∈ X ∧ (∀ z∈X. z ≤ Max (X ∩ {...})) by auto
      } note le-Sup = this
    fix z assume *: z ∈ X ∧ (∀ y∈X. y ≤ z)
    with le have z ≤ Max (X ∩ {...})
      by auto
    moreover have Max (X ∩ {...}) ≤ z
      using * ex by auto
    ultimately show z = Max (X ∩ {...})
      by auto
  qed }
  then have Sup X ∈ X ∧ (∀ y∈X. y ≤ Sup X)
    unfolding Sup-int-def by (rule theI') }
  note Sup-int = this

  { fix x :: int and X :: int set assume x ∈ X bdd-above X then show x ≤ Sup X
    using Sup-int[of X] by auto }
  note le-Sup = this
{ fix x :: int and X :: int set assume X ≠ {} \land y, y ∈ X \implies y ≤ x then show Sup X ≤ x using Sup-int[of X] by (auto simp: bdd-above-def) }

note Sup-le = this

{ fix x :: int and X :: int set assume x ∈ X bdd-below X then show Inf X ≤ x using le-Sup[of uminus ' X] by (auto simp: Inf-int-def) }

{ fix x :: int and X :: int set assume X ≠ {} \land y, y ∈ X \implies x ≤ y then show x ≤ Inf X using Sup-le[of uminus ' X − x] by (force simp: Inf-int-def) }

qed
end

lemma interval-cases:
fixes S :: ‘a :: conditionally-complete-linorder set
assumes ivl: \land a b x. a ∈ S \implies b ∈ S \implies a ≤ x \implies x ≤ b \implies x ∈ S
shows ∃ a b. S = {} ∨ S = UNIV ∨ S = {..b} ∨ S = {a..} ∨ S = {a..<} ∨ S = {a..<b} ∨ S = {a..} ∨ S = {a..<} ∨ S = {a..<b} ∨ S = {a..<} ∨ S = {a..<b} ∨ S = {a..b}
proof −
  define lower upper where lower = {x. \exists s∈S. s ≤ x} and upper = {x. \exists s∈S. x ≤ s}
  with ivl have S = lower \cap upper
    by auto
  moreover have \exists a. upper = UNIV \lor upper = {} \lor upper = {.. a} \lor upper = {..< a}
  proof
    assume *: bdd-above S \land S ≠ {}
    from * have upper ⊆ {.. Sup S}
      by (auto simp: upper-def intro: cSup-upper2)
    moreover from * have {..< Sup S} ⊆ upper
      by (force simp add: less-cSup-iff upper-def subset-eq Ball-def)
    ultimately have upper = {.. Sup S} \lor upper = {..< Sup S}
    unfolding iel-disj-un(2)[symmetric] by auto
    then show ?thesis by auto
  next
    assume ¬ (bdd-above S \land S ≠ {})
    then have upper = UNIV \lor upper = {}
      by (auto simp: upper-def bdd-above-def not-le dest: less-imp-le)
    then show ?thesis
      by auto
  qed
qed
moreover
have \( \exists b. \text{lower} = \text{UNIV} \lor \text{lower} = \{ \} \lor \text{lower} = \{ b .. \} \lor \text{lower} = \{ b <.. \} \)

proof cases
  assume \( *: \text{bdd-below} \ S \land S \neq \{ \} \)
  from \( * \) have \( \text{lower} \subseteq \{ \text{Inf} \ S .. \} \)
    by (auto simp: lower-def intro: cInf-lower2)
  moreover from \( * \) have \( \{ \text{Inf} \ S .. \} \subseteq \text{lower} \)
    by (force simp add: cInf-less-iff lower-def subset-eq Ball-def)
  ultimately have \( \text{lower} = \{ \text{Inf} \ S .. \} \lor \text{lower} = \{ \text{Inf} \ S <.. \} \)
  unfolding \( \text{iol-disj-un}(1)[\text{symmetric}] \) by auto
  then show \( ?\text{thesis} \) by auto
next
  assume \( \neg (\text{bdd-below} \ S \land S \neq \{ \}) \)
  then have \( \text{lower} = \text{UNIV} \lor \text{lower} = \{ \} \)
    by (auto simp: lower-def bdd-below-def not-le dest: less-imp-le)
  then show \( ?\text{thesis} \) by auto
qed

ultimately show \( ?\text{thesis} \)
unfolding greaterThanAtMost-def greaterThanLessThan-def atLeastAtMost-def atLeastLessThan-def
by (metis inf-bot-left inf-bot-right inf-top.left-neutral inf-top.right-neutral)
qed

lemma \( \text{cSUP-eq-cINF-D} \):
  fixes \( f : . \Rightarrow \text{b: conditionally-complete-lattice} \)
  assumes eq: \( (\bigsqcup x \in A. f x) = (\bigsqcup x \in A. f x) \)
    and bdd: \( \text{bdd-above} \ (f \ A) \) \( \text{bdd-below} \ (f \ A) \)
    and \( a : a \in A \)
  shows \( f a = (\bigsqcup x \in A. f x) \)
proof (rule antisym)
  show \( f a \leq \bigsqcup (f \ A) \)
    by (metis a bdd by (auto simp: cINF-lower))
  show \( \bigsqcup (f \ A) \leq f a \)
    using a bdd by (auto simp: cSUP-upper)
qed

lemma \( \text{cSUP-UNION} \):
  fixes \( f : . \Rightarrow \text{b: conditionally-complete-lattice} \)
  assumes ne: \( A \neq \{ \} \wedge x \in A \Rightarrow B(x) \neq \{ \} \)
    and bdd-UN: \( \text{bdd-above} \ (\bigcup x \in A. f \ B x) \)
  shows \( (\bigcup z \in \bigcup x \in A. B x. f z) = (\bigcup x \in A. \bigcup z \in B x. f z) \)
proof
  have \( \text{bdd}: \bigwedge x. x \in A \Rightarrow \text{bdd-above} \ (f \ B x) \)
    using bdd-UN by (meson UN-upper bdd-above-mono)
  obtain \( M \) where \( \bigwedge x \ y. x \in A \Rightarrow y \in B(x) \Rightarrow f y \leq M \)
    using bdd-UN by (auto simp: bdd-above-def)
  then have \( \text{bdd2}: \text{bdd-above} \ ((\lambda x. \bigcup z \in B x. f z) \ A) \)
    unfolding \( \text{bdd-above-def} \) by (force simp: bdd cSUP-le-iff ne(2))
have \( (\bigcup z \in \bigcup x \in A. B. x. f z) \leq (\bigcup z \in \bigcup x \in A. B. x. f z) \)
using assms by (fastforce simp add: intro!: cSUP-least intro: cSUP-upper2 simp: bdd2 bdd)
moreover have \( (\bigcup z \in A. \bigcup z \in B. x. f z) \leq (\bigcup z \in \bigcup x \in A. B. x. f z) \)
using assms by (fastforce simp add: intro!: cSUP-least intro: cSUP-upper simp: image-UN bdd-UN)
ultimately show \(?thesis\)
by (rule order-antisym)
qed

lemma cINF-UNION:
fixes \( f :: \alpha \Rightarrow 'b::conditionally-complete-lattice \)
assumes ne: \( A \neq \{\} \\land x. x \in A \implies B(x) \neq \{\} \)
and bdd-UN: \( \text{bdd-below} (\bigcup x \in A. f ^ ' B x) \)
shows \( (\bigcap z \in \bigcup x \in A. B. x. f z) = (\bigcap x \in A. \bigcap z \in B. x. f z) \)
proof –
have bdd: \( \land x. x \in A \implies \text{bdd-below} (f ^ ' B x) \)
using bdd-UN by (meson UN-upper bdd-below-mono)
obtain M where \( \land x y. x \in A \implies y \in B(x) \implies f y \geq M \)
using bdd-UN by (auto simp: bdd-below-def)
then have bdd2: \( \text{bdd-below} ((\lambda x. \bigcap z \in B. x. f z) ^ ' A) \)
unfolding bdd-below-def by (force simp: bdd le-cINF-iff ne(2))
have \( (\bigcap z \in \bigcup x \in A. B. x. f z) \leq (\bigcap x \in A. \bigcap z \in B. x. f z) \)
using assms by (fastforce simp add: intro!: cINF-greatest intro: cINF-lower simp: bdd2 bdd)
moreover have \( (\bigcap z \in A. \bigcap z \in B. x. f z) \leq (\bigcap z \in \bigcup x \in A. B. x. f z) \)
using assms by (fastforce simp add: intro!: cINF-greatest intro: cINF-lower2 simp: bdd bdd-UN bdd2)
ultimately show \(?thesis\)
by (rule order-antisym)
qed

lemma cSup-abs-le:
fixes \( S :: ('a::linordered-idom,conditionally-complete-linorder) \) set
shows \( S \neq \{\} \implies (\land x. x \in S \implies |x| \leq a) \implies |\text{Sup} S| \leq a \)
apply (auto simp add: abs-le-iff intro: cSup-least)
by (metis bdd-above1 cSup-upper neg-le-iff-le order-trans)

end

92 Factorial Function, Rising Factorials

theory Factorial
imports Groups-List
begin

92.1 Factorial Function

context semiring-char-0
begin

definition fact :: nat ⇒ 'a
  where fact-prod: fact n = of-nat (∏ {1..n})

lemma fact-prod-Suc: fact n = of-nat (prod Suc {0..<n})
  unfolding fact-prod using atLeast0LessThan prod.atLeast1-atMost-eq by auto

lemma fact-prod-rev: fact n = of-nat (∏ i = 0..<n. n − i)
  proof
    have prod Suc {0..<n} = ∏ {1..n}
      by (simp add: atLeast0LessThan prod.atLeast1-atMost-eq)
    then have prod Suc {0..<n} = prod ((−) (n + 1)) {1..n}
      using prod.atLeastAtMost-rev [of λ i. i 1 n] by presburger
    then show ?thesis
      unfolding fact-prod-Suc by (simp add: atLeast0LessThan prod.atLeast1-atMost-eq)
  qed

lemma fact-0 [simp]: fact 0 = 1
  by (simp add: fact-prod)

lemma fact-1 [simp]: fact 1 = 1
  by (simp add: fact-prod)

lemma fact-Suc-0 [simp]: fact (Suc 0) = 1
  by (simp add: fact-prod)

lemma fact-Suc [simp]: fact (Suc n) = of-nat (Suc n) * fact n
  by (simp add: fact-prod atLeastAtMostSuc-conv algebra-simps)

lemma fact-2 [simp]: fact 2 = 2
  by (simp add: numeral-2-eq-2)

lemma fact-split: k ≤ n ⇒ fact n = of-nat (prod Suc {n − k..<n}) * fact (n − k)
  by (simp add: fact-prodSuc prod.union-disjoint [symmetric]
    iel-disj-an ac-simps of-nat-mult [symmetric])

end

lemma of-nat-fact [simp]: of-nat (fact n) = fact n
  by (simp add: fact-prod)

lemma of-int-fact [simp]: of-int (fact n) = fact n
  by (simp only: fact-prod of-int-of-nat-eq)

lemma fact-reduce: n > 0 ⇒ fact n = of-nat n * fact (n − 1)
  by (cases n) auto
lemma fact-nonzero [simp]: \( \text{fact } n \neq (0 :: 'a::\{semiring-char-0,semiring-no-zero-divisors\}) \)
using of-nat-0-neq by (induct n) auto

lemma fact-mono-nat: \( m \leq n \Rightarrow \text{fact } m \leq (\text{fact } n :: \text{nat}) \)
by (induct n) (auto simp: le-Suc-eq)

lemma fact-in-Nats: \( \text{fact } n \in \mathbb{N} \)
by (induct n) auto

lemma fact-in-Ints: \( \text{fact } n \in \mathbb{Z} \)
by (induct n) auto

context assumes SORT-CONSTRAINT('a::linordered-semidom)
begin

lemma fact-mono: \( m \leq n \Rightarrow \text{fact } m \leq (\text{fact } n :: 'a) \)
by (metis of-nat-fact of-nat-le-iff fact-mono-nat)

lemma fact-ge-1 [simp]: \( \text{fact } n \geq (1 :: 'a) \)
by (metis le0 fact-0 fact-mono)

lemma fact-gt-zero [simp]: \( \text{fact } n > (0 :: 'a) \)
using fact-ge-1 less-le-trans zero-less-one by blast

lemma fact-ge-zero [simp]: \( \text{fact } n \geq (0 :: 'a) \)
by (simp add: less-imp-le)

lemma fact-not-neg [simp]: \( \neg \text{fact } n < (0 :: 'a) \)
by (simp add: not-less-iff-gr-or-eq)

lemma fact-le-power: \( \text{fact } n \leq (\text{of-nat } (n^n) :: 'a) \)
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then have \*: \( \text{fact } n \leq (\text{of-nat } (\text{Suc } n^n) :: 'a) \)
  by (rule order-trans) (simp add: power-mono del: of-nat-power)
  have fact (Suc n) = (of-nat (Suc n) * fact n :: 'a)
  by (simp add: algebra-simps)
  also have \ldots \leq of-nat (Suc n) * of-nat (Suc n^n)
  by (simp add: * ordered-comm-semiring-class.comm-mult-left-mono del: of-nat-power)
  also have \ldots \leq of-nat (Suc n ^ Suc n)
  by (metis of-nat-mult order-refl power-Suc)
  finally show ?case.
qed

end
lemma fact-less-mono-nat: \( 0 < m \implies m < n \implies \text{fact} m < (\text{fact} n :: \text{nat}) \)
by (induct n) (auto simp: less-Suc-eq)

lemma fact-less-mono: \( 0 < m \implies m < n \implies \text{fact} m < (\text{fact} n :: 'a::linordered-semidom) \)
by (metis of-nat-fact of-nat-less-iff fact-less-mono-nat)

lemma fact-less-mono: \( 0 < m \implies m < n \implies \text{fact} m < (\text{fact} n :: 'a::linordered-semidom) \)
by (induct n) (auto simp: dvdI le-Suc-eq)

lemma fact-ge-Suc-0-nat [simp]: \( \text{fact} n \geq \text{Suc} 0 \)
by (metis One-nat-def fact-ge-1)

lemma dvd-fact: \( 1 \leq m \implies m \leq n \implies m \text{ dvd} \text{ fact} n \)
by (induct n) (auto simp: dvdI le-Suc-eq)

lemma fact-ge-self: \( \text{fact} n \geq n \)
by (cases n = 0) (simp-all add: dvd-imp-le dvd-fact)

lemma fact-dvd: \( n \leq m \implies \text{fact} n \text{ dvd} (\text{fact} m :: 'a::linordered-semidom) \)
by (induct m) (auto simp: le-Suc-eq)

lemma fact-mod: \( m \leq n \implies \text{fact} n \text{ mod} (\text{fact} m :: 'a::semidom-modulo, linordered-semidom) = 0 \)
by (simp add: mod-eq-0-iff-dvd fact-dvd)

lemma fact-eq-fact-times: assumes m \( \geq n \) shows \( \text{fact} m = \text{fact} n \times \prod \{\text{Suc} n .. m\} \)
unfolding fact-prod
by (metis add.commute assms le-add1 le-add-diff-inverse of-nat-id plus-1-eq-Suc prod.ub-add-nat)

lemma fact-div-fact: assumes m \( \geq n \) shows \( \text{fact} m \text{ div} \text{ fact} n = \prod \{n + 1 .. m\} \)
by (simp add: fact-eq-fact-times [OF assms])

lemma fact-num-eq-if: \( \text{fact} m = (\text{if} m = 0 \text{ then} 1 \text{ else} \text{of-nat} m \ast \text{fact} (m - 1)) \)
by (cases m) auto

lemma fact-div-fact-le-pow: assumes r \( \leq n \) shows \( \text{fact} n \text{ div} \text{ fact} (n - r) \leq n \sim r \)
proof -
have r \( \leq n \implies \prod \{n - r .. n\} = (n - r) \times \prod \{\text{Suc} (n - r) .. n\} \) for r
by (subst prod.insert[symmetric]) (auto simp: atLeastAtMostInsertL)
with assms show \?thesis
by (induct r rule: nat.induct) (auto simp add: fact-div-fact Suc-diff-Suc mult-le-mono)
qed

lemma prod-Suc-fact: \( \text{prod} \{0 ..< n\} = \text{fact} n \)
by (simp add: fact-prod-Suc)

lemma prod-Suc-Suc-fact: prod Suc {Suc 0..<n} = fact n
proof (cases n = 0)
  case True
  then show ?thesis by simp
next
  case False
  have prod Suc {Suc 0..<n} = Suc 0 * prod Suc {Suc 0..<n}
    by simp
  also have ... = prod Suc (insert 0 {Suc 0..<n})
    by simp
  also have insert 0 {Suc 0..<n} = {0..<n}
    using False by auto
  finally show ?thesis
    by (simp add: fact-prod-Suc)
qed

lemma fact-numeral: fact (numeral k) = numeral k * fact (pred-numeral k)
  — Evaluation for specific numerals
  by (metis fact-Suc numeral-eq-Suc of-nat-numeral)

92.2 Pochhammer’s symbol: generalized rising factorial

See https://en.wikipedia.org/wiki/Pochhammer_symbol.

context comm-semiring-1
begin

definition pochhammer :: 'a ⇒ nat ⇒ 'a
  where pochhammer-prod: pochhammer a n = prod (λi. a + of-nat i) {0..<n}

lemma pochhammer-prod-rev: pochhammer a n = prod (λi. a + of-nat (n - i)) {1..n}
  using prod.atLeastLessThan-rev-at-least-atMost [of λi. a + of-nat i 0 n]
  by (simp add: pochhammer-prod)

lemma pochhammer-Suc-prod: pochhammer a (Suc n) = prod (λi. a + of-nat i) {0..n}
  by (simp add: pochhammer-prod atLeastLessSuc-atLeastAtMost)

lemma pochhammer-Suc-prod-rev: pochhammer a (Suc n) = prod (λi. a + of-nat (n - i)) {0..n}
  using prod.atLeastSuc-atMost-Suc-shift
  by (simp add: pochhammer-prod-rev prod.atLeastSuc-atMost-Suc-shift del: prod.cl-ivl-Suc)

lemma pochhammer-0 [simp]: pochhammer a 0 = 1
  by (simp add: pochhammer-prod)

lemma pochhammer-1 [simp]: pochhammer a 1 = a
by (simp add: pochhammer-prod lessThan-Suc)

lemma pochhammer-Suc0 [simp]: pochhammer a (Suc 0) = a
  by (simp add: pochhammer-prod lessThan-Suc)

lemma pochhammer-Suc: pochhammer a (Suc n) = pochhammer a n * (a + of-nat n)
  by (simp add: pochhammer-prod atLeast0-lessThan-Suc ac-simps)
end

lemma pochhammer-nonneg:
  fixes x :: 'a :: linordered-semidom
  shows x > 0 ==> pochhammer x n ≥ 0
  by (induction n) (auto simp: pochhammer-Suc intro!: mult-nonneg-nonneg add-nonneg-nonneg)

lemma pochhammer-pos:
  fixes x :: 'a :: linordered-semidom
  shows x > 0 ==> pochhammer x n > 0
  by (induction n) (auto simp: pochhammer-Suc intro!: mult-pos-pos add-pos-nonneg)

context comm-semiring-1
begin

lemma pochhammer-of-nat: pochhammer (of-nat x) n = of-nat (pochhammer x n)
  by (simp add: pochhammer-prod Factorial.pochhammer-prod)
end

context comm-ring-1
begin

lemma pochhammer-of-int: pochhammer (of-int x) n = of-int (pochhammer x n)
  by (simp add: pochhammer-prod Factorial.pochhammer-prod)
end

lemma pochhammer-rec: pochhammer a (Suc n) = a * pochhammer (a + 1) n
  by (simp add: pochhammer-prod prod.atLeast0-lessThan-Suc-shift ac-simps del: prod.op-iol-Suc)

lemma pochhammer-rec': pochhammer z (Suc n) = (z + of-nat n) * pochhammer z n
  by (simp add: pochhammer-prod prod.atLeast0-lessThan-Suc ac-simps)

lemma pochhammer-fact: fact n = pochhammer 1 n
  by (simp add: pochhammer-prod fact-prod-Suc)

lemma pochhammer-of-nat-eq-0-lemma: k > n ==> pochhammer (− (of-nat n ::
THEORY “Factorial”

'(a:: idom)) k = 0

by (auto simp add: pochhammer-prod)

lemma pochhammer-of-nat-eq-0-lemma':
assumes kn: k ≤ n
shows pochhammer (- (of-nat n :: 'a::{idom,ring-char-0})) k ≠ 0
proof (cases k)
  case 0
  then show ?thesis by simp
next
  case (Suc h)
  then show ?thesis
    apply (simp add: pochhammer-Suc-prod)
    using Suc kn
    apply (auto simp add: algebra-simps)
  done
qed

lemma pochhammer-of-nat-eq-0-iff:
pochhammer (- (of-nat n :: 'a::{idom,ring-char-0})) k = 0 ←→ k > n
(is ?l = ?r)
using pochhammer-of-nat-eq-0-lemma[of n k, where ?'a=a]
pochhammer-of-nat-eq-0-lemma'[of k n, where ?'a = 'a]
by (auto simp add: not-le[symmetric])

lemma pochhammer-0-left:
pochhammer 0 n = (if n = 0 then 1 else 0)
by (induction n) (simp-all add: pochhammer-rec)

lemma pochhammer-eq-0-iff: pochhammer a n = (0::'a::field-char-0) ←→ (∃k < n. a = − of-nat k)
by (auto simp add: pochhammer-prod eq-neg-iff-add-eq-0)

lemma pochhammer-eq-0-mono:
pochhammer a n = (0::'a::field-char-0) ⇒ m ≥ n ⇒ pochhammer a m = 0
unfolding pochhammer-eq-0-iff by auto

lemma pochhammer-neq-0-mono:
pochhammer a m ≠ (0::'a::field-char-0) ⇒ m ≥ n ⇒ pochhammer a n ≠ 0
unfolding pochhammer-eq-0-iff by auto

lemma pochhammer-minus:
pochhammer (- b) k = ((− 1) ^ k :: 'a::comm-ring-1) * pochhammer (b − of-nat k + 1) k
proof (cases k)
  case 0
  then show ?thesis by simp
next
  case (Suc h)
have eq: ((− 1) ^ Suc h :: 'a) = (∏ i = 0..h. − 1)
using prod-constant [where A={0.. h} and y=− 1 :: 'a]
by auto
with Suc show ?thesis
using pochhammer-Suc-prod-rev [of b − of-nat k + 1]
by (auto simp add: pochhammer-Suc-prod prod, distrib [symmetric] eq of-nat-diff simp del: prod-constant)
qed

lemma pochhammer-minus': pochhammer (b − of-nat k + 1) k = ((− 1) ^ k :: 'a::comm-ring-1) * pochhammer (− b) k
by (simp add: pochhammer-minus)

lemma pochhammer-same: pochhammer (− of-nat n) n =
((− 1) ^ n :: 'a::{semiring-char-0,comm-ring-1,semiring-no-zero-divisors}) * fact n
unfolding pochhammer-minus
by (simp add: of-nat-diff pochhammer-fact)

lemma pochhammer-product': pochhammer z (n + m) = pochhammer z n * pochhammer (z + of-nat m) m
proof (induct n arbitrary: z)
  case 0
  then show ?case by simp
next
  case (Suc n z)
  have pochhammer z (Suc n) * pochhammer (z + of-nat (Suc n)) m =
    z * (pochhammer (z + 1) n * pochhammer (z + 1 + of-nat n) m)
    by (simp add: pochhammer-rec ac-simps)
  also note Suc[symmetric]
  also have z * pochhammer (z + 1) (n + m) = pochhammer z (Suc (n + m))
    by (subst pochhammer-rec) simp
  finally show ?case
    by simp
qed

lemma pochhammer-product:
  m ≤ n =⇒ pochhammer z n = pochhammer z m * pochhammer (z + of-nat m) (n − m)
  using pochhammer-product'[of z m n − m] by simp

lemma pochhammer-times-pochhammer-half:
  fixes z :: 'a::field-char-0
  shows pochhammer z (Suc n) * pochhammer (z + 1/2) (Suc n) = (∏ k=0..2*n+1. z + of-nat k / 2)
  proof (induct n)
    case 0
    then show ?case
by (simp add: atLeast0-atMost-Suc)
next
  case (Suc n)
define n' where n' = Suc n
  have pochhammer z (Suc n') * pochhammer (z + 1 / 2) (Suc n') =
    (pochhammer z n' * pochhammer (z + 1 / 2) n') * ((z + of-nat n') * (z +
    1/2 + of-nat n'))
    (is - = - * ?A)
    by (simp-all add: pochhammer-rec' mult-ac)
  also have ?A = (z + of-nat (Suc (2 * n + 1)) / 2) * (z + of-nat (Suc (Suc (2
    * n + 1)))) / 2)
    (is - = ?B)
    by (simp add: field-simps n'-def)
  also note Suc[folded n'-def]
  also have (∏ k=0..2 * n + 1. z + of-nat k / 2) * ?B = (∏ k=0..2 * Suc n +
    1. z + of-nat k / 2)
    by (simp add: atLeast0-atMost-Suc)
finally show ?case
  by (simp add: n'-def)
qed

lemma pochhammer-double:
  fixes z :: 'a::field-char-0
  shows pochhammer (2 * z) (2 * n) = of-nat (2^(2*n)) * pochhammer z n *
    pochhammer (z+1/2) n
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have pochhammer (2 * z) (2 * (Suc n)) = pochhammer (2 * z) (2 * n) *
    (2 * (z + of-nat n)) * (2 * (z + of-nat n) + 1)
    by (simp add: pochhammer-rec' ac-simps)
  also note Suc
  also have of-nat (2 ^ (2 * n)) * pochhammer z n * pochhammer (z + 1/2) n *
    (2 * (z + of-nat n)) * (2 * (z + of-nat n) + 1) =
    of-nat (2 ^ (2 * (Suc n))) * pochhammer z (Suc n) * pochhammer (z + 1/2)
    (Suc n)
    by (simp add: field-simps pochhammer-rec')
  finally show ?case .
qed

lemma fact-double:
  fact (2 * n) = (2 ^ (2 * n)) * pochhammer (1 / 2) n * fact n :: 'a::field-char-0
  using pochhammer-double[of 1/2::'a n] by (simp add: pochhammer-fact)

lemma pochhammer-absorb-comp: (r - of-nat k) * pochhammer (- r) k = r *
  pochhammer (-r + 1) k
(is ?lhs = ?rhs)
for $r :: 'a::comm-ring-1$
proof
  have ?lhs = pochhammer ($- r$) (Suc $k$)
    by (subst pochhammer-rec') (simp add: algebra-simps)
  also have \ldots = ?rhs
    by (subst pochhammer-rec) simp
finally show ?thesis .
qed

92.3 Misc

lemma fact-code [code]:
  fact $n = (of-nat (fold-atLeastAtMost-nat ((*)) \ 2 \ \ n \ 1) :: 'a::semiring-char-0)$
proof
  have fact $n = (of-nat (\prod \{1..n\}) :: 'a)
    by (simp add: fact-prod)
  also have $\prod \{1..n\} = \prod \{2..n\}$
    by (intro prod_mono_neutral-right) auto
  also have \ldots = fold-atLeastAtMost-nat ((*)) \ 2 \ \ n \ 1
    by (simp add: prod-atLeastAtMost-code)
finally show ?thesis .
qed

lemma pochhammer-code [code]:
  pochhammer $a \ n =$
  (if $n = 0$ then 1
    else fold-atLeastAtMost-nat ($\lambda n \ acc. (a + of-nat n) * \ acc$) \ 0 \ \ (n - 1) \ 1)
by (cases $n$)
  (simp-all add: pochhammer-prod prod-atLeastAtMost-code [symmetric]
    atLeastLessThanSuc-atLeastAtMost)

end

93 Binomial Coefficients, Binomial Theorem, Inclusion-exclusion Principle

theory Binomial
  imports Presburger Factorial
begin

93.1 Binomial coefficients

This development is based on the work of Andy Gordon and Florian Kammueller.

Combinatorial definition

definition binomial :: nat $\Rightarrow$ nat $\Rightarrow$ nat (infixl choose 65)
where $n$ choose $k = \text{card} \{K \in \text{Pow} \{0..<n\}. \ \text{card} \ K = k\}$
lemma binomial-mono:
  assumes \( m \leq n \) shows \( \text{choose } m \leq \text{choose } n \)

proof
  have \( \{ K. \ K \subseteq \{ 0..<m \} \land \text{card } K = k \} \subseteq \{ K. \ K \subseteq \{ 0..<n \} \land \text{card } K = k \} \)
    using assms by auto
  then show \( \text{thesis} \)
    by (simp add: binomial-def card-mono)
qed

theorem n-subsets:
  assumes finite A
  shows \( \text{card } \{ B. \ B \subseteq A \land \text{card } B = k \} = \text{choose } A \) choose \( k \)

proof
  from assms obtain \( f \) where bij: bij-betw \( f \) \( \{ 0..<\text{card } A \} \) \( A \)
    by (blast dest: ex-bij-betw-nat-finite)
  then have \( \text{simp}: \text{card } (f ^' C) = \text{card } C \) if \( C \subseteq \{ 0..<\text{card } A \} \) for \( C \)
    by (meson bij-betw-imp-inj-on bij-betw-subset card-image that)
  from bij have bij-betw \( (\text{image } f) (\text{Pow } \{ 0..<\text{card } A \} ) (\text{Pow } A) \)
    by (rule bij-betw-Pow)
  then have \( \text{inj-on } (\text{image } f) (\text{Pow } \{ 0..<\text{card } A \} ) \)
    by (rule bij-betw-imp-inj-on)
  moreover have \( \{ K. \ K \subseteq \{ 0..<\text{card } A \} \land \text{card } K = k \} \subseteq \text{Pow } \{ 0..<\text{card } A \} \)
    by auto
  ultimately have \( \text{inj-on } (\text{image } f) \{ K. \ K \subseteq \{ 0..<\text{card } A \} \land \text{card } K = k \} \)
    by (rule inj-on-subset)
  then have \( \text{card } \{ K. \ K \subseteq \{ 0..<\text{card } A \} \land \text{card } K = k \} = \text{card } (\text{image } f ^' \{ K. \ K \subseteq \{ 0..<\text{card } A \} \land \text{card } K = k \} ) \) (is - = card ?C)
    by (simp add: card-image)
  also have \( ?C = \{ K. \ K \subseteq f ^' \{ 0..<\text{card } A \} \land \text{card } K = k \} \)
    by (auto elim!: subset-imageE)
  also have \( f ^' \{ 0..<\text{card } A \} = A \)
    by (meson bij bij-betw-def)
  finally show \( \text{thesis} \)
    by (simp add: binomial-def)
qed

Recursive characterization
lemma binomial-n-0 [simp]: \( \text{choose } 0 = 1 \)

proof
  have \( \{ K \in \text{Pow } \{ 0..<n \}. \ \text{card } K = 0 \} = \{ \} \)
    by (auto dest: finite-subset)
  then show \( \text{thesis} \)
    by (simp add: binomial-def)
qed

lemma binomial-0-Suc [simp]: \( \text{choose } \text{Suc } k = 0 \)
  by (simp add: binomial-def)
lemma binomial-Suc-Suc [simp]: Suc n choose Suc k = (n choose k) + (n choose Suc k)
proof -
  let ?P = λn k. {K. K ⊆ {0..<n} ∧ card K = k}
  let ?Q = ?P (Suc n) (Suc k)
  have inj: inj-on (insert n) (?P n k)
    by (rule (auto; metis atLeastLessThan-iff insert-iff less-irrefl subsetCE))
  have disjoint: insert n ' ?P n k ∩ ?P n (Suc k) = {}
    by auto
  also have ?Q = {K ∈ ?Q. n ∈ K} ∪ {K ∈ ?Q. n /∈ K}
    by auto
  also have {K ∈ ?Q. n ∈ K} = insert n ' ?P n k (is ?A = ?B)
    proof (rule set-eqI)
      fix K
      have K-finite: finite K if K ⊆ insert n {0..<n}
        using that by (rule finite-subset) simp-all
      have Suc-card-K: Suc (card K − Suc 0) = card K if n ∈ K
        and finite K
      proof -
        from n ∈ K obtain L where K = insert n L and n /∈ L
          by (blast elim: Set.set-insert)
        with that show ?thesis by (simp add: card-un-disjoint card-image)
      qed
      show K ∈ ?A ←→ K ∈ ?B
        by (subst in-image-insert-iff)
        (auto simp add: card-insert-remove subset-eq-atLeast0-lessThan-finite
          Diff-subset-conv K-finite Suc-card-K)
      qed
    also have {K ∈ ?Q. n /∈ K} = ?P n (Suc k)
      by (auto simp add: atLeast0-lessThan-Suc)
    finally show ?thesis using inj disjoint
      by (simp add: binomial-def card-Un-disjoint card-image)
  qed

lemma binomial-eq-0: n < k ⇒ n choose k = 0
  by (auto simp add: binomial-def dest: subset-eq-atLeast0-lessThan-card)

lemma zero-less-binomial: k ≤ n ⇒ n choose k > 0
  by (induct n k rule: diff-induct) simp-all

lemma binomial-eq-0-iff [simp]: n choose k = 0 ↔ n < k
  by (metis binomial-eq-0 less-numeral-extra(3) not-less zero-less-binomial)

lemma zero-less-binomial-iff [simp]: n choose k > 0 ↔ k ≤ n
  by (metis binomial-eq-0-iff not-less0 not-less zero-less-binomial)

lemma binomial-n-n [simp]: n choose n = 1
  by (induct n) (simp-all add: binomial-eq-0)
lemma binomial-Suc-n [simp]: Suc n choose n = Suc n
  by (induct n) simp-all

lemma binomial-1 [simp]: n choose Suc 0 = n
  by (induct n) simp-all

lemma choose-one: n choose 1 = n for n :: nat
  by simp

lemma choose-reduce-nat:
  0 < n ⇒ 0 < k ⇒ n choose k = ((n - 1) choose (k - 1)) + ((n - 1) choose k)
  using binomial-Suc-Suc [of n - 1 k - 1] by simp

lemma Suc-times-binomial-eq:
  Suc n * (n choose k) = (Suc n choose Suc k) * Suc k
proof (induction n arbitrary: k)
  case 0
  then show ?case by auto
next
  case (Suc n)
  show ?case
    proof (cases k)
      case (Suc k')
      then show ?thesis
        using Suc.IH by (simp add: add-mult-distrib add-mult-distrib2 le-Suc-eq binomial-eq-0)
    qed auto
  qed

lemma binomial-le-pow2: n choose k ≤ 2^n
proof (induction n arbitrary: k)
  case 0
  then show ?case using le-less less-le-trans by fastforce
next
  case (Suc n)
  show ?case
    proof (cases k)
      case (Suc k')
      then show ?thesis
        using Suc.IH by (simp add: add-le-mono mult-2)
    qed auto
  qed

The absorption property.

lemma Suc-times-binomial:
  Suc k * (Suc n choose Suc k) = Suc n * (n choose k)
  using Suc-times-binomial-eq by auto
This is the well-known version of absorption, but it’s harder to use because of the need to reason about division.

**lemma** binomial-Suc-Suc-eq-times: \((\text{Suc } n \text{ choose } \text{Suc } k) = (\text{Suc } n \ast (n \text{ choose } k))\)  
\(\text{div } \text{Suc } k\)
  by (simp add: Suc-times-binomial-eq del: mult-Suc mult-Suc-right)

Another version of absorption, with \(-1\) instead of \(\text{Suc}\).

**lemma** times-binomial-minus1-eq: \(\text{\(\theta\) \(<\) \(k\) \(\implies\) \(\text{k} \ast (n \text{ choose } k) = n \ast ((n - 1)\) \text{ choose } (k - 1))\)}\)
  using Suc-times-binomial-eq [where \(n = n - 1\) and \(k = k - 1\)]
  by (auto split: nat-diff-split)

### 93.2 The binomial theorem (courtesy of Tobias Nipkow):

Avigad’s version, generalized to any commutative ring

**theorem** (in comm-semiring-1) binomial-ring:
\((a + b \vdash 'a)^{\underline{n}} = (\sum k \leq n. (\text{of-nat } (n \text{ choose } k)) \ast a^\underline{k} \ast b^{\underline{n-k}})\)

**proof** (induct \(n\))
\(\text{case } \theta\)  
then show \(?case by simp\)
next
\(\text{case } (\text{Suc } n)\)
  have \(\text{decomp: } \{0..n+1\} = \{0\} \cup \{n + 1\} \cup \{1..n\} \text{ and } \text{decomp2: } \{0..n\} = \{0\} \cup \{1..n\}\)
  by auto
  have \((a + b)^\underline{(n+1)} = (a + b) \ast (\sum k \leq n. (\text{of-nat } (n \text{ choose } k)) \ast a^\underline{k} \ast b^{\underline{n-k}})\)
    using Suc.hyps by simp
  also have \(\ldots = a \ast (\sum k \leq n. (n \text{ choose } k) \ast a^\underline{k} \ast b^{\underline{n-k}}) + \)
    \(b \ast (\sum k \leq n. (n \text{ choose } k) \ast a^\underline{k} \ast b^{\underline{n-k}})\)
    by (rule distrib-right)
  also have \(\ldots = (\sum k \leq n. (n \text{ choose } k) \ast a^\underline{k} \ast b^{\underline{n-k}}) + \)
    \((\sum k \leq n. (n \text{ choose } k) \ast a^\underline{k} \ast b^{\underline{n-k}}) + \)
    \((\sum k = 1..n+1. \text{ of-nat } (n \text{ choose } (k - 1)) \ast a^\underline{k} \ast b^{\underline{n + 1 - k}})\)
    by (simp add: atMost-atLeast0 sum.shift-bounds-cl-Suc-ivl Suc-diff-le field-simps del: sum.cl-ivl-Suc)
  also have \(\ldots = b^{\underline{n + 1}} + \)
    \((\sum k = 1..n. \text{ of-nat } (n \text{ choose } k) \ast a^\underline{k} \ast b^{\underline{n + 1 - k}}) + (a^{\underline{n + 1}} + \)
    \((\sum k = 1..n. \text{ of-nat } (n \text{ choose } (k - 1)) \ast a^\underline{k} \ast b^{\underline{n + 1 - k}})\)
    using sum.nat-ivl-Suc' [of \(1\) \(\lambda k. \text{ of-nat } (n \text{ choose } (k-1)) \ast a^\underline{k} \ast b^{\underline{n + 1 - k}}(n + 1 - k))]
    by (simp add: sum.atLeast-Suc-atMost atMost-atLeast0)
  also have \(\ldots = a^{\underline{(n + 1)}} + b^{\underline{(n + 1)}} + \)
    \((\sum k = 1..n. \text{ of-nat } (n + 1 \text{ choose } k) \ast a^\underline{k} \ast b^{\underline{n + 1 - k}})\)
    by (auto simp add: field-simps sum.distrib [symmetric] choose-reduce-nat)
  also have \(\ldots = (\sum k \leq n+1. (n + 1 \text{ choose } k) \ast a^\underline{k} \ast b^{\underline{n + 1 - k}})\)
using decomp by (simp add: atMost-atLeast0 field-simps)
finally show ?case
  by simp
qed

Original version for the naturals.
corollary binomial: \((a + b :: nat)^n = (\sum k \leq n. \text{of-nat (n choose k)} \cdot a^k \cdot \text{b}^{(n - k)})\)
using binomial-ring [of int a int b n]

lemma binomial-fact-lemma: \(k \leq n \implies \text{fact k} \cdot \text{fact (n - k)} \cdot \text{(n choose k)} = \text{fact n}\)
proof (induct n arbitrary; k rule: nat-less-induct)
  fix n k
  assume H: \(\forall m < n. \forall x \leq m. \text{fact x} \cdot \text{fact (m - x)} \cdot (m \text{ choose x}) = \text{fact m}\)
  assume kn: \(k \leq n\)
  let ?ths = \(\text{fact k} \cdot \text{fact (n - k)} \cdot (n \text{ choose k}) = \text{fact n}\)
  consider n = 0 \lor k = 0 \lor n = k \mid \text{m h where n = Suc m k = Suc h h < m}
  using kn by atomize-elim presburger
  then show \(\text{fact k} \cdot \text{fact (n - k)} \cdot (n \text{ choose k}) = \text{fact n}\)
  proof cases
    case 1
    with kn show ?thesis by auto
  next
    case 2
    note n = \(\text{n = Suc m}\)
    note k = \(\text{k = Suc h}\)
    note hm = \(\text{h < m}\)
    have mn: \(m < n\)
      using n by arith
    have hm: \(h \leq m\)
      using hm by arith
    have km: \(k \leq m\)
      using km k n kn by arith
    have m - h = Suc (m - Suc h)
      using km km hm by arith
    with km k have \(\text{fact (m - h)} = (m - h) \cdot \text{fact (m - k)}\)
      by simp
    with n k have \(\text{fact k} \cdot \text{fact (n - k)} \cdot (n \text{ choose k}) = \text{k} \cdot (\text{fact h} \cdot (m - h) \cdot (m \text{ choose h})) + \text{(m - h)} \cdot (\text{fact k} \cdot \text{fact (m - k)} \cdot (m \text{ choose k}))\)
      by (simp add: field-simps)
    also have \(\ldots = (k + (m - h)) \cdot \text{fact m}\)
      using H[rule-format, OF mn hm'] H[rule-format, OF mn km]
      by (simp add: field-simps)
  finally show ?thesis
  using k n km by simp
lemma binomial-fact':
  assumes k ≤ n
  shows \( \binom{n}{k} = \frac{\text{fact } n}{\text{fact } k \cdot \text{fact } (n - k)} \)
  using binomial-fact-lemma [OF assms]
  by (metis fact-nonzero mult-eq-0-iff nonzero-mult-div-cancel-left)

lemma binomial-fact:
  assumes kn: k ≤ n
  shows \( \text{of-nat } \binom{n}{k} = \frac{\text{fact } n}{\text{fact } k \cdot \text{fact } (n - k)} \)
  using binomial-fact-lemma [OF kn]
  by (metis (mono_tags, lifting) fact-nonzero mult-eq-0-iff nonzero-mult-div-cancel-left
                   of-nat-fact of-nat-mult)

lemma fact-binomial:
  assumes k ≤ n
  shows \( \text{fact } k \cdot \text{of-nat } \binom{n}{k} = \frac{\text{fact } n}{\text{fact } (n - k)} \)
  unfolding binomial-fact [OF assms]
  by (simp add: field-simps)

lemma binomial-fact-pow: \( \sum_{k \leq n} \binom{n}{k} \leq n^s \)
proof (cases s ≤ n)
  case True
  then show \?thesis
    by (smt (verit) binomial-fact-lemma mult assoc mult.commute
                     fact-div-fact-le-pow fact-nonzero nonzero-mult-div-cancel-right)
qde (simp add: binomial-eq-0)

lemma choose-two: \( \binom{n}{2} = n \cdot (n - 1) \div 2 \)
proof (cases n ≥ 2)
  case False
  then have n = 0 ∨ n = 1
    by auto
  then show \?thesis by auto
next
  case True
  define m where m = n - 2
  with True have n = m + 2
    by simp
  then have \( \text{fact } n = m \cdot (m - 1) \cdot \text{fact } (n - 2) \)
    by (simp add: fact-prod-Suc atLeast0-lessThan-Suc algebra-simps)
  with True show \?thesis
    by (simp add: binomial-fact')
qde

lemma choose-row-sum: \( \sum_{k \leq n} \binom{n}{k} = 2^n \)
using binomial [of 1 1 n] by (simp add: numeral-2-eq-2)
lemma sum-choose-lower: \( \sum_{k \leq n. \ (r+k) \ choose \ k} = \text{Suc} \ (r+n) \ choose \ n \)
by (induct n) auto

lemma sum-choose-upper: \( \sum_{k \leq n. \ k \ choose \ m} = \text{Suc} \ n \ choose \ \text{Suc} \ m \)
by (induct n) auto

lemma choose-alternating-sum:
\[ n > 0 \implies (\sum_{i \leq n. \ (-1)^i \ * \ of-nat \ (n \ choose \ i)}) = (0 :: 'a::comm-ring-1) \]
using binomial-ring[of \ -1 :: 'a 1 n]
by (simp add: atLeast0AtMost mult-of-nat-commute zero-power)

lemma choose-even-sum:
assumes \( n > 0 \)
shows \( 2 \ * (\sum_{i \leq n. \ \text{if even } i \ then \ of-nat \ (n \ choose \ i) \ else \ 0}) = (2 ^ n :: 'a::comm-ring-1) \)
proof
have \( 2 ^ n = (\sum_{i \leq n. \ \text{of-nat} \ (n \ choose \ i)}) - (\sum_{i \leq n. \ (-1) ^ i \ * \ of-nat \ (n \ choose \ i)}) \)
using choose-row-sum[of \ n]
by (simp add: choose-alternating-sum assms atLeast0AtMost of-nat-sum[ symmetric])
also have \( \ldots = 2 ^ n \)
by (simp add: sum-subtractf)
also have \( \ldots = 2 ^ n \)
by (subst sum-distrib-left, intro sum.cong) simp-all
finally show \( \ldots \)
qed

lemma choose-odd-sum:
assumes \( n > 0 \)
shows \( 2 ^ n \ * (\sum_{i \leq n. \ \text{if odd } i \ then \ of-nat \ (n \ choose \ i) \ else \ 0}) = (2 ^ n :: 'a::comm-ring-1) \)
proof
have \( 2 ^ n = (\sum_{i \leq n. \ \text{of-nat} \ (n \ choose \ i)}) - (\sum_{i \leq n. \ (-1) ^ i \ * \ of-nat \ (n \ choose \ i)}) \)
using choose-row-sum[of \ n]
by (simp add: choose-alternating-sum assms atLeast0AtMost of-nat-sum[ symmetric])
also have \( \ldots = 2 ^ n \)
by (simp add: sum-subtractf)
also have \( \ldots = 2 ^ n \)
by (subst sum-distrib-left, intro sum.cong) simp-all
finally show \( \ldots \)
qed

NW diagonal sum property

lemma sum-choose-diagonal:
assumes \( m \leq n \)
shows \( \sum_{k \leq m. \ (n - k) \ choose \ (m - k)} = \text{Suc} \ n \ choose \ m \)
proof
have \( \sum_{k \leq m. \ (n - k) \ choose \ (m - k)} = \sum_{k \leq m. \ (n - m + k) \ choose \ k} \)

NW diagonal sum property
using sum.atLeastAtMost-rev [of λk. (n - k) choose (m - k) 0 m] assms
by (simp add: atMost-atLeast0)
also have ... = Suc (n - m + m) choose m
by (rule sum-choose-lower)
also have ... = Suc n choose m
using assms by simp
finally show ?thesis.
qed

93.3 Generalized binomial coefficients

definition gbinomial :: 'a::{semidom-divide,semiring-char-0} ⇒ nat ⇒ 'a (infixl gchoose 65)
where gbinomial-prod-rev: a gchoose k = prod (λi. a - of_nat i) {0..<k} div fact k

lemma gbinomial-0 [simp]:
a gchoose 0 = 1
0 gchoose (Suc k) = 0
by (simp-all add: gbinomial-prod-rev prod.atLeast0-lessThan-Suc-shift del: prod.op-ivl-Suc)

lemma gbinomial-Suc: a gchoose (Suc k) = prod (λi. a - of_nat i) {0..k} div fact (Suc k)
by (simp add: gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost)

lemma gbinomial-1 [simp]: a gchoose 1 = a
by (simp add: gbinomial-prod-rev lessThan-Suc)

lemma gbinomial-Suc0 [simp]: a gchoose Suc 0 = a
by (simp add: gbinomial-prod-rev lessThan-Suc)

lemma gbinomial-mult-fact: fact k * (a gchoose k) = (∏ i = 0..<k. a - of_nat i)
for a :: 'a::field-char-0
by (simp-all add: gbinomial-prod-rev field-simps)

lemma gbinomial-mult-fact': (a gchoose k) * fact k = (∏ i = 0..<k. a - of_nat i)
for a :: 'a::field-char-0
using gbinomial-mult-fact [of k a] by (simp add: ac-simps)

lemma gbinomial-pochhammer: a gchoose k = (−1) ^ k * pochhammer (− a) k / fact k
for a :: 'a::field-char-0
proof (cases k)
case (Suc k)
then have a gchoose k = pochhammer (a - of_nat k') (Suc k') / ((1 + of_nat k') * fact k')
by (simp add: gbinomial-prod-rev pochhammer-prod-rev prod.atLeastLessThanSuc-atLeastAtMost
prod.atLeast-Suc-atMost-Suc-shift of-nat-diff flip: power-mult-distrib prod.cl-ivl-Suc)
then show ?thesis
by (simp add: pochhammer-minus Suc)
qed auto

lemma gbinomial-pochhammer': a gchoose k = pochhammer (a - of-nat k + 1) k / fact k
  for a :: 'a::field-char-0
proof
  have a gchoose k = ((-1) ^ k * (-1) ^ k) * pochhammer (a - of-nat k + 1) k / fact k
    by (simp add: gbinomial-pochhammer pochhammer-minus mult-ac)
  also have (-1 :: 'a) ^ k * (-1) ^ k = 1
    by (subst power-add [symmetric]) simp
  finally show ?thesis
    by simp
qed

lemma gbinomial-binomial: n gchoose k = n choose k
proof (cases k ≤ n)
  case False
  then have n < k
    by (simp add: not-le)
  then have 0 ∈ ((-1) n) ' {0..<k}
    by auto
  then have prod ((-1) n) {0..<k} = 0
    by (auto intro: prod-zero)
  with n < k show ?thesis
    by (simp add: gbinomial-prod-rev prod-zero)
next
  case True
  define m where m = n - k
  with True have *: prod ((-1) n) {0..<k} = \[\prod_{i = 0..<m+k. of-nat (m+k-i)}\] auto
  from True have n choose k = fact n div (fact k * fact (n - k))
    by (rule binomial-fact')
  with * show ?thesis
    by (simp add: gbinomial-prod-rev mult.commute [of fact k] div-mult2-eq fact-div-fact)
qed

lemma of-nat-gbinomial: of-nat (n gchoose k) = (of-nat n gchoose k :: 'a::field-char-0)
proof (cases k ≤ n)
  case False
  then show ?thesis
    by (simp add: not-le gbinomial-binomial binomial-eq-0 gbinomial-prod-rev)
next
  case True
  define m where m = n - k
  with True have n = m + k
    by arith
  from n have fact n = ((\[\prod_{i = 0..<m+k. of-nat (m+k-i)}\] :: 'a)
    by (simp add: fact-prod-rev)
also have \( \ldots = \big( \prod_{i \in \{0..<k\} \cup \{k..<m+k\} \} \text{of-nat} (m+k-i) \big) :: \alpha \)  
by (simp add: idl-disj-un)

finally have fact \( n = \big( \text{fact} m \ast \big( \prod_{i = 0..<k} \text{of-nat} m \ast \text{of-nat} k \ast \text{of-nat} i \big) \big) :: \alpha \)

by (simp add: ivl-disj-un)

using \( \text{prod.shift-bounds-nat-idl} \ [\alpha \cdot \text{of-nat} (m+k-i) :: \alpha 0 k m] \)

by (simp add: fact-prod-rev [of m]
prod.union-disjoint of-nat-diff)

then have fact \( n / \text{fact} (n-k) = \big( \prod_{i = 0..<k} \text{of-nat} (n-k) \big) :: \alpha \)

by (simp add: field-simps gbinomial-pochhammer)

then show \( \text{?thesis} \)
by simp

d qed

lemma gbinomial-gbinomial: \( \text{of-nat} (n \text{ choose} k) = (\text{of-nat} n \text{ gchoose} k :: \alpha::\text{field-char-0}) \)

by (simp add: gbinomial-binomial [symmetric] of-nat-gbinomial)

setup “Sign.add-const-constraint (const-name \\text{\small gbinomial}, SOME typ (\alpha::\text{field-char-0} \Rightarrow \text{nat} \Rightarrow \alpha))”

lemma gbinomial-mult-1:
fixes \( a :: \alpha::\text{field-char-0} \)

shows \( a \ast (a \text{ gchoose} k) = \text{of-nat} k \ast (a \text{ gchoose} k) + \text{of-nat} (\text{Suc} k) \ast (a \text{ gchoose} (\text{Suc} k)) \)

(is \( ?l = ?r \))

proof –

have \( ?r = (\big(-1\big)^k \ast \text{pochhammer} (a) \ast \text{fact} k) \ast (\text{of-nat} (\text{Suc} k) + (\text{Suc} k) \text{ gchoose} k) \)

unfolding gbinomial-pochhammer pochhammer-Suc right-diff-distrib power-Suc

by (auto simp add: field-simps simp del: of-nat-Suc)

also have \( \ldots = ?l \)

by (simp add: field-simps gbinomial-pochhammer)

finally show \( \text{?thesis} \)

d qed

lemma gbinomial-mult-1′:
\( (a \text{ gchoose} k) \ast a = \text{of-nat} k \ast (a \text{ gchoose} k) + \text{of-nat} (\text{Suc} k) \ast (a \text{ gchoose} (\text{Suc} k)) \)

for \( a :: \alpha::\text{field-char-0} \)

by (simp add: mult.commute gbinomial-mult-1)

lemma gbinomial-Suc-Suc: \( (a + 1) \text{ gchoose} (\text{Suc} k) = a \text{ gchoose} k + (a \text{ gchoose} (\text{Suc} k)) \)

for \( a :: \alpha::\text{field-char-0} \)

proof (cases k)

case 0
then show \( \text{thesis} \) by simp

next

\textbf{case} \((\text{Suc } h)\)

\textbf{have} eq0: \((\prod i \in \{1..k\}. (a + 1) - \text{of-nat } i) = (\prod i \in \{0..h\}. a - \text{of-nat } i)\)

\textbf{proof} (rule prod.reindex-cong)

\textbf{show} \({\{1..k\} = \text{Suc } ' \{0..h\}}\)

\textbf{using} \text{Suc} \textbf{by} (auto simp add: image-Suc-atMost)

\textbf{qed} auto

\textbf{have} \ldots = \(\text{(a choose Suc } h) \times \text{of-nat } (\text{Suc } (Suc \ h)) + (\prod i = 0..Suc \ h. a - \text{of-nat } i)\)

\textbf{apply} (simp only: \text{gbinomial-mult-fact field-simps mult.left-commute [of - 2]})

\textbf{apply} (simp del: \text{fact-Suc add: fact-Suc [of Suc } h\text{] field-simps gbinomial-mult-fact mult.left-commute [of - 2] atLeastLessThanSuc-atLeastAtMost})

\textbf{done}

\textbf{also have} \ldots = \(\text{(fact } (Suc \ h) \times (a choose Suc } h) \times \text{of-nat } (Suc \ (Suc \ h)) + (\prod i = 0..Suc \ h. a - \text{of-nat } i)\)

\textbf{by} (simp only: \text{fact-Suc mult.commute mult.left-commute of-nat-fact of-nat-id of-nat-mult})

\textbf{also have} \ldots = \(\text{of-nat } (Suc \ (Suc \ h)) \times (\prod i = 0..h. a - \text{of-nat } i) + (\prod i = 0..Suc \ h. a - \text{of-nat } i)\)

\textbf{unfolding} \text{gbinomial-mult-fact atLeastLessThanSuc-atLeastAtMost} \textbf{by} auto

\textbf{also have} \ldots = \(\prod i = 0..Suc \ h. a - \text{of-nat } i) + (\text{of-nat } h \times (\prod i = 0..h. a - \text{of-nat } i) + 2 * (\prod i = 0..h. a - \text{of-nat } i)\)

\textbf{by} (simp add: field-simps)

\textbf{also have} \ldots = \(\text{(a choose Suc } h) \times (\text{fact } (Suc \ h)) \times (\text{of-nat } (Suc \ k)) + (\prod i \in \{0..Suc \ h\}. a - \text{of-nat } i)\)

\textbf{unfolding} \text{gbinomial-mult-fact'} \textbf{by} (simp add: comm-semiring-class.distrib field-simps Suc atLeastLessThanSuc-atLeastAtMost)

\textbf{also have} \ldots = \(\prod i \in \{0..k\}. (a + 1) - \text{of-nat } i\)

\textbf{unfolding} \text{gbinomial-mult-fact'} \textbf{atLeast0-atMost-Suc} \textbf{by} (simp add: field-simps Suc atLeastLessThanSuc-atLeastAtMost)

\textbf{also have} \ldots = \(\prod i \in \{0..k\}. (a + 1) - \text{of-nat } i\)

\textbf{using} eq0 \textbf{by} (simp add: Suc prod.atLeast0-atMost-Suc-shift del: prod.cl-iel-Suc)

\textbf{also have} \ldots = \(\text{(fact } (Suc \ k)) \times ((a + 1) \text{ gchoose } (Suc \ k))\)

\textbf{by} (simp only: \text{gbinomial-mult-fact atLeastLessThanSuc-atLeastAtMost})

\textbf{finally show} \text{thesis}

\textbf{using} \text{fact-nonzero [of Suc } k\text{] by} auto

\textbf{qed}

\textbf{lemma} \text{gbinomial-reduce-nat:} \(0 < k \Rightarrow a \text{ gchoose } k = (a - 1) \text{ gchoose } (k - 1)\)
+ ((a - 1) \text{ choose } k) \\
{\text{for } a :: 'a::field-char-0} \\
{\text{by (metis Suc-pred' diff-add-cancel gbinomial-Suc-Suc)}}

\textbf{lemma} \text{ choose-row-sum-weighted}: \\
(\sum k = 0..m. (r \text{ choose } k) * (r/2 - \text{ of-nat } k)) = \text{ of-nat}(\text{Suc } m) / 2 * (r \text{ choose } (\text{Suc } m)) \\
{\text{for } r :: 'a::field-char-0} \\
{\text{by (induct } m) (simp-all add: field-simps distrib gbinomial-mult-1)}

\textbf{lemma} \text{ binomial-symmetric}: \\
\text{assumes } kn: k \leq n \\
\text{shows } n \text{ choose } k = n \text{ choose } (n - k) \\
\text{proof} - \\
\text{have } kn': n - k \leq n \\
\text{using } kn \text{ by arith} \\
\text{from binomial-fact-lemma[OF } kn] \text{ binomial-fact-lemma[OF } kn'] \\
\text{have } fact k * fact (n - k) * (n \text{ choose } k) = fact (n - k) * fact (n - (n - k)) \\
* (n \text{ choose } (n - k)) \\
\text{by simp} \\
\text{then show } \text{?thesis} \\
\text{using } kn \text{ by simp} \\
\text{qed}

\textbf{lemma} \text{ choose-linear-sum}: \\
(\sum j \leq m. ((n + j) \text{ choose } n)) = ((n + m + 1) \text{ choose } (n + 1)) \\
(\sum j \leq m. ((n + j) \text{ choose } n)) = ((n + m + 1) \text{ choose } m) \\
\text{proof} - \\
\text{show } (\sum j \leq m. ((n + j) \text{ choose } n)) = ((n + m + 1) \text{ choose } (n + 1)) \\
\text{by (induct } m) \text{ simp-all} \\
\text{also have } \ldots = (n + m + 1) \text{ choose } m \\
\text{by (subst binomial-symmetric) simp-all} \\
\text{finally show } (\sum j \leq m. ((n + j) \text{ choose } n)) = (n + m + 1) \text{ choose } m . \\
\text{qed}

\textbf{lemma} \text{ choose-linear-sum}: (\sum i \leq n. i * (n \text{ choose } i)) = n * 2 ^ (n - 1) \\
\text{proof} (\text{cases } n) \\
\text{case } 0 \\
\text{then show } \text{?thesis by simp} \\
\text{next} \\
\text{case } (\text{Suc } m) \\
\text{have } (\sum i \leq n. i * (n \text{ choose } i)) = (\sum i \leq \text{Suc } m. i * (\text{Suc } m \text{ choose } i)) \\
\text{by (simp add: Suc) } \\
\text{also have } \ldots = \text{Suc } m * 2 ^ m \\
\text{unfolding sum.atMost-Suc-shift Suc-times-binomial sum-distrib-left[ symmetric]} \\
\text{by (simp add: choose-row-sum)} \\
\text{finally show } \text{?thesis} \\
\text{using } \text{Suc by simp} \\
\text{qed}
lemma choose-alternating-linear-sum:

assumes $n \neq 1$

shows $(\sum_{i \leq n} (-1)^i \cdot \text{ofo-nat } i \cdot \text{ofo-nat } (n \text{ choose } i)) :: \text{ 'a::comm-ring-1} = 0$

proof (cases $n$

  case $0$

  then show ?thesis by simp

next

  case $(\text{Suc } m)$

  with assms have $m > 0$

  by simp

  have $(\sum_{i \leq n} (-1)^i \cdot \text{ofo-nat } i \cdot \text{ofo-nat } (n \text{ choose } i)) :: \text{ 'a} =$

  $(\sum_{i \leq \text{Suc } m} (-1)^i \cdot \text{ofo-nat } i \cdot \text{ofo-nat } (\text{Suc } m \text{ choose } i))$

  by (simp add: Suc)

  also have $\ldots = (\sum_{i \leq m} (-1)^i \cdot \text{ofo-nat } (\text{Suc } i) \cdot \text{ofo-nat } (\text{Suc } i \cdot (\text{Suc } m \text{ choose } \text{Suc } i)))$

  by (simp only: sum.atMost-Suc-shift sum-distrib-left[| symmetric |] mult-ac of-ofo-nat-mult)

  simp

  also have $\ldots = \text{ofo-nat } (\text{Suc } m) \cdot (\sum_{i \leq m} (-1)^i \cdot \text{ofo-nat } (m \text{ choose } i))$

  by (subst sum-distrib-left, rule sum.cong[OF refl], subst Suc-times-binomial[

  simp add: algebra-simps sum.distrib Suc-diff-le])

  also have $(\sum_{i \leq m} (-1)^i \cdot \text{ofo-nat } (m \text{ choose } i)) = 0$

  using choose-alternating-sum[OF $\langle m > 0 \rangle$] by simp

  finally show ?thesis

  by simp

qed

lemma vandermonde: $(\sum_{k \leq r} (m \text{ choose } k) \cdot (n \text{ choose } (r - k))) = (m + n)$

choose $r$

proof (induct $n$ arbitrary: $r$

  case $0$

  have $(\sum_{k \leq r} (m \text{ choose } k) \cdot (\text{0 \text{ choose } (r - k)})) = (\sum_{k \leq r} \text{if } k = r \text{ then } (m \text{ choose } k) \text{ else } 0)$

  by (intro sum.cong) simp-all

  also have $\ldots = m \text{ choose } r$

  by simp

  finally show ?case

  by simp

next

  case $(\text{Suc } n \cdot r)$

  show ?case

  by (cases $r$) (simp-all add: Suc[| symmetric |] algebra-simps sum.distrib Suc-diff-le)

qed

lemma choose-square-sum: $(\sum_{k \leq n} (n \text{ choose } k)^2) = ((2 \cdot n) \text{ choose } n)$

using vandermonde$[\text{of } n \cdot n]$

by (simp add: power2-eq-square mult-2 binomial-symmetric[| symmetric |])

lemma pochhammer-binomial-sum:
fixes a b :: 'a::comm-ring-1

shows pochhammer (a + b) n = (∏ k≤n. of-nat (n choose k) * pochhammer a k * pochhammer b (n – k))

proof (induction n arbitrary: a b)

case 0

then show ?case by simp

next
case (Suc n a b)

have (∑ k≤Suc n. of-nat (Suc n choose k) * pochhammer a k * pochhammer b (Suc n – k)) =
    (∑ i≤n. of-nat (n choose i) * pochhammer a (Suc i) * pochhammer b (n – i)) +
    ((∑ i≤n. of-nat (n choose Suc i) * pochhammer a (Suc i) * pochhammer b (n – i)) +
      pochhammer b (Suc n))

  by (subst sum.atMost-Suc-shift) (simp add: ring-distrib sum.distrib)

also have (∑ i≤n. of-nat (n choose i) * pochhammer a (Suc i) * pochhammer b (Suc n – i)) =
        a * pochhammer ((a + 1) + b) n

  by (subst Suc) (simp add: sum-distrib-left pochhammer-rec mult-ac)

also have (∑ i≤n. of-nat (n choose Suc i) * pochhammer a (Suc i) * pochhammer b (n – i)) +
      pochhammer b (Suc n) =
    (∑ i=0..Suc n. of-nat (n choose i) * pochhammer a i * pochhammer b (Suc n – i))

    apply (subst sum.atLeast-Suc-atMost, simp)

    apply (simp add: sum.shift-bounds-cl-Suc-atMost del: sum.cl-ivl-Suc)

    done

also have . . . = (∑ i≤n. of-nat (n choose i) * pochhammer a i * pochhammer b (Suc n – i))

  using Suc by (intro sum.mono-neutral-right) (auto simp: not-le binomial-eq-0)

also have . . . = (∑ i≤n. of-nat (n choose i) * pochhammer a i * pochhammer b (Suc n – i))

  by (intro sum.cong) (simp-all add: Suc-diff-le)

also have . . . = b * pochhammer ((a + 1) + b) n

  by (subst Suc) (simp add: sum-distrib-left mult-ac pochhammer-rec)

also have a * pochhammer ((a + 1) + b) n + b * pochhammer ((a + 1) + b) n =

  pochhammer (a + b) (Suc n)

  by (simp add: pochhammer-rec algebra-simps)

finally show ?case ..

qed

Contributed by Manuel Eberl, generalised by LCP. Alternative definition of the binomial coefficient as ∏ i<k. (n – i) / (k – i).

lemma gbinomial-altdef-of-nat: a choose k = (∏ i = 0..<k. (a – of-nat i) / of-nat (k – i)) :: 'a

for k :: nat and a :: 'a::field-char-0

by (simp add: prod-dividef gbinomial-prod-rev fact-prod-rev)
lemma gbinomial-ge-n-over-k-pow-k:
  fixes k :: nat
  and a :: 'a::linordered-field
  assumes of-nat k ≤ a
  shows (a / of-nat k :: 'a) ^ k ≤ a \choose k
proof –
  have x: 0 ≤ a
    using assms of-nat-0-le-iff order-trans by blast
  have (a / of-nat k :: 'a) ^ k = (∏ i = 0..< k. a / of-nat k :: 'a)
    by simp
  also have . . . ≤ a \choose k
proof –
  have \(\forall i. i < k \Rightarrow 0 \leq a / of-nat k\)
    by (simp add: x zero-le-divide-iff)
  moreover have a / of-nat k ≤ (a - of-nat i) / of-nat (k - i) if i < k for i
proof –
  from assms have a * of-nat i ≥ of-nat (i * k)
    by (metis mult.commute mult-le-cancel-right of-nat-less-0-iff of-nat-mult)
  then have a * of-nat k - a * of-nat i ≤ a * of-nat k - of-nat (i * k)
    by arith
  then have a * of-nat (k - i) ≤ (a - of-nat i) * of-nat k
    using (i < k) by (simp add: algebra-simps zero-less-mult-iff of-nat-diff)
  then have a * of-nat (k - i) ≤ (a - of-nat i) * (of-nat k :: 'a)
    by blast
  with assms show \(?\)thesis
    using (i < k) by (simp add: field-simps)
  qed
  ultimately show \(?\)thesis
  unfolding gbinomial-altdef-of-nat
  by (intro prod-mono) auto
  qed
finally show \(?\)thesis.
qed

lemma gbinomial-negated-upper: (a \choose k) = (-1) ^ k * ((of-nat k - a - 1) \choose k)
  by (simp add: gbinomial-pochhammer pochhammer-minus algebra-simps)

lemma gbinomial-minus: ((-a) \choose k) = (-1) ^ k * ((a + of-nat k - 1) \choose k)
  by (subst gbinomial-negated-upper) (simp add: add-ac)

lemma Suc-times-gbinomial: of-nat (Suc k) * ((a + 1) \choose (Suc k)) = (a + 1) * (a \choose k)
proof (cases k)
  case 0
  then show \(?\)thesis by simp
next
THEORY "Binomial"

case (Suc b)
then have \((a + 1) \text{ choose } \text{Suc } b\) = \((\prod i = 0..\text{Suc } b. \ a + \ (1 - \text{of-nat } i)) \ / \ \text{fact } (b + 2)\)
  by (simp add: field-simps gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost)
also have \((\prod i = 0..\text{Suc } b. \ a + \ (1 - \text{of-nat } i)) = (a + 1) * (\prod i = 0..b. \ a - \text{of-nat } i)\)
  by (simp add: prod.atLeast0-atMost-Suc-shift del: prod.cl-ivl-Suc)
also have \(\ldots \ / \ \text{fact } (b + 2) = (a + 1) / \ \text{of-nat } (\text{Suc } b) * (a \text{ choose } \text{Suc } b)\)
  by (simp-all add: gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost)
finally show ?thesis by (simp add: Suc)
qed

lemma gbinomial-factors: \((a + 1) \text{ choose } \text{Suc } k\) = \((a + 1) / \ \text{of-nat } (\text{Suc } k)\) * \(a \text{ choose } k\)
proof (cases k)
case 0
then show ?thesis by simp
next
case (Suc b)
then have \((a + 1) \text{ choose } \text{Suc } b\) = \((\prod i = 0..\text{Suc } b. \ a + \ (1 - \text{of-nat } i)) / \ \text{fact } (b + 2)\)
  by (simp add: field-simps gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost)
also have \((\prod i = 0..\text{Suc } b. \ a + \ (1 - \text{of-nat } i)) = (a + 1) * (\prod i = 0..b. \ a - \text{of-nat } i)\)
  by (simp add: prod.atLeast0-atMost-Suc-shift del: prod.cl-ivl-Suc)
also have \(\ldots / \ \text{fact } (b + 2) = (a + 1) / \ \text{of-nat } (\text{Suc } b) * (a \text{ choose } \text{Suc } b)\)
  by (simp-all add: gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost)
finally show ?thesis by (simp add: Suc)
qed

lemma gbinomial-rec: \((a + 1) \text{ choose } \text{Suc } k\) = \((a \text{ choose } k) * ((a + 1) / \ \text{of-nat } (\text{Suc } k))\)
using gbinomial-mult-1[of a k]
  by (subst gbinomial-Suc-Suc) (simp add: field-simps del: of-nat-Suc, simp add: algebra-simps)

lemma gbinomial-of-nat-symmetric: \(k \leq n \implies \text{of-nat } n \text{ choose } k = \text{of-nat } n \text{ choose } (n - k)\)
using binomial-symmetric[of k n] by (simp add: binomial-gbinomial [symmetric])

The absorption identity (equation 5.5 [3, p. 157]):
\[
\binom{r}{k} = \frac{r}{k} \binom{r - 1}{k - 1}, \quad \text{integer } k \neq 0.
\]

lemma gbinomial-absorption': \(k > 0 \implies a \text{ choose } k = (a / \ \text{of-nat } k) * (a - 1\)
THEORY “Binomial”

\[ \binom{k-1}{k} \]

using \binom{-1}{k-1}
by (simp-all add: \binom{-1}{k-1} del: of-nat-Suc)

The absorption identity is written in the following form to avoid division by \(k\) (the lower index) and therefore remove the \(k \neq 0\) restriction [3, p. 157]:

\[
k\binom{r}{k} = r\binom{r-1}{k-1}, \quad \text{integer } k.
\]

\textbf{lemma} \ gbinomial-absorption: \ of-nat \ (Suc k) * (a \ gchoose Suc k) = a * ((a - 1) \ gchoose k)
using \ gbinomial-absorption[of Suc k a] by (simp add: field-simps del: of-nat-Suc)

The absorption identity for natural number binomial coefficients:

\textbf{lemma} \ binomial-absorption: \ Suc k * (n \ choose Suc k) = n * ((n - 1) \ choose k)
by (cases n) (simp-all add: binomial-eq-0 Suc-times-binomial del: binomial-Suc-Suc mult-Suc)

The absorption companion identity for natural number coefficients, following the proof by GKP [3, p. 157]:

\textbf{lemma} \ binomial-absorb-comp: \ (n - k) * (n \ choose k) = n * ((n - 1) \ choose k)
(is ?lhs = ?rhs)
\textbf{proof} (cases n \leq k)
case \ True
then show \ ?thesis by auto
next
case \ False
then have \ ?rhs = Suc ((n - 1) - k) * (n \ choose Suc ((n - 1) - k))
using \ binomial-symmetric[of k n - 1] \ binomial-absorption[of (n - 1) - k n]
by \ simp
also have \ Suc ((n - 1) - k) = n - k
using \ False \ by \ simp
also have \ n \ choose \ ... = n \ choose \ k
using \ False \ by \ (intro \ binomial-symmetric \ [symmetric]) \ simp-all
finally show \ ?thesis ..
qed

The generalised absorption companion identity:

\textbf{lemma} \ gbinomial-absorb-comp: \ (a - of-nat k) * (a \ gchoose k) = a * ((a - 1) \ gchoose k)
using \ gbinomial-pochhammer-absorb-comp[of a k] by (simp add: gbinomial-pochhammer)

\textbf{lemma} \ gbinomial-addition-formula:
\ a \ gchoose (Suc k) = ((a - 1) \ gchoose (Suc k)) + ((a - 1) \ gchoose k)
using \ gbinomial-Suc-Suc[of a - 1 k] by (simp add: algebra-simps)

\textbf{lemma} \ binomial-addition-formula:
THEORY “Binomial”

0 < n ⇒ n choose (Suc k) = ((n - 1) choose (Suc k)) + ((n - 1) choose k)
by (subst choose-reduce-nat) simp-all

Equation 5.9 of the reference material [3, p. 159] is a useful summation formula, operating on both indices:

\[ \sum_{k \leq n} \binom{r + k}{k} = \binom{r + n + 1}{n}, \text{ integer } n. \]

lemma gbinomial-parallel-sum: \( (\sum_{k \leq n} (a + \text{of-nat } k) \text{ gchoose } k) = (a + \text{of-nat } n + 1) \text{ gchoose } n \)
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc m)
  then show ?case
  using gbinomial-Suc-Suc[of (a + of-nat m + 1) m]
  by (simp add: add-ac)
qed

93.4 Summation on the upper index

Another summation formula is equation 5.10 of the reference material [3, p. 160], aptly named summation on the upper index:

\[ \sum_{0 \leq k \leq n} \binom{k}{m} = \binom{n + 1}{m + 1}, \text{ integers } m, n \geq 0. \]

lemma gbinomial-sum-up-index:
\( (\sum_{j = 0..n} (\text{of-nat } j \text{ gchoose } k) :: 'a::field-char-0) = (\text{of-nat } n + 1) \text{ gchoose } (k + 1) \)
proof (induct n)
  case 0
  show ?case
  using gbinomial-Suc-Suc[of 0 k]
  by (cases k) auto
next
  case (Suc n)
  then show ?case
  using gbinomial-Suc-Suc[of of-nat (Suc n) :: 'a k]
  by (simp add: add-ac)
qed

lemma gbinomial-index-swap:
\( ((-1) ^ k) * ((- (\text{of-nat } n) - 1) \text{ gchoose } k) = ((-1) ^ n) * ((- (\text{of-nat } k) - 1) \text{ gchoose } n) \)
is ?lhs = ?rhs
proof

have ?lhs = (of-nat (k + n) gchoose k)
  by (subst gbinomial-negated-upper) (simp add: power-mult-distrib [symmetric])
also have . . . = (of-nat (k + n) gchoose n)
  by (subst gbinomial-of-nat-symmetric) simp-all
also have . . . = ?rhs
  by (subst gbinomial-negated-upper) simp
finally show ?thesis.
qed

lemma gbinomial-sum-lower-neg: \( \sum_{k \leq m} (a \binom{k}{n}) \cdot (-1)^k = (-1)^m \cdot (a - 1 \binom{m+1}{n}) \)
proof
  have ?lhs = \( \sum_{k \leq m} - (a + 1) + of-nat k \binom{k}{n} \)
    by (intro sum.cong[OF refl]) (subst gbinomial-negated-upper, simp add: power-mult-distrib)
  also have . . . = \( a + of-nat m \binom{m}{n} \)
    by (subst gbinomial-parallel-sum) simp
  also have . . . = ?rhs
    by (subst gbinomial-negated-upper) (simp add: power-mult-distrib)
finally show ?thesis.
qed

lemma gbinomial-partial-row-sum:
\( \sum_{k \leq m} (0 a \binom{k}{n}) \cdot (a/2 - of-nat k) \)
proof (induct m)
  case 0
  then show ?case by simp
next
case (Suc mm)
  then have \( \sum_{k \leq Suc mm} (a \binom{k}{n}) \cdot ((a/2) - of-nat k) \)
    = \( (of-nat m + 1)/2 \cdot (a \binom{m+1}{n}) \)
    by (simp add: field-simps)
  also have . . . = \( a \cdot (a - 1 \binom{Suc mm}{n}) / 2 \)
    by (subst gbinomial-absorb-comp) (rule refl)
  also have . . . = \( (of-nat (Suc mm) + 1) / 2 \cdot (a \binom{Suc mm + 1}{n}) \)
    by (subst gbinomial-absorption [symmetric]) simp
finally show ?case.
qed

lemma sum-bounds-lt-plus1: \( \sum_{k < mm} f (Suc k) = (\sum_{k=1..mm} f k) \)
by (induct mm) simp-all

lemma gbinomial-partial-sum-poly:
\( \sum_{k \leq m} (of-nat m + a \binom{k}{n}) \cdot x^k \cdot y^{(m-k)} \)
= \( \sum_{k \leq m} (-a \binom{k}{n}) \cdot (-x)^k \cdot (x + y)^{(m-k)} \)
(is ?lhs m = ?rhs m)
proof (induction m)
case 0
  then show ?case by simp
next
  case (Suc mm)
  define G where \( G i k = (\text{of-nat } i + \text{a choose } k) \cdot x^k \cdot y^{(i - k)} \) for \( i \)
  define S where \( S = \text{?lhs} \)
  have \( \text{SG-def: } S = (\chi_i. (\sum k \leq i. (G i k))) \)
    unfolding \( \text{S-def G-def ..} \)

  have \( \text{S (Suc mm)} = G (\text{Suc mm}) 0 + (\sum k=\text{Suc 0..Suc mm}. G (\text{Suc mm}) k) \)
    using \( \text{SG-def by (simp add: sum.atLeast-Suc-atMost atLeast0AtMost [symmetric]} \)
  also have \( (\sum k=\text{Suc 0..Suc mm}. G (\text{Suc mm}) k) = (\sum k=0..\text{mm}. G (\text{Suc mm}) (\text{Suc k})) \)
    by (subst sum.shift-bounds-cl-Suc-ivl) simp
  also have \( \ldots = (\sum k=0..\text{mm}. ((\text{of-nat mm} + \text{a choose } (\text{Suc k}))) + \)
    \( (\text{of-nat mm} + \text{a choose } k) \cdot x^{(\text{Suc k})} \cdot y^{(\text{mm} - k)}) \)
    unfolding \( \text{G-def by (subst gbinomial-addition-formula) simp} \)
  also have \( \ldots = (\sum k=0..\text{mm}. (\text{of-nat mm} + \text{a choose } (\text{Suc k})) \cdot x^{(\text{Suc k})} \cdot y^{(\text{mm} - k)}) \)
    by [subst sum.distrib [symmetric]] (simp add: algebra-simps)
  also have \( (\sum k=0..\text{mm}. (\text{of-nat mm} + \text{a choose } \text{Suc k})) \cdot x^{(\text{Suc k})} \cdot y^{(\text{mm} - k)}) \)
    by (simp only: atLeast0AtMost lessThan-Suc-atMost)
  also have \( \ldots = (\sum k<\text{Suc mm}. (\text{of-nat mm} + \text{a choose } (\text{Suc k})) \cdot x^{(\text{Suc k})} \cdot y^{(\text{mm} - k)}) \)
    by simp: atLeast0AtMost lessThan-Suc-atMost
  also have \( ?A = (\sum k=1..\text{mm}. (\text{of-nat mm} + \text{a choose k}) \cdot x^{\text{k}} \cdot y^{(\text{mm} - k - 1)}) \)
  proof (subst sum.bounds-lt-plus1 [symmetric], intro cong[OF refl], clarify)
    fix k
    assume \( k < \text{mm} \)
    then have \( \text{mm} - k = \text{mm} - \text{Suc k + 1} \)
      by linarith
    then show \( (\text{of-nat mm} + \text{a choose } \text{Suc k}) \cdot x^{(\text{Suc k})} \cdot y^{(\text{mm} - k)} = \)
      \( (\text{of-nat mm} + \text{a choose } \text{Suc k}) \cdot x^{(\text{Suc k})} \cdot y^{(\text{mm} - \text{Suc k + 1})} \)
      by (simp only:)
    qed
  also have \( \ldots + ?B = y \cdot (\sum k=1..\text{mm}. (\text{G mm k}) + (\text{of-nat mm} + \text{a choose } (\text{Suc mm})) \cdot x^{(\text{Suc mm})} \)
    unfolding \( \text{G-def by (subst sum.distrib-left) simp add: algebra-simps} \)
  also have \( (\sum k=0..\text{mm}. (\text{of-nat mm} + \text{a choose k}) \cdot x^{(\text{Suc k})} \cdot y^{(\text{mm} - k)}) \)
    \( = x \cdot (\text{S mm}) \)
    unfolding \( \text{S-def by (subst sum.distrib-left) simp add: atLeast0AtMost algebra-simps} \)
  also have \( (G (\text{Suc mm}) 0) = y \cdot (\text{G mm 0}) \)
proof

finally have \( S (\text{Suc } mm) = \)
\( y \ast (G mm \theta + (\sum k=1..mm. (G mm k))) + (\text{of-nat } mm + a \text{ gchoose (Suc } mm)) \ast x \cdot \langle\text{Suc } mm\rangle + x \ast (S mm) \)

by (simp add: ring-distrib)
also have \( G mm \theta + (\sum k=1..mm. (G mm k)) = S mm \)

by (simp add: sum.atLeast-Suc-atMost[symmetric] G-def atLeast0AtMost)

finally have \( S (\text{Suc } mm) = (x + y) \ast (S mm) + (\text{of-nat } mm + a \text{ gchoose (Suc } mm)) \ast x \cdot \langle\text{Suc } mm\rangle \)

by (simp add: algebra-simps)
also have \( (\text{of-nat } mm + a \text{ gchoose (Suc } mm)) = (-1) \cdot \langle\text{Suc } mm\rangle \ast (- a \text{ gchoose (Suc } mm)) \)

by (subt gbinomial-negated-upper simp)
also have \( (-1) \cdot \langle\text{Suc } mm\rangle \ast (- a \text{ gchoose (Suc } mm)) \ast \langle\text{Suc } mm\rangle = \)
\( (- a \text{ gchoose (Suc } mm)) \ast \langle\text{Suc } mm\rangle \)

by (simp add: power-minus[simp only])
also have \( (x + y) \ast \langle\text{Suc } mm\rangle + \ldots = (x + y) \ast ?rhs mm + (- a \text{ gchoose (Suc } mm)) \ast \langle\text{Suc } mm\rangle \)

by (simp)
finally show \( ?\text{case} \)

by (simp only: S-def)

\[ \quad \] 

lemma gbinomial-partial-sum-poly-xpos:
\( \sum k \leq m. (\text{of-nat } m + a \text{ gchoose } k) \ast x \cdot k \ast y \cdot (m-k)) = \)
\( \sum k \leq m. (\text{of-nat } k + a - 1 \text{ gchoose } k) \ast x \cdot k \ast (x + y) \cdot (m-k)) \) (is \( ?\text{lhs} = ?\text{rhs} \))

proof
have \( ?\text{lhs} = \sum k \leq m. (- a \text{ gchoose } k) \ast \langle\text{Suc } mm\rangle \ast (- x) \cdot \langle\text{Suc } mm\rangle \ast \langle\text{Suc } mm\rangle \)

by (simp add: gbinomial-partial-sum-poly)
also have \( \ldots = \sum k \leq m. (-1) \cdot \langle\text{Suc } mm\rangle \ast (\text{of-nat } k + - a - 1 \text{ gchoose } k) \ast (- x) \cdot \langle\text{Suc } mm\rangle \)

by (metis (no-types, opaque-lifting) gbinomial-negated-upper)
also have \( \ldots = ?\text{rhs} \)

by (intro sum.cong) (auto simp flip; power-mult-distrib)
finally show \( \text{thesis} \)

\[ \quad \] 

lemma binomial-r-part-sum: \( \sum k \leq m. (2 \ast m + 1 \text{ choose } k) = 2 \cdot \langle\text{Suc } mm\rangle \)

proof
have \( 2 \ast 2 \cdot \langle\text{Suc } mm\rangle = \sum k = 0..(2 \ast m + 1). (2 \ast m + 1 \text{ choose } k) \)

using choose-rou-sum[where \( n=2 \ast m + 1 \)]
by (simp add: atMost-atLeast0)
also have \( \sum k = 0..(2 \ast m + 1). (2 \ast m + 1 \text{ choose } k) = \)
THEORY "Binomial"

(∑ k = 0..m. (2 * m + 1 choose k)) + 
(∑ k = m+1..2*m+1. (2 * m + 1 choose k))
using sum.ab-add-nat[of 0 m λk. 2 * m + 1 choose k m+1]
by (simp add: mult-2)
also have (∑ k = m+1..2*m+1. (2 * m + 1 choose k)) = 
(∑ k = 0..m. (2 * m + 1 choose (k + (m + 1))))
by (subst sum.shift-bounds-cl-nat-ivl [symmetric]) (simp add: mult-2)
also have ... = (∑ k = 0..m. (2 * m + 1 choose (m - k)))
by (intro sum.cong[OF refl], subst binomial-symmetric) simp-all
also have ... = (∑ k = 0..m. (2 * m + 1 choose k))
using sum.atLeastAtMost-rev [of λk. 2 * m + 1 choose (m - k) 0 m]
by simp
also have ... + ... = 2 * ...
by simp
finally show ?thesis
by (subst (asm mult-cancel1)) (simp add: atLeast0AtMost)
qed

lemma gbinomial-r-part-sum: (∑ k≤m. (2 * (of-nat m) + 1 gchoose k)) = 2 ^ (2 * m)
(is ?lhs = ?rhs)
proof -
have ?lhs = of-nat (∑ k≤m. (2 * m + 1) choose k)
  by (simp add: binomial-gbinomial add-ac)
also have ... = of-nat (2 ^ (2 * m))
  by (subst binomial-r-part-sum) (rule refl)
finally show ?thesis by simp
qed

lemma gbinomial-sum-nat-pow2:
(∑ k≤m. (of-nat (m + k) gchoose k :: 'a::field-char-0) / 2 ^ k) = 2 ^ m
(is ?lhs = ?rhs)
proof -
have 2 ^ m * 2 ^ m = (2 ^ (2*m) :: 'a)
  by (induct m) simp-all
also have ... = (∑ k≤m. (2 * (of-nat m) + 1 gchoose k))
  using gbinomial-r-part-sum..
also have ... = (∑ k≤m. (of-nat (m + k) gchoose k) * 2 ^ (m - k))
  using gbinomial.partial-sum-poly-xpos[where x=1 and y=1 and a=of-nat m + 1 and m=m]
  by (simp add: add-ac)
also have ... = 2 ^ m * (∑ k≤m. (of-nat (m + k) gchoose k) / 2 ^ k)
  by (subst sum-distrib-left) (simp add: algebra-simps power-diff)
finally show ?thesis
  by (subst (asm mult-left-cancel)) simp-all
qed

lemma gbinomial-trinomial-revision:
  assumes k ≤ m

THEORY “Binomial”

shows (a gchoose m) * (of-nat m gchoose k) = (a gchoose k) * (a − of-nat k gchoose (m − k))
proof −
  have (a gchoose m) * (of-nat m gchoose k) = (a gchoose m) * fact m / (fact k * fact (m − k))
    using assms by (simp add: binomial-gbinomial [symmetric] binomial-fact)
  also have . . . = (a gchoose k) * (a − of-nat k gchoose (m − k))
    using assms by (simp add: gbinomial-pochhammer power-diff pochhammer-product)
  finally show ?thesis .
qed

Versions of the theorems above for the natural-number version of *choose*

lemma binomial-altdef-of-nat:
  k ≤ n =⇒ of-nat (n choose k) = (∏ i = 0..<k. of-nat (n − i) / of-nat (k − i)) :: 'a
  for n k :: nat and x :: 'a:field-char-0
  by (simp add: gbinomial-altdef-of-nat binomial-gbinomial of-nat-diff)

lemma binomial-ge-n-over-k-pow-k: k ≤ n =⇒ (of-nat n / of-nat k :: 'a) ∨ k ≤ of-nat (n choose k)
  for k n :: nat and x :: 'a:linordered-field
  by (simp add: gbinomial-ge-n-over-k-pow-k binomial-gbinomial of-nat-diff)

lemma binomial-le-pow:
  assumes r ≤ n
  shows n choose r ≤ n ^ r
proof −
  have n choose r ≤ fact n div fact (n − r)
    using assms by (subst binomial-fact-lemma [symmetric]) auto
  with fact-div-fact-le-pow [OF assms] show ?thesis
    by auto
qed

lemma binomial-altdef-nat:
  k ≤ n =⇒ n choose k = fact n div (fact k * fact (n − k))
  for k n :: nat
  by (subst binomial-fact-lemma [symmetric]) auto

lemma choose-dvd:
  assumes k ≤ n shows fact k * fact (n − k) dvd (fact n :: 'a:linordered-semidom)
  unfolding dvd-def
proof
  show fact n = fact k * fact (n − k) * of-nat (n choose k)
    by (metis assms binomial-fact-lemma of-nat-fact of-nat-mult)
qed

lemma fact-fact-dvd-fact:
  fact k * fact n dvd (fact (k + n) :: 'a:linordered-semidom)
  by (metis add.commute add-diff-cancel-left choose-dvd le-add2)
lemma choose-mult-lemma:
\((m + r + k) \text{ choose } (m + k)) * ((m + k) \text{ choose } k) = ((m + r + k) \text{ choose } k) * ((m + r) \text{ choose } m)\)
(is ?lhs = -)
proof -
  have ?lhs =
    \(\frac{\text{fact } (m + r + k)}{\text{fact } (m + k) * \text{fact } (m + r - m)} * \left(\frac{\text{fact } (m + k)}{\text{fact } (m + r) * \text{fact } k * \text{fact } m}\right)\)
    by (simp add: binomial-altdef-nat)
  also have ... = \(\text{fact } (m + r + k) * \text{fact } (m + k)\)
    by (metis add-implies-diff add-le-mono1 choose-dvd diff-cancel2 div-mult-div-if-dvd le-add1 le-add2)
  also have ... = \(\text{fact } (m + r + k) * \text{fact } (m + r)\)
    by simp
  also have ... =
    \(\frac{\text{fact } (m + r + k)}{\text{fact } k * \text{fact } (m + r)} * \left(\frac{\text{fact } (m + r)}{\text{fact } k * \text{fact } m}\right)\)
    by (auto simp: div-mult-div-if-dvd fact-fact-dvd algebra-simps)
finally show ?thesis
  by (simp add: binomial-altdef-nat mult.commute)
qed

The 'Subset of a Subset' identity.

lemma choose-mult:
\(k \leq m \Longrightarrow m \leq n \Longrightarrow (n \text{ choose } m) * (m \text{ choose } k) = (n \text{ choose } k) * ((n - k) \text{ choose } (m - k))\)
using choose-mult-lemma [of m-k n-m k] by simp

lemma of-nat-binomial-eq-mult-binomial-Suc:
assumes \(k \leq n\)
shows \((\text{of-nat :: } (\text{nat } \Rightarrow ('a :: field-char-0))) (n \text{ choose } m) = \text{of-nat } (n + 1 - k) / \text{of-nat } (n + 1) * \text{of-nat } (\text{Suc } n \text{ choose } k)\)
proof (cases k)
case 0 then show ?thesis
  using of-nat-neq-0 by auto
next
case \(\text{Suc } l\)
  have \((\text{of-nat } (n + 1) * (\Pi_{i=0..k}. \text{of-nat } (n - i))) = (\text{of-nat :: } (\text{nat } \Rightarrow 'a)) (n + 1 - k) * (\Pi_{i=0..k}. \text{of-nat } (\text{Suc } n - i))\)
    using prod.atLeast0-lessThan-Suc [where ?'a = 'a, symmetric, of_a. of-nat (Suc n - i) k]
    by (simp add: ac-simps prod.atLeast0-lessThan-Suc-shift del: prod.op-iwl-Suc)
  also have ... = \((\text{of-nat :: } (\text{nat } \Rightarrow 'a)) (\text{Suc } n - k) * (\Pi_{i=0..k}. \text{of-nat } (\text{Suc } n - i))\)
THEORY “Binomial” 1879

by (simp add: Suc atLeast0-atMost-Suc atLeastLessThanSuc-atLeastAtMost)
also have \ldots = (of-nat :: (nat \Rightarrow 'a)) (n + 1 - k) * (\prod_{i=0..<k. of-nat (Suc n - i)''})
  by (simp only: Suc-eq-plus1)
finally have (\prod_{i=0..<k. of-nat (n - i)}) = (of-nat :: (nat \Rightarrow 'a)) (n + 1 - k)
/ of-nat (n + 1) * (\prod_{i=0..<k. of-nat (Suc n - i)})
using of-nat-neq-0 by (auto simp: mult.commute divide.simps)
with assms show \?thesis
by (simp add: binomial-altdef-of-nat prod-dividef)
qed

93.5 More on Binomial Coefficients

The number of nat lists of length \( m \) summing to \( N \) is \( N + m - 1 \) choose \( N \):

lemma card-length-sum-list-rec:
  assumes \( m \geq 1 \)
  shows \( \text{card \{l::nat list. length l = m \& sum-list l = N\}} = \)
  \( \text{card \{l. length l = (m - 1) \& sum-list l = N\}} + \)
  \( \text{card \{l. length l = m \& sum-list l + 1 = N\}} \)
  \( \text{(is \text{card \?C = card \?A + card \?B\)}} \)
proof
  let \?A' = \{l. length l = m \& sum-list l = N \& hd l = 0\}
  let \?B' = \{l. length l = m \& sum-list l = N \& hd l \neq 0\}
  let \?f = \lambda l. 0 \# l
  let \?g = \lambda l. (hd l + 1) \# tl l
  have 1: \( \text{xs \neq [] \Rightarrow x = hd xs \Rightarrow x \# tl xs = xs for x :: nat and xs\)}
    by simp
  have 2: \( \text{xs \neq [] \Rightarrow sum-list(tl xs) = sum-list xs - hd xs for xs :: nat list\)}
    by (auto simp add: neq-Nil-conv)
  have f: bij-betw \?f \?A \?A'
    by (rule bij-betw-byWitness[where \( f' = tl\)]) (use assms in auto simp: 2 1 simp flip: length-0-conv)
  have g: bij-betw \?g \?B \?B'
    apply (rule bij-betw-byWitness[where \( f' = \lambda l. (hd l - 1) \# tl l\)])
    using assms
    by (auto simp: 2 simp flip: length-0-conv intro!: 3)
  have fin: \( \text{finite \{xs. size xs = M \& set xs \subseteq \{0..<N\}\} for M N :: nat\)}
    using finite-lists-length-eq[OF finite-atLeastLessThan] conj-commute by auto
  have fin-A: finite \?A using \( \text{fin[of - N + 1]} \)
    by (intro finite-subset[where \?A = \?A and \?B = \{xs. size xs = m - 1 \& set xs \subseteq \{0..<N\+1\}\}])
    (auto simp: member-le-sum-list less-Suc-eq-le)
  have fin-B: finite \?B
    by (intro finite-subset[where \?A = \?B and \?B = \{xs. size xs = m \& set xs \subseteq \{0..<N\}\}])
    (auto simp: member-le-sum-list less-Suc-eq-le fin)
have uni: \(?C = ?A' \cup ?B'\)
by auto
have disj: \(?A' \cap ?B' = \{\}\) by blast
have card \(?C = \text{card}(\(?A' \cup ?B'\))\)
using uni by simp
also have \(\ldots = \text{card} \ ?A + \text{card} \ ?B\)
using card-Un-disjoint[OF \ ! \ - \ disj] bij-betw-finite[OF f] bij-betw-finite[OF g]
bij-betw-same-card[OF f] bij-betw-same-card[OF g] fin-A fin-B
by presburger
finally show \(?\text{thesis}\).
Qed

lemma \text{card-length-sum-list}: card \{l::nat list. size l = m \land \text{sum-list} l = N\} = (N + m - 1) choose N
— by Holden Lee, tidied by Tobias Nipkow

proof (cases \(m\))
  case 0
  then show \(?\text{thesis}\)
  by (cases N) (auto cong: conj-cong)
next
case (Suc \(m\))
  have \(m: m \geq 1\)
  by (simp add: Suc)
  then show \(?\text{thesis}\)
  proof (induct N + m - 1 arbitrary: N m)
    case 0 — In the base case, the only solution is [0].
    have \(\{l::nat list. \text{length} l = \text{Suc} 0 \land (\forall n \in \text{set} l. n = 0)\} = \{[0]\}\)
    by (auto simp: length-Suc-conv)
    have \(m = 1 \land N = 0\)
    using 0 by linarith
    then show \(?\text{case}\)
    by simp
next
case (Suc k)
  have \(c1: \text{card} \{l::nat list. \text{size} l = (m - 1) \land \text{sum-list} l = N\} = (N + (m - 1) - 1) \choose N\)
  proof (cases \(m = 1\))
    case True
    with Suc.hyps have \(N \geq 1\)
    by auto
    with True show \(?\text{thesis}\)
    by (simp add: binomial-eq-0)
next
case False
then show \(?\text{thesis}\)
  using Suc by fastforce
qed
from Suc have \(c2: \text{card} \{l::nat list. \text{size} l = m \land \text{sum-list} l + 1 = N\} =\)
(if \(N > 0\) then ((\(N - 1\) + \(m - 1\)) choose \((N - 1)\) else 0)
proof
  have *, n > 0 ⟹ Suc m = n ⟷ m = n − 1 for m n
  by arith

from Suc have N > 0 ⟹
  card {l::nat list. size l = m ∧ sum-list l + 1 = N} =
  ((N − 1) + m − 1) choose (N − 1)
  by (simp add: *)
then show ?thesis
  by auto
qed

from Suc.prems have
  card {l::nat list. size l = (m − 1) ∧ sum-list l = N} +
  card {l::nat list. size l = m ∧ sum-list l + 1 = N} = (N + m − 1)
choose N
  by (auto simp add: c1 c2 choose-reduce-nat)
then show ?thesis
  using card-length-sum-list-rec[OF Suc.prems] by auto
qed

lemma card-disjoint-shuffles:
  assumes set xs ∩ set ys = {}
  shows card (shuffles xs ys) = (length xs + length ys) choose length xs
using assms
proof
  (induction xs ys rule: shuffles.induct)
  case (3 x xs y ys)
  have shuffles (x # xs) (y # ys) = (#) x ' shuffles xs (y # ys) ∪ (#) y ' shuffles
    (x # xs) ys
    by (rule shuffles.simps)
  also have card . . . = card ((#) x ' shuffles xs (y # ys)) + card ((#) y ' shuffles
    (x # xs) ys)
    by (rule card-Un-disjoint) (insert 3.prems, auto)
  also have card ((#) x ' shuffles xs (y # ys)) = card (shuffles xs (y # ys))
    by (rule card-image) auto
  also have . . . = (length xs + length (y # ys)) choose length xs
    using 3.prems by (intro 3.IH) auto
  also have card ((#) y ' shuffles (x # xs) ys) = card (shuffles (x # xs) ys)
    by (rule card-image) auto
  also have . . . = (length (x # xs) + length ys) choose length (x # xs)
    using 3.prems by (intro 3.IH) auto
  also have length xs + length (y # ys) choose length xs + . . . =
    (length (x # xs) + length (y # ys)) choose length (x # xs) by simp
  finally show ?case .
qed

lemma Suc-times-binomial-add: Suc a * (Suc (a + b) choose Suc a) = Suc b *
  (Suc (a + b) choose a)
  — by Lukas Bulwahn
proof
  have dvd: Suc a * (fact a * fact b) dvd fact (Suc (a + b)) for a b
using fact-fact-dvd-fact[of Suc a b, where 'a=nat]
by (simp only: fact-Suc add-Suc[symmetric] of-nat-id mult.assoc)

have Suc a * (fact (Suc (a + b))) div (Suc a * fact a * fact b)) =
Suc a * fact (Suc (a + b)) div (Suc a * (fact a * fact b))
by (subst div-mult-swap[symmetric]; simp only: mult.assoc dvd)

also have . . . = Suc b * fact (Suc (a + b)) div (Suc b * (fact a * fact b))
using dvd[of b a] by (subst div-mult-swap[symmetric]; simp only: dvd)

finally show ☐thesis
by (subst (1 2) binomial-altdef-nat)
(simp-all only: ac-simps dvd)

qed

93.6 Inclusion-exclusion principle

Ported from HOL Light by lcp

lemma Inter-over-Union:
\[ \bigcap \{ \bigcup (F \cdot x) \mid x \in S \} = \bigcup \{ \bigcap (G \cdot s) \mid G. \forall x \in S. G x \in F x \} \]
proof
  have \( \forall x. \forall s \in S. \exists X \in F s. x \in X \Longrightarrow \exists G. (\forall x \in S. G x \in F x) \land (\forall s \in S. x \in G s) \) 
  by metis
  then show ☐thesis 
  by (auto simp flip: all-simps ex-simps)
qed

lemma subset-insert-lemma:
\[ \{ T. T \subseteq (\text{insert } a S) \land P T \} = \{ T. T \subseteq S \land P T \} \cup \{ \text{insert } a T \mid T. T \subseteq S \land P(\text{insert } a T) \} \]
(is ☐L=☐R)
proof
  show ☐L \subseteq ☐R 
  by (smt (verit) UnI1 UnI2 insert-Diff mem-Collect-eq subsetI subset-insert-iff)
qed blast

Versions for additive real functions, where the additivity applies only to
some specific subsets (e.g. cardinality of finite sets, measurable sets with
bounded measure. (From HOL Light)

locale Incl-Excl =

fixes P :: 'a set ⇒ bool and f :: 'a set ⇒ 'b::ring-1
assumes disj-add: \[ P S; P T; \text{disjnt } S T \Rightarrow f(S \cup T) = f S + f T \]
and empty: \[ P{} \]
and Int: \[ P S; P T \Rightarrow P(S \cap T) \]
and Un: \[ P S; P T \Rightarrow P(S \cup T) \]
and Diff: \[ P S; P T \Rightarrow P(S - T) \]

begin
lemma f-empty [simp]: \( f(\{\} ) = 0 \)
using disj-add empty by fastforce

lemma f-Un-Int: \([P \ S; \ P \ T] \implies f(S \cup T) + f(S \cap T) = f(S) + f(T)\)
by (smt (verit, ccf0-threshold) Groups.add_ac(2) Incl-Excl.Diff Incl-Excl.Int Incl-Excl-axioms Int-Diff-Un Int-Diff-disjoint Int-absorb Un-Diff Un-Int-eq(2) disj-add disjnt-def group-cancel.add2 sup-bot.right-neutral)

lemma restricted-indexed:
assumes finite \( A \) and \( X: \bigwedge a . \ a \in A \implies P(X \ a) \)
shows \( f(\bigcup (X \cdot A)) = (\sum B \mid B \subseteq A \land B \neq \{\} . \ (-1)^\mathrm{\scriptsize card}(B) + 1) \ast f(\bigcap (X \cdot B))\)
proof
have \([\text{finite } A; \ \text{card } A = n; \ \forall a \in A. \ P( X \ a)] \implies f(\bigcup (X \cdot A)) = (\sum B \mid B \subseteq A \land B \neq \{\} . \ (-1)^\mathrm{\scriptsize card}(B) + 1) \ast f(\bigcap (X \cdot B))\)
for \( n \ X \ \text{and} \ A :: 'c\ \text{set} \)
proof (induction \( n \ \text{arbitrary}; \ A \ \text{X rule}; \ \text{less-induct})
case \( \text{less } n \ 0 \ A \ 0 \ X \)
show \( \text{?case} \)
proof (cases \( n=0=0 \))
case True
with \( \text{less show} \ \text{?thesis} \)
by fastforce
next
case False
with \( \text{less.premss} \ \text{obtain} \ A \ n \ a \ \text{where} *: n=0 = \text{Suc } n \ A 0 = \text{insert } a \ A \ a \notin A \)
card \( A = n \) \text{finite } A
by (metis card-Suc-eq-finite not0-implies-Suc)
with \( \text{less have} \ P( X \ a) \) by blast
have APX: \( \forall a \in A. \ P( X \ a) \)
by (simp add: * less.premss)
have PUXA: \( P(\bigcup (X \cdot A)) \)
using \( \text{finite } A \cdot APX \)
by (induction) (auto simp: empty Un)
have \( f(\bigcup (X \cdot A)) = f( X \ a \cup \bigcup (X \cdot A)) \)
by (simp add: *)
also have \( \ldots = f( X \ a) + f(\bigcup (X \cdot A)) - f( X a \cap \bigcup (X \cdot A)) \)
using f-Un-Int add-diff-cancel PUXA \( P( X a) \) by metis
also have \( \ldots = f( X a) - (\sum B \mid B \subseteq A \land B \neq \{\} . \ (-1)^\mathrm{\scriptsize card}(B) + 1) \ast f(\bigcap (X \cdot B))\)
by (simp add: *)
\( -1)^\mathrm{\scriptsize card}(B) \ast f( X a \cap \bigcup (X \cdot B))\)

proof
have \( 1: f(\bigcup i \in A. X \ a \cap X \ i) = (\sum B \mid B \subseteq A \land B \neq \{\} . \ (-1)^\mathrm{\scriptsize card}(B) + 1) \ast f(\bigcap i \in B. X \ a \cap X \ i) \)
using less.Ind \( \{n\ a\ \lambda i. \ X \ a \cap X \ i\} \ APX \text{ Int } \text{Sup} \ (P \ X a) \) by (simp add: *)
have \( 2: X \ a \cap \bigcup (X \cdot A) = (\bigcup i \in A. X \ a \cap X \ i) \)
by auto
have \( 3: f(\bigcup (X \cdot A)) = (\sum B \mid B \subseteq A \land B \neq \{\} . \ (-1)^\mathrm{\scriptsize card}(B) + 1) \)

* \( f \left( \bigcap (X \ i \ B) \right) \)

using less_IH [of \( n \ A \ X \)] APX Int \( P (X a) \) by (simp add: *)

show ?thesis

unfolding 3 2 1

by (simp add: sum-negf)

qed

also have ...

= (\( \sum B \mid B \subseteq A0 \land B \neq {} \). \( -1 \) \( \sim \) (card \( B + 1 \) \( \ast \) \( f \left( \bigcap (X \ i \ B) \right) \))

proof

have F:

\{ insert a B \mid B \subseteq A \} = insert a \ ' Pow A \ \& \ \{ B \mid B \subseteq A \land B \neq {} \}

by auto

have G:

\( \sum B \in Pow A \). \( -1 \) \( \sim \) card \( \{ \text{insert} \ a \ B \mid B \subseteq A \land B \neq {} \} \) = \( -1 \) \( \sim \) card 

\( B \ast f \left( X a \cap \bigcap (X \ i \ B) \right) \)

using B \( \ast \) by (auto simp add: card-insert-if finite B)

qed

have disj:\( \{ B. B \subseteq A \land B \neq {} \} \cap \{ \text{insert} \ a \ B \mid B \subseteq A \} = {} \}

using \* by blast

have inj: inj-on (insert a) (Pow A)

using \* by fastforce

show ?thesis

apply (simp add: \* subset-insert-lemma sum.union-disjoint disj sum-negf)

apply (simp add: F G sum-negf sum.reindex [OF inj] o-def sum-diff *)

done

qed

finally show ?thesis .

qed

lemma restricted:

assumes finite A \( \land \) a. a \( \in \) A \( \Longrightarrow \) P a

shows \( f(\bigcup A) = (\sum B \mid B \subseteq A \land B \neq {} \). \( -1 \) \( \sim \) (card \( B + 1 \) \( \ast \) \( f \left( \bigcap B \right) \))

using restricted-indexed [of A \( \lambda x. x \)] assms by auto

end

93.7 Versions for unrestrictedly additive functions

lemma Incl-Excl-UN:
fixes f :: 'a set ⇒ 'b::ring_1
assumes ∩S T. disjoint S T ⟹ f(S ∪ T) = f S + f T finite A
shows f(∪ (G ∩ A)) = (∑ B | B ⊆ A ∧ B ≠ {}. (∼1) ∨ (card B + 1) * f (∩ (G ∩ B)))
proof –
  interpret Incl-Excl λx. True f
  by (simp add: Incl-Excl.intro assms(1))
show ?thesis
  using restricted-assms by blast
qed

lemma Incl-Excl-Union:
  fixes f :: 'a set ⇒ 'b::ring_1
  assumes ∩S T. disjoint S T ⟹ f(S ∪ T) = f S + f T finite A
  shows f(∪ A) = (∑ B | B ⊆ A ∧ B ≠ {}. (∼1) ∨ (card I + 1) * int (card (∩ I)))
  using Incl-Excl-UN [of f A λX. X] assms by simp

The famous inclusion-exclusion formula for the cardinality of a union

lemma int-card-UNION:
  assumes finite A ∩k. K ∈ A ⇒ finite K
  shows int (card (∪ A)) = (∑ I | I ⊆ A ∧ I ≠ {}. (∼1) ∨ (card I + 1) * int (card (∩ I)))
  proof –
    interpret Incl-Excl finite int o card
    proof qed (auto simp add: card-UN-disjnt)
    show ?thesis
      using restricted-assms by auto
  qed

A more conventional form

lemma inclusion-exclusion:
  assumes finite A ∩k. K ∈ A ⇒ finite K
  shows int(card(∪ A)) = (∑ n=1..card A. (∼1) ∨ (Suc n) * (∑ B | B ⊆ A ∧ card B = n. int (card (∩ B)))) (is ?>R)
  proof –
    have fin: finite {I. I ⊆ A ∧ I ≠ {}}
      by (simp add: assms)
    have ∩k. [Suc 0 ≤ k; k ≤ card A] ⇒ ∃B⊆A. B ≠ {} ∧ k = card B
      by (metis (mono_tags, lifting) Suc-le-D Zero-neq-Suc card-eq0-iff obtain-subset-with-card-n)
    with finite A: finite-subset
    have card-eq: card ′ {I. I ⊆ A ∧ I ≠ {}} = {1..card A}
      using not-less-eq-eq cardinality mono by (fastforce simp: image-iff)
    have int(card(∪ A))
      = (∑ y = 1..card A. ∑ I∈{x. x ⊆ A ∧ x ≠ {}} ∧ card x = y}. (∼1) ∨ y * int (card (∩ I)))
      by (simp add: int-card-UNION assms sum.image-gen [OF fin, where g=card] card-eq)
    also have ... = ?R
proof –

have \{B. B \subseteq A \land B \neq \emptyset \land \card B = k\} = \{B. B \subseteq A \land \card B = k\}
if Suc 0 \leq k and k \leq \card A for k
using that by auto
then show ?thesis
by (clarsimp simp add: sum-negf simp flip: sum-distrib-left)
qed

finally show ?thesis.
qed

lemma card-UNION:
assumes finite A and \(\bigwedge K. K \in A \implies \text{finite } K\)
shows \(\card{\bigcup A} = \text{nat}\left(\sum_{I} I \subseteq A \land I \neq \emptyset\right) \cdot (-1)^{\card{\bigcap I}} \cdot \text{int} \left(\card{\bigcup I}\right)\)
by (simp only: flip: int-card-UNION[of assms])

lemma card-UNION-nonneg:
assumes finite A and \(\bigwedge K. K \in A \implies \text{finite } K\)
shows \(\sum_{I} I \subseteq A \land I \neq \emptyset. (-1)^{\card{\bigcap I}} \cdot \text{int} \left(\card{\bigcup I}\right) \geq 0\)
using int-card-UNION[of assms] by presburger

93.8 General "Moebius inversion" inclusion-exclusion principle

This 'symmetric' form is from Ira Gessel: 'Symmetric Inclusion-Exclusion'

lemma sum-Un-eq:
\(\{S \cap T = \emptyset; S \cup T = U; \text{finite } U\} \implies (\sum f S + \sum f T = \sum f U)\)
by (metis finite-Un sum.union_disjoint)

lemma card-adjust-lemma: [inj-on f S; x = y + \card{f'S}]
\(\implies x = y + \card S\)
by (simp add: card_image)

lemma card-subsets-step:
assumes finite S x \notin S U \subseteq S
shows \(\card{T. T \subseteq (\text{insert } x S) \land U \subseteq T \land \text{odd}(\card T)}\)
= \(\card{T. T \subseteq S \land U \subseteq T \land \text{odd}(\card T)} + \card{T. T \subseteq S \land U \subseteq T \land \text{even}(\card T)}\)
\(\land \text{even}(\card T)\)
= \(\card{T. T \subseteq (\text{insert } x S) \land U \subseteq T \land \text{even}(\card T)} + \card{T. T \subseteq S \land U \subseteq T \land \text{even}(\card T)}\)
\(\land \text{odd}(\card T)\)

proof –

have inj: inj-on (insert x) \(\{T. T \subseteq S \land P T\}\) for P
using assms by (auto simp: inj-on_def)

have [simp]: finite \(\{T. T \subseteq S \land P T\}\) finite (insert x \(\{T. T \subseteq S \land P T\}\))
for P
using [finite S] by auto

have [simp]: disjoint \(\{T. T \subseteq S \land P T\}\) \(\{\text{insert } x \{T. T \subseteq S \land Q T\}\}\) for P Q
using assms by (auto simp: disjoint_if)

PROOF
have eq: \{T. T \subseteq S \land U \subseteq T \land P T\} \cup \{T. T \subseteq S \land U \subseteq T \land Q T\} = \{T. T \subseteq \text{insert} x S \land U \subseteq T \land P T\} (\text{is } ?L = ?R)
if \bigwedge A. A \subseteq S \implies Q (\text{insert} x A) \iff P A \bigwedge A. \neg Q A \iff P A \text{ for } P Q
proof
show ?L \subseteq ?R
  by (clarsimp simp: image_iff subset_iff) (meson subsetI that)
show ?R \subseteq ?L
  using \langle U \subseteq S \rangle
  by (clarsimp simp: image_iff) (smt \langle verit \rangle insert_iff mk_disjoint_insert subset_iff that)
qed

have \[\text{simp}:\] \bigwedge A. A \subseteq S = \implies even \langle \text{card} (\text{insert} x A) \rangle \iff odd \langle \text{card} A \rangle
by (metis \langle finite S \rangle \langle x \notin S \rangle \card_insert_disjoint even_Suc finite_subset subsetD)

qed
fixes $f :: 'a set \Rightarrow 'b :: ring_1$
assumes $S: \forall S. \text{finite } S \Rightarrow g S = (\sum T \in \text{Pow } S. (\neg 1) \cdot \text{card } T \cdot f T)$
and $\text{finite } S$
shows $f S = (\sum T \in \text{Pow } S. (\neg 1) \cdot \text{card } T \cdot g T)$
proof

have $(\neg 1) \cdot \text{card } T \cdot g T = (\neg 1) \cdot \text{card } T \cdot (\sum U \mid U \subseteq S \land U \subseteq T. (\neg 1) \cdot \text{card } U \cdot f U)$
if $T \subseteq S$ for $T$

proof
have $(\neg 1) \cdot \text{card } U \cdot f U = (\neg 1) \cdot \text{card } U \cdot (\sum U \mid U \subseteq S \land U \subseteq T. (\neg 1) \cdot \text{card } U \cdot f U)$
using that by auto

show $?\text{thesis}$
using that by (simp add: $\langle \text{finite } S \rangle$ finite-subset $\langle \# \rangle$)

qed

also have $\ldots = (\sum U \in \text{Pow } S. \text{if } U = S \text{ then } f S \text{ else } 0)$

proof
have $\ldots = (\sum U \in \text{Pow } S. \text{if } U = S \text{ then } f S \text{ else } 0)$
by auto

show $?\text{thesis}$
apply (rule sum.cong [OF refl])
by (simp add: $\langle \text{finite } S \rangle$ sum-alternating-cancels card-subsupersets-even-odd $\langle \text{finite } S \rangle$ $\langle \text{power-add } \rangle$

qed

also have $\ldots = f S$
by (simp add: $\langle \text{finite } S \rangle$)

finally show $?\text{thesis}$
by presburger

qed

The more typical non-symmetric version.

lemma inclusion-exclusion-mobius:

fixes $f :: 'a set \Rightarrow 'b :: ring_1$
assumes $S: \forall S. \text{finite } S \Rightarrow g S = \text{sum } f \langle \text{Pow } S \rangle$ and $\text{finite } S$
shows $f S = (\sum T \in \text{Pow } S. (\neg 1) \cdot \text{card } S - \text{card } T) \cdot g T$ (is $\ldots = ?\text{rhs}$)

proof

have $(\neg 1) \cdot \text{card } S \cdot f S = (\sum T \in \text{Pow } S. (\neg 1) \cdot \text{card } T \cdot g T)$
by (rule inclusion-exclusion-symmetric; simp add: assms flip; power-add mult_assoc)

then have $(\neg 1) \cdot \text{card } S \cdot (\neg 1) \cdot \text{card } S \cdot f S = ((\neg 1) \cdot \text{card } S) \cdot (\sum T \in \text{Pow } S. (\neg 1) \cdot \text{card } T \cdot g T)$
by (simp add: mult-ac)
then have \( f S = (\sum T \in \text{Pow } S, (-1)^{-(\text{card } S + \text{card } T)} \cdot g T) \)
by (simp add: sum-distrib-left flip: power-add mult.assoc)
also have \( \ldots = \text{rhs} \)
by (simp add: finite S card-mono neg-one-power-add-eq-neg-one-power-diff)
finally show \( \text{thesis} \).
qed

93.9 Executable code

lemma gbinomial-code [code]:
\( a \choose k =
\begin{cases}
if k = 0 \text{ then } 1 \\
else \text{ fold-atLeastAtMost-nat}(\lambda \text{ acc}. (a - of-nat k) \cdot \text{acc}) 0 (k - 1) 1 / \text{fact} k
\end{cases}
\)
by (cases k)
(simp-all add: gbinomial-prod-rev prod-atLeastAtMost-code [symmetric]
atLeastLessThanSuc-atLeastAtMost)

lemma binomial-code [code]:
\( n \choose k =
\begin{cases}
if k > n \text{ then } 0 \\
else if 2 \cdot k > n \text{ then } \text{n choose } (n - k) \\
else \text{fold-atLeastAtMost-nat}(\cdot) (n - k + 1) n 1 \text{ div } \text{fact } k
\end{cases}
\)
proof -
\{ 
assume \( k \leq n \)
then have \( \{1..n\} = \{1..n-k\} \cup \{n-k+1..n\} \) by auto
then have \( (\text{fact } n :: \text{nat}) = \text{fact } (n-k) \cdot \prod\{n-k+1..n\} \)
by (simp add: prod.union-disjoint fact-prod)
\}
thен show \( \text{thesis} \) by (auto simp: binomial-altdef-nat mult-ac prod-atLeastAtMost-code)
qed

end

94 Misc lemmas on division, to be sorted out finally

theory Divides
imports Parity
begin

class unique-euclidean-semiring-numeral = linordered-euclidean-semiring + discrete-linordered-semidom +
assumes div-less [no-atp]: \( 0 \leq a \Longrightarrow a < b \Longrightarrow a \text{ div } b = 0 \)
and mod-less [no-atp]: \( 0 \leq a \Longrightarrow a < b \Longrightarrow a \text{ mod } b = a \)
and div-positive [no-atp]: \( 0 < b \Longrightarrow b \leq a \Longrightarrow a \text{ div } b > 0 \)
and mod-less-eq-dividend [no-atp]: \( 0 \leq a \Longrightarrow a \text{ mod } b \leq a \)
and pos-mod-bound [no-atp]: $0 < b \implies a \mod b < b$
and pos-mod-sign [no-atp]: $0 < b \implies 0 \leq a \mod b$
and mod-mult2-eq [no-atp]: $0 \leq c \implies a \mod (b \cdot c) = b \cdot (a \div b \mod c) + a \mod b$
and div-mult2-eq [no-atp]: $0 \leq c \implies a \div (b \cdot c) = a \div b \div c$

hide-fact (open) div-less mod-less div-positive mod-less-eq-dividend pos-mod-bound pos-mod-sign mod-mult2-eq div-mult2-eq

context unique-euclidean-semiring-numeral
begin

context begin

qualified lemma discrete [no-atp]:
\begin{align*}
a < b & \iff a + 1 \leq b \\
& \text{by (fact less-iff-succ-less-eq)}
\end{align*}

qualified lemma divmod-digit-1 [no-atp]:
\begin{align*}
\text{assumes } 0 & \leq a \text{ and } b \leq a \mod (2 \cdot b) \\
\text{shows } & 2 \cdot (a \div (2 \cdot b)) + 1 = a \div b \text{ (is } \ ?P) \\
& \text{and } a \mod (2 \cdot b) - b = a \mod b \text{ (is } \ ?Q)
\end{align*}

proof –
\begin{align*}
& \text{from assms mod-less-eq-dividend } [\text{of } a \cdot 2 \cdot b] \ \text{have } b \leq a \\
& \text{by (auto intro: trans)} \\
& \text{with } \langle 0 < b \rangle \ \text{have } 0 < a \div b \text{ by (auto intro: div-positive)} \\
& \text{then have } [\text{simp}]: 1 \leq a \div b \text{ by (simp add: discrete)} \\
& \text{with } \langle 0 < b \rangle \ \text{have mod-less; } a \mod b < b \text{ by (simp add: pos-mod-bound)} \\
& \text{define } w \text{ where } w = a \div b \mod 2 \\
& \text{then have } w\text{-exhaust: } w = 0 \lor w = 1 \text{ by auto} \\
& \text{have mod-w: } a \mod (2 \cdot b) = a \mod b + b \cdot w \\
& \text{by (simp add: w-def mod-mult2-eq ac-simps)} \\
& \text{from assms w-exhaust have } w = 1 \\
& \text{using mod-less by (auto simp add: mod-w)} \\
& \text{with mod-w have mod: } a \mod (2 \cdot b) = a \mod b + b \cdot w \text{ by simp} \\
& \text{have } 2 \cdot (a \div (2 \cdot b)) = a \div b - w \\
& \text{by (simp add: w-def div-mult2-eq minus-mod-eq-mult-div ac-simps)} \\
& \text{with } \langle w = 1 \rangle \ \text{have } \text{div: } 2 \cdot (a \div (2 \cdot b)) = a \div b - 1 \text{ by simp} \\
& \text{then show } \ ?P \text{ and } \ ?Q \\
& \text{by (simp-all add: div mod add-implies-diff [symmetric])}
\end{align*}

qed

qualified lemma divmod-digit-0 [no-atp]:
\begin{align*}
\text{assumes } 0 & < b \text{ and } a \mod (2 \cdot b) < b \\
\text{shows } & 2 \cdot (a \div (2 \cdot b)) = a \div b \text{ (is } \ ?P) \\
& \text{and } a \mod (2 \cdot b) = a \mod b \text{ (is } \ ?Q)
\end{align*}

proof –
\begin{align*}
& \text{define } w \text{ where } w = a \div b \mod 2
\end{align*}
then have w-exhaust: \( w = 0 \lor w = 1 \) by auto
have mod-w: \( a \mod (2 * b) = a \mod b + b * w \)
  by (simp add: w-def mod-mult2-eq ac-simps)
moreover have \( b \leq a \mod b + b \)
proof
  from \( 0 < b \) pos-mod-sign have \( 0 \leq a \mod b \) by blast
  then have \( 0 + b \leq a \mod b + b \) by (rule add-right-mono)
  then show \( ?thesis \) by simp
qed
moreover note assms w-exhaust
ultimately have \( w = 0 \) by auto
with mod-w have \( \mod: \ a \mod (2 * b) = a \mod b \) by simp
have \( 2 * (a \div (2 * b)) = a \div b - w \)
  by (simp add: w-def div-mult2-eq minus-mod-eq-mult-div ac-simps)
with \( \langle w = 0 \rangle \) have \( \div: 2 * (a \div (2 * b)) = a \div b \) by simp
then show \( ?P \) and \( ?Q \)
  by (simp-all add: div mod)
qed

qualified lemma mod-double-modulus [no-atp]:
assumes \( m > 0 \ \ x \geq 0 \)
shows \( x \mod (2 * m) = x \mod m \lor x \mod (2 * m) = x \mod m + m \)
proof (cases \( x \mod (2 * m) < m \))
  case True
  thus \( ?thesis \) using assms using divmod-digit-0(2)[of m x] by auto
next
case False
  hence \( x \mod (2 * m) - m = x \mod m \)
  using assms by (intro divmod-digit-1) auto
  hence \( x \mod (2 * m) = x \mod m + m \)
  by (subst \( * \) [symmetric], subst le-add-diff-inverse2) (use False in auto)
  thus \( ?thesis \) by simp
qed

end

end

instance nat :: unique-euclidean-semiring-numeral
  by standard
  (auto simp add: div-greater-zero-iff div-mult2-eq mod-mult2-eq)

instance int :: unique-euclidean-semiring-numeral
  by standard (auto intro: zmod-le-nonneg-dividend simp add:
    pos-imp-zdiv-pos-iff zmod-zmult2-eq zdiv-zmult2-eq)

context
begin
qualified lemma \( \text{zmod-eq-0D [dest!]} \): \( \exists q. \ m = d \ast q \) if \( m \text{ mod } d = 0 \) for \( m, d :: \text{int} \)
using that by auto

qualified lemma \( \text{div-geq [no-atp]} \): \( m \div n = \text{Suc} ((m - n) \text{ div } n) \) if \( 0 < n \) and \( \neg \ m < n \) for \( m, n :: \text{nat} \)
by (rule le-div-geq) (use that in \( \langle \text{simp-all add: not-less} \rangle \))

qualified lemma \( \text{mod-geq [no-atp]} \): \( m \mod n = (m - n) \mod n \) if \( \neg m < n \) for \( m, n :: \text{nat} \)
by (rule le-mod-geq) (use that in \( \langle \text{simp add: not-less} \rangle \))

qualified lemma \( \text{mod-eq-0D [no-atp]} \): \( \exists q. \ m = d \ast q \) if \( m \mod d = 0 \) for \( m, d :: \text{nat} \)
using that by (auto simp add: mod-eq-0-iff-dvd)

qualified lemma \( \text{pos-mod-conj [no-atp]} \): \( 0 < b \implies 0 \leq a \mod b \land a \mod b < b \) for \( a, b :: \text{int} \)
by simp

qualified lemma \( \text{neg-mod-conj [no-atp]} \): \( b < 0 \implies a \mod b \leq 0 \land b < a \mod b \) for \( a, b :: \text{int} \)
by simp

qualified lemma \( \text{zmod-eq-0-iff [no-atp]} \): \( m \mod d = 0 \iff (\exists q. m = d \ast q) \) for \( m, d :: \text{int} \)
by (auto simp add: mod-eq-0-iff-dvd)

qualified lemma \( \text{div-positive-int [no-atp]} \):
\( k \div l > 0 \) if \( k \geq l \) and \( l > 0 \) for \( k, l :: \text{int} \)
using that by (simp add: nonneg1-imp-zdiv-pos-iff)

end

code-identifier

code-module Divides \( \to \) (SML) Arith and (OCaml) Arith and (Haskell) Arith

done

95 Main HOL

Classical Higher-order Logic – only “Main”, excluding real and complex numbers etc.

done

classical higher-order logic – only “Main”, excluding real and complex numbers etc.

done
THEORY “Main”

Mirabelle
Extraction
Nunchaku
BNF-Greatest-Fixpoint
Filter
Conditionally-Complete-Lattices
Binomial
GCD
Divides

begin

95.1 Namespace cleanup

hide-const (open)
czero cinfinite cfinite csum cone ctwo Csum cprod cexp image2 image2p vimage2p
Gr Grp collect
fsts snds setl setr convol pick-middlep fstOp sndOp csquare relImage relInvImage
Succ Shift
shift proj id-bnf

hide-fact (open) id-bnf-def type-definition-id-bnf-UNIV

95.2 Syntax cleanup

no-notation
ordLeq2 (infix <=o 50) and
ordLeq3 (infix <= 50) and
ordLess2 (infix <o 50) and
ordIso2 (infix =o 50) and
card-of (| -|) and
BNF-Cardinal-Arithmetic.csum (infixr + 65) and
BNF-Cardinal-Arithmetic.cprod (infixr * 80) and
BNF-Cardinal-Arithmetic.cexp (infixr ~ 90) and
BNF-Def.convol ((-/-))

bundle cardinal-syntax
begin

notation
ordLeq2 (infix <=o 50) and
ordLeq3 (infix <= 50) and
ordLess2 (infix <o 50) and
ordIso2 (infix =o 50) and
card-of (| -|) and
BNF-Cardinal-Arithmetic.csum (infixr + 65) and
BNF-Cardinal-Arithmetic.cprod (infixr * 80) and
BNF-Cardinal-Arithmetic.cexp (infixr ~ 90)

alias cinfinite = BNF-Cardinal-Arithmetic.cinfinite
alias czero = BNF-Cardinal-Arithmetic.czero
alias cone = BNF-Cardinal-Arithmetic.cone
alias ctwo = BNF-Cardinal-Arithmetic.ctwo

end

95.3 Lattice syntax

bundle lattice-syntax
begin

notation
bot (⊥) and
top (⊤) and
inf (infixl ⊓ 70) and
sup (infixl ⊔ 65) and
Inf (⨆ - [900] 900) and
Sup (⨆ - [900] 900)

syntax
-INF1 :: pttrns ⇒ 'b ⇒ 'b ((3_/ -) [0, 10] 10)
-INF :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3_/ ∈) [0, 0, 10] 10)
-SUP1 :: pttrns ⇒ 'b ⇒ 'b ((3_/ -) [0, 10] 10)
-SUP :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3_/ ∈) [0, 0, 10] 10)

end

bundle no-lattice-syntax
begin

no-notation
bot (⊥) and
top (⊤) and
inf (infixl ⊓ 70) and
sup (infixl ⊔ 65) and
Inf (⨆ - [900] 900) and
Sup (⨆ - [900] 900)

no-syntax
-INF1 :: pttrns ⇒ 'b ⇒ 'b ((3_/ -) [0, 10] 10)
-INF :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3_/ ∈) [0, 0, 10] 10)
-SUP1 :: pttrns ⇒ 'b ⇒ 'b ((3_/ -) [0, 10] 10)
-SUP :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3_/ ∈) [0, 0, 10] 10)

end

unbundle no-lattice-syntax
end
96 Archimedean Fields, Floor and Ceiling Functions

theory Archimedean-Field
imports Main
begin

lemma cInf-abs-ge:
  fixes S :: 'a::{linordered-idom,conditionally-complete-linorder} set
  assumes S ≠ {} and bdd: ∀x. x∈S ⇒ |x| ≤ a
  shows |Inf S| ≤ a
  proof
    have Sup (uminus ' S) = − (Inf S)
    proof (rule antisym)
      have ∃x. x ∈ S ⇒ bdd-above (uminus ' S)
        using bdd by (force simp: abs-le_iff bdd-above_def)
      then show − (Inf S) ≤ Sup (uminus ' S)
        by (meson cInf-greatest [OF S ≠ {}] cSUP-upper minus-le_iff)
    next
      have ∗: ∀x. x ∈ S ⇒ Inf S ≤ x
        by (meson abs-le_iff bdd bdd-below_def cInf-lower minus_le_iff)
      show Sup (uminus ' S) ≤ − Inf S
        using ∗; by (force intro: ∗ cSup-least)
    qed
  with cSup-abs-le [of uminus ' S] assms show ?thesis
  by fastforce
  qed

lemma cSup-asclose:
  fixes S :: 'a::{linordered-idom,conditionally-complete-linorder} set
  assumes S ≠ {} and b: ∀x∈S. |x − l| ≤ e
  shows |Sup S − l| ≤ e
  proof
    have ∗: |x − l| ≤ e ↔ l − e ≤ x ∧ x ≤ l + e for x l e :: 'a
      by arith
    have bdd-above S
      using b by (auto intro!: bdd-aboveI[of - l + e])
    with S b show ?thesis
      unfolding ∗ by (auto intro!: cSup-upper2 cSup-least)
  qed

lemma cInf-asclose:
  fixes S :: 'a::{linordered-idom,conditionally-complete-linorder} set
  assumes S ≠ {} and b: ∀x∈S. |x − l| ≤ e
  shows |Inf S − l| ≤ e
  proof
THEORY "Archimedean-Field"

have *: |x - l| ≤ e ↔ l - e ≤ x ∧ x ≤ l + e for x l e :: 'a
by arith
have bdd-below S
  using b by (auto intro!: bdd-belowI[of - l - e])
with S b show ?thesis
  unfolding * by (auto intro!: cInf-lower2 cInf-greatest)
qed

96.1 Class of Archimedean fields

Archimedean fields have no infinite elements.

class archimedean-field = linordered-field +
  assumes ex-le-of-int: ∃ z. x ≤ of-int z

lemma ex-less-of-int: ∃ z. x < of-int z
  for x :: 'a::archimedean-field
proof –
  from ex-le-of-int obtain z where x ≤ of-int z ..
  then have x < of-int (z + 1) by simp
  then show ?thesis ..
qed

lemma ex-of-int-less: ∃ z. of-int z < x
  for x :: 'a::archimedean-field
proof –
  from ex-less-of-int obtain z where x < of-int z ..
  then have of-int (− z) < x by simp
  then show ?thesis ..
qed

lemma reals-Archimedean2: ∃ n. x < of-nat n
  for x :: 'a::archimedean-field
proof –
  obtain z where x < of-int z
    using ex-less-of-int ..
  also have ... ≤ of-int (int (nat z))
    by simp
  also have ... = of-nat (nat z)
    by (simp only: of-int-of-nat-eq)
  finally show ?thesis ..
qed

lemma real-arch-simple: ∃ n. x ≤ of-nat n
  for x :: 'a::archimedean-field
proof –
  obtain n where x < of-nat n
    using reals-Archimedean2 ..
  then have x ≤ of-nat n
    by simp
then show \textit{thesis} ..

qed

Archimedean fields have no infinitesimal elements.

\textbf{lemma} \texttt{reals-Archimedean}:
\begin{verbatim}
fixes x :: 'a::archimedean-field
assumes 0 < x
shows \exists n. inverse (of-nat (Suc n)) < x
proof (auto)
from \langle 0 < x \rangle have 0 < inverse x
by (rule positive-imp-inverse-positive)
obtain n where inverse x < of-nat n
using reals-Archimedean \[OF \langle 0 < x \rangle\]
then obtain m where inverse x < of-nat (Suc m)
using \langle 0 < inverse x \rangle by (cases n)
then have inverse (of-nat (Suc m)) < inverse (inverse x)
using \langle 0 < inverse x \rangle by (rule less-imp-inverse-less)
then have inverse (of-nat (Suc m)) < x
using \langle 0 < x \rangle by (simp add: nonzero-inverse-inverse-eq)
then show \textit{thesis} ..
qed
\end{verbatim}

\textbf{lemma} \texttt{ex-inverse-of-nat-less}:
\begin{verbatim}
fixes x :: 'a::archimedean-field
assumes 0 < x
shows \exists n > 0. inverse (of-nat n) < x
using \texttt{reals-Archimedean} \[OF \langle 0 < x \rangle\] by auto
\end{verbatim}

\textbf{lemma} \texttt{ex-less-of-nat-mult}:
\begin{verbatim}
fixes x :: 'a::archimedean-field
assumes 0 < x
shows \exists n. y < of-nat n * x
proof (auto)
obtain n where y / x < of-nat n
using \texttt{reals-Archimedean} \[OF \langle 0 < x \rangle\]
with \langle 0 < x \rangle have y < of-nat n * x
by (simp add: pos-divide-less-eq)
then show \textit{thesis} ..
qed
\end{verbatim}

\section{96.2 Existence and uniqueness of floor function}

\textbf{lemma} \texttt{exists-least-lemma}:
\begin{verbatim}
assumes \lnot P 0 and \exists n. P n
shows \exists n. \lnot P n \land P (Suc n)
proof (auto)
from \langle \exists n. P n \rangle have P (Least P)
by (rule \texttt{LeastI-ex})
with \langle \lnot P 0 \rangle obtain n where Least P = Suc n
\end{verbatim}
by (cases Least P) auto
then have \( n < \text{Least P} \)
  by simp
then have \( \neg P \ n \)
  by (rule not-less-Least)
then have \( \neg P \ n \land P \ (\text{Suc} \ n) \)
  using \( \langle P \ (\text{Least P}) \rangle \/ \langle \text{Least P} = \text{Suc} \ n \rangle \) by simp
then show \( \text{thesis} \).
qed

lemma floor-exists:
  fixes \( x :: 'a::archimedean-field \)
  shows \( \exists z. \text{of-int} \ z \leq x \land x < \text{of-int} \ (z + 1) \)
proof (cases \( \theta \leq x \))
  case True
  then have \( \neg x < \text{of-nat} \ 0 \)
    by simp
  then have \( \exists n. \neg x < \text{of-nat} \ n \land x < \text{of-nat} \ (\text{Suc} \ n) \)
    using reals-Archimedean\(2 \) by (rule exists-least-lemma)
  then obtain \( n \) where \( \neg x < \text{of-nat} \ n \land x < \text{of-nat} \ (\text{Suc} \ n) \) ..
  then have of-int \( (\text{int} \ n) \) \( \leq x \land x < \text{of-int} \ (\text{int} \ n + 1) \)
    by simp
  then show \( \text{thesis} \).
next
  case False
  then have \( \neg x \leq \text{of-nat} \ 0 \)
    by simp
  then have \( \exists n. \neg x \leq \text{of-nat} \ n \land -x \leq \text{of-nat} \ (\text{Suc} \ n) \)
    using real-arch-simple by (rule exists-least-lemma)
  then obtain \( n \) where \( \neg x \leq \text{of-nat} \ n \land -x \leq \text{of-nat} \ (\text{Suc} \ n) \) ..
  then have of-int \( (-\text{int} \ n - 1) \) \( \leq x \land x < \text{of-int} \ (-\text{int} \ n - 1 + 1) \)
    by simp
  then show \( \text{thesis} \).
qed

lemma floor-exists1:\( \exists!z. \text{of-int} \ z \leq x \land x < \text{of-int} \ (z + 1) \)
for \( x :: 'a::archimedean-field \)
proof (rule ex-ex1I)
  show \( \exists z. \text{of-int} \ z \leq x \land x < \text{of-int} \ (z + 1) \)
    by (rule floor-exists)
next
  fix \( y \ z \)
  assume \( \text{of-int} \ y \leq x \land x < \text{of-int} \ (y + 1) \)
  and \( \text{of-int} \ z \leq x \land x < \text{of-int} \ (z + 1) \)
  with le-less-trans \( \text{of-int} \ y x \text{of-int} \ (z + 1) \)
  le-less-trans \( \text{of-int} \ z x \text{of-int} \ (y + 1) \) show \( y = z \)
  by (simp del: of-int-add)
qed
96.3 Floor function

class floor-ceiling = archimedean-field +
  fixes floor :: 'a ⇒ int (\lfloor . \rfloor)
  assumes floor-correct: of-int \lfloor x \rfloor \leq x \land x < of-int \lfloor x + 1 \rfloor

lemma floor-unique: of-int z \leq x \Longrightarrow x < of-int z + 1 \Longrightarrow \lfloor x \rfloor = z
  using floor-correct \lfloor of x \rfloor \illem exists1 \lfloor of x \rfloor by auto

lemma of-int-floor-le [simp]: of-int \lfloor x \rfloor \leq x
  using floor-correct ..

lemma le-floor-iff: z \leq \lfloor x \rfloor \longleftrightarrow of-int z \leq x
  proof
  assume z \leq \lfloor x \rfloor
  then have (of-int z :: 'a) \leq of-int \lfloor x \rfloor by simp
  also have of-int \lfloor x \rfloor \leq x by (rule of-int-floor-le)
  finally show of-int z \leq x .
  next
  assume of-int z \leq x
  also have x < of-int \lfloor x + 1 \rfloor using floor-correct ..
  finally show z \leq \lfloor x \rfloor by (simp del: of-int-add)
  qed

lemma floor-less-cancel: \lfloor x \rfloor < \lfloor y \rfloor \Longrightarrow x < y
  by (metis floor-correct floor-unique less-floor-iff not-le order-refl)

lemma floor-mono:
  assumes x \leq y
  shows \lfloor x \rfloor \leq \lfloor y \rfloor
  proof
    have of-int \lfloor x \rfloor \leq x by (rule of-int-floor-le)
    also note \lfloor x \rfloor \leq \lfloor y \rfloor
    finally show \?thesis by (simp add: le-floor-iff)
  qed
by (auto simp add: not-le [symmetric] floor-mono)

lemma floor-of-int [simp]: ⌊of-int z⌋ = z
  by (rule floor-unique) simp-all

lemma floor-of-nat [simp]: ⌊of-nat n⌋ = int n
  using floor-of-int [of of-nat n] by simp

lemma le-floor-add: ⌊x⌋ + ⌊y⌋ ≤ ⌊x + y⌋
  by (simp only: le-floor-iff of-int-add add-mono of-int-floor-le)

Floor with numerals.

lemma floor-zero [simp]: ⌊0⌋ = 0
  using floor-of-int [of 0] by simp

lemma floor-one [simp]: ⌊1⌋ = 1
  using floor-of-int [of 1] by simp

lemma floor-numeral [simp]: ⌊numeral v⌋ = numeral v
  using floor-of-int [of numeral v] by simp

lemma floor-neg-numeral [simp]: ⌊−numeral v⌋ = −numeral v
  using floor-of-int [of −numeral v] by simp

lemma zero-le-floor [simp]: 0 ≤ ⌊x⌋ ←→ 0 ≤ x
  by (simp add: le-floor-iff)

lemma one-le-floor [simp]: 1 ≤ ⌊x⌋ ←→ 1 ≤ x
  by (simp add: le-floor-iff)

lemma numeral-le-floor [simp]: numeral v ≤ ⌊x⌋ ←→ numeral v ≤ x
  by (simp add: le-floor-iff)

lemma neg-numeral-le-floor [simp]: −numeral v ≤ ⌊x⌋ ←→ −numeral v ≤ x
  by (simp add: le-floor-iff)

lemma zero-less-floor [simp]: 0 < ⌊x⌋ ←→ 1 ≤ x
  by (simp add: less-floor-iff)

lemma one-less-floor [simp]: 1 < ⌊x⌋ ←→ 2 ≤ x
  by (simp add: less-floor-iff)

lemma numeral-less-floor [simp]: numeral v < ⌊x⌋ ←→ numeral v + 1 ≤ x
  by (simp add: less-floor-iff)

lemma neg-numeral-less-floor [simp]: −numeral v < ⌊x⌋ ←→ −numeral v + 1 ≤ x
  by (simp add: less-floor-iff)
lemma floor-le-zero [simp]: \( \lfloor x \rfloor \leq 0 \iff x < 1 \)
by (simp add: floor-le-iff)

lemma floor-le-one [simp]: \( \lfloor x \rfloor \leq 1 \iff x < 2 \)
by (simp add: floor-le-iff)

lemma floor-le-numeral [simp]: \( \lfloor x \rfloor \leq \text{numeral } v \iff x < \text{numeral } v + 1 \)
by (simp add: floor-le-iff)

lemma floor-le-neg-numeral [simp]: \( \lfloor x \rfloor \leq -\text{numeral } v \iff x < -\text{numeral } v + 1 \)
by (simp add: floor-le-iff)

lemma floor-less-zero [simp]: \( \lfloor x \rfloor < 0 \iff x < 0 \)
by (simp add: floor-less-iff)

lemma floor-less-one [simp]: \( \lfloor x \rfloor < 1 \iff x < 1 \)
by (simp add: floor-less-iff)

lemma floor-less-numeral [simp]: \( \lfloor x \rfloor < \text{numeral } v \iff x < \text{numeral } v \)
by (simp add: floor-less-iff)

lemma floor-less-neg-numeral [simp]: \( \lfloor x \rfloor < -\text{numeral } v \iff x < -\text{numeral } v \)
by (simp add: floor-less-iff)

lemma le-mult-floor-Ints: 
assumes \( 0 \leq a \in \text{Ints} \)
shows \( \text{of-int } (\lfloor a \rfloor * \lfloor b \rfloor) \leq (\text{of-int } (a * b) :: \text{'a :: linordered-idom}) \)
by (metis Ints-cases assms floor-less-iff floor-of-int linorder-not-less mult-left-mono of-int-floor-le of-int-less-iff of-int-mult)

Addition and subtraction of integers.

lemma floor-add-int: \( \lfloor x \rfloor + z = \lfloor x + \text{of-int } z \rfloor \)
using floor-correct [of x] by (simp add: floor-unique[symmetric])

lemma int-add-floor: \( z + \lfloor x \rfloor = \lfloor \text{of-int } z + x \rfloor \)
using floor-correct [of x] by (simp add: floor-unique[symmetric])

lemma one-add-floor: \( \lfloor x \rfloor + 1 = \lfloor x + 1 \rfloor \)
using floor-add-int [of x 1] by simp

lemma floor-diff-of-int [simp]: \( \lfloor x - \text{of-int } z \rfloor = \lfloor x \rfloor - z \)
using floor-add-int [of x - z] by (simp add: algebra-simps)

lemma floor-uminus-of-int [simp]: \( - (\text{of-int } z) = -z \)
by (metis floor-diff-of-int [of 0] diff-0 floor-zero)

lemma floor-diff-numeral [simp]: \( \lfloor x - \text{numeral } v \rfloor = \lfloor x \rfloor - \text{numeral } v \)
using floor-diff-of-int [of x numeral v] by simp
lemma floor-diff-one [simp]: \([x - 1] = \lfloor x \rfloor - 1\)
using floor-diff-of-int [of \(x \cdot 1\)] by simp

lemma le-mult-floor:
assumes \(0 \leq a\) and \(0 \leq b\)
shows \([a] \cdot [b] \leq [a \cdot b]\)
proof
have of-int \([a] \leq a\) and of-int \([b] \leq b\)
by (auto intro: of-int-floor-le)
then have of-int \((a) \cdot \lfloor b \rfloor \) \(\leq a \cdot b\)
using assms by (auto intro!: mult-mono)
also have \(a \cdot b < \lfloor a \cdot b \rfloor + 1\)
using floor-correct[of \(a \cdot b\)] by auto
finally show ?thesis
unfolding of-int-less-iff by simp
qed

lemma floor-divide-of-int-eq: \(\lfloor \text{of-int } k \div \text{of-int } l \rfloor = k \div l\)
for \(k\) \(\cdot\) \(l::\text{int}\)
proof (cases \(l = 0\))
case True
then show ?thesis by simp
next
case False
have of-int \(\lfloor \text{of-int } (k \mod l) \div \text{of-int } l \rfloor = 0\)
proof (cases \(l > 0\))
case True
then show ?thesis
by (auto intro: floor-unique)
next
case False
obtain \(r\) where \(r = -l\)
by blast
then have \(l = -r\)
by simp
with \(l \neq 0\) False have \(r > 0\)
by simp
with \(l\) show ?thesis
using pos-mod-bound [of \(r\)]
by (auto simp add: zmod-zminus2-eq-if less-le field-simps intro: floor-unique)
qed

have (of-int \(k ::\) \('a\)) = of-int \((k \div l) \cdot l + k \mod l\)
by simp

also have \(\ldots = \lfloor \text{of-int } (k \div l) + \text{of-int } (k \mod l) / \text{of-int } l \rfloor \div \text{of-int } l\)
using False by (simp only: of-int-add) (simp add: field-simps)
finally have (of-int \(k / \text{of-int } l ::\) \('a\)) = \(\ldots / \text{of-int } l\)
by simp
then have (of-int \(k / \text{of-int } l ::\) \('a\)) = of-int \((k \div l) + \text{of-int } (k \mod l) / \text{of-int } l\)
using False by (simp only: simp add: field-simps)
then have \([\text{of-int } k \div \text{of-int } l :: 'a] = [\text{of-int } (k \div l) + \text{of-int } (k \mod l) \div \text{of-int } l :: 'a]\)
  by simp
then have \([\text{of-int } k \div \text{of-int } l :: 'a] = [\text{of-int } (k \mod l) \div \text{of-int } l + \text{of-int } (k \div l) :: 'a]\)
  by (simp add: ac-simps)
then have \([\text{of-int } k \div \text{of-int } l :: 'a] = [\text{of-int } (k \mod l) \div \text{of-int } l :: 'a] + k \div l\)
  by (simp add: floor-add-int)
with * show ?thesis
  by simp
qed

lemma floor-divide-of-int-eq: \([\text{of-nat } m \div \text{of-nat } n :: \text{nat}] = \text{of-nat } (m \div n)\)
  for m n :: nat
  by (metis floor-divide-of-int-eq of-int-of-nat-eq linordered-euclidean-semiring-class.of-nat-div)

lemma floor-divide-lower:
  fixes q :: 'a::floor-ceiling
  shows q > 0 =⇒ \([\text{of-int } \lceil p / q \rceil \times q \leq p]\)
  using of-int-floor-le pos-le-divide-eq by blast

lemma floor-divide-upper:
  fixes q :: 'a::floor-ceiling
  shows q > 0 =⇒ p < (\text{of-int } \lceil p / q \rceil + 1) \times q
  by (meson floor-eq-iff pos-divide-less-eq)

96.4 Ceiling function

definition ceiling :: 'a::floor-ceiling ⇒ int (\lfloor - \rfloor)
  where \([x] = - \lfloor -x \rfloor\)

lemma ceiling-correct: \([\text{of-int } \lceil x \rceil - 1 < x \land x \leq \text{of-int } \lceil x \rceil]\)
  unfolding ceiling-def using floor-correct [of - x]
  by (simp add: le-minus-iff)

lemma ceiling-unique: \([\text{of-int } z - 1 < x =⇒ x \leq \text{of-int } z =⇒ \lceil x \rceil = z]\)
  unfolding ceiling-def using floor-unique [of - z - x] by simp

lemma ceiling-eq-iff: \([\lceil x \rceil = a =⇒ \text{of-int } a - 1 < x \land x \leq \text{of-int } a]\)
  using ceiling-correct ceiling-unique by auto

lemma le-of-int-ceiling [simp]: \(x \leq \text{of-int } \lceil x \rceil\)
  using ceiling-correct ..

lemma ceiling-le-iff: \([x] \leq z =⇒ z \leq \text{of-int } z\)
  unfolding ceiling-def using le-floor-iff [of - z - x] by auto

lemma less-ceiling-iff: \(z < \lceil x \rceil =⇒ \text{of-int } z < x\)
by (simp add: not-le [symmetric] ceiling-le-iff)

lemma ceiling-less-iff: \([x] < z \iff x \leq \text{of-int } z - 1\)
using ceiling-le-iff [of \(x\ z - 1\)] by simp

lemma le-ceiling-iff: \(z \leq [x] \iff \text{of-int } z - 1 < x\)
by (simp add: not-less [symmetric] ceiling-less-iff)

lemma ceiling-mono: \(x \geq y = \implies [x] \geq [y]\)
unfolding ceiling-def by (simp add: floor-mono)

lemma ceiling-less-cancel: \([x] < [y] \implies x < y\)
by (auto simp add: not-le [symmetric] ceiling-mono)

lemma ceiling-of-int [simp]: \([\text{of-int } z] = z\)
by (rule ceiling-unique simp-all)

lemma ceiling-of-nat [simp]: \([\text{of-nat } n] = \text{int } n\)
using ceiling-of-int [of \(\text{of-nat } n\)] by simp

lemma ceiling-add-le: \([x + y] \leq [x] + [y]\)
by (simp only: ceiling-le-iff of-int-add add-mono le-of-int-ceiling)

lemma mult-ceiling-le:
assumes \(0 \leq a \text{ and } 0 \leq b\)
shows \([a * b] \leq [a] * [b]\)
by (metis assms ceiling-le-iff ceiling-mono le-of-int-ceiling mult-mono of-int-mult)

lemma mult-ceiling-le-Ints:
assumes \(0 \leq a \text{ a } \in \text{Ints}\)
shows \([\text{of-int } \{a * b\}] :: \text{linordered-idom} \leq \text{of-int}\{[a] * [b]\}\)
by (metis Ints-cases assms ceiling-le-iff ceiling-of-int le-of-int-ceiling mult-left-mono of-int-le-iff of-int-mult)

lemma finite-int-segment:
fixes \(a :: \text{floor-ceiling}\)
shows finite \(\{x \in \text{Z. } a \leq x \land x \leq b\}\)
proof 
  have finite \(\{\text{ceiling } a..\text{floor } b\}\)
  by simp
  moreover have \(\{x \in \text{Z. } a \leq x \land x \leq b\} = \text{of-int } \{\text{ceiling } a..\text{floor } b\}\)
  by (auto simp: le-floor-iff ceiling-le-iff elim!: Ints-cases)
  ultimately show \(?\text{thesis}\)
  by simp
qed

corollary finite-abs-int-segment:
fixes \(a :: \text{floor-ceiling}\)
shows finite \(\{k \in \text{Z. } |k| \leq a\}\)
using finite-int-segment [of \(-a\ a\)] by (auto simp add: abs-le-iff conj-commute minus-le-iff)

96.4.1 Ceiling with numerals.

lemma ceiling-zero [simp]: \([0] = 0\)
using ceiling-of-int [of 0] by simp

lemma ceiling-one [simp]: \([1] = 1\)
using ceiling-of-int [of 1] by simp

lemma ceiling-numeral [simp]: \([\text{numeral } v] = \text{numeral } v\)
using ceiling-of-int [of \text{numeral } v] by simp

lemma ceiling-neg-numeral [simp]: \([- \text{numeral } v] = - \text{numeral } v\)
using ceiling-of-int [of \(-\text{numeral } v\)] by simp

lemma ceiling-le-zero [simp]: \([x] \leq 0 \iff x \leq 0\)
by (simp add: ceiling-le-iff)

lemma ceiling-le-one [simp]: \([x] \leq 1 \iff x \leq 1\)
by (simp add: ceiling-le-iff)

lemma ceiling-le-numeral [simp]: \([x] \leq \text{numeral } v \iff x \leq \text{numeral } v\)
by (simp add: ceiling-le-iff)

lemma ceiling-le-neg-numeral [simp]: \([x] \leq - \text{numeral } v \iff x \leq - \text{numeral } v\)
by (simp add: ceiling-le-iff)

lemma ceiling-less-zero [simp]: \([x] < 0 \iff x \leq -1\)
by (simp add: ceiling-less-iff)

lemma ceiling-less-one [simp]: \([x] < 1 \iff x \leq 0\)
by (simp add: ceiling-less-iff)

lemma ceiling-less-numeral [simp]: \([x] < \text{numeral } v \iff x \leq \text{numeral } v - 1\)
by (simp add: ceiling-less-iff)

lemma ceiling-less-neg-numeral [simp]: \([x] < - \text{numeral } v \iff x \leq - \text{numeral } v - 1\)
by (simp add: ceiling-less-iff)

lemma zero-le-ceiling [simp]: \(0 \leq [x] \iff -1 < x\)
by (simp add: le-ceiling-iff)

lemma one-le-ceiling [simp]: \(1 \leq [x] \iff 0 < x\)
by (simp add: le-ceiling-iff)

lemma numeral-le-ceiling [simp]: \(\text{numeral } v \leq [x] \iff \text{numeral } v - 1 < x\)
THEORY “Archimedean-Field”

by (simp add: le-ceiling-iff)

lemma neg-numeral-le-ceiling [simp]: \(-\text{numeral } v \leq \lfloor x \rfloor \longleftrightarrow -\text{numeral } v - 1 < x\)
  by (simp add: le-ceiling-iff)

lemma zero-less-ceiling [simp]: \(0 < \lfloor x \rfloor \longleftrightarrow 0 < x\)
  by (simp add: less-ceiling-iff)

lemma one-less-ceiling [simp]: \(1 < \lfloor x \rfloor \longleftrightarrow 1 < x\)
  by (simp add: less-ceiling-iff)

lemma numeral-less-ceiling [simp]: \(\text{numeral } v < \lfloor x \rfloor \longleftrightarrow \text{numeral } v < x\)
  by (simp add: less-ceiling-iff)

lemma neg-numeral-less-ceiling [simp]: \(-\text{numeral } v < \lfloor x \rfloor \longleftrightarrow -\text{numeral } v < x\)
  by (simp add: less-ceiling-iff)

lemma ceiling-altdef: \(\lceil x \rceil = (\text{if } x = \text{of-int } \lfloor x \rfloor \text{ then } \lfloor x \rfloor \text{ else } \lfloor x \rfloor + 1)\)
  by (intro ceiling-unique; simp, linarith?)

lemma floor-le-ceiling [simp]: \(\lfloor x \rfloor \leq \lceil x \rceil\)
  by (simp add: ceiling-altdef)

96.4.2 Addition and subtraction of integers.

lemma ceiling-add-of-int [simp]: \(\lceil x + \text{of-int } z \rceil = \lceil x \rceil + z\)
  using ceiling-correct [of x]
  by (simp add: ceiling-def)

lemma ceiling-add-numeral [simp]: \(\lceil x + \text{numeral } v \rceil = \lceil x \rceil + \text{numeral } v\)
  using ceiling-add-of-int [of x numeral v]
  by simp

lemma ceiling-add-one [simp]: \(\lceil x + 1 \rceil = \lceil x \rceil + 1\)
  using ceiling-add-of-int [of x 1]
  by simp

lemma ceiling-diff-of-int [simp]: \(\lceil x - \text{of-int } z \rceil = \lceil x \rceil - z\)
  using ceiling-add-of-int [of x - z]
  by (simp add: algebra-simps)

lemma ceiling-diff-numeral [simp]: \(\lceil x - \text{numeral } v \rceil = \lceil x \rceil - \text{numeral } v\)
  using ceiling-diff-of-int [of x numeral v]
  by simp

lemma ceiling-diff-one [simp]: \(\lceil x - 1 \rceil = \lceil x \rceil - 1\)
  using ceiling-diff-of-int [of x 1]
  by simp

lemma ceiling-split [linarith-split]: \(P \[t\] \longleftrightarrow (\forall i. \text{of-int } i - 1 < t \land t \leq \text{of-int } i \rightarrow P i)\)
  by (auto simp add: ceiling-unique ceiling-correct)

lemma ceiling-diff-floor-le-1: \(\lceil x \rceil - \lfloor x \rfloor \leq 1\)
proof
  have of-int ⌈x⌉ − 1 < x using ceiling-correct[of x] by simp
  also have x < of-int ⌈x⌉ + 1 using floor-correct[of x] by simp-all
  finally have of-int ([x] − ⌈x⌉) < (of-int 2::'a) by simp
  then show ?thesis unfolding of-int-less-iff by simp
qed

lemma nat-approx-posE:
  fixes e :: 'a::{archimedean_field,floor-ceiling}
  assumes 0 < e obtains n :: nat where 1 / of-nat (Suc n) < e
proof
  have (1::'a) / of-nat (Suc (nat [1/e])) < 1 / of-int ([1/e])
  proof (rule divide-strict-left-mono)
    show (of-int [1 / e]::'a) < of-nat (Suc (nat [1 / e]))
      using assms by (simp add: field-simps)
    show (0::'a) < of-nat (Suc (nat [1 / e])) * of-int [1 / e]
      using assms by (auto simp: zero-less-mult-iff pos-add-strict)
  qed auto
  also have 1 / of-int ([1/e]) ≤ 1 / (1/e)
    by (rule divide-left-mono) (auto simp: 0 < e ceiling-correct)
  also have . . . = e by simp
  finally show 1 / of-nat (Suc (nat [1 / e])) < e
    by metis
qed

lemma ceiling-divide-upper:
  fixes q :: 'a::floor-ceiling
  shows q > 0 ⇒ p ≤ of-int (ceiling (p / q)) * q
  by (meson divide-le-eq le-of-int-ceiling)

lemma ceiling-divide-lower:
  fixes q :: 'a::floor-ceiling
  shows q > 0 ⇒ (of-int ⌈p / q⌉ − 1) * q < p
  by (meson ceiling-eq-iff pos-less-divide-eq)

96.5 Negation

lemma floor-minus: ⌈− x⌉ = − ⌈x⌉
  unfolding ceiling-def by simp

lemma ceiling-minus: ⌈− x⌉ = − ⌈x⌉
  unfolding ceiling-def by simp
96.6 Natural numbers

lemma of-nat-floor: \( r \geq 0 \implies \text{of-nat} (\text{nat } \lfloor r \rfloor) \leq r \)
  by simp

lemma of-nat-ceiling: \( \text{of-nat} (\text{nat } \lceil r \rceil) \geq r \)
  by (cases \( r \geq 0 \)) auto

lemma of-nat-int-floor [simp]: \( x \geq 0 \implies \text{of-nat} (\text{nat } \lfloor x \rfloor) = \text{of-int} \lfloor x \rfloor \)
  by auto

lemma of-nat-int-ceiling [simp]: \( x \geq 0 \implies \text{of-nat} (\text{nat } \lceil x \rceil) = \text{of-int} \lceil x \rceil \)
  by auto

96.7 Frac Function

definition frac :: 'a \Rightarrow 'a::floor-ceiling
  where frac \( x \equiv x - \text{of-int} \lfloor x \rfloor \)

lemma frac-lt-1: \( \text{frac} x < 1 \)
  by (simp add: frac-def) linarith

lemma frac-eq-0-iff [simp]: \( \text{frac} x = 0 \longleftrightarrow x \in \mathbb{Z} \)
  by (simp add: frac-def) (metis Ints-cases Ints-of-int floor-of-int)

lemma frac-ge-0 [simp]: \( \text{frac} x \geq 0 \)
  unfolding frac-def by linarith

lemma frac-gt-0-iff [simp]: \( \text{frac} x > 0 \longleftrightarrow x \notin \mathbb{Z} \)
  by (metis frac-eq-0-iff frac-ge-0 le-less less-irrefl)

lemma frac-of-int [simp]: \( \text{frac} (\text{of-int} z) = 0 \)
  by (simp add: frac-def)

lemma frac-frac [simp]: \( \text{frac} (\text{frac} x) = \text{frac} x \)
  by (simp add: frac-def)

lemma floor-add: \( \lfloor x + y \rfloor = (\text{if} \ \text{frac} x + \text{frac} y < 1 \ \text{then} \ \lfloor x \rfloor + \lfloor y \rfloor \ \text{else} \ (\lfloor x \rfloor + \lfloor y \rfloor) + 1) \)
  proof
    have \( x + y < 1 + (\text{of-int} \lfloor x \rfloor + \text{of-int} \lfloor y \rfloor) \implies \lfloor x + y \rfloor = \lfloor x \rfloor + \lfloor y \rfloor \)
      by (metis add.commute floor-unique le-floor-add le-floor-iff of-int-add)
  moreover
    have \( \neg x + y < 1 + (\text{of-int} \lfloor x \rfloor + \text{of-int} \lfloor y \rfloor) \implies \lfloor x + y \rfloor = 1 + (\lfloor x \rfloor + \lfloor y \rfloor) \)
      apply (simp add: floor-eq-iff)
      apply (auto simp add: algebra-simps)
      apply linarith
      done
  ultimately show \( \text{thesis} \) by (auto simp add: frac-def algebra-simps)
  qed
lemma floor-add2[simp]: \( x \in \mathbb{Z} \lor y \in \mathbb{Z} \implies \lfloor x + y \rfloor = \lfloor x \rfloor + \lfloor y \rfloor \)
by (metis add.commute add.left-neutral frac-lt-1 floor-add frac-eq-0-iff)

lemma frac-add:
\[
\frac{x + y}{=} (if \frac{x}{+} \frac{y}{<} \mathbb{Z} then \frac{x}{+} \frac{y}{else} (\frac{x}{+} \frac{y}{-} \mathbb{Z}))
\]
by (simp add: frac-def floor-add)

lemma frac-unique-iff: \( \frac{x}{=} a \leftrightarrow x - a \in \mathbb{Z} \land 0 \leq a \land a < 1 \)
for \( x :: \mathbb{Z} \)
apply (auto simp: Ints-def frac-def algebra-simps floor-unique)
apply linarith
done

lemma frac-eq: \( \frac{x}{=} x \leftrightarrow 0 \leq x \land x < 1 \)
by (simp add: frac-unique-iff)

lemma frac-neg: \( \frac{-x}{=} (if x \in \mathbb{Z} then 0 else 1 - \frac{x}{}) \)
for \( x :: \mathbb{Z} \)
apply (auto simp add: frac-unique-iff)
apply (simp add: frac-def)
apply (meson frac-lt-1 less-iff-diff-less-0 not-le not-less-iff-gr-or-eq)
done

lemma frac-in-Ints-iff [simp]: \( \frac{x}{=} x \leftrightarrow x \in \mathbb{Z} \)
proof safe
assume \( \frac{x}{=} x \in \mathbb{Z} \)
hence of-int \( \lfloor x \rfloor + \frac{x}{\in} \mathbb{Z} \) by auto
also have of-int \( \lfloor x \rfloor + \frac{x}{} x \in \mathbb{Z} \) by (simp add: frac-def)
finally show \( x \in \mathbb{Z} \).
qed (auto simp: frac-def)

lemma frac-1-eq: \( \frac{x+1}{=} \frac{x}{} \)
by (simp add: frac-def)

96.8 Fractional part arithmetic

Many thanks to Stepan Holub

lemma frac-non-zero: \( \frac{x}{=} 0 \implies \frac{-x}{=} 1 - \frac{x}{} \)
using frac-eq-0-iff frac-neg by metis

lemma frac-add-simps [simp]:
\[
\frac{\frac{a}{+} b}{=} \frac{a}{+} b \frac{a}{+} \frac{b}{+} \frac{a}{+} b \]
by (simp-all add: frac-add)

lemma frac-neg-frac: \( \frac{-x}{=} \frac{-x}{} \)
unfolding frac-neg frac-fra by force
lemma frac-diff-simp: \( \frac{y - x}{y + \frac{1}{2}} = \frac{y}{x} \)

unfolding diff-conv-add-uminus frac-add frac-neg frac...

lemma frac-diff: \( \frac{a - b}{a + (-b)} \)

unfolding frac-add-simps(1)

unfolding ab-group-add-class.ab-diff-conv-add-uminus[symmetric] frac-diff-simp...

lemma frac-diff-pos: \( a = \frac{y}{x} \) 

show \( \frac{y}{x} = \frac{y - x}{x} \)

using frac-diff-pos by force

lemma frac-diff-neg: assumes \( \frac{y}{x} \) < \( \frac{y}{x} \)

shows \( \frac{y}{x} - \frac{x}{x} \)

proof

have \( x \notin \mathbb{Z} \)

unfolding frac-gt-0 iff [symmetric]

using assms frac-ge-0[of y] by order

have \( frac y + (1 + - frac x) < 1 \)

using frac-lt-1[of x] assms by fastforce

show \( \frac{y}{x} < \frac{y}{x} \)

using frac-diff-conv-add-uminus frac-add frac-neg

if-not-P[\( of x \notin \mathbb{Z} \)] if-P[of \( \frac{y}{x} + (1 + - frac x) < 1 \)]

by simp

qed

lemma frac-diff-eq: assumes \( \frac{y}{x} = \frac{y}{x} \)

shows \( \frac{y}{x} - \frac{x}{x} \)

by (simp add: assms frac-diff-pos)

lemma frac-diff-zero: assumes \( \frac{y}{x} = \frac{y}{x} \)

shows \( \frac{y}{x} = \frac{y}{x} \)

using frac-add-simps(1)[of \( \frac{y}{x} = \frac{y}{x} \), symmetric]

unfolding assms add.group-left-neutral diff-add-cancel.

lemma frac-neg-eq-iff: \( \frac{-x}{-y} \leftrightarrow \frac{x}{y} \)

using add.inverse-inverse frac-neg frac by metis

96.9 Rounding to the nearest integer

definition round :: 'a::floor-ceiling \Rightarrow int

where round \( x = \lfloor x + \frac{1}{2} \rfloor \)

lemma of-int-round-ge: \( of-int (round x) \geq x - \frac{1}{2} \)

and of-int-round-le: \( of-int (round x) \leq x + \frac{1}{2} \)

and of-int-round-abs-le: \( |of-int (round x) - x| \leq \frac{1}{2} \)

and of-int-round-gt: \( of-int (round x) > x - \frac{1}{2} \)

proof

from floor-correct[of \( x + \frac{1}{2} \)] have \( x + \frac{1}{2} < of-int (round x) + \frac{1}{2} \)
by (simp add: round-def)
from add-strict-right-mono[OF this, of -1] show A: of-int (round x) > x - 1/2
  by simp
then show of-int (round x) ≥ x - 1/2
  by simp
from floor-correct[of x + 1/2] show of-int (round x) ≤ x + 1/2
  by (simp add: round-def)
with A show \|of-int (round x) - x\| ≤ 1/2
  by linarith
qed

lemma round-of-int [simp]: round (of-int n) = n
  unfolding round-def by (subst floor-eq-iff) force

lemma round-0 [simp]: round 0 = 0
  using round-of-int[of 0] by simp

lemma round-1 [simp]: round 1 = 1
  using round-of-int[of 1] by simp

lemma round-numeral [simp]: round (numeral n) = numeral n
  using round-of-int[of numeral n] by simp

lemma round-neg-numeral [simp]: round (-numeral n) = -numeral n
  using round-of-int[of -numeral n] by simp

lemma round-of-nat [simp]: round (of-nat n) = of-nat n
  using round-of-int[of int n] by simp

lemma round-mono: x ≤ y ⇒ round x ≤ round y
  unfolding round-def
proof (rule floor-unique)
  assume x - 1 / 2 < of-int y
  from add-strict-left-mono[OF this, of 1] show x + 1 / 2 < of-int y + 1
    by simp
qed

lemma round-unique': \|x - of-int n\| < 1/2 ⇒ round x = n
  by (subst (asm) abs-less-iff, rule round-unique) (simp-all add: field-simps)

lemma round-altdef: round x = (if frac x ≥ 1/2 then \lfloor x\rfloor else \lceil x\rceil)
  by (cases frac x ≥ 1/2)
  (rule round-unique, (simp add: frac-def field-simps ceiling-altdef; linarith)+)[2])+

lemma floor-le-round: \lfloor x\rfloor ≤ round x
unfolding round-def by (intro floor-mono) simp

lemma ceiling-ge-round: \( \lceil x \rceil \geq \text{round } x \)
unfolding round-altdef by simp

lemma round-diff-minimal: \(|z - \text{of-int} \ (\text{round } z)| \leq |z - \text{of-int } m|\)
proof (cases of-int m \geq z)
case True
then have \(|z - \text{of-int} \ (\text{round } z)| \leq |\text{of-int } \lceil z \rceil - z|\)
unfolding round-altdef by (simp add: field-simps ceiling-altdef frac-def) linarith
also have \(|z - \text{of-int } \lceil z \rceil| \geq 0\)
by linarith
with True have \(|z - \text{of-int } \lceil z \rceil - z| \leq |z - \text{of-int } m|\)
by (simp add: ceiling-le-iff)
finally show \(\text{thesis}\).
next
case False
then have \(|z - \text{of-int} \ (\text{round } z)| \leq |\text{of-int } \lfloor z \rfloor - z|\)
unfolding round-altdef by (simp add: field-simps ceiling-altdef frac-def) linarith
also have \(|z - \text{of-int } \lfloor z \rfloor| \geq 0\)
by linarith
with False have \(|z - \text{of-int } \lfloor z \rfloor - z| \leq |z - \text{of-int } m|\)
by (simp add: le-floor-iff)
finally show \(\text{thesis}\).
qed

end

97 Rational numbers

theory Rat
imports Archimedean-Field
begin

97.1 Rational numbers as quotient

97.1.1 Construction of the type of rational numbers

definition ratrel :: (int \times int) \Rightarrow (int \times int) \Rightarrow bool
where ratrel = (\lambda x y. snd x \neq 0 \land snd y \neq 0 \land fst x * snd y = fst y * snd x)

lemma ratrel-iff [simp]:
ratrel x y \iff snd x \neq 0 \land snd y \neq 0 \land fst x * snd y = fst y * snd x
by (simp add: ratrel-def)

lemma exists-ratrel-refl: \(\exists x. \text{ratrel } x x\)
by (auto intro!: one-neq-zero)
lemma symp-ratrel: symp ratrel
  by (simp add: ratrel-def symp-def)

lemma transp-ratrel: transp ratrel
proof (rule transpI, unfold split-paired-all)
  fix a b a' b' a'' b'' :: int
  assume *: ratrel (a, b) (a', b')
  assume **: ratrel (a', b') (a'', b'')
  have b' * (a * b'') = b'' * (a * b') by simp
  also from * have a * b' = a' * b by auto
  also have b'' * (a' * b) = b * (a' * b'') by simp
  also from ** have a' * b'' = a'' * b' by auto
  also have b * (a'' * b') = b' * (a'' * b) by simp
  moreover from ** have b' ≠ 0 by auto
  ultimately have a * b'' = a'' * b by simp
  with * ** show ratrel (a, b) (a'', b'') by auto
qed

lemma part-equivp-ratrel: part-equivp ratrel
  by (rule part-equivpI [OF exists-ratrel-refl symp-ratrel transp-ratrel])

quotient-type rat = int × int / partial: ratrel
  morphisms Rep-Rat Abs-Rat
  by (rule part-equivp-ratrel)

lemma Domainp-cr-rat [transfer-domain-rule]: Domainp pcr-rat = (λx. snd x ≠ 0)
  by (simp add: rat.domain-eq)

97.1.2 Representation and basic operations

lift-definition Fract :: int ⇒ int ⇒ rat
  is λa b. if b = 0 then (0, 1) else (a, b)
  by simp

lemma eq-rat:
  ∀a b c d. b ≠ 0 ⟹ d ≠ 0 ⟹ Fract a b = Fract c d ⟷ a * d = c * b
  ∀a. Fract a 0 = Fract 0 1
  ∀a c. Fract 0 a = Fract 0 c
  by (transfer, simp)+

lemma Rat-cases [case-names Fract, cases type: rat]:
  assumes that: ∀a b. q = Fract a b ⟹ b > 0 ⟹ coprime a b ⟹ C
  shows C
proof –
  obtain a b :: int where q = Fract a b and b ≠ 0
    by transfer simp
  let ?a = a div gcd a b
let $b = b \div \gcd a b$
from $b$ have $b * \gcd a b = b$
  by simp
with $b$ have $b \neq 0$
  by fastforce
with $q \ b$ have $q2 \cdot q = \text{Fract} \ ?a \ ?b$
  by (simp add: eq-rat dvd-div-mult mult.commute [of $a$])
from $b$ have coprime: coprime $?a \ ?b$
  by (auto intro: div-gcd-coprime)
show $C$
proof (cases $b > 0$
  case True
  then have $?b > 0$
    by (simp add: nonneg1-imp-zdiv-pos-iff)
  from $q2$ this coprime show $C$ by (rule that)
next
  case False
  have $q = \text{Fract} (- ?a) (- ?b)$
    unfolding $q2$ by transfer simp
  moreover from False $b$ have $- ?b > 0$
    by (simp add: pos-imp-zdiv-neg-iff)
  moreover from coprime have coprime $(- ?a) (- ?b)$
    by simp
  ultimately show $C$
    by (rule that)
qed

lemma Rat-induct [case-names Fract, induct type: rat]:
assumes $\bigwedge a \ b. \ b > 0 \implies \text{coprime } a \ b \implies P \ (\text{Fract } a \ b)$
shows $P \ q$
using assms by (cases $q$) simp

instantiation rat :: field
begin

lift-definition zero-rat :: rat is $(0, 1)$
  by simp

lift-definition one-rat :: rat is $(1, 1)$
  by simp

lemma Zero-rat-def: $0 = \text{Fract} 0 1$
  by transfer simp

lemma One-rat-def: $1 = \text{Fract} 1 1$
  by transfer simp

lift-definition plus-rat :: rat $\Rightarrow$ rat $\Rightarrow$ rat
is \( \lambda x \ y. \ (\text{fst} \ x \ * \ \text{snd} \ y + \ \text{fst} \ y \ * \ \text{snd} \ x, \ \text{snd} \ x \ * \ \text{snd} \ y) \)

by (auto simp: distrib-right) (simp add: ac-simps)

lemma add-rat [simp]:
  assumes \( b \neq 0 \) and \( d \neq 0 \)
  shows \( \text{Fract} \ a \ b + \text{Fract} \ c \ d = \text{Fract} \ (a \ * \ d + c \ * \ b) \ (b \ * \ d) \)
  using assms by transfer simp

lift-definition uminus-rat :: \text{rat} \\
  is \( \lambda x. \ (\neg \ \text{fst} \ x, \ \text{snd} \ x) \)

by simp

lemma minus-rat [simp]: \( - \text{Fract} \ a \ b = \text{Fract} \ (\neg \ a) \ b \)

by transfer simp

lemma minus-rat-cancel [simp]: \( \text{Fract} \ (\neg \ a) \ (\neg \ b) = \text{Fract} \ a \ b \)

by (cases \( b = 0 \)) (simp-all add: eq-rat)

definition diff-rat-def: \( q - r = q + - r \) for \( q \ r :: \text{rat} \)

lemma diff-rat [simp]: \( b \neq 0 \Rightarrow d \neq 0 \Rightarrow \text{Fract} \ a \ b - \text{Fract} \ c \ d = \text{Fract} \ (a \ * \ d - c \ * \ b) \ (b \ * \ d) \)

by (simp add: diff-rat-def)

lift-definition times-rat :: \text{rat} \\
  is \( \lambda x \ y. \ (\text{fst} \ x \ * \ \text{fst} \ y, \ \text{snd} \ x \ * \ \text{snd} \ y) \)

by (simp add: ac-simps)

lemma mult-rat [simp]: \( \text{Fract} \ a \ b \ * \ \text{Fract} \ c \ d = \text{Fract} \ (a \ * \ c) \ (b \ * \ d) \)

by transfer simp

lemma mult-rat-cancel: \( c \neq 0 \Rightarrow \text{Fract} \ (c \ * \ a) \ (c \ * \ b) = \text{Fract} \ a \ b \)

by transfer simp

lift-definition inverse-rat :: \text{rat} \\
  is \( \lambda x. \) if \( \text{fst} \ x = 0 \) then \( (0, 1) \) else \( \text{snd} \ x, \ \text{fst} \ x \)

by (auto simp add: mult.commute)

lemma inverse-rat [simp]: \( \text{inverse} \ (\text{Fract} \ a \ b) = \text{Fract} \ b \ a \)

by transfer simp

definition divide-rat-def: \( q \ \text{div} \ r = q \ \text{inverse} \ r \) for \( q \ r :: \text{rat} \)

lemma divide-rat [simp]: \( \text{Fract} \ a \ b \ \text{div} \ \text{Fract} \ c \ d = \text{Fract} \ (a \ * \ d) \ (b \ * \ c) \)

by (simp add: divide-rat-def)

instance

proof
  fix \( q \ r \ s :: \text{rat} \)

  show \( (q \ * \ r) \ * \ s = q \ * \ (r \ * \ s) \)
by transfer simp
show \( q \cdot r = r \cdot q \)
  by transfer simp
show \( 1 \cdot q = q \)
  by transfer simp
show \( (q + r) + s = q + (r + s) \)
  by transfer \( (\text{simp add: algebra-simps}) \)
show \( q + r = r + q \)
  by transfer simp
show \( 0 + q = q \)
  by transfer simp
show \( -(q + q) = 0 \)
  by transfer simp
show \( q - r = q + (-r) \)
  by \( (\text{fact diff-rat-def}) \)
show \( (q + r) \cdot s = q \cdot s + r \cdot s \)
  by transfer \( (\text{simp add: algebra-simps}) \)
show \( (0 :: \text{rat}) \neq 1 \)
  by transfer simp
show \( \text{inverse } q \cdot q = 1 \) if \( q \neq 0 \)
  using that by transfer simp
show \( q \div r = q \cdot \text{inverse } r \)
  by \( (\text{fact divide-rat-def}) \)
show \( \text{inverse } 0 = (0 :: \text{rat}) \)
  by transfer simp

qed

end

lemma \text{div-add-self1-no-field \ [simp]}:
  assumes \text{NO-MATCH} \( (x :: 'b :: \text{field}) \ b \ (b :: 'a :: \text{euclidean-semiring-cancel}) \neq 0 \)
  shows \( (b + a) \div b = a \div b + 1 \)
  using \text{assms}(2) by \( (\text{fact div-add-self1}) \)

lemma \text{div-add-self2-no-field \ [simp]}:
  assumes \text{NO-MATCH} \( (x :: 'b :: \text{field}) \ b \ (b :: 'a :: \text{euclidean-semiring-cancel}) \neq 0 \)
  shows \( (a + b) \div b = a \div b + 1 \)
  using \text{assms}(2) by \( (\text{fact div-add-self2}) \)

lemma \text{of-nat-rat}: \text{of-nat } k = \text{Fract } (\text{of-nat } k) 1
  by \( (\text{induct } k) \ (\text{simp-all add: Zero-rat-def One-rat-def}) \)

lemma \text{of-int-rat}: \text{of-int } k = \text{Fract } k 1
  by \( (\text{cases } k \ \text{rule: int-diff-cases}) \ (\text{simp add: of-nat-rat}) \)

lemma \text{Fract-of-nat-eq}: \text{Fract } (\text{of-nat } k) 1 = \text{of-nat } k
THEORY "Rat"

by (rule of-nat-rat [symmetric])

lemma Fract-of-int-eq: Fract k 1 = of-int k
  by (rule of-int-rat [symmetric])

lemma rat-number-collapse:
  Fract 0 k = 0
  Fract 1 1 = 1
  Fract (numeral w) 1 = numeral w
  Fract (− numeral w) 1 = − numeral w
  Fract (− 1) 1 = − 1
  Fract k 0 = 0
  using Fract-of-int-eq [of numeral w]
  and Fract-of-int-eq [of − numeral w]
  by (simp-all add: Zero-rat-def One-rat-def eq-rat)

lemma rat-number-expand:
  0 = Fract 0 1
  1 = Fract 1 1
  numeral k = Fract (numeral k) 1
  − 1 = Fract (− 1) 1
  − numeral k = Fract (− numeral k) 1
  by (simp-all add: rat-number-collapse)

lemma Rat-cases-nonzero [case-names Fract 0]:
  assumes Fract: ⋀ a b. q = Fract a b ≡ b > 0 ≡ a ≠ 0 ≡ coprime a b ≡
  C
  and 0: q = 0 ⇒ C
  shows C
  proof (cases q = 0)
    case True
    then show C using 0 by auto
    next
    case False
    then obtain a b where *: q = Fract a b > 0 coprime a b
    by (cases q) auto
    with False have 0 ≠ Fract a b
    by simp
    with *b > 0* have a ≠ 0
    by (simp add: Zero-rat-def eq-rat)
    with Fract * show C by blast
  qed

97.1.3 Function normalize

lemma Fract-coprime: Fract (a div gcd a b) (b div gcd a b) = Fract a b
  proof (cases b = 0)
    case True
    then show ?thesis
by (simp add: eq-rat)

next

case False

moreover have \( b \div \gcd a b = b \)
  by (rule dvd-div-mult-self) simp

ultimately have \( b \div \gcd a b \neq 0 \)
  by simp

then have \( b \div \gcd a b \neq 0 \)
  by fastforce

with False show ?thesis
  by (simp add: eq-rat dvd-div-mult mult.commute [of a])

qed

definition normalize :: \( \mathbb{int} \times \mathbb{int} \Rightarrow \mathbb{int} \times \mathbb{int} \)
where normalize \( p = \)
  (if snd \( p \) > 0 then (fst \( p \) = gcd (fst \( p \)) (snd \( p \)) in (fst \( p \) div a, snd \( p \) div a))
  else if snd \( p \) = 0 then \( (0, 1) \)
  else (let a = \( -\ gcd (fst \( p \)) (snd \( p \)) in (fst \( p \) div a, snd \( p \) div a))))

lemma normalize-crossproduct:
  assumes \( q \neq 0 s \neq 0 \)
  assumes normalize \((p, q) = \) normalize \((r, s)\)
  shows \( p * s = r * q \)
proof –
  have \(*\): \( p * s = q * r \)
    if \( p * gcd r s = sgn (q * s) * r * gcd p q \) and \( q * gcd r s = sgn (q * s) * s * gcd p q \)

proof –
  from that have \((p * gcd r s) * (sgn (q * s) * s * gcd p q) = \)
    \((q * gcd r s) * (sgn (q * s) * r * gcd p q)\)
  by simp

with assms show ?thesis
  by (auto simp add: ac-simps sgn-mult sgn-0-0)

qed

from assms show ?thesis
  by (auto simp: normalize-def Let-def dvd-div-div-eq-mult mult.commute sgn-mult split: if-splits intro: *)

qed

lemma normalize-eq: normalize \((a, b) = (p, q) \) \( \Rightarrow \) Fract \( p q = \) Fract \( a b \)
by (auto simp: normalize-def Let-def dvd-div-div-eq-mult mult.commute sgn-mult split: if-splits)

lemma normalize-denom-pos: normalize \( r = (p, q) \) \( \Rightarrow \) \( q > 0 \)

lemma normalize-coprime: normalize \( r = (p, q) \) \( \Rightarrow \) coprime \( p q \)
by (auto simp: normalize-def Let-def dvd-div-neg div-gcd-coprime split: if-splits)
lemma normalize-stable [simp]: $q > 0 \implies \text{coprime } p q \implies \text{normalize } (p, q) = (p, q)$
  by (simp add: normalize-def)

lemma normalize-denom-zero [simp]: $\text{normalize } (p, 0) = (0, 1)$
  by (simp add: normalize-def)

lemma normalize-negative [simp]: $q < 0 \implies \text{normalize } (p, q) = \text{normalize } (-p, -q)$
  by (simp add: normalize-def Let-def dvd-div-neg dvd-neg-div)

Decompose a fraction into normalized, i.e. coprime numerator and denominator:

definition quotient-of :: rat $\Rightarrow$ int $\times$ int
  where quotient-of $x$ = (THE pair. $x = \text{Fract } \text{fst } \text{pair} \text{snd } \text{pair} \land \text{snd } \text{pair} > 0 \land \text{coprime } \text{fst } \text{pair} \text{snd } \text{pair})$ (snd pair))

lemma quotient-of-unique: $\exists! p. r = \text{Fract } \text{fst } p \text{snd } p \land \text{snd } p > 0 \land \text{coprime } \text{fst } p \text{snd } p$
  proof (cases $r$)
    case (Fract $a b$)
    then have $r = \text{Fract } \text{fst } (a, b) \text{snd } (a, b) \land \text{snd } (a, b) > 0 \land \text{coprime } \text{fst } (a, b) \text{snd } (a, b)$
      by auto
    then show $?thesis$
      proof (rule ex1I)
        fix $p$
        assume $r: r = \text{Fract } \text{fst } p \text{snd } p \land \text{snd } p > 0 \land \text{coprime } \text{fst } p \text{snd } p$
        obtain $c d$ where $p = (c, d)$ by (cases $p$)
        with $r$ have $\text{Fract'}: r = \text{Fract } c d d > 0 \text{ coprime } c d$
          by simp-all
        have $(c, d) = (a, b)$
          proof (cases $a = 0$)
            case True
            with $\text{Fract Fract'}$ show $?thesis$
              by (simp add: eq-rat)
          next
            case False
            with $\text{Fract Fract'}$ have $*: c * b = a * d \land c \neq 0$
              by (auto simp add: eq-rat)
            then have $c * b > 0 \iff a * d > 0$
              by auto
            with $(b > 0) \land (d > 0)$ have $a > 0 \iff c > 0$
              by (simp add: zero-less-mult-iff)
            with $(a \neq 0) \land (c \neq 0)$ have $\text{sgn: sgn } a = \text{sgn } c$
              by (auto simp add: not-less)
            from $(\text{coprime } a b) \land (\text{coprime } c d)$ have $|a| * |d| = |c| * |b| \iff |a| = |c| \land$
\(|d| = |b|\)

\[\begin{align*}
&|d| = |b| \\
&\text{by (\textit{simp add: coprime-crossproduct-int})} \\
&\text{with } \langle b > 0 \rangle \cdot (d > 0) \text{ have } |a| \times d = |c| \times b \iff |a| = |c| \land d = b \\
&\text{by \textit{simp}} \\
&\text{then have } a \times \text{sgn } a \times d = c \times \text{sgn } c \times b \iff a \times \text{sgn } a = c \times \text{sgn } c \land d = b \\
&\text{by (\textit{simp add: abs-sgn})} \\
&\text{with } \text{sgn} \times \text{show } \theta\text{thesis} \\
&\text{by (auto \textit{simp add: sgn-0-0})} \\
&\text{qed} \\
&\text{with } \text{show } p = (a, b) \\
&\text{by \textit{simp}} \\
&\text{qed} \\
&\text{qed} \\
&\text{lemma quotient-of-Fract [code]: quotient-of } (\text{Fract } a b) = \text{normalize } (a, b) \\
&\text{proof} \\
&\text{have } \text{Fract } a b = \text{Fract } (fst (\text{normalize } (a, b))) (snd (\text{normalize } (a, b))) \text{ (is } \?\text{Fract)} \\
&\text{by (rule \textit{sym}) (auto intro: normalize-eq)} \\
&\text{moreover have } 0 < \text{snd } (\text{normalize } (a, b)) \text{ (is } \?\text{denom-pos)} \\
&\text{by (cases \text{normalize } (a, b)) (rule normalize-denom-pos, \textit{simp})} \\
&\text{moreover have } \text{coprime } (fst (\text{normalize } (a, b))) (snd (\text{normalize } (a, b))) \text{ (is } \?\text{coprime)} \\
&\text{by (rule normalize-coprime) \textit{simp}} \\
&\text{ultimately have } \theta\text{Fract } \land \theta\text{denom-pos } \land \theta\text{coprime by \textit{blast}} \\
&\text{then have } (\text{THE } p. \text{Fract } a b = \text{Fract } (fst p) (snd p) \land 0 < \text{snd } p \land \\
&\text{coprime } (fst p) (snd p)) = \text{normalize } (a, b) \\
&\text{by (rule the1-equality [OF quotient-of-unique])} \\
&\text{then show } \theta\text{thesis by (simp add: quotient-of-def)} \\
&\text{qed} \\
&\text{lemma quotient-of-number [simp]:} \\
&\text{quotient-of } 0 = (0, 1) \\
&\text{quotient-of } 1 = (1, 1) \\
&\text{quotient-of } (\text{numeral } k) = (\text{numeral } k, 1) \\
&\text{quotient-of } (- 1) = (- 1, 1) \\
&\text{quotient-of } (- \text{ numeral } k) = (- \text{ numeral } k, 1) \\
&\text{by (simp-all add: rat-number-expand quotient-of-Fract)} \\
&\text{lemma quotient-of-eq: quotient-of } (\text{Fract } a b) = (p, q) \Longrightarrow \text{Fract } p q = \text{Fract } a b \\
&\text{by (simp add: quotient-of-Fract normalize-eq)} \\
&\text{lemma quotient-of-denom-pos: quotient-of } r = (p, q) \Longrightarrow q > 0 \\
&\text{by (cases } r \rangle \text{ simp add: quotient-of-Fract normalize-denom-pos)} \\
&\text{lemma quotient-of-denom-pos': } \text{snd } (\text{quotient-of } r) > 0 \\
&\text{using quotient-of-denom-pos [of } r \rangle \text{ by (simp add: prod-eq-iff)} \\
&\text{lemma quotient-of-coprime: quotient-of } r = (p, q) \Longrightarrow \text{coprime } p q
by (cases r) (simp add: quotient-of-Fract normalize-coprime)

lemma quotient-of-inject:
  assumes quotient-of a = quotient-of b
  shows a = b
proof
  obtain p q r s where a = Fract p q and b = Fract r s and q > 0 and s > 0
    by (cases a, cases b)
  with assms show ?thesis
    by (simp add: eq-rat quotient-of-Fract normalize-crossproduct)
qed

lemma quotient-of-inject-eq: quotient-of a = quotient-of b ⟷ a = b
by (auto simp add: quotient-of-inject)

97.1.4 Various

lemma Fract-of-int-quotient: Fract k l = of-int k / of-int l
by (simp add: Fract-of-int-eq [symmetric])

lemma Fract-add-one: n ≠ 0 ⟹ Fract (m + n) n = Fract m n + 1
by (simp add: rat-number-expand)

lemma quotient-of-div:
  assumes r: quotient-of r = (n,d)
  shows r = of-int n / of-int d
proof
  from theI' [OF quotient-of-unique [of r], unfolded r [unfolded quotient-of-def]]
  have r = Fract n d by simp
  then show ?thesis using Fract-of-int-quotient
    by simp
qed

97.1.5 The ordered field of rational numbers

lift-definition positive :: rat ⇒ bool
  is λx. 0 < fst x * snd x
proof clarsimp
  fix a b c d :: int
  assume b ≠ 0 and d ≠ 0 and a * d = c * b
  then have a * d * b * d = c * b * b * d
    by simp
  then have a * b * d^2 = c * d * b^2
    unfolding power2_eq_square by (simp add: ac_simps)
  then have 0 < a * b * d^2 ⟷ 0 < c * d * b^2
    by simp
  then show 0 < a * b ⟷ 0 < c * d
    using (b ≠ 0) and (d ≠ 0)
    by (simp add: zero_less_mult_iff)
lemma positive-zero: \( \neg \text{positive } 0 \)
by transfer simp

lemma positive-add: positive \( x \implies \text{positive } y \implies \text{positive } (x + y) \)
apply transfer
apply (auto simp add: zero-less-mult-iff add-pos-pos add-neg-neg mult-pos-pos mult-neg-neg)
done

lemma positive-mult: positive \( x \implies \text{positive } y \implies \text{positive } (x \ast y) \)
apply transfer
by (metis fst-conv mult.commute mult-pos-neg2 snd-conv zero-less-mult-iff)

lemma positive-minus: \( \neg \text{positive } x \implies x \neq 0 \implies \text{positive } (\neg x) \)
by transfer (auto simp: neq-iff zero-less-mult-iff mult-less-0-iff)

instantiation rat :: linordered-field
begin

definition \( x < y \iff \text{positive } (y - x) \)
definition \( x \leq y \iff x < y \lor x = y \) for \( x y :: \text{rat} \)
definition \(|a| = (\text{if } a < 0 \text{ then } \neg a \text{ else } a)\) for \( a :: \text{rat} \)
definition \( \text{sgn } a = (\text{if } a = 0 \text{ then } 1 \text{ else } -1)\) for \( a :: \text{rat} \)

instance proof
fix \( a b c :: \text{rat} \)
show \(|a| = (\text{if } a < 0 \text{ then } \neg a \text{ else } a)\)
by (rule abs-rat-def)
show \( a < b \iff a \leq b \land \neg b \leq a \)
unfolding less-eq-rat-def less-rat-def
using positive-add positive-zero by force
show \( a \leq a \)
unfolding less-eq-rat-def by simp
show \( a \leq b \implies b \leq c \implies a \leq c \)
unfolding less-eq-rat-def less-rat-def
using positive-add by fastforce
show \( a \leq b \implies b \leq a \implies a = b \)
unfolding less-eq-rat-def less-rat-def
using positive-add positive-zero by fastforce
show \( a \leq b \implies a + b \leq c + b \)
unfolding less-eq-rat-def less-rat-def by auto
show \( \text{sgn } a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } -1)\)
by (rule sgn-rat-def)
show a ≤ b ∨ b ≤ a
  unfolding less-eq-rat-def less-rat-def
  by (auto dest!: positive-minus)
show a < b ⇒ 0 < c ⇒ c * a < c * b
  unfolding less-rat-def
  by (metis diff-zero positive-mult right-diff-distrib)
qed

end

instantiation rat :: distrib-lattice
begin
  definition (inf :: rat ⇒ rat ⇒ rat) = min
  definition (sup :: rat ⇒ rat ⇒ rat) = max
  instance
    by standard (auto simp add: inf-rat-def sup-rat-def max-min-distrib2)
end

lemma positive-rat: positive (Fract a b) ←→ 0 < a * b
  by transfer simp

lemma less-rat [simp]:
  b ≠ 0 ⇒ d ≠ 0 ⇒ Fract a b < Fract c d ←→ (a * d) * (b * d) < (c * b) *
  (b * d)
  by (simp add: less-rat-def positive-rat algebra-simps)

lemma le-rat [simp]:
  b ≠ 0 ⇒ d ≠ 0 ⇒ Fract a b ≤ Fract c d ←→ (a * d) * (b * d) ≤ (c * b) *
  (b * d)
  by (simp add: le-less eq-rat)

lemma abs-rat [simp, code]: |Fract a b| = Fract |a| |b|
  by (auto simp add: abs-rat-def zabs-def Zero-rat-def not-less le-less eq-rat zero-less-mult-iff)

lemma sgn-rat [simp, code]: sgn (Fract a b) = of-int (sgn a * sgn b)
  unfolding Fract-of-int-eq
  by (auto simp: zsgn-def sgn-rat-def Zero-rat-def eq-rat)
    (auto simp: rat-number-collapse not-less le-less zero-less-mult-iff)

lemma Rat-induct-pos [case-names Fract, induct type: rat]:
  assumes step: ∀a b. 0 < b ⇒ P (Fract a b)
  shows P q
  proof (cases q)
    case (Fract a b)
    have step': P (Fract a b) if b: b < 0 for a b :: int
proof – 
from $b$ have $0 < - b$
  by simp
then have $P (\text{Fract} (- a) (- b))$
  by (rule step)
then show $P (\text{Fract} a b)$
  by (simp add: order-less-imp-not-eq [OF $b$])
qed
from Fract show $P q$
  by (auto simp add: linorder-neq_iff step step')
qed

lemma zero-less-Fract-iff: $0 < b \Rightarrow 0 < \text{Fract} a b \longleftrightarrow 0 < a$
  by (simp add: Zero-rat_def zero-less_mult_iff)

lemma Fract-less-zero-iff: $0 < b \Rightarrow \text{Fract} a b < 0 \longleftrightarrow a < 0$
  by (simp add: Zero-rat_def mult_less_0_iff)

lemma zero-le-Fract-iff: $0 < b \Rightarrow 0 \leq \text{Fract} a b \longleftrightarrow 0 \leq a$
  by (simp add: Zero-rat_def zero_le_mult_iff)

lemma Fract-le-zero-iff: $0 < b \Rightarrow \text{Fract} a b \leq 0 \longleftrightarrow a \leq 0$
  by (simp add: Zero-rat_def mult_le_0_iff)

lemma one-less-Fract-iff: $0 < b \Rightarrow 1 < \text{Fract} a b \longleftrightarrow b < a$
  by (simp add: One-rat_def mult_less_cancel_right_disj)

lemma Fract-less-one-iff: $0 < b \Rightarrow \text{Fract} a b < 1 \longleftrightarrow a < b$
  by (simp add: One-rat_def mult_less_cancel_right_disj)

lemma one-le-Fract-iff: $0 < b \Rightarrow 1 \leq \text{Fract} a b \longleftrightarrow b \leq a$
  by (simp add: One-rat_def mult_le_cancel_right)

lemma Fract-le-one-iff: $0 < b \Rightarrow \text{Fract} a b \leq 1 \longleftrightarrow a \leq b$
  by (simp add: One-rat_def mult_le_cancel_right)

97.1.6 Rationals are an Archimedean field

lemma rat-floor-lemma: of-int $(a \div b) \leq \text{Fract} a b \land \text{Fract} a b < of-int (a \div b + 1)$
proof –
  have $\text{Fract} a b = \text{of-int} (a \div b) + \text{Fract} (a \mod b) b$
    by (cases $b = 0$) (simp, simp add: of-int-rat)
moreover have $0 \leq \text{Fract} (a \mod b) b \land \text{Fract} (a \mod b) b < 1$
  unfolding Fract-of-int_quotient
    by (rule linorder_cases [of $b 0$]) (simp-all add: divide_nonpos_neg)
ultimately show $?thesis$ by simp
qed
instance rat :: archimedean-field
proof
  show \( \exists z. r \leq \text{of-int } z \) for \( r :: \text{rat} \)
proof (induct \( r \))
  case (Fract a b)
  have Fract a b \( \leq \) of-int (a div b + 1)
  using rat-floor-lemma \([a b]\) by simp
  then show \( \exists z. \text{Fract } a b \leq \text{of-int } z \) ..
qed

instantiation rat :: floor-ceiling
begin

definition floor-rat :: \( \text{rat} \Rightarrow \text{int} \)
where \( \lfloor x \rfloor = (\text{THE } z. \text{of-int } z \leq x \land x < \text{of-int } (z + 1)) \) for \( x :: \text{rat} \)

instance
proof
  show \( \text{of-int } \lfloor x \rfloor \leq x \land x < \text{of-int } \lfloor x \rfloor + 1 \) for \( x :: \text{rat} \)
    unfolding floor-rat-def using floor-exists1 by (rule theI’)
qed

end

lemma floor-Fract [simp]: \( \lfloor \text{Fract } a b \rfloor = a \text{ div } b \)
  by (simp add: Fract-of-int-quotient floor-divide-of-int-eq)

97.2 Linear arithmetic setup

declaration \( K \) \( (\text{Lin-Arith.add-inj-thms @\{thms of-int-le-iff [THEN iffD2] of-int-eq-iff [THEN iffD2]\}}) \)

\( \text{RS iffD2} *\)

\( \text{#> Lin-Arith.add-inj-const (const-name \text{of-nat}, typ nat \Rightarrow \text{rat})} \)

\( \text{#> Lin-Arith.add-inj-const (const-name \text{of-int}, typ int \Rightarrow \text{rat})} \)

97.3 Embedding from Rationals to other Fields

context field-char-0
begin

lift-definition of-rat :: \( \text{rat} \Rightarrow \text{a} \)
  is \( \lambda x. \text{of-int } (\text{fst } x) / \text{of-int } (\text{snd } x) \)
  by (auto simp: nonzero-divide-eq-eq nonzero-eq-divide-eq (simp only: of-int-mult [symmetric]))

end
lemma of-rat-rat: \( b \neq 0 \implies \text{of-rat} (\text{Fract} \ a \ b) = \text{of-int} \ a / \text{of-int} \ b \)
by transfer simp

lemma of-rat-0 [simp]: \( \text{of-rat} \ 0 = 0 \)
by transfer simp

lemma of-rat-1 [simp]: \( \text{of-rat} \ 1 = 1 \)
by transfer simp

lemma of-rat-add: \( \text{of-rat} (a + b) = \text{of-rat} \ a + \text{of-rat} \ b \)
by transfer (simp add: add-frac-eq)

lemma of-rat-minus: \( \text{of-rat} (- a) = - \text{of-rat} \ a \)
by transfer simp

lemma of-rat-neg-one [simp]: \( \text{of-rat} (-1) = -1 \)
by (simp add: of-rat-minus)

lemma of-rat-diff: \( \text{of-rat} (a - b) = \text{of-rat} \ a - \text{of-rat} \ b \)
using of-rat-add [of a - b] by (simp add: of-rat-minus)

lemma of-rat-mult: \( \text{of-rat} (a * b) = \text{of-rat} \ a * \text{of-rat} \ b \)
by transfer (simp add: divide-inverse nonzero-inverse-mult-distrib ac-simps)

lemma of-rat-sum: \( \text{of-rat} \sum\limits_a \in A. f a = \sum\limits_a \in A. \text{of-rat} (f a) \)
by (induct rule: infinite-finite-induct) (auto simp: of-rat-add)

lemma of-rat-prod: \( \text{of-rat} \prod\limits_a \in A. f a = \prod\limits_a \in A. \text{of-rat} (f a) \)
by (induct rule: infinite-finite-induct) (auto simp: of-rat-mult)

lemma nonzero-of-rat-inverse: \( a \neq 0 \implies \text{of-rat} \text{inverse} \ a = \text{inverse} \ (\text{of-rat} \ a) \)
by (rule inverse-unique [symmetric]) (simp add: of-rat-mult [symmetric])

lemma of-rat-inverse: \( \text{of-rat} \text{inverse} \ a :: 'a::{field-char-0} = \text{inverse} \ (\text{of-rat} \ a) \)
by (cases a = 0) (simp-all add: nonzero-of-rat-inverse)

lemma nonzero-of-rat-divide: \( b \neq 0 \implies \text{of-rat} (a / b) = \text{of-rat} \ a / \text{of-rat} \ b \)
by (simp add: divide-inverse of-rat-mult nonzero-of-rat-inverse)

lemma of-rat-divide: \( \text{of-rat} (a / b) :: 'a::{field-char-0} = \text{of-rat} \ a / \text{of-rat} \ b \)
by (cases b = 0) (simp-all add: nonzero-of-rat-divide)

lemma of-rat-power: \( \text{of-rat} (a ^ n) :: 'a::{field-char-0} = \text{of-rat} \ a ^ n \)
by (induct n) (simp-all add: of-rat-mult)

lemma of-rat-eq-iff [simp]: \( \text{of-rat} \ a = \text{of-rat} \ b \iff a = b \)
apply transfer
apply (simp add: nonzero-divide-eq-eq nonzero-eq-divide-eq flip: of-int-mult)
done

lemma of-rat-eq-0-iff [simp]: of-rat a = 0 ↔ a = 0
  using of-rat-eq-iff [of - 0] by simp

lemma zero-eq-of-rat-iff [simp]: 0 = of-rat a ↔ 0 = a
  by simp

lemma of-rat-eq-1-iff [simp]: of-rat a = 1 ↔ a = 1
  using of-rat-eq-iff [of - 1] by simp

lemma one-eq-of-rat-iff [simp]: 1 = of-rat a ↔ 1 = a
  by simp

lemma of-rat-less: (of-rat r :: 'a::linordered-field) < of-rat s ↔ r < s
proof (induct r, induct s)
  fix a b c d :: int
  assume not-zero: b > 0 d > 0
  then have b * d > 0 by simp
  have of-int-divide-less-eq:
    (of-int a :: 'a) / of-int b < of-int c / of-int d ↔
    (of-int a :: 'a) * of-int d < of-int c * of-int b
    using not-zero by (simp add: pos-less-divide-eq pos-divide-less-eq)
  show (of-rat (Fract a b) :: 'a::linordered-field) < of-rat (Fract c d) ↔
    Fract a b < Fract c d
    using not-zero (b * d > 0)
qed

lemma of-rat-less-eq: (of-rat r :: 'a::linordered-field) ≤ of-rat s ↔ r ≤ s
  unfolding le-less by (auto simp add: of-rat-less)

lemma of-rat-le-0-iff [simp]: (of-rat r :: 'a::linordered-field) ≤ 0 ↔ r ≤ 0
  using of-rat-less-eq [of r 0, where 'a = 'a] by simp

lemma zero-le-of-rat-iff [simp]: 0 ≤ (of-rat r :: 'a::linordered-field) ↔ 0 ≤ r
  using of-rat-less-eq [of 0 r, where 'a = 'a] by simp

lemma of-rat-le-1-iff [simp]: (of-rat r :: 'a::linordered-field) ≤ 1 ↔ r ≤ 1
  using of-rat-less-eq [of r 1] by simp

lemma one-le-of-rat-iff [simp]: 1 ≤ (of-rat r :: 'a::linordered-field) ↔ 1 ≤ r
  using of-rat-less-eq [of 1 r] by simp

lemma of-rat-less-0-iff [simp]: (of-rat r :: 'a::linordered-field) < 0 ↔ r < 0
  using of-rat-less [of r 0, where 'a = 'a] by simp

lemma zero-less-of-rat-iff [simp]: 0 < (of-rat r :: 'a::linordered-field) ↔ 0 < r
  using of-rat-less [of 0 r, where 'a = 'a] by simp
lemma of-rat-less-1-iff [simp]: (of-rat r :: 'a::linordered-field) < 1 \iff r < 1
using of-rat-less [of r 1] by simp

lemma one-less-of-rat-iff [simp]: 1 < (of-rat r :: 'a::linordered-field) \iff 1 < r
using of-rat-less [of 1 r] by simp

lemma of-rat-eq-id [simp]: of-rat = id
proof
  show of-rat a = id a for a
  by (induct a) (simp add: of-rat-rat Fract-of-int-eq [symmetric])
qed

lemma abs-of-rat [simp]:
  \lfloor of-rat r \rfloor = (of-rat \lfloor r \rfloor :: 'a :: linordered-field)
  by (cases r \geq 0) (simp-all add: not-le of-rat-minus)

Collapse nested embeddings.

lemma of-rat-of-nat-eq [simp]: of-rat (of-nat n) = of-nat n
by (induct n) (simp-all add: of-rat-add)

lemma of-rat-of-int-eq [simp]: of-rat (of-int z) = of-int z
by (cases z rule: int-diff-cases) (simp add: of-rat-diff)

lemma of-rat-numeral-eq [simp]: of-rat (numeral w) = numeral w
using of-rat-of-int-eq [of numeral w] by simp

lemma of-rat-neg-numeral-eq [simp]: of-rat (\neg numeral w) = \neg numeral w
using of-rat-of-int-eq [of \neg numeral w] by simp

lemma of-rat-floor [simp]:
  \lfloor of-rat r \rfloor = \lfloor r \rfloor
  by (cases r) (simp add: Fract-of-int-quotient of-rat-divide floor-divide-of-int-eq)

lemma of-rat-ceiling [simp]:
  \lceil of-rat r \rceil = \lceil r \rceil
  using of-rat-floor [of \neg r] by (simp add: of-rat-minus ceiling-def)

lemmas zero-rat = Zero-rat-def
lemmas one-rat = One-rat-def

abbreviation rat-of-nat :: nat \Rightarrow rat
where rat-of-nat \equiv of-nat

abbreviation rat-of-int :: int \Rightarrow rat
where rat-of-int \equiv of-int
97.4 The Set of Rational Numbers

context field-char-0

begin

definition Rats :: 'a set (Q)
  where Q = range of-rat

end

lemma Rats-cases [cases set: Rats]:
  assumes q ∈ Q
  obtains (of-rat) r where q = of-rat r
proof –
  from q ∈ Q have q ∈ range of-rat
    by (simp only: Rats-def)
  then obtain r where q = of-rat r ..
  then show thesis ..
qed

lemma Rats-cases':
  assumes (x :: 'a :: field-char-0) ∈ Q
  obtains a b where b > 0 coprime a b x = of-int a / of-int b
proof –
  from assms obtain r where x = of-rat r
    by (auto simp: Rats-def)
  obtain a b where quot: quotient-of r = (a,b) by force
  have b > 0 using quotient-of-denom-pos[OF quot] .
  moreover have coprime a b using quotient-of-coprime[OF quot] .
  moreover have x = of-int a / of-int b unfolding (x = of-rat r)
    quotient-of-div[OF quot] by (simp add: of-rat-divide)
  ultimately show ?thesis using that by blast
qed

lemma Rats-of-rat [simp]: of-rat r ∈ Q
  by (simp add: Rats-def)

lemma Rats-of-int [simp]: of-int z ∈ Q
  by (subst of-rat-of-int-eq [symmetric]) (rule Rats-of-rat)

lemma Ints-subset-Rats: Z ⊆ Q
  using Ints-cases Rats-of-int by blast

lemma Rats-of-nat [simp]: of-nat n ∈ Q
  by (subst of-rat-of-nat-eq [symmetric]) (rule Rats-of-rat)

lemma Nats-subset-Rats: N ⊆ Q
  using Ints-subset-Rats Nats-subset-Ints by blast

lemma Rats-number-of [simp]: numeral w ∈ Q
by (subst of-rat-numeral-eq [symmetric]) (rule Rats-of-rat)

lemma Rats-0 [simp]: \( 0 \in \mathbb{Q} \)
unfolding Rats-def by (rule range-eqI) (rule of-rat-0 [symmetric])

lemma Rats-1 [simp]: \( 1 \in \mathbb{Q} \)
unfolding Rats-def by (rule range-eqI) (rule of-rat-1 [symmetric])

lemma Rats-add [simp]: \( a \in \mathbb{Q} \Rightarrow b \in \mathbb{Q} \Rightarrow a + b \in \mathbb{Q} \)
by (metis Rats-cases Rats-of-rat of-rat-add)

lemma Rats-minus-iff [simp]: \( -a \in \mathbb{Q} \leftrightarrow a \in \mathbb{Q} \)
by (metis Rats-cases Rats-of-rat add.inverse-inverse of-rat-minus)

lemma Rats-diff [simp]: \( a \in \mathbb{Q} \Rightarrow b \in \mathbb{Q} \Rightarrow a - b \in \mathbb{Q} \)
by (metis Rats-add Rats-minus-iff diff-conv-add-uminus)

lemma Rats-mult [simp]: \( a \in \mathbb{Q} \Rightarrow b \in \mathbb{Q} \Rightarrow a \ast b \in \mathbb{Q} \)
by (metis Rats-cases Rats-of-rat of-rat-mult)

lemma Rats-inverse [simp]: \( a \in \mathbb{Q} \Rightarrow \text{inverse } a \in \mathbb{Q} \)
for \( a :: 'a::field-char-0 \)
by (metis Rats-cases Rats-of-rat of-rat-inverse)

lemma Rats-divide [simp]: \( a \in \mathbb{Q} \Rightarrow b \in \mathbb{Q} \Rightarrow a / b \in \mathbb{Q} \)
for \( a b :: 'a::field-char-0 \)
by (simp add: divide-inverse)

lemma Rats-power [simp]: \( a \in \mathbb{Q} \Rightarrow a ^ n \in \mathbb{Q} \)
for \( a :: 'a::field-char-0 \)
by (metis Rats-cases Rats-of-rat of-rat-power)

lemma Rats-sum [intro]: \( (\forall x. x \in A \Rightarrow f x \in \mathbb{Q}) \Rightarrow \text{sum } f A \in \mathbb{Q} \)
by (induction A rule: infinite-finite-induct) auto

lemma Rats-prod [intro]: \( (\forall x. x \in A \Rightarrow f x \in \mathbb{Q}) \Rightarrow \text{prod } f A \in \mathbb{Q} \)
by (induction A rule: infinite-finite-induct) auto

lemma Rats-induct [case-names of-rat, induct set: Rats]: \( q \in \mathbb{Q} \Rightarrow (\forall r. P \ (\text{of-rat } r)) \Rightarrow P \ q \)
by (rule Rats-cases) auto

lemma Rats-infinite: \( \neg \text{finite } \mathbb{Q} \)
by (auto dest!: finite-imageD simp: inj-on-def infinite-UNIV-char-0 Rats-def)

lemma Rats-add-iff: \( a \in \mathbb{Q} \lor b \in \mathbb{Q} \Rightarrow a + b \in \mathbb{Q} \leftrightarrow a \in \mathbb{Q} \land b \in \mathbb{Q} \)
by (metis Rats-add Rats-diff add-diff-cancel add-diff-cancel-left)

lemma Rats-diff-iff: \( a \in \mathbb{Q} \lor b \in \mathbb{Q} \Rightarrow a - b \in \mathbb{Q} \leftrightarrow a \in \mathbb{Q} \land b \in \mathbb{Q} \)
THEORY "Rat"

by (metis Rats-add-iff diff-add-cancel)

lemma Rats-mult-iff: $a \in \mathbb{Q} - \{0\} \lor b \in \mathbb{Q} - \{0\} \implies a \cdot b \in \mathbb{Q} \iff a \in \mathbb{Q} \land b \in \mathbb{Q}$
  by (metis Diff-iff Rats-divide Rats-mult insertI1 mult.commute nonzero-divide-eq-eq)

lemma Rats-inverse-iff [simp]: $\text{inverse } a \in \mathbb{Q} \iff a \in \mathbb{Q}$
  using Rats-inverse by force

lemma Rats-divide-iff: $a \in \mathbb{Q} - \{0\} \lor b \in \mathbb{Q} - \{0\} \implies a \div b \in \mathbb{Q} \iff a \in \mathbb{Q} \land b \in \mathbb{Q}$
  by (metis Rats-divide Rats-mult-iff divide-eq-0-iff divide-inverse nonzero-mult-div-cancel-right)

97.5 Implementation of rational numbers as pairs of integers

Formal constructor

definition Frct :: int × int ⇒ rat
  where [simp]: Frct $p = \text{Fract } (\text{fst } p) (\text{snd } p)$

lemma [code abstype]: Frct (quotient-of $q$) = $q$
  by (cases $q$) (auto intro: quotient-of-eq)

Numerals

declare quotient-of-Fract [code abstract]

definition of-int :: int ⇒ rat
  where [code-abbrev]: of-int = Int.of-int

hide-const (open) of-int

lemma quotient-of-int [code abstract]: quotient-of (Rat.of-int $a$) = $(a, 1)$
  by (simp add: of-int-def of-int-rat quotient-of-Fract)

lemma [code-unfold]: numeral $k = \text{Rat.of-int } (\text{numeral } k)$
  by (simp add: Rat.of-int-def)

lemma [code-unfold]: $- \text{numeral } k = \text{Rat.of-int } (- \text{numeral } k)$
  by (simp add: Rat.of-int-def)

lemma Frct-code-post [code-post]:
  Frct $(0, a) = 0$
  Frct $(a, 0) = 0$
  Frct $(1, 1) = 1$
  Frct $(\text{numeral } k, 1) = \text{numeral } k$
  Frct $(1, \text{numeral } k) = 1 / \text{numeral } k$
  Frct $(\text{numeral } k, \text{numeral } l) = \text{numeral } k / \text{numeral } l$
  Frct $(− a, b) = − \text{Frct } (a, b)$
  Frct $(a, − b) = − \text{Frct } (a, b)$
  $− (\text{Frct } q) = \text{Frct } q$
  by (simp-all add: Fract-of-int-quotient)
Operations

**lemma rat-zero-code** [code abstract]: quotient-of 0 = (0, 1)
by (simp add: Zero-rat-def quotient-of-Fract normalize-def)

**lemma rat-one-code** [code abstract]: quotient-of 1 = (1, 1)
by (simp add: One-rat-def quotient-of-Fract normalize-def)

**lemma rat-plus-code** [code abstract]:
quotient-of (p + q) = (let (a, c) = quotient-of p; (b, d) = quotient-of q
in normalize (a * d + b * c, c * d))
by (cases p, cases q) (simp add: quotient-of-Fract)

**lemma rat-uminus-code** [code abstract]:
quotient-of (- p) = (let (a, b) = quotient-of p in (- a, b))
by (cases p) (simp add: quotient-of-Fract)

**lemma rat-minus-code** [code abstract]:
quotient-of (p - q) = (let (a, c) = quotient-of p; (b, d) = quotient-of q
in normalize (a * d - b * c, c * d))
by (cases p, cases q) (simp add: quotient-of-Fract)

**lemma rat-times-code** [code abstract]:
quotient-of (p * q) = (let (a, c) = quotient-of p; (b, d) = quotient-of q
in normalize (a * b, c * d))
by (cases p, cases q) (simp add: quotient-of-Fract)

**lemma rat-inverse-code** [code abstract]:
quotient-of (inverse p) = (let (a, b) = quotient-of p
in if a = 0 then (0, 1) else (sgn a * b, |a|))
proof (cases p)
case (Fract a b)
then show ?thesis
by (cases 0::int a rule: linorder-cases) (simp-all add: quotient-of-Fract ac-simps)
qed

**lemma rat-divide-code** [code abstract]:
quotient-of (p / q) = (let (a, c) = quotient-of p; (b, d) = quotient-of q
in normalize (a * d, c * b))
by (cases p, cases q) (simp add: quotient-of-Fract)

**lemma rat-abs-code** [code abstract]:
quotient-of |p| = (let (a, b) = quotient-of p in (|a|, b))
by (cases p) (simp add: quotient-of-Fract)

**lemma rat-sgn-code** [code abstract]: quotient-of (sgn p) = (sgn (fst (quotient-of

proof (cases p)
  case (Fract a b)
  then show "thesis"
    by (cases 0::int a rule: linorder-cases) (simp-all add: quotient-of-Fract)
qed

lemma rat-floor-code [code]: [p] = (let (a, b) = quotient-of p in a div b)
  by (cases p) (simp add: quotient-of-Fract floor-Fract)

instantiation rat :: equal
begin

definition [code]: HOL.equal a b ←→ quotient-of a = quotient-of b

instance
  by standard (simp add: equal-rat-def quotient-of-inject-eq)

lemma rat-eq-refl [code nbe]: HOL.equal (r::rat) r ←→ True
  by (rule equal-refl)

end

lemma rat-less-eq-code [code]:
  p ≤ q ←→ (let (a, c) = quotient-of p; (b, d) = quotient-of q in a * d ≤ c * b)
  by (cases p, cases q) (simp add: quotient-of-Fract mult.commute)

lemma rat-less-code [code]:
  p < q ←→ (let (a, c) = quotient-of p; (b, d) = quotient-of q in a * d < c * b)
  by (cases p, cases q) (simp add: quotient-of-Fract mult.commute)

lemma [code]: of-rat p = (let (a, b) = quotient-of p in of-int a / of-int b)
  by (cases p) (simp add: quotient-of-Fract of-rat-rat)

Quickcheck

context
  includes term-syntax
begin

definition valuerm-fract :: int × (unit ⇒ Code-Evaluation.term) ⇒
  int × (unit ⇒ Code-Evaluation.term) ⇒
  rat × (unit ⇒ Code-Evaluation.term)
  where [code-unfold]: valuerm-fract k l = Code-Evaluation.valuermify Fract {·} k

end

instantiation rat :: random
begin

context

includes state-combinator-syntax

begin

definition

Quickcheck-Random.random i =
Quickcheck-Random.random i ◦→ (λnum. Random.range i ◦→ (λdenom. Pair
(let j = int-of-integer (integer-of-natural (denom + 1))
in valterm-fract num (j, λu. Code-Evaluation.term-of j))))

instance ..

end

end

instantiation rat :: exhaustive

begin

definition

exhaustive-rat f d =
Quickcheck-Exhaustive.exhaustive
(λl. Quickcheck-Exhaustive.exhaustive
(λk. f (Fract k (int-of-integer (integer-of-natural l) + 1))) d) d

instance ..

end

instantiation rat :: full-exhaustive

begin

definition

full-exhaustive-rat f d =
Quickcheck-Exhaustive.full-exhaustive
(λ(l, -). Quickcheck-Exhaustive.full-exhaustive
(λk. f
(let j = int-of-integer (integer-of-natural l) + 1
in valterm-fract k (j, λu. Code-Evaluation.term-of j))) d) d

instance ..

end

instance rat :: partial-term-of ..

lemma [code]:
partial-term-of (ty :: rat itself) (Quickcheck-Narrowing.Narrowing-variable p tt) ≡
Code-Evaluation.Free (STR "..") (Typerep.Typerep (STR "Rat" rat) [])
partial-term-of (ty :: rat itself) (Quickcheck-Narrowing.Narrowing-constructor 0 [l, k]) ≡
Code-Evaluation.App
(Code-Evaluation.Const (STR "Rat Frct")
(Typerep.Typerep (STR "fun")
[Typerep.Typerep (STR "Product-Type.prod")
Typerep.Typerep (STR "Int.int") [], Typerep.Typerep (STR "Int.int") []],
Typerep.Typerep (STR "Rat.rat") []))
(Code-Evaluation.App
(Code-Evaluation.Const (STR "Product-Type.Pair")
(Typerep.Typerep (STR "fun")
[Typerep.Typerep (STR "Int.int") [],
Typerep.Typerep (STR "fun")
[Typerep.Typerep (STR "Int.int") [],
Typerep.Typerep (STR "Product-Type.prod")
Typerep.Typerep (STR "Int.int") [], Typerep.Typerep (STR "Int.int") []]))
(partial-term-of (TYPE(int)) l) (partial-term-of (TYPE(int)) k)
by (rule partial-term-of-anything)+

instantiation rat :: narrowing
begin

definition narrowing =
Quickcheck-Narrowing.apply
(Quickcheck-Narrowing.apply
(Quickcheck-Narrowing.cons (λnom denom Fract nom denom) narrowing)
narrowing)

instance ..

end

97.6 Setup for Nitpick

declaration
Nitpick-HOL.register-frac-type type-name (rat)
[(const-name Abs-Rat, const-name Nitpick.Abs-Frac),
(const-name zero-rat-inst.zero-rat, const-name Nitpick.zero-frac),
(const-name one-rat-inst.one-rat, const-name Nitpick.one-frac),
(const-name plus-rat-inst.plus-rat, const-name Nitpick.plus-frac),
(const-name times-rat-inst.times-rat, const-name Nitpick.times-frac),
(const-name uminus-rat-inst.uminus-rat, const-name Nitpick.uminus-frac),
(const-name inverse-rat-inst.inverse-rat, const-name Nitpick.inverse-frac),
(const-name Rat rat, const-name Nitpick.Rat),
(const-name Fract nom denom, const-name Nitpick.Frac nom denom),
(const-name Int int, const-name Nitpick.Int int)]
(const-name \(\text{ord-rat-inst}\) less-rat),
(const-name \(\text{ord-rat-inst}\) less-eq-rat),
(const-name \(\text{Nitpick.less-eq-frac}\)),
(const-name \(\text{field-char-0-class.of-rat}\), const-name \(\text{Nitpick.of-frac}\))]

lemmas [nitpick-unfold] =
inverse-rat-inst.inverse-rat
one-rat-inst.one-rat ord-rat-inst.less-rat
ord-rat-inst.less-eq-rat plus-rat-inst.plus-rat times-rat-inst.times-rat
uminus-rat-inst.uminus-rat zero-rat-inst.zero-rat

97.7 Float syntax

syntax -Float :: float-const \(\Rightarrow\) 'a (-)

parse-translation \(\langle\)
let
  fun mk-frac str =
    let
      val \{mant = i, exp = n\} = Lexicon.read-float str;
      val exp = Syntax.const const-syntax \(\langle\text{Power.power}\rangle\);
      val ten = Numeral.mk-number-syntax 10;
      val exp10 = if n = 1 then ten else exp \$ ten \$ Numeral.mk-number-syntax n;
    in Syntax.const const-syntax \(\langle\text{Fields.inverse-divide}\rangle\) \$ Numeral.mk-number-syntax i \$ exp10 end;

      \(\)\n      fun float-tr [\(c\ as\ Const\ \langle\text{syntax-const \(-\text{constrain}\), -}\rangle\) \$ t \$ u] = c \$ float-tr [t] \$ u
      | float-tr [t as Const \(\langle\text{str}, -\rangle\)] = mk-frac str
      | float-tr ts = raise TERM (float-tr, ts);
    in \[\langle\text{syntax-const \(-\text{Float}, K\ float-tr\}\rangle\end

Test:

lemma 123.456 = \(-111.111 + 200 + 30 + 4 + 5/10 + 6/100 + (7/1000::\text{rat})\)
  \(\)by simp

97.8 Hiding implementation details

hide-const \(\langle\text{open}\rangle\) normalize positive

lifting-update rat.lifting
lifting-forget rat.lifting

end
98 Development of the Reals using Cauchy Sequences

theory Real
imports Rat
begin

This theory contains a formalization of the real numbers as equivalence classes of Cauchy sequences of rationals. See the AFP entry Dedekind-Real for an alternative construction using Dedekind cuts.

98.1 Preliminary lemmas

Useful in convergence arguments

lemma inverse-of-nat-le:
  fixes n::nat shows [n ≤ m; n≠0] ⇒ 1 / of-nat m ≤ (1::'a::linordered-field) / of-nat n
  by (simp add: frac-le)

lemma add-diff-add: (a + c) − (b + d) = (a − b) + (c − d)
  for a b c d :: 'a::ab-group-add
  by simp

lemma minus-diff-minus: − a − − b = − (a − b)
  for a b :: 'a::ab-group-add
  by simp

lemma mult-diff-mult: (x * y − a * b) = x * (y − b) + (x − a) * b
  for x y a b :: 'a::ring
  by (simp add: algebra-simps)

lemma inverse-diff-inverse:
  fixes a b :: 'a::division-ring
  assumes a ≠ 0 and b ≠ 0
  shows inverse a − inverse b = − (inverse a * (a − b) * inverse b)
  using assms by (simp add: algebra-simps)

lemma obtain-pos-sum:
  fixes r :: rat assumes r: 0 < r
  obtains s t where 0 < s and 0 < t and r = s + t
  proof
  from r show 0 < r/2 by simp
  from r show 0 < r/2 by simp
  show r = r/2 + r/2 by simp
  qed
98.2 Sequences that converge to zero

definition vanishes :: (nat ⇒ rat) ⇒ bool
  where vanishes X ←→ (∀ r>0. ∃ k. ∀ n≥k. |X n| < r)

lemma vanishesI: (∀ r. 0 < r ⇒ ∃ k. ∀ n≥k. |X n| < r) ⇒ vanishes X
  unfolding vanishes-def by simp

lemma vanishesD: vanishes X ⇒ 0 < r ⇒ ∃ k. ∀ n≥k. |X n| < r
  unfolding vanishes-def by simp

lemma vanishes-const [simp]: vanishes (λn. c) ⇔ c = 0
  proof (cases c = 0)
    case True
    then show ?thesis
      by (simp add: vanishesI)
  next
    case False
    then show ?thesis
      unfolding vanishes-def
      using zero-less-abs-iff by blast
  qed

lemma vanishes-minus: vanishes X ⇒ vanishes (λn. − X n)
  unfolding vanishes-def by simp

lemma vanishes-add:
  assumes X: vanishes X
  and Y: vanishes Y
  shows vanishes (λn. X n + Y n)
  proof (rule vanishesI)
    fix r :: rat
    assume 0 < r
    then obtain s t where s: 0 < s and t: 0 < t and r: r = s + t
      by (rule obtain-pos-sum)
    obtain i where i: ∀ n≥i. |X n| < s
      using vanishesD [OF X s] ..
    obtain j where j: ∀ n≥j. |Y n| < t
      using vanishesD [OF Y t] ..
    have ∀ n≥max i j. |X n + Y n| < r
      proof clarsimp
        fix n
        assume n: i ≤ n j ≤ n
        have |X n + Y n| ≤ |X n| + |Y n|
          by (rule abs-triangle-ineq)
        also have ... < s + t
          by (simp add: add-strict-mono i j n)
        finally show |X n + Y n| < r
          by (simp only: r)
      qed
  qed
then show $\exists k. \forall n \geq k. |X n + Y n| < r$. 

qed

lemma `vanishes-diff`:
assumes `vanishes X` `vanishes Y`
shows `vanishes (λn. X n - Y n)`
unfolding `diff-conv-add-uminus` by (intro `vanishes-add` `vanishes-minus` assms)

lemma `vanishes-mult-bounded`:
assumes `X` [simp]: $\exists a > 0. \forall n. |X n| < a$
assumes `Y` [simp]: [OF `vanishes` (λn. Y n)]
shows `vanishes (λn. X n * Y n)`
proof (rule `vanishesI`)
fix `r :: rat`
assume `r`: $0 < r$
obtain `a` [simp]: $a > 0. \forall n. |X n| < a$
  using `X` [simp]: blast
obtain `b` [simp]: $b: 0 < b. r = a * b$
proof
  show $0 < r / a$ using `r a` [simp]
  show $r = a * (r / a)$ using `a` [simp]
qed

obtain `k` where [simp]: $k: \forall n \geq k. |Y n| < b$
  using `vanishesD` [OF `vanishes` `Y b` [simp] `k` ..]
  have $\forall n \geq k. |X n * Y n| < r$
    by (simp add: `b` [simp] abs-mult mult-strict-mono' `a `k`)
then show $\exists k. \forall n \geq k. |X n * Y n| < r$. 

qed

98.3 Cauchy sequences

definition `cauchy` [simp]: $\forall r > 0. \exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r$

lemma `cauchyI` [simp]: $\forall r > 0. \exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r \Longrightarrow cauchy X$
  unfolding `cauchy-def` by simp

lemma `cauchyD` [simp]: $cauchy X \Longrightarrow 0 < r \Longrightarrow \exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r$
  unfolding `cauchy-def` by simp

lemma `cauchy-const` [simp]: `cauchy` (λn. `x`)
  unfolding `cauchy-def` by simp

lemma `cauchy-add` [simp]:
  assumes `X`: `cauchy X` and `Y`: `cauchy Y`
  shows `cauchy` (λn. X n + Y n)
  proof (rule `cauchyI`)
    fix `r :: rat`
assume $0 < r$
then obtain $s$ $t$ where $s$: $0 < s$ and $t$: $0 < t$ and $r$: $r = s + t$
  by (rule obtain-pos-sum)
obtain $i$ where $i$: $\forall m \geq i$. $\forall n \geq i$. $|X m - X n| < s$
  using cauchyD [OF $X s$] ..
obtain $j$ where $j$: $\forall m \geq j$. $\forall n \geq j$. $|Y m - Y n| < t$
  using cauchyD [OF $Y t$] ..
have $\forall m \geq max i j. \forall n \geq max i j. |(X m + Y m) - (X n + Y n)| < r$
proof clarsimp
  fix $m$ $n$
  assume $*: i \leq m$ $j \leq m$ $i \leq n$ $j \leq n$
  have $|(X m + Y m) - (X n + Y n)| \leq |X m - X n| + |Y m - Y n|$
    unfolding add-diff-add by (rule abs-triangle-ineq)
  also have $\ldots < s + t$
    by (rule add-strict-mono) (simp-all add: $i j *$)
  finally show $|(X m + Y m) - (X n + Y n)| < r$ by (simp only: $r$)
qed
then show $\exists k. \forall m \geq k. \forall n \geq k. |(X m + Y m) - (X n + Y n)| < r$ ..
qed

lemma cauchy-minus [simp]:
  assumes $X$: cauchy $X$
  shows cauchy $(\lambda n. - X n)$
  using assms unfolding cauchy-def
  unfolding minus-diff-minus abs-minus-cancel .

lemma cauchy-diff [simp]:
  assumes cauchy $X$ cauchy $Y$
  shows cauchy $(\lambda n. X n - Y n)$
  using assms unfolding diff-cone-add-uminus by (simp del: add-uminus-conv-diff)

lemma cauchy-imp-bounded:
  assumes cauchy $X$
  shows $\exists b > 0. \forall n. |X n| < b$
proof
  obtain $k$ where $k$: $\forall m \geq k. \forall n \geq k. |X m - X n| < 1$
    using cauchyD [OF assms zero-less-one] ..
  show $\exists b > 0. \forall n. |X n| < b$
    by (rule linorder-le-cases)
  assume $n \leq k$
  then have $|X n| \leq Max (abs ' X ' [..k]) + 1$ by simp
next
  fix $n :: nat$
  show $|X n| \leq Max (abs ' X ' [..k]) + 1$
    by (rule linorder-le-cases)
  assume $n \leq k$
  then have $|X n| \leq Max (abs ' X ' [..k])$ by simp
then show \(|X n| < \text{Max} \left(\text{abs} ' X ' \{..k\}\right) + 1\) by \(\text{simp}\) 
next 
assume \(k \leq n\) 
have \(|X n| = |X k + (X n - X k)|\) by \(\text{simp}\) 
also have \(|X k + (X n - X k)| \leq |X k| + |X n - X k|\) 
by (rule \text{abs-triangle-ineq}) 
also have \(\ldots < \text{Max} \left(\text{abs} ' X ' \{..k\}\right) + 1\) 
by (rule \text{add-le-less-mono} (\text{simp-all add: } k \leq n)) 
finally show \(|X n| < \text{Max} \left(\text{abs} ' X ' \{..k\}\right) + 1\) .
qed 
qed 
qed 

\text{lemma cauchy-mult} [\text{simp}]:
\text{assumes } X: \text{cauchy } X \text{ and } Y: \text{cauchy } Y 
\text{shows } \text{cauchy} \left(\lambda n. \ X n * Y n\right) 
\text{proof (rule cauchy1)} 
fix \(r::\text{rat}\) assume \(0 < r\) 
then obtain \(u v\) where \(u: 0 < u\) and \(v: 0 < v\) and \(r = u + v\) 
by (rule \text{obtain-pos-sum}) 
obtain \(a\) where \(a: 0 < a \ \forall n. |X n| < a\) 
using \text{cauchy-imp-bounded} [OF \(X\)] by \text{blast} 
obtain \(b\) where \(b: 0 < b \ \forall n. |Y n| < b\) 
using \text{cauchy-imp-bounded} [OF \(Y\)] by \text{blast} 
obtain \(s t\) where \(s: 0 < s\) and \(t: 0 < t\) and \(r = a * t + s * b\) 
proof 
show \(0 < v/b\) using \(v b(1)\) \(\text{by simp}\) 
show \(0 < u/a\) using \(u a(1)\) \(\text{by simp}\) 
show \(r = a * (u/a) + (v/b) * b\) 
using \(a(1) b(1)\) \(r = a + v\) \(\text{by simp}\) 
qed 
obtain \(i\) where \(i: \forall m \geq i. \forall n \geq i. |X m - X n| < s\) 
using \text{cauchyD} [OF \(X s\)] .. 
obtain \(j\) where \(j: \forall m \geq j. \forall n \geq j. |Y m - Y n| < t\) 
using \text{cauchyD} [OF \(Y t\)] .. 
have \(\forall m \geq max i j. \forall n \geq max i j. |X m * Y m - X n * Y n| < r\) 
proof \text{clarsimp} 
fix \(m n\) 
assume \(*: i \leq m j \leq m i \leq n j \leq n\) 
have \(|X m * Y m - X n * Y n| = |X m * (Y m - Y n) + (X m - X n) * Y n|\) 
unfolding \text{mult-diff-mult} ..
also have \(\ldots \leq |X m * (Y m - Y n)| + |(X m - X n) * Y n|\) 
by (rule \text{abs-triangle-ineq}) 
also have \(\ldots = |X m| * |Y m - Y n| + |X m - X n| * |Y n|\) 
unfolding \text{abs-mult} ..
also have \(\ldots < a * t + s * b\) 
by (\text{simp-all add: add-strict-mono mult-strict-mono'} a b i j *) 
finally show \(|X m * Y m - X n * Y n| < r\)
THEORY "Real"

proof (simp only: r)
qed

then show \( \exists k. \forall m \geq k. \forall n \geq k. |X m * Y m - X n * Y n| < r \) ..
qed

lemma cauchy-not-vanishes-cases:
assumes X: cauchy X
assumes nz: \( \neg \) vanishes X
shows \( \exists b > 0. \exists k. (\forall n \geq k. b < - X n) \lor (\forall n \geq k. b < X n) \)
proof
  obtain r where \( 0 < r \) and \( r: \forall k. \exists n \geq k. r \leq |X n| \)
    using nz unfolding vanishes-def by (auto simp add: not-less)
  obtain s t where s: \( 0 < s \) and \( t: 0 < t \) and \( r = s + t \)
    using \( \langle 0 < r \rangle \) by (rule obtain-pos-sum)
  obtain i where i: \( \forall m \geq i. \forall n \geq i. |X m - X n| < s \)
    using cauchyD [OF X s] ..
  obtain k where i \( \leq k \) and \( r \leq |X k| \)
    using r by blast
  have k: \( \forall n \geq k. |X n - X k| < s \)
    using i \( \langle i \leq k \rangle \) by auto
  have X k \( \leq - r \lor r \leq X k \)
    using \( \langle r \leq |X k| \rangle \) by auto
  then have \( \langle \forall n \geq k. t < - X n \rangle \lor (\forall n \geq k. t < X n) \)
    unfolding \( \langle r = s + t \rangle \) using k by auto
  then have \( \exists k. (\forall n \geq k. t < - X n) \lor (\forall n \geq k. t < X n) \) ..
  then show \( \exists t > 0. \exists k. (\forall n \geq k. t < - X n) \lor (\forall n \geq k. t < X n) \)
    using t by auto
qed

lemma cauchy-not-vanishes:
assumes X: cauchy X
and nz: \( \neg \) vanishes X
shows \( \exists b > 0. \exists k. \forall n \geq k. b < |X n| \)
using cauchy-not-vanishes-cases [OF assms]
by (elim ex-forward conj-forward asm-rl) auto

lemma cauchy-inverse [simp]:
assumes X: cauchy X
and nz: \( \neg \) vanishes X
shows cauchy (\( \lambda n. \) inverse (\( X n \)))
proof (rule cauchyI)
fix r :: rat
assume \( 0 < r \)
obtain b i where b: \( 0 < b \) and \( i: \forall n \geq i. b < |X n| \)
  using cauchy-not-vanishes [OF X nz] by blast
from b i have nz: \( \forall n \geq i. X n \neq 0 \) by auto
obtain s where s: \( 0 < s \) and \( r: r = inverse b * s * inverse b \)
proof
  show \( 0 < b * r * b \) by (simp add: \( \langle 0 < r \rangle \) b)
show \( r = \text{inverse} \ b \ast (b \ast r \ast b) \ast \text{inverse} \ b \)
using \( b \) by simp

qed

obtain \( j \) where \( j: \forall m \geq j. \forall n \geq j. |X m - X n| < s \)
using cauchyD \([OF X \ s]\) ..
have \( \forall m \geq \max i \ j. \forall n \geq \max i \ j. |\text{inverse} \ (X \ m) - \text{inverse} \ (X \ n)| < r \)
proof clarsimp
fix \( m \ n \)
assume \( r : i \leq m \ j \leq m \ i \leq n \ j \leq n \)
have \( |\text{inverse} \ (X \ m) - \text{inverse} \ (X \ n)| = \text{inverse} \ |X \ m| \ast |X \ m - X \ n| \ast \text{inverse} \ |X \ n| \)
by (simp add: inverse-diff-inverse nz \ast abs-mult)
also have \( \ldots < \text{inverse} \ b \ast s \ast \text{inverse} \ b \)
by (simp add: mult-strict-mono less-imp-inverse-less i j b \ast s)
finally show \( |\text{inverse} \ (X \ m) - \text{inverse} \ (X \ n)| < r \)
by (simp only: r ..)

qed

lemma vanishes-diff-inverse:
assumes \( X: \text{cauchy} \ X \sim \text{vanishes} \ X \)
and \( Y: \text{cauchy} \ Y \sim \text{vanishes} \ Y \)
and \( XY: \text{vanishes} \ (\lambda n. X \ n - Y \ n) \)
shows \( \text{vanishes} \ (\lambda n. \text{inverse} \ (X \ n) - \text{inverse} \ (Y \ n)) \)
proof (rule vanishesI)
fix \( r :: \text{rat} \)
assume \( r : 0 < r \)
obtain \( a \ i \ ) where \( a : 0 < a \) and \( i : \forall n \geq i. \ a < |X \ n| \)
using cauchy-not-vanishes \([OF X]\) by blast
obtain \( b \ j \ ) where \( b : 0 < b \) and \( j : \forall n \geq j. \ b < |Y \ n| \)
using cauchy-not-vanishes \([OF Y]\) by blast
obtain \( s \ ) where \( s : 0 < s \) and \( \text{inverse} \ a \ast s \ast \text{inverse} \ b = r \)
proof
show \( 0 < a \ast r \ast b \)
using \( a \ b \) by simp
show \( \text{inverse} \ a \ast (a \ast r \ast b) \ast \text{inverse} \ b = r \)
using \( a \ b \) by simp
qed

obtain \( k \ ) where \( k : \forall n \geq k. |X \ n - Y \ n| < s \)
using vanishesD \([OF XY \ s]\) ..
have \( \forall n \geq \max (\max i \ j) \ k. |\text{inverse} \ (X \ n) - \text{inverse} \ (Y \ n)| < r \)
proof clarsimp
fix \( n \)
assume \( n: i \leq n \ j \leq n \ k \leq n \)
with \( i \ j \ a \ b \) have \( X \ n \neq 0 \) and \( Y \ n \neq 0 \)
by auto
then have \( |\text{inverse} \ (X \ n) - \text{inverse} \ (Y \ n)| = \text{inverse} \ |X \ n| \ast |X \ n - Y \ n| \ast \text{inverse} \ |Y \ n| \)
by (simp add: inverse-diff-inverse abs-mult)
also have ... < inverse a * s * inverse b
by (intro mult-strict-mono' less-imp-inverse-less) (simp-all add: a b i j k n)
also note (inverse a * s * inverse b = r)
finally show \[| (x n) - (y n)| < r .\]
qed
then show \[\exists k. \forall n \geq k. | (x n) - (y n)| < r .\]
qed

98.4 Equivalence relation on Cauchy sequences

definition realrel :: (nat ⇒ rat) ⇒ (nat ⇒ rat) ⇒ bool
where realrel = (λX Y. cauchy X ∧ cauchy Y ∧ vanishes (λn. X n - Y n))

lemma realrelI [intro?]: cauchy X ⇒ cauchy Y ⇒ vanishes (λn. X n - Y n) ⇒ realrel X Y
  by (simp add: realrel-def)

lemma realrel-refl: cauchy X ⇒ realrel X X
  by (simp add: realrel-def)

lemma simp-realrel: simp realrel
  by (simp add: abs-minus-commute realrel-def symp-def vanishes-def)

lemma transp-realrel: transp realrel
  unfolding realrel-def
  by (rule transpI) (force simp add: dest: vanishes-add)

lemma part-equivp-realrel: part-equivp realrel
  by (blast intro: part-equivpI symp-realrel transp-realrel realrel-refl cauchy-const)

98.5 The field of real numbers

quotient-type real = nat ⇒ rat / partial: realrel
  morphisms rep-real Real
  by (rule part-equivp-realrel)

lemma cr-real-eq: pcr-real = (λx y. cauchy x ∧ Real x = y)
  unfolding real,pcr-cr-eq cr-real-def realrel-def by auto

lemma Real-induct [induct type: real]:
  assumes \(\forall X. cauchy X ⇒ P (Real X)\)
  shows P x
proof (induct x)
case (1 X)
then have cauchy X by (simp add: realrel-def)
then show P (Real X) by (rule assms)
qed

lemma eq-Real: cauchy X ⇒ cauchy Y ⇒ Real X = Real Y ⇒ vanishes (λn. X n - Y n)
using real.rel-eq-transfer
unfolding real.pcr-cr-eq cr-real-def rel-fun-def realrel-def by simp

lemma Domainp-pcr-real [transfer-domain-rule]: Domainp pcr-real = cauchy
  by (simp add: real.domain-eq realrel-def)

instantiate real :: field
begin

lift-definition zero-real :: real is λn. 0
  by (simp add: realrel-refl)

lift-definition one-real :: real is λn. 1
  by (simp add: realrel-refl)

lift-definition plus-real :: real ⇒ real ⇒ real is λX Y n. X n + Y n
  unfolding realrel-def add-diff-add
  by (simp only: cauchy-add vanishes-add simp-thms)

lift-definition uminus-real :: real ⇒ real is λX n. −X n
  unfolding realrel-def minus-diff-minus
  by (simp only: cauchy-minus vanishes-minus simp-thms)

lift-definition times-real :: real ⇒ real ⇒ real is λX Y n. X n ∗ Y n
  proof –
    fix f1 f2 f3 f4
    have [cauchy f1; cauchy f4; vanishes (λn. f1 n - f2 n); vanishes (λn. f3 n - f4 n)]
      ⇒ vanishes (λn. f1 n ∗ (f3 n - f4 n) + f4 n ∗ (f1 n - f2 n))
      by (simp add: vanishes-add vanishes-mult-bounded cauchy-imp-bounded)
    then show [realrel f1 f2; realrel f3 f4] ⇒ realrel (λn. f1 n ∗ f3 n) (λn. f2 n ∗ f4 n)
      by (simp add: mult.commute realrel-def mult-diff-mult)
  qed

lift-definition inverse-real :: real ⇒ real is λX. if vanishes X then (λn. 0) else (λn. inverse (X n))
  proof –
    fix X Y
    assume realrel X Y
    then have X: cauchy X and Y: cauchy Y and XY: vanishes (λn. X n − Y n)
      by (simp-all add: realrel-def)
    have vanishes X ←→ vanishes Y
      proof
        assume vanishes X
        from vanishes-diff [OF this XY] show vanishes Y
          by simp
      next
        assume vanishes Y
  qed
from \texttt{vanishes-add} \([OF\ this\ XY]\) \textbf{show} \texttt{vanishes X}
by \texttt{simp}
\textbf{qed}
then \textbf{show} \texttt{thesis X Y}
by (\texttt{simp add: \texttt{vanishes-diff-inverse} X Y XY realrel-def})
\textbf{qed}

\textbf{definition} \(x - y = x + - y\) \textbf{for} \(x y :: \texttt{real}\)

\textbf{definition} \(x \div y = x \ast \texttt{inverse} y\) \textbf{for} \(x y :: \texttt{real}\)

\textbf{lemma} \texttt{add-Real}: \texttt{cauchy X \implies \texttt{cauchy Y \implies Real X + Real Y = Real (\lambda n. X n + Y n)}}
\textbf{using} \texttt{plus-real.transfer} \textbf{by} (\texttt{simp add: \texttt{cr-real-eq} rel-fun-def)

\textbf{lemma} \texttt{minus-Real}: \texttt{cauchy X \implies \texttt{- Real X = Real (\lambda n. - X n)}}
\textbf{using} \texttt{uminus-real.transfer} \textbf{by} (\texttt{simp add: \texttt{cr-real-eq} rel-fun-def)

\textbf{lemma} \texttt{diff-Real}: \texttt{cauchy X \implies \texttt{cauchy Y \implies Real X - Real Y = Real (\lambda n. X n - Y n)}}
\textbf{by} (\texttt{simp add: \texttt{minus-Real} add-Real minus-real-def)

\textbf{lemma} \texttt{mult-Real}: \texttt{cauchy X \implies \texttt{cauchy Y \implies Real X \ast Real Y = Real (\lambda n. X n \ast Y n)}}
\textbf{using} \texttt{times-real.transfer} \textbf{by} (\texttt{simp add: \texttt{cr-real-eq} rel-fun-def)

\textbf{lemma} \texttt{inverse-Real}:
\texttt{cauchy X \implies \texttt{inverse (Real X) = (if \texttt{vanishes X then 0 else Real (\lambda n. \texttt{inverse} (X n))}})
\textbf{using} \texttt{inverse-real.transfer} \texttt{zero-real.transfer} \texttt{unfolding} \texttt{cr-real-eq rel-fun-def by} (\texttt{simp split: if-split-asn, \texttt{metis})

\textbf{instance}
\textbf{proof}
fix \(a\) \(b\) \(c :: \texttt{real}\)
show \(a + b = b + a\)
by \texttt{transfer} (\texttt{simp add: \texttt{ac-simps} realrel-def)
show \((a + b) + c = a + (b + c)\)
by \texttt{transfer} (\texttt{simp add: \texttt{ac-simps} realrel-def)
show \(0 + a = a\)
by \texttt{transfer} (\texttt{simp add: realrel-def)
show \(- a + a = 0\)
by \texttt{transfer} (\texttt{simp add: realrel-def)
show \(a - b = a + - b\)
by (\texttt{rule minus-real-def)
show \((a \ast b) \ast c = a \ast (b \ast c)\)
by \texttt{transfer} (\texttt{simp add: \texttt{ac-simps} realrel-def)
show \(a \ast b = b \ast a\)
by \texttt{transfer} (\texttt{simp add: \texttt{ac-simps} realrel-def)
show \( 1 \ast a = a \)
   by transfer (simp add: ac-simps realrel-def)
show \((a + b) \ast c = a \ast c + b \ast c\)
   by transfer (simp add: distrib-right realrel-def)
show \((0::real) \neq (1::real)\)
   by transfer (simp add: realrel-def)

have vanishes \((\lambda n. \text{inverse} (X n) \ast X n - 1)\) if \(X\): cauchy \(X\) \(\neg\) vanishes \(X\) for \(X\)
proof (rule vanishesI)
  fix \(r::\text{rat}\)
  assume \(0 < r\)
  obtain \(b k\) where \(b > 0\ \forall n \geq k. \ b < |X n|\)
  using \(X\) cauchy-not-vanishes by blast
  then show \(\exists k. \forall n \geq k. |\text{inverse} (X n) \ast X n - 1| < r\)
  using \(\langle 0 < r \rangle\) by force
  qed
then show \(a \neq 0 = \Rightarrow \text{inverse} a \ast a = 1\)
by transfer (simp add: realrel-def)
show \(a \div b = a \ast \text{inverse} b\)
by (rule divide-real-def)
show \(\text{inverse} (0::real) = 0\)
by transfer (simp add: realrel-def)
qed

end

98.6 Positive reals

lift-definition positive :: real \Rightarrow bool
is \(\lambda X. \exists r \geq 0. \exists k. \forall n \geq k. r < X n\)
proof –
  have \(1: \exists r \geq 0. \exists k. \forall n \geq k. r < Y n\)
  if \(*::\text{realrel} X Y\) and \(\ast\::\exists r \geq 0. \exists k. \forall n \geq k. r < X n\) for \(X\ Y\)
  proof –
  from \(*\ have \ XY\): vanishes \((\lambda n. X n - Y n)\)
  by (simp-all add: realrel-def)
  from \(\ast\ obtain \ r \ i\ where \ 0 < r\ and\ i\ : \forall n \geq i. r < X n\)
  by blast
  obtain \(s t\) where \(s: 0 < s\ and\ t: 0 < t\ and\ r: r = s + t\)
  using \(\langle 0 < r \rangle\) by (rule obtain-pos-sum)
  obtain \(j\) where \(j: \forall n \geq j. |X n - Y n| < s\)
  using vanishesD \([OF XY s]\) 
  have \(\forall n \geq \max i j. t < Y n\)
  proof clarsimp
  fix \(n\)
  assume \(n: i \leq n j \leq n\)
  have \(|X n - Y n| < s\ and\ r < X n\)
  using \(i j n\) by simp-all
  then show \(t < Y n\) by (simp add: r)
qed

then show \textit{thesis} using \textit{t} by blast
qed

fix \(X\ Y\) assume realrel \(X\ Y\)
then have \(\text{realrel} \ X\ Y\) and \(\text{realrel} \ Y\ X\)
  using symp-realrel by (auto simp: symp-def)
then show \(\text{thesis} \ X\ Y\)
  by (safe elim: 1)
qed

lemma \textit{positive-Real}: \(\text{cauchy} \ X\ =\Rightarrow\text{positive} \ (\text{Real} \ X)\ \leftrightarrow\ (\exists r>0. \ \exists k. \ \forall n\geq k. \ r \ < X n)\)
using positive.transfer by (simp add: cr-real-eq rel-fun-def)

lemma \textit{positive-zero}: \(\neg\text{positive} \ 0\)
by transfer auto

lemma \textit{positive-add}:
assumes \(\text{positive} \ x\ \text{positive} \ y\)
shows \(\text{positive} \ (x + y)\)
proof
  have \(\ast\): \(\forall n\geq i. \ a < x n; \ \forall n\geq j. \ b < y n; \ 0 < a; \ 0 < b; \ n \geq \max i j\)
    \(\Rightarrow\ a + b < x n + y n\ \text{for} \ x\ y\ \text{and} \ a \ b::\text{rat}\ \text{and} \ i\ j\ n::\text{nat}\)
  by (simp add: add-strict-mono)
show \(\text{thesis}\)
  using \textit{assms}
  by transfer (blast intro: \(\ast\) \ pos-add-strict)
qed

lemma \textit{positive-mult}:
assumes \(\text{positive} \ x\ \text{positive} \ y\)
shows \(\text{positive} \ (x * y)\)
proof
  have \(\ast\): \(\forall n\geq i. \ a < x n; \ \forall n\geq j. \ b < y n; \ 0 < a; \ 0 < b; \ n \geq \max i j\)
    \(\Rightarrow\ a * b < x n + y n\ \text{for} \ x\ y\ \text{and} \ a \ b::\text{rat}\ \text{and} \ i\ j\ n::\text{nat}\)
  by (simp add: mult-strict-mono)
show \(\text{thesis}\)
  using \textit{assms}
  by transfer (blast intro: \(\ast\) \ mult-pos-pos)
qed

lemma \textit{positive-minus}: \(\neg\text{positive} \ x\ \Rightarrow\ x \neq 0\ \Rightarrow\text{positive} \ (\ - \ x)\)
apply transfer
apply (simp add: realrel-def)
apply (blast dest: cauchy-not-vanishes-cases)
done

instantiation \textit{real} :: \textit{linordered-field}
begin

definition \(x < y\ \longleftrightarrow\ \text{positive} \ (y - x)\)
definition \( x \leq y \iff x < y \lor x = y \) for \( x, y :: \text{real} \)

definition \( |a| = (\text{if } a < 0 \text{ then } -a \text{ else } a) \) for \( a :: \text{real} \)

definition \( \text{sgn} \ a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } -1) \) for \( a :: \text{real} \)

instance
proof
  fix \( a, b, c :: \text{real} \)
  show \( |a| = (\text{if } a < 0 \text{ then } -a \text{ else } a) \)
    by (rule abs-real-def)
  show \( a < b \iff a \leq b \land \neg b \leq a \)
    a \leq b \implies b \leq c \implies a \leq c \quad a \leq a \)
    a \leq b \implies b \leq a \implies a = b \)
    a \leq b \implies c + a \leq c + b \)
    unfolding less-eq-real-def less-real-def
    by (force simp add: positive-zero dest: positive-add) +
  show \( \text{sgn} \ a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } -1) \)
    by (rule sgn-real-def)
  show \( a \leq b \lor b \leq a \)
    by (auto dest!: positive-minus simp: less-eq-real-def less-real-def)
  show \( a < b \implies 0 < c \implies c \cdot a < c \cdot b \)
    unfolding less-real-def
    by (force simp add: algebra-simps dest: positive-mult)
qed

end

instantiation \( \text{real} :: \text{distrib-lattice} \)
begin

definition \( (\text{inf} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}) = \text{min} \)

definition \( (\text{sup} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}) = \text{max} \)

instance
  by standard (auto simp add: inf-real-def sup-real-def max-min-distrib2)
end

lemma of-nat-Real: \( \text{of-nat} \ x = \text{Real} \ (\lambda n. \text{of-nat} \ x) \)
  by (induct \( x \)) (simp-all add: zero-real-def one-real-def add-Real)

lemma of-int-Real: \( \text{of-int} \ x = \text{Real} \ (\lambda n. \text{of-int} \ x) \)
  by (cases \( x \) rule: int-diff-cases) (simp add: of-nat-Real diff-Real)

lemma of-rat-Real: \( \text{of-rat} \ x = \text{Real} \ (\lambda n. \ x) \)
proof (induct \( x \))
THEORY “Real”

\begin{verbatim}
  case (Fract a b)
  then show ?case
  apply (simp add: Fract-of-int-quotient of-rat-divide)
  apply (simp add: of-int-Real divide-inverse inverse-Real mult-Real)
  done

  qed

instance real :: archimedean-field
  proof
    show \( \exists z. x \leq \text{of-int } z \) for \( x :: \text{real} \)
    proof (induct x)
      case (1 X)
      then obtain b where \( 0 < b \) and \( |X n| < b \)
        by (blast dest: cauchy-imp-bounded)
      then have \( \text{Real } X < \text{of-int } (\lceil b \rceil + 1) \)
        using 1
        apply (simp add: of-int-Real less-real-def diff-Real positive-Real)
        apply (rule-tac x=1 in exI)
        apply (simp add: algebra-simps)
        by (metis abs-ge-self le-less-trans le-of-int-ceiling less-le)
      then show ?case
        using less-eq-real-def by blast
    qed

  qed

instantiation real :: floor-ceiling
  begin

  definition [code del]: \( \lfloor x :: \text{real} \rfloor = (\text{THE } z. \text{of-int } z \leq x \land x < \text{of-int } (z + 1)) \)

  instance
    proof
      show \( \text{of-int } \lfloor x \rfloor \leq x \land x < \text{of-int } (\lfloor x \rfloor + 1) \) for \( x :: \text{real} \)
        unfolding floor-real-def using floor-exists1 by (rule theI')
    qed

  end

98.7 Completeness

lemma not-positive-Real:
  assumes cauchy X shows \( \neg \text{positive } (\text{Real } X) \leftrightarrow (\forall r>0. \exists k. \forall n\geq k. X n \leq r) \) (is \( ?lhs = ?rhs \))
  unfolding positive-Real [OF assms]
  proof (intro iffI allI notI impl)
    show \( \exists k. \forall n\geq k. X n \leq r \) if \( r \) \( \neg (\exists r>0. \exists k. \forall n\geq k. r < X n) \) and \( 0 < r \) for \( r \)
    proof
      obtain s t where \( s > 0 \) t > 0 r = s+t
      using \( r > 0 \) obtain-pos-sum by blast
  qed
\end{verbatim}
obtain \( k \) where \( k: \bigwedge m. \lfloor m \geq k; n \geq k \rfloor \implies |X m - X n| < t \)
using cauchyD [OF assms \( t > 0 \)] by blast
obtain \( n \) where \( n \geq k \) \( X n \leq s \)
by (meson \( r \langle 0 < s \rangle \) not-less)
then have \( X l \leq r \) if \( l \geq n \) for \( l \)
using \( k \) [OF \( n \geq k \) \( \) of \( l \)]
that \( \langle r = s + t \rangle \) by linarith
then show \( \) thesis
by blast
qed (meson le-cases not-le)

lemma le-Real
assumes cauchy X cauchy Y
shows Real X \( \leq \) Real Y = \( (\forall r > 0. \exists k. \forall n \geq k. X n \leq Y n + r) \)
unfolding not-less [symmetric, where 'a=real] less-real-def
apply (simp add: diff-Real not-positive-Real assms)
apply (simp add: diff-le-eq ac-simps)
done

lemma le-RealI
assumes Y: cauchy Y
shows \( \forall n. x \leq \text{of-rat} (Y n) \implies x \leq \text{Real} Y \)
proof (induct x)
fix X
assume X: cauchy X and \( \forall n. \text{Real} X \leq \text{of-rat} (Y n) \)
then have le: \( \forall m. 0 < r \implies \exists k. \forall n \geq k. X n \leq Y m + r \)
by (simp add: of-rat-Real le-Real)
then have \( \exists k. \forall n \geq k. X n \leq Y n + r \) if \( 0 < r \) for \( r :: \text{rat} \)
proof –
from that obtain s t where: \( 0 < s \) and \( t: 0 < t \) and \( r: r = s + t \)
by (rule obtain-pos-sum)
obtain i where: i: \( \forall m \geq i. \forall n \geq i. |Y m - Y n| < s \)
using cauchyD [OF Y s] ..
obtain j where: j: \( \forall n \geq j. X n \leq Y i + t \)
using le [OF i] ..
have \( \forall n \geq \text{max} i j. X n \leq Y n + r \)
proof clarsimp
fix n
assume n: i \( \leq n \) j \( \leq n \)
have X n \( \leq Y i + t \)
using n j by simp
moreover have \( |Y i - Y n| < s \)
using n i by simp
ultimately show X n \( \leq Y n + r \)
unfolding r by simp
qed
then show \( \) thesis ..
qed
then show \( \) Real X \( \leq \) Real Y
by (simp add: of-rat-Real le-Real X Y)

qed

lemma Real-leI:
assumes X: cauchy X
assumes le: \(\forall n. \text{of-rat } (X n) \leq y\)
shows Real X \(\leq y\)

proof
  have \(-y \leq -\text{Real } X\)
  by (simp add: minus-Real X le-RealI of-rat-minus le)
  then show \(?thesis\) by simp

qed

lemma less-RealD:
assumes cauchy Y
shows \(x < \text{Real } Y \implies \exists n. x < \text{of-rat } (Y n)\)

apply (erule contrapos-pp)
apply (simp add: not-less)
apply (erule Real-leI [OF assms])
done

lemma of-nat-less-two-power [simp]: \(\text{of-nat } n < (2::'a::linordered-idom)^n\)

apply (induct n)
apply simp
apply (metis add-le-less-mono mult-2 of-nat-Suc one-le-numeral one-le-power power-Suc)
done

lemma complete-real:
fixes S :: real set
assumes \(\exists x. x \in S\) and \(\exists z. \forall x \in S. x \leq z\)
shows \(\exists y. (\forall x \in S. x \leq y) \land (\forall z. (\forall x \in S. x \leq z) \implies y \leq z)\)

proof
  obtain x where x: x \(\in S\) using assms(1) ..
  obtain z where z: \(\forall x \in S. x \leq z\) using assms(2) ..

define P where P x \(\leftrightarrow (\forall y \in S. y \leq \text{of-rat } x)\) for x
obtain a where a: \(\neg P\) a

proof
  have \(\text{of-int } \lfloor x - 1 \rfloor \leq x - 1\) by (rule of-int-floor-le)
  also have \(x - 1 < x\) by simp
  finally have \(\text{of-int } \lfloor x - 1 \rfloor < x\).
  then have \(\neg x \leq \text{of-int } \lfloor x - 1 \rfloor\) by (simp only: not-le)
  then show \(\neg P (\text{of-int } \lfloor x - 1 \rfloor)\)
    unfolding \(P\)-def of-rat-of-int-eq using x by blast

qed

obtain b where b: P b

proof
  show P \(\text{of-int } \lceil z \rceil\)
unfolding $P$-def of-rat-of-int-eq
proof
  fix $y$ assume $y \in S$
  then have $y \leq z$ using $z$ by simp
  also have $z \leq \text{of-int } \lceil z \rceil$ by (rule le-of-int-ceiling)
  finally show $y \leq \text{of-int } \lceil z \rceil$.
qed
qed

define $\text{avg}$ where $\text{avg } x y = x/2 + y/2$ for $x y :: \text{rat}$
define $\text{bisect}$ where $\text{bisect } = (\lambda(x, y). \text{if } P(\text{avg } x y) \text{ then } (x, \text{avg } x y) \text{ else } (\text{avg } x y, y))$
define $A$ where $A n = \text{fst } ((\text{bisect } ^n n) (a, b))$ for $n$
define $B$ where $B n = \text{snd } ((\text{bisect } ^n n) (a, b))$ for $n$
define $C$ where $C n = \text{avg } (A n) (B n)$ for $n$
have $A$-0 [simp]: $A 0 = a$ unfolding $A$-def by simp
have $B$-0 [simp]: $B 0 = b$ unfolding $B$-def by simp
have $A$-Suc [simp]: $\forall n. A (\text{Suc } n) = (\text{if } P(C n) \text{ then } A n \text{ else } C n)$
  unfolding $A$-def $B$-def $C$-def $\text{bisect-def } \text{split-def}$ by simp
have $B$-Suc [simp]: $\forall n. B (\text{Suc } n) = (\text{if } P(C n) \text{ then } C n \text{ else } B n)$
  unfolding $A$-def $B$-def $C$-def $\text{bisect-def } \text{split-def}$ by simp
have width: $B n - A n = (b - a) / 2^n$ for $n$
proof (induct $n$)
  case (Suc $n$)
  then show ?case
by (simp add: $C$-def eq-divide-eq avg-def algebra-simps)
qed simp
have twos: $\exists n. y / 2^n < r$ if $0 < r$ for $y r :: \text{rat}$
proof –
  obtain $n$ where $y / r < \text{rat-of-nat } n$
  using $0 < r$ by reals-Archimedean2 blast
  then have $\exists n. y < r * 2^n$
  by (metis divide-less-eq less-trans mult.commute of-nat-less-two-power that)
  then show ?thesis
by (simp add: field-split-simps)
qed
have $PA$: $\neg P (A n)$ for $n$
  by (induct $n$) (simp-all add: $a$)
have $PB$: $P (B n)$ for $n$
  by (induct $n$) (simp-all add: $b$)
have ab: $a < b$
  using $a b$ unfolding $P$-def
  by (meson leI less-le-trans of-rat-less)
have $AB$: $A n < B n$ for $n$
  by (induct $n$) (simp-all add: ab $C$-def avg-def)
have $A i \leq A j \land B j \leq B i$ if $i < j$ for $i j$
  using that
proof (induction rule: less-Suc-induct)
case (1 i)
  then show \(?case
  apply (clarsimp simp add: C-def avg-def add-divide-distrib [symmetric])
  apply (rule AB [THEN less-imp-le])
  done
qed

then have A-mono: \( A i \leq A j \) and B-mono: \( B j \leq B i \) if \( i \leq j \) for \( i, j \)
  by (metis eq-refl le-neq-implies-less that)
have cauchy-lemma: cauchy X if \( \forall n. i \geq n \implies A n \leq X i \land X i \leq B n \) for X
  proof (rule cauchyI)
    fix \( r \)
    assume \( \exists r. \exists k \rightarrow B k - A k \)
    have \( \exists r. \forall n \geq k. |X m - X n| < r \)
      by (simp add: * abs-rat-def diff-mono that)
    also have ... < r
      by (simp add: k width)
    finally show \(?thesis \).
  qed
then show \( \exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r \)
  by blast
qed

have cauchy A
  by (rule cauchy-lemma) (meson AB A-mono B-mono dual-order.strict-implies-order le-less-trans)
have cauchy B
  by (rule cauchy-lemma) (meson AB A-mono B-mono dual-order.strict-implies-order le-less-trans)
have \( \forall x \in S. x \leq \text{Real} B \)
proof
  fix \( x \)
  assume \( x \in S \)
  then show \( x \leq \text{Real} B \)
    using PB [unfolded P-def \( \langle \text{cauchy} B \rangle \)]
    by (simp add: le-RealI)
qed

moreover have \( \forall z. (\forall x \in S. x \leq z) \rightarrow \text{Real} A \leq z \)
  by (meson PA Real-leI P-def \( \langle \text{cauchy} A \rangle \) le-cases order.trans)
moreover have vanishes (\( \lambda n. (b - a) / 2 ^ {\sim n} \))
proof (rule vanishesI)
  fix \( r :: \text{rat} \)
  assume \( \exists r. \exists k \rightarrow |b - a| / 2 ^ {\sim k} < r \)
  using twos by blast
  have \( \forall n \geq k. |(b - a) / 2 ^ {\sim n}| < r \)
proof  clarify
  fix n
  assume n: k ≤ n
  have \(|(b - a) / 2 ^ n| = |b - a| / 2 ^ n\)
    by simp
  also have ... ≤ |b - a| / 2 ^ k
    using n by (simp add: divide-left-mono)
  also note k
  finally show \(|(b - a) / 2 ^ n| < r\).
qed

then show \(\exists k. \forall n \geq k. \|(b - a) / 2 ^ n\| < r\).
qed

then have \(\text{Real } B = \text{Real } A\)
  by (simp add: eq-Real \(\langle \text{cauchy } A \rangle \langle \text{cauchy } B \rangle \text{ width}\))
ultimately show \(\exists y. (\forall x \in S. x \leq y) \land (\forall z. (\forall x \in S. x \leq z) \longrightarrow y \leq z)\)
  by force
qed

instantiation real :: linear-continuum
begin

98.8  Supremum of a set of reals

definition \(\text{Sup } X = (\text{LEAST } z::\text{real. } \forall x \in X. x \leq z)\)
definition \(\text{Inf } X = - \text{Sup } (\text{uminus } \langle X \rangle)\) for \(X::\text{real set}\)

instance
proof
  show Sup-upper: \(x \leq \text{Sup } X\)
    if \(x \in X\) bdd-above \(X\)
    for \(x::\text{real and } X::\text{real set}\)
  proof --
    from that obtain \(s\) where \(s::(\forall y \in X. y \leq s) \land (\forall z. (\forall y \in X. y \leq z) \longrightarrow s \leq z)\)
      using complete-real[of \(X\)] unfolding bdd-above-def by blast
    then show \(?\text{thesis}\)
      unfolding Sup-real-def by (rule LeastI2-order) (auto simp: that)
  qed
  show Sup-least: \(\text{Sup } X \leq z\)
    if \(X \neq \{\}\) and \(z::(\forall x \in X \longrightarrow x \leq z)\)
    for \(z::\text{real and } X::\text{real set}\)
  proof --
    from that obtain \(s\) where \(s::(\forall y \in X. y \leq s) \land (\forall z. (\forall y \in X. y \leq z) \longrightarrow s \leq z)\)
      using complete-real[of \(X\)] by blast
    then have \(\text{Sup } X = s\)
      unfolding Sup-real-def by (best intro: Least-equality)
    also from \(s\) \(z\) have ... \(\leq z\)
      by blast
    finally show \(?\text{thesis}\).
  qed
show Inf X ≤ x if x ∈ X bdd-below X
for x :: real and X :: real set
  using Sup-upper [of ~x uminus ' X] by (auto simp: Inf-real-def that)
show z ≤ Inf X if X ≠ {} \(x, x \in X \implies z ≤ x\)
for z :: real and X :: real set
  using Sup-least [of ~x uminus ' X ~ z] by (force simp: Inf-real-def that)
show \(\exists a b :: real. a \neq b\)
  using zero-neq-one by blast
qed

end

98.9 Hiding implementation details
hide-const (open) vanishes cauchy positive Real

declare Real-induct [induct del]
declare Abs-real-induct [induct del]
declare Abs-real-cases [cases del]

lifting-update real.lifting
lifting-forget real.lifting

98.10 Embedding numbers into the Reals
abbreviation real-of-nat :: nat ⇒ real
  where real-of-nat ≡ of-nat

abbreviation real :: nat ⇒ real
  where real ≡ of-nat

abbreviation real-of-int :: int ⇒ real
  where real-of-int ≡ of-int

abbreviation real-of-rat :: rat ⇒ real
  where real-of-rat ≡ of-rat

declare [[coercion-enabled]]
declare [[coercion of-nat :: nat ⇒ int]]
declare [[coercion of-nat :: nat ⇒ real]]
declare [[coercion of-int :: int ⇒ real]]

declare [[coercion-map map]]
declare [[coercion-map λf g h x. g (h (f x))]]
declare [[coercion-map λf g (x,y). (f x, g y)]]
declare of-int-eq-0-iff [algebra, presburger]
lemma \text{int-less-real-le}: \, n < m \iff \text{real-of-int } n + 1 \leq \text{real-of-int } m

proof
\begin{itemize}
  \item have \((0::\text{real}) \leq 1\) by (metis less-eq-real-def zero-less-one)
  \item then show \(?\text{thesis}\)
    \begin{itemize}
      \item by (metis floor-of-int less-floor-iff)
    \end{itemize}
\end{itemize}
qed

lemma \text{int-le-real-less}: \, n \leq m \iff \text{real-of-int } n < \text{real-of-int } m + 1
by (meson int-less-real-le not-le)

lemma \text{real-of-int-div-aux}:
\begin{align*}
(\text{real-of-int } x) / (\text{real-of-int } d) &= \text{real-of-int } (x \div d) + (\text{real-of-int } (x \mod d)) / (\text{real-of-int } d)
\end{align*}

proof
\begin{itemize}
  \item have \(x = (x \div d) \times d + x \mod d\)
    \begin{itemize}
      \item by auto
    \end{itemize}
  \item then have \(\text{real-of-int } x = \text{real-of-int } (x \div d) \times \text{real-of-int } d + \text{real-of-int}(x \mod d)\)
    \begin{itemize}
      \item by (metis of-int-add of-int-mult)
    \end{itemize}
  \item then have \(\text{real-of-int } x / \text{real-of-int } d = \ldots / \text{real-of-int } d\)
    \begin{itemize}
      \item by simp
    \end{itemize}
  \item then show \(?\text{thesis}\)
    \begin{itemize}
      \item by (auto simp add: add-divide-distrib algebra-simps)
    \end{itemize}
\end{itemize}
qed

lemma \text{real-of-int-div}:
\begin{align*}
d \mid n \Rightarrow \text{real-of-int } (n \div d) &= \text{real-of-int } n / \text{real-of-int } d \text{ for } d, n :: \text{int}
\end{align*}
by (simp add: real-of-int-div-aux)

lemma \text{real-of-int-div2}: \(0 \leq \text{real-of-int } n / \text{real-of-int } x = \text{real-of-int } (n \div x)\)

proof (cases \(x = 0\))
\begin{itemize}
  \item case False
  \item then show \(?\text{thesis}\)
    \begin{itemize}
      \item by (metis diff-ge-0-iff-ge floor-divide-of-int-eq of-int-floor-le)
    \end{itemize}
\end{itemize}
qed simp
lemma real-of-int-div3: real-of-int n / real-of-int x - real-of-int (n div x) ≤ 1
  apply (simp add: algebra-simps)
  by (metis add.commute floor-correct floor-divide-of-int-eq less-eq-real-def of-int-1 of-int-add)

lemma real-of-int-div4: real-of-int (n div x) ≤ real-of-int n / real-of-int x
  using real-of-int-div2 [of n x] by simp

98.11 Embedding the Naturals into the Reals

lemma real-of-card: real (card A) = sum (λx. 1) A
  by simp

lemma nat-less-real-le: n < m → real n + 1 ≤ real m
  proof
    have ‹n < m → Suc n ≤ m›
      by (simp add: less-eq-Suc-le)
    also have ‹. . . → real (Suc n) ≤ real m›
      by (simp only: of-nat-le-iff)
    also have ‹. . . → real n + 1 ≤ real m›
      by (simp add: ac-simps)
    finally show ?thesis.
  qed

lemma nat-le-real-less: n ≤ m → real n < real m + 1
  by (meson nat-less-real-le not-le)

lemma real-of-nat-div-aux: real x / real d = real (x div d) + real (x mod d) / real d
  proof
    have x = (x div d) * d + x mod d
      by auto
    then have real x = real (x div d) * real d + real (x mod d)
      by (metis of-nat-add of-nat-mult)
    then have real x / real d = . . . / real d
      by simp
    then show ?thesis
      by (auto simp add: add-divide-distrib algebra-simps)
  qed

lemma real-of-nat-div: d dvd n → real(n div d) = real n / real d
  by (subst real-of-nat-div-aux) (auto simp add: dvd-eq-mod-eq-0 [symmetric])

lemma real-of-nat-div2: 0 ≤ real n / real x - real (n div x) for n x :: nat
  apply (simp add: algebra-simps)
  by (metis floor-divide-of-nat-eq of-int-floor-le of-int-of-nat-eq)

lemma real-of-nat-div3: real n / real x - real (n div x) ≤ 1 for n x :: nat
THEORY "Real"

by (metis of-int-of-nat-eq real-of-int-div3 of-nat-div)

lemma real-of-nat-div4: real (n div x) ≤ real n / real x for n x :: nat
  using real-of-nat-div2 [of n x] by simp

lemma real-binomial-eq-mult-binomial-Suc:
  assumes k ≤ n
  shows real (n choose k) = (n + 1 - k) / (n + 1) * (Suc n choose k)
  using assms

98.12 The Archimedean Property of the Reals

lemma real-arch-inverse: 0 < e ←→ (∃ n::nat. n ≠ 0 ∧ 0 < inverse (real n) ∧ inverse (real n) < e)
  using reals-Archimedean[of e] less-trans[of 0 1 / real n e for n::nat]
  by (auto simp add: field-simps cong: conj-cong simp del: of-nat-Suc)

lemma reals-Archimedean3: 0 < x =⇒ ∀ y. ∃ n. y < real n * x
  by (auto intro: ex-less-of-nat-mult)

lemma real-archimedian-rdiv-eq-0:
  assumes x0: x ≥ 0
    and c: c ≥ 0
    and xc: ∀ m::nat. m > 0 =⇒ real m * x ≤ c
  shows x = 0
  by (metis reals-Archimedean3 dual-order order-iff-strict le0 le-less-trans not-le x0 xc)

lemma inverse-Suc: inverse (Suc n) > 0
  using of-nat-0-less-iff positive-imp-inverse-positive zero-less-Suc by blast

lemma Archimedean-eventually-inverse:
  fixes ε::real shows (∀ x in sequentially. inverse (real (Suc n)) < ε) ←→ 0 < ε
  (is ?lhs=?rhs)
  proof
    assume ?lhs
    then show ?rhs
      unfolding eventually-at-top-dense using inverse-Suc order-less-trans by blast
  next
    assume ?rhs
    then obtain N where inverse (Suc N) < ε
      using reals-Archimedean by blast
    moreover have inverse (Suc n) ≤ inverse (Suc N) if n ≥ N for n
      using inverse-Suc that by fastforce
    ultimately show ?lhs
      unfolding eventually-sequentially
      using order-le-less-trans by blast
On the relationship between two different ways of converting to 0

**Lemma** Inter-eq-Inter-inverse-Suc:  
\[ \forall r', r < r \implies A r' \subseteq A r \]

**Proof**

\[ \forall x \in A \varepsilon \]  
\[ \text{if } x : \forall n. x \in A (\text{inverse} (\text{Suc} n)) \text{ and } \varepsilon > 0 \text{ for } x \text{ and } \varepsilon : \text{real} \]

**Proof –**

\[ \text{obtain } n \text{ where } \text{inverse} (\text{Suc} n) < \varepsilon \]
\[ \text{using } (\varepsilon > 0), \text{reals-Archimedean by blast} \]
\[ \text{with asms } x \text{ show } \text{thesis} \]
\[ \text{by blast} \]

**QED**

then show \( \bigcap n . A (\text{inverse} (\text{Suc} n)) \subseteq (\bigcap \varepsilon \in \{0 <..\}. A \varepsilon) \)
by auto

**QED (use inverse-Suc in fastforce)**

### 98.13 Rationals

**Lemma** Rats-abs-iff[simp]:  
\[ [(x : real)] \in \mathbb{Q} \iff x \in \mathbb{Q} \]
by(simp add: abs-real-def split: if-splits)

**Lemma** Rats-eq-int-div-int: \( \mathbb{Q} = \{\text{real-of-int } i / \text{real-of-int } j \mid i, j \neq 0\} \)  
(is - = ?S)

**Proof**

show \( \mathbb{Q} \subseteq ?S \)

**Proof**

fix \( x : \text{real} \)
assume \( x \in \mathbb{Q} \)
then obtain \( r \) where \( x = \text{of-rat } r \)
unfolding Rats-def ..
have \( \text{of-rat } r \in ?S \)
by (cases r) (auto simp add: of-rat-rat)
then show \( x \in ?S \)
using \( (x = \text{of-rat } r) \) by simp

**QED**

next

show \( ?S \subseteq \mathbb{Q} \)

**Proof** (auto simp: Rats-def)

fix \( i, j : \text{int} \)
assume \( j \neq 0 \)
then have \( \text{real-of-int } i / \text{real-of-int } j = \text{of-rat} (\text{Fract } i j) \)
by (simp add: of-rat-rat)
then show \( \text{real-of-int } i / \text{real-of-int } j \in \text{range } \text{of-rat} \)
by blast

**QED**
qed

lemma Rats-eq-int-div-nat: $\mathbb{Q} = \{ \text{real-of-int } i / \text{real } n \mid i, n. n \neq 0 \}$
proof (auto simp: Rats-eq-int-div-int)
  fix $i \cdot j :: \text{int}$
  assume $j \neq 0$
  show $\exists (i'::\text{int}) (n::\text{nat}). \text{real-of-int } i / \text{real-of-int } j = \text{real-of-int } i' / \text{real } n \land 0 < n$
  proof (cases $j > 0$)
    case True
    then have $\text{real-of-int } i / \text{real-of-int } j = \text{real-of-int } i / \text{real } (\text{nat } j) \land 0 < \text{nat } j$
    by simp
    then show $?\text{thesis}$ by blast
  next
    case False
    with $\langle j \neq 0 \rangle$
    have $\text{real-of-int } i / \text{real-of-int } j = \text{real-of-int } (-i) / \text{real } (\text{nat } (-j)) \land 0 < \text{nat } (-j)$
    by simp
    then show $?\text{thesis}$ by blast
  qed

next
  fix $i :: \text{int}$ and $n :: \text{nat}$
  assume $0 < n$
  then have $\text{real-of-int } i / \text{real } n = \text{real-of-int } i / \text{real-of-int}(\text{int } n) \land \text{int } n \neq 0$
  by simp
  then show $\exists i' j. \text{real-of-int } i / \text{real } n = \text{real-of-int } i' / \text{real-of-int } j \land j \neq 0$
  by blast
qed

lemma Rats-abs-nat-div-natE:
assumes $x \in \mathbb{Q}$
obtains $m n :: \text{nat}$ where $n \neq 0$ and $|x| = \text{real } m / \text{real } n$ and coprime $m n$
proof
  from $\langle x \in \mathbb{Q} \rangle$
  obtain $i :: \text{int}$ and $n :: \text{nat}$ where $n \neq 0$ and $x = \text{real-of-int } i / \text{real } n$
  by (auto simp add: Rats-eq-int-div-nat)
  then have $|x| = \text{real } (\text{nat } |i|) / \text{real } n$ by simp
  then obtain $m :: \text{nat}$ where $x\text{-rat}: |x| = \text{real } m / \text{real } n$ by blast
  let $\?gcd = \text{gcd } m n$
  from $\langle n \neq 0 \rangle$
  have $\text{gcd}: \?gcd \neq 0$ by simp
  let $?k = m \text{ div } \?gcd$
  let $?l = n \text{ div } \?gcd$
  let $\?gcd' = \text{gcd } ?k \cdot ?l$
  have $\?gcd \text{ dvd } m$ ..
  then $\text{have } \text{gcd-k}: \?gcd \ast ?k = m$
    by (rule dvd-mult-div-cancel)
  have $\?gcd \text{ dvd } n$ ..
  then $\text{have } \text{gcd-l}: \?gcd \ast ?l = n$
by (rule dvd-mult-div-cancel)
from \( \langle n \neq 0 \rangle \) and \( \text{gcd}-l \) have \( \text{gcd} \ast \overline{n} \neq 0 \) by simp
then have \( \overline{l} \neq 0 \) by (blast dest!: mult-not-zero)
moreover
have \( |x| = \text{real } \overline{k} / \text{real } \overline{l} \)
proof –
from gcd have \( \text{real } \overline{k} / \text{real } \overline{l} = \text{real } (\text{gcd} \ast \overline{k}) / \text{real } (\text{gcd} \ast \overline{l}) \)
by (simp add: real-of-nat-div)
also from gcd-k and gcd-l have \( \ldots = \text{real } m / \text{real } n \) by simp
also from x-rat have \( \ldots = |x| \).
finally show \( \text{thesis} \).
qed
moreover
have \( \text{gcd} \prime = 1 \)
proof –
have \( \text{gcd} \ast \overline{\text{gcd} \prime} = \text{gcd} (\text{gcd} \ast \overline{k}) (\text{gcd} \ast \overline{l}) \)
by (rule gcd-mult-distrib-nat)
with gcd-k gcd-l have \( \text{gcd} \ast \overline{\text{gcd} \prime} = \text{gcd} \) by simp
with gcd show \( \text{thesis} \) by auto
qed
then have coprime \( \overline{k} \overline{l} \)
by (simp only: coprime-iff-gcd-eq-1)
ultimately show \( \text{thesis} \).
qed

98.14 Density of the Rational Reals in the Reals

This density proof is due to Stefan Richter and was ported by TN. The original source is *Real Analysis* by H.L. Royden. It employs the Archimedean property of the reals.

lemma Rats-dense-in-real:
fixes \( x :: \text{real} \)
assumes \( x < y \)
shows \( \exists \overline{r} \in \text{Q}. \ x < \overline{r} \land \overline{r} < y \)
proof –
from \( \langle x < y \rangle \) have \( 0 < y - x \) by simp
with reals-Archimedean obtain \( q :: \text{nat} \) where \( q \) : inverse (real \( q \)) < \( y - x \) and \( 0 < q \)
by blast
define \( p \) where \( p = \lfloor y * \text{real } q \rfloor - 1 \)
define \( r \) where \( r = \text{of-int } p / \text{real } q \)
from \( q \) have \( x < y - \text{inverse} (\text{real } q) \)
by simp
also from \( \langle 0 < q \rangle \) have \( y - \text{inverse} (\text{real } q) \leq r \)
by (simp add: r-def p-def le-divide-eq left-diff-distrib)
finally have \( x < r \).
moreover from \( \langle 0 < q \rangle \) have \( r < y \)
by (simp add: r-def p-def divide-less-eq diff-less-eq less-ceiling-iff [symmetric])
moreover have \( r \in \text{Q} \)
by \(\text{simp add: \textit{r-def}}\)
ultimately show \textit{thesis by blast}
\textit{qed}

\textbf{lemma} \textit{of-rat-dense}:
\textit{fixes} \(x\ y::\text{real}\)
\textit{assumes} \(x < y\)
\textit{shows} \(\exists q::\text{rat}.\ x < \text{of-rat}\ q\ \land\ \text{of-rat}\ q < y\)
\textit{using} \textit{Rats-dense-in-real} [\(\text{OF}\ (x < y)\)]
\textit{by} (\textit{auto elim: Rats-cases})

\textbf{98.15} \textbf{Numerals and Arithmetic}
\textbf{declaration} \(\langle\)
\textit{K\ (Lin-Arith.add-inj-const (const-name \textit{of-nat}, typ \textit{nat}\ \Rightarrow\ \textit{real})\)}
\(\rangle\)

\textbf{98.16} \textbf{Simprules combining} \(x + y\) \textbf{and} \(0\)
\textbf{lemma} \textit{real-add-minus-iff} [\textit{simp}]: \(x + -a = 0 \longleftrightarrow x = a\)
\textit{for} \(x\ a::\text{real}\)
\textit{by} \textit{arith}

\textbf{lemma} \textit{real-add-less-0-iff} : \(x + y < 0 \longleftrightarrow y < -x\)
\textit{for} \(x\ y::\text{real}\)
\textit{by} \textit{auto}

\textbf{lemma} \textit{real-0-less-add-iff} : \(0 < x + y \longleftrightarrow -x < y\)
\textit{for} \(x\ y::\text{real}\)
\textit{by} \textit{auto}

\textbf{lemma} \textit{real-add-le-0-iff} : \(x + y \leq 0 \longleftrightarrow y \leq -x\)
\textit{for} \(x\ y::\text{real}\)
\textit{by} \textit{auto}

\textbf{lemma} \textit{real-0-le-add-iff} : \(0 \leq x + y \longleftrightarrow -x \leq y\)
\textit{for} \(x\ y::\text{real}\)
\textit{by} \textit{auto}

\textbf{98.17} \textbf{Lemmas about powers}
\textbf{lemma} \textit{two-realpow-ge-one} : \((1::\text{real}) \leq 2 ^ n\)
\textit{by} \textit{simp}

\textbf{declare} \textit{sum-squares-eq-zero-iff [simp]} \textit{sum-power2-eq-zero-iff [simp]}

\textbf{lemma} \textit{real-minus-mult-self-le} [\textit{simp}]: \(- (u * u) \leq x * x\)
\textit{for} \(u\ x::\text{real}\)
by (rule order-trans [where y = 0]) auto

lemma realpow-square-minus-le [simp]: \(-u^2 \leq x^2\)
  for u x :: real
  by (auto simp add: power2_eq_square)

98.18 Density of the Reals

lemma field-lbound-gt-zero: \(0 < d1 \Longrightarrow 0 < d2 \Longrightarrow \exists e. 0 < e \land e < d1 \land e < d2\)
  for d1 d2 :: 'a:linordered-field
  by (rule exI [where x = min d1 d2 / 2]) (simp add: min_def)

lemma field-less-half-sum: \(x < y \Longrightarrow x < (x + y) / 2\)
  for x y :: 'a:linordered-field
  by auto

lemma field-sum-of-halves: \(x / 2 + x / 2 = x\)
  for x :: 'a:linordered-field
  by simp

98.19 Archimedean properties and useful consequences

Bernoulli’s inequality

proposition Bernoulli-inequality:
  fixes x :: real
  assumes \(-1 \leq x\)
  shows \(1 + n * x \leq (1 + x)^n\)
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have \(1 + Suc n * x \leq 1 + (Suc n) * x + n * x^2\)
    by (simp add: algebra_simps)
  also have \(\ldots = (1 + x) * (1 + n * x)\)
    by (auto simp: power2_eq_square algebra_simps)
  also have \(\ldots \leq (1 + x)^Suc n\)
    using Suc.hyps assms mult-left-mono by fastforce
  finally show ?case.
qed

corollary Bernoulli-inequality-even:
  fixes x :: real
  assumes even n
  shows \(1 + n * x \leq (1 + x)^n\)
proof (cases \(-1 \leq x \lor n=0\))
  case True
  then show ?thesis
by (auto simp: Bernoulli-inequality)
next
  case False
  then have real \( n \geq 1 \)
    by simp
  with False have \( n \times x \leq -1 \)
    by (metis linear minus-zero mult.commute mult.left-neutral mult-left-mono-neg
      neg-le-iff-le order-trans zero-le-one)
  then have \( 1 + n \times x \leq 0 \)
    by auto
  also have ... \( \leq (1 + x)^n \)
    using assms
    using zero-le-even-power by blast
  finally show \(?\)thesis \.
qed

corollary real-arch-pow:
  fixes \( x :: real \)
  assumes \( x \times 1 < x \)
  shows \( \exists n. y < x^n \)
proof
  from \( x \) have \( x0: x - 1 > 0 \)
    by arith
  from reals-Archimedean3[OF x0, rule-format, of y]
  obtain \( n :: nat \) where \( n: y < real n \times (x - 1) \)
    by metis
  from \( x0 \) have \( x00: x - 1 \geq -1 \)
    by arith
  from Bernoulli-inequality[OF \( x00 \), of \( n \)]
  have \( y < x^n \)
    by auto
  then show \(?\)thesis by metis
qed

corollary real-arch-pow-inv:
  fixes \( x \ y :: real \)
  assumes \( y \times y > 0 \)
    and \( x1: x < 1 \)
  shows \( \exists n. x^n < y \)
proof (cases \( x > 0 \))
  case True
  with \( x1 \) have \( ix: 1 < 1/x \)
    by (simp add: field-simps)
  from real-arch-pow[OF ix, of \( 1/y \)]
  obtain \( n \) where \( 1/y < (1/x)^n \)
    by blast
  then show \(?\)thesis using \( y \times 1 > 0 \),
    by (auto simp add: field-simps)
next
  case False
  with \( y \times 1 \) show \(?\)thesis
    by (metis less-le-trans not-less power-one-right)
qed
lemma \texttt{forall-pos-mono}:
\[(\forall d\ e::\text{real}.\ d < e \Rightarrow P d \Rightarrow P e) \Rightarrow \]
\[(\forall n::\text{nat}.\ n \neq 0 \Rightarrow P \ (\text{inverse} \ (\text{real} \ n))) \Rightarrow (\exists e.\ 0 < e \Rightarrow P e)\]
\text{by (metis real-arch-inverse)}

lemma \texttt{forall-pos-mono-1}:
\[(\forall d\ e::\text{real}.\ d < e \Rightarrow P d \Rightarrow P e) \Rightarrow \]
\[(\forall n.\ P \ (\text{inverse} \ (\text{real} \ (\text{Suc} \ n)))) \Rightarrow 0 < e \Rightarrow P e\]
\text{using reals-Archimedean by blast}

lemma \texttt{Archimedean-eventually-pow}:
\texttt{fixes x::real} \texttt{assumes 1 < x} \texttt{shows \(\forall F \ n \text{ in sequentially.} \ b < x ^ n\)}
\texttt{proof –}
\texttt{\ obtaining N where \(\forall n.\ n \geq N \Rightarrow b < x ^ n\)}
\texttt{\ by (metis assms le-less order-less-trans power-strict-increasing-iff real-arch-pow)}
\texttt{\ then show \(\exists\text{thesis}\)}
\texttt{\ using eventually-sequentially by blast}
\texttt{qed}

lemma \texttt{Archimedean-eventually-pow-inverse}:
\texttt{fixes x::real} \texttt{assumes |x| < 1 \(\varepsilon > 0\)} \texttt{shows \(\forall F \ n \text{ in sequentially.} \ |x^n| < \varepsilon\)}
\texttt{proof (cases x = 0)}
\texttt{\ case True}
\texttt{\ then show \(\exists\text{thesis}\)}
\texttt{\ by (simp add: assms eventually-at-top-dense zero-power)}
\texttt{\ next}
\texttt{\ case False}
\texttt{\ then have \(\forall F \ n \text{ in sequentially.} \ \text{inverse} \ \varepsilon < \text{inverse} \ |x| ^ n\)}
\texttt{\ by (simp add: Archimedean-eventually-pow assms(1) one-less-inverse)}
\texttt{\ then show \(\exists\text{thesis}\)}
\texttt{\ by eventually-elim (metis \(\varepsilon > 0\) inverse-less-imp-less-power-abs power-inverse)}
\texttt{qed}

98.20 Floor and Ceiling Functions from the Reals to the Integers

lemma \texttt{real-of-nat-less-numeral-iff [simp]}: \(\text{real} \ n < \text{numeral} \ w \iff n < \text{numeral} \ w\)
\texttt{for n :: nat}
\texttt{by (metis of-nat-less-iff of-nat-numeral)}

lemma \texttt{numeral-less-real-of-nat-iff [simp]}: \(\text{numeral} \ w < \text{real} \ n \iff \text{numeral} \ w < n\)
\texttt{for n :: nat}
\texttt{by (metis of-nat-less-iff of-nat-numeral)}
lemma numeral-le-real-of-nat-iff [simp]: numeral n ≤ real m ↔ numeral n ≤ m
  for m :: nat
  by (metis not-le real-of-nat-less-numeral-iff)

lemma of-int-floor-cancel [simp]: of-int ⌊x⌋ = x ↔ (∃n::int. x = of_int n)
  by (metis Ints-cases of-int-floor-cancel)

lemma floor-frac [simp]: ⌊frac r⌋ = 0
  by (simp add: frac-def)

lemma frac-1 [simp]: frac 1 = 0
  by (simp add: frac-def)

lemma frac-in-Rats-iff [simp]:
  fixes r::'a::{floor-ceiling,field-char-0}
  shows frac r ∈ 'Q ↔ r ∈ 'Q
  by (metis Rats-add Rats-diff Rats-of-int diff-add-cancel frac-def)

lemma floor-eq: real_of_int n < x ⟹ x < real_of_int n + 1 ⟹ ⌊x⌋ = n
  by linarith

lemma floor-eq2: real_of_int n ≤ x ⟹ x < real_of_int n + 1 ⟹ ⌊x⌋ = n
  by (fact floor-unique)

lemma floor-eq3: real n < x ⟹ x < real (Suc n) ⟹ nat ⌊x⌋ = n
  by linarith

lemma floor-eq4: real n ≤ x ⟹ x < real (Suc n) ⟹ nat ⌊x⌋ = n
  by linarith

lemma real-of-int-floor-ge-diff-one [simp]: r − 1 ≤ real_of_int ⌊r⌋
  by linarith

lemma real-of-int-floor-gt-diff-one [simp]: r − 1 < real_of_int ⌊r⌋
  by linarith

lemma real-of-int-floor-add-one-ge [simp]: r ≤ real_of_int ⌊r⌋ + 1
  by linarith

lemma real-of-int-floor-add-one-gt [simp]: r < real_of_int ⌊r⌋ + 1
  by linarith

lemma floor-divide-real-eq-div:
  assumes 0 ≤ b
  shows ⌊a / real_of_int b⌋ = [a] div b
proof (cases $b = 0$)
  case True
  then show thesis by simp
next
  case False
  with assms have $b > 0$ by simp
  have $j = i \div b$
    if real-of-int $i \leq a a < 1 + \text{real-of-int } i$
    real-of-int $j \times \text{real-of-int } b \leq a a < \text{real-of-int } b + \text{real-of-int } j \times \text{real-of-int } b$
    for $i j :: \text{int}$
proof
    from that have $i < b + j \times b$
    by (metis le-less-trans of-int-add of-int-less-iff of-int-mult)
  moreover have $j \times b < 1 + i$
proof
    have real-of-int $(j \times b) < \text{real-of-int } i + 1$
    using $(a < 1 + \text{real-of-int } i \leq \text{real-of-int } j \times \text{real-of-int } b \leq a)$ by force
    then show $j \times b < 1 + i$ by linarith
  qed
ultimately have $(j - i \div b) \times b \leq i \mod b i \mod b < ((j - i \div b) + 1) \times b$
by (auto simp: field-simps)
  then have $(j - i \div b) \times b < 1 \times b 0 \times b < ((j - i \div b) + 1) \times b$
  using pos-mod-bound [OF $b$, of $i$] pos-mod-sign [OF $b$, of $i$]
  by linarith+
  then show thesis using $b$ unfolding mult-less-cancel-right by auto
  qed
with $b$ show thesis by (auto split: floor-split simp: field-simps)
  qed

lemma floor-one-divide-eq-div-numeral [simp]:
  $[1 / \text{numeral } b :: \text{real}] = 1 \div \text{numeral } b$
by (metis floor-divide-of-int-eq of-int-1 of-int-numeral)

lemma floor-minus-one-divide-eq-div-numeral [simp]:
  $[- (1 / \text{numeral } b :: \text{real})] = -1 \div \text{numeral } b$
by (metis (mono_tags, opaque-lifting) div-minus-right minus-divide-right
  floor-divide-of-int-eq of-int-neg-numeral of-int-1)

lemma floor-divide-eq-div-numeral [simp]:
  $[\text{numeral } a / \text{numeral } b :: \text{real}] = \text{numeral } a \div \text{numeral } b$
by (metis floor-divide-of-int-eq of-int-numeral)

lemma floor-minus-divide-eq-div-numeral [simp]:
  $[- (\text{numeral } a / \text{numeral } b :: \text{real})] = -\text{numeral } a \div \text{numeral } b$
by (metis divide-minus-left floor-divide-of-int-eq of-int-neg-numeral of-int-numeral)

lemma of-int-ceiling-cancel [simp]: of-int $[x] = x \leftrightarrow (\exists n :: \text{int}. \ x = \text{of-int } n)$
using ceiling-of-int by metis
lemma of-int-ceiling [simp]: \( a \in \mathbb{Z} \implies \text{of-int (ceiling } a \text{) } = a \)
by (metis Ints-cases of-int-ceiling-cancel)

lemma ceiling-eq: \( \text{of-int } n < x \implies x \leq \text{of-int } n + 1 \implies \lfloor x \rfloor = n + 1 \)
by (simp add: ceiling-unique)

lemma of-int-ceiling-diff-one-le [simp]: \( \text{of-int } \lceil r \rceil - 1 \leq r \)
by linarith

lemma of-int-ceiling-le-add-one [simp]: \( \text{of-int } \lceil r \rceil \leq r + 1 \)
by linarith

lemma ceiling-le: \( x \leq \text{of-int } a \implies \lceil x \rceil \leq a \)
by (simp add: ceiling-le-iff)

lemma ceiling-divide-eq-div: \( \lceil \text{of-int } a / \text{of-int } b \rceil = - (a \text{ div } b) \)
by (metis ceiling-def floor-divide-of-int-eq minus-divide-left of-int-minus)

lemma ceiling-divide-eq-div-numeral [simp]:
\( \lceil \text{numeral } a / \text{numeral } b :: \text{real} \rceil = - (\text{numeral } a \text{ div } \text{numeral } b) \)
using ceiling-divide-eq-div[of numeral a numeral b] by simp

lemma ceiling-minus-divide-eq-div-numeral [simp]:
\( - (\text{numeral } a / \text{numeral } b :: \text{real}) = - (\text{numeral } a \text{ div } \text{numeral } b) \)
using ceiling-divide-eq-div[of - numeral a numeral b] by simp

The following lemmas are remnants of the erstwhile functions natfloor and natceiling.

lemma nat-floor-neg: \( x \leq 0 \implies \text{nat } \lfloor x \rfloor = 0 \)
for \( x :: \text{real} \)
by linarith

lemma le-nat-floor: \( \text{real } x \leq a \implies x \leq \text{nat } \lfloor a \rfloor \)
by linarith

lemma le-mult-nat-floor: \( \text{nat } \lfloor a \rfloor \times \text{nat } \lfloor b \rfloor \leq \text{nat } \lfloor a \times b \rfloor \)
by (cases \( 0 \leq a \land 0 \leq b \))
(auto simp add: nat-mult-distrib[symmetric] nat-mono le-mult-floor)

lemma nat-ceiling-le-eq [simp]: \( \text{nat } \lceil x \rceil \leq a \iff x \leq \text{real } a \)
by linarith

lemma real-nat-ceiling-ge: \( x \leq \text{real } (\text{nat } \lfloor x \rfloor) \)
by linarith

lemma Rats-no-top-le: \( \exists q \in \mathbb{Q}. x \leq q \)
for \( x :: \text{real} \)
by (auto intro!: bexI[of - of-nat (\text{nat } \lfloor x \rfloor)]) linarith
lemma Rats-no-bot-less: \( \exists q \in \mathbb{Q}, q < x \) for \( x : \mathbb{R} \)
by (auto intro!: bexI[of - of-int (\lfloor x \rfloor - 1)]) linarith

\section{98.21 Exponentiation with floor}

lemma floor-power:
assumes \( x = \text{of-int} \ \lfloor x \rfloor \)
shows \( \lfloor x \rfloor ^ n = \lfloor x \rfloor ^ n \)
proof
have \( x ^ n = \text{of-int} (\lfloor x \rfloor ^ n) \)
using assms by (induct n arbitrary: \( x \)) simp-all
then show \( \text{thesis} \) by (metis floor-of-int)
qed

lemma floor-numeral-power [simp]: \( \lfloor \text{numeral} x ^ n \rfloor = \text{numeral} x ^ n \)
by (metis floor-of-int of-int-numeral of-int-power)

lemma ceiling-numeral-power [simp]: \( \lceil \text{numeral} x ^ n \rceil = \text{numeral} x ^ n \)
by (metis ceiling-of-int of-int-numeral of-int-power)

\section{98.22 Implementation of rational real numbers}

Formal constructor

definition Ratreal :: \( \mathbb{R} \to \mathbb{R} \)
where [code-abbrev, simp]: \( \text{Ratreal} = \text{real-of-rat} \)

code-datatype Ratreal

Quasi-Numerals

lemma [code-abbrev]:
real-of-rat (numeral k) = numeral k
real-of-rat (- numeral k) = - numeral k
real-of-rat (rat-of-int a) = real-of-int a
by simp-all

lemma [code-post]:
real-of-rat 0 = 0
real-of-rat 1 = 1
real-of-rat (- 1) = - 1
real-of-rat (1 / numeral k) = 1 / numeral k
real-of-rat (numeral k / numeral l) = numeral k / numeral l
real-of-rat (- (1 / numeral k)) = - (1 / numeral k)
real-of-rat (- (numeral k / numeral l)) = - (numeral k / numeral l)
by (simp-all add: of-rat-divide of-rat-minus)

Operations

lemma zero-real-code [code]: \( 0 = \text{Ratreal} \ 0 \)
by simp
lemma one-real-code [code]: 1 = Ratreal 1
  by simp

instantiation real :: equal
begin

definition HOL.equal x y ←→ x - y = 0 for x :: real

instance by standard (simp add: equal-real-def)

lemma real-equal-code [code]: HOL.equal (Ratreal x) (Ratreal y) ←→ HOL.equal x y
  by (simp add: equal-real-def equal)

lemma [code nbe]: HOL.equal x x ←→ True
  for x :: real
  by (rule equal-refl)

end

lemma real-less-eq-code [code]: Ratreal x ≤ Ratreal y ←→ x ≤ y
  by (simp add: af-rat-less-eq)

lemma real-less-code [code]: Ratreal x < Ratreal y ←→ x < y
  by (simp add: af-rat-less)

lemma real-plus-code [code]: Ratreal x + Ratreal y = Ratreal (x + y)
  by (simp add: af-rat-add)

lemma real-times-code [code]: Ratreal x * Ratreal y = Ratreal (x * y)
  by (simp add: af-rat-mult)

lemma real-uminus-code [code]: − Ratreal x = Ratreal (− x)
  by (simp add: af-rat-minus)

lemma real-minus-code [code]: Ratreal x − Ratreal y = Ratreal (x - y)
  by (simp add: af-rat-diff)

lemma real-inverse-code [code]: inverse (Ratreal x) = Ratreal (inverse x)
  by (simp add: af-rat-inverse)

lemma real-divide-code [code]: Ratreal x / Ratreal y = Ratreal (x / y)
  by (simp add: af-rat-divide)

lemma real-floor-code [code]: ⌊Ratreal x⌋ = ⌊x⌋
  by (metis Ratreal-def floor-le-iff floor-unique le-floor-iff
       of-int-floor-le of-rat-of-int-eq real-less-eq-code)

Quickcheck
context
  includes term-syntax
begin

definition
  valterm-ratreal :: rat × (unit ⇒ Code-Evaluation.term) ⇒ real × (unit ⇒ Code-Evaluation.term)
  where [code-unfold]: valterm-ratreal k = Code-Evaluation.valtermify Ratreal {·} k
end

instantiation real :: random
begin

context
  includes state-combinator-syntax
begin

definition
  Quickcheck-Random.random i = Quickcheck-Random.random i ◦→ (λr. Pair (valterm-ratreal r))

instance ..

end

end

instantiation real :: exhaustive
begin

definition
  exhaustive-real f d = Quickcheck-Exhaustive.exhaustive (λr. f (Ratreal r)) d

instance ..

end

instantiation real :: full-exhaustive
begin

definition
  full-exhaustive-real f d = Quickcheck-Exhaustive.full-exhaustive (λr. f (valterm-ratreal r)) d

instance ..

end
instantiation real :: narrowing
begin

definition
  narrowing-real = Quickcheck-Narrowing.apply (Quickcheck-Narrowing.cons Ratreal) narrowing

instance..

end

98.23 Setup for Nitpick

declaration

Nitpick-HOL.register-frac-type type-name‹real›

[(const-name‹zero-real-inst.zero-real›, const-name‹Nitpick.zero-frac›),
 (const-name‹one-real-inst.one-real›, const-name‹Nitpick.one-frac›),
 (const-name‹plus-real-inst.plus-real›, const-name‹Nitpick.plus-frac›),
 (const-name‹times-real-inst.times-real›, const-name‹Nitpick.times-frac›),
 (const-name‹uminus-real-inst.uminus-real›, const-name‹Nitpick.uminus-frac›),
 (const-name‹inverse-real-inst.inverse-real›, const-name‹Nitpick.inverse-frac›),
 (const-name‹ord-real-inst.less-real›, const-name‹Nitpick.less-frac›),
 (const-name‹ord-real-inst.less-eq-real›, const-name‹Nitpick.less-eq-frac›)]

lemmas[nitpick-unfold] = inverse-real-inst.inverse-real one-real-inst.one-real
ord-real-inst.less-real ord-real-inst.less-eq-real plus-real-inst.plus-real
times-real-inst.times-real uminus-real-inst.uminus-real
zero-real-inst.zero-real

98.24 Setup for SMT

ML-file ‹Tools/SMT/smt-real.ML›
ML-file ‹Tools/SMT/z3-real.ML›

lemma[z3-rule]:
  0 + x = x
  x + 0 = x
  0 * x = 0
  1 * x = x
  -x = -1 * x
  x + y = y + x
  for x y :: real
  by auto

lemma[smt-arith-multiplication]:
  fixes A B :: real and p n :: real
  assumes A ≤ B 0 < n p > 0
  shows (A / n) * p ≤ (B / n) * p
using assms by (auto simp: field-simps)

lemma [smt-arith-multiplication]:
  fixes A B :: real and p n :: real
  assumes A < B 0 < n p > 0
  shows \((A / n) * p < (B / n) * p\)
  using assms by (auto simp: field-simps)

lemma [smt-arith-multiplication]:
  fixes A B :: real and p n :: int
  assumes A \leq B 0 < n p > 0
  shows \((A / n) * p \leq (B / n) * p\)
  using assms by (auto simp: field-simps)

lemma [smt-arith-multiplication]:
  fixes A B :: real and p n :: int
  assumes A < B 0 < n p > 0
  shows \((A / n) * p < (B / n) * p\)
  using assms by (auto simp: field-simps)

lemmas [smt-arith-multiplication] =
  verit-le-mono-div[THEN mult-left-mono, unfolded int-distrib, of - - \langle nat (floor (- :: real))\rangle]
  disp-le-mono[THEN mult-left-mono, unfolded int-distrib, of - - \langle nat (floor (- :: real))\rangle]
  verit-le-mono-div-int[THEN mult-left-mono, unfolded int-distrib, of - - \langle floor (- :: real)\rangle]
  \langle floor (- :: real)\rangle]
  zdv-mono1[THEN mult-left-mono, unfolded int-distrib, of - - \langle floor (- :: real)\rangle]
  arg-cong[of - - \langle\lambda a :: real. a / real (n::nat) * real (p::nat)\rangle for n p :: nat, THEN sym]
  arg-cong[of - - \langle\lambda a :: real. a / real-of-int n * real-of-int p\rangle for n p :: int, THEN sym]
  arg-cong[of - - \langle\lambda a :: real. a / n * p\rangle for n p :: real, THEN sym]

lemmas [smt-arith-simplify] =
  floor-one floor-numeral div-by-1 times-divide-eq-right
  nonzero-mult-div-cancel-left division-ring-divide-zero div-0
  divide-minus-left zero-less-divide-iff

98.25 Setup for Argo

ML-file \langle Tools/Argo/argo-real.ML\rangle

end

99 Topological Spaces

theory Topological-Spaces
imports Main

begin

class open = 
  fixes open :: 'a set ⇒ bool

class topological-space = open +
  assumes open-UNIV [simp, intro]: open UNIV
  assumes open-Int [intro]: open S ⇒ open T ⇒ open (S ∩ T)
  assumes open-Union [intro]: ∀ S∈K. open S ⇒ open (⋃ K)

begin

definition closed :: 'a set ⇒ bool
  where closed S ←→ open (− S)

lemma open-empty [continuous-intros, intro, simp]: open {}
  using open-Union [of {}] by simp

lemma open-Un [continuous-intros, intro]: open S ⇒ open T ⇒ open (S ∪ T)
  using open-Union [of {S, T}] by simp

lemma open-UN [continuous-intros, intro]: ∀ x∈A. open (B x) ⇒ open (∪ x∈A. B x)
  using open-Union [of B ' A] by simp

lemma open-Inter [continuous-intros, intro]: finite S ⇒ ∀ T∈S. open T ⇒ open (∩ S)
  by (induction set: finite) auto

lemma open-INT [continuous-intros, intro]: finite A ⇒ ∀ x∈A. open (B x) ⇒ open (∩ x∈A. B x)
  using open-Inter [of B ' A] by simp

lemma openI:
  assumes \( \forall x. x \in S \implies \exists T. \text{open } T \land x \in T \land T \subseteq S \)
  shows open S

proof –
  have open (⋃ {T. open T \land T \subseteq S}) by auto
  moreover have ⋃ {T. open T \land T \subseteq S} = S by (auto dest!: assms)
  ultimately show open S by simp

qed

lemma open-subopen: open S ←→ (\( \forall x\in S. \exists T. \text{open } T \land x \in T \land T \subseteq S \))
  by (auto intro: openI)
lemma closed-empty [continuous-intros, intro, simp]: closed {}
  unfolding closed-def by simp

lemma closed-Un [continuous-intros, intro]: closed S \implies closed T \implies closed (S \cup T)
  unfolding closed-def by auto

lemma closed-UNIV [continuous-intros, intro, simp]: closed UNIV
  unfolding closed-def by simp

lemma closed-Int [continuous-intros, intro]: closed S \implies closed T \implies closed (S \cap T)
  unfolding closed-def by auto

lemma closed-INT [continuous-intros, intro]: \forall x \in A. closed (B x) \implies closed (∩ x \in A. B x)
  unfolding closed-def uminus-Inf by auto

lemma closed-Union [continuous-intros, intro]: finite S \implies \forall T \in S. closed T \implies closed (∪ S)
  by (induct set: finite) auto

lemma closed-UN [continuous-intros, intro]:
  finite A \implies \forall x \in A. closed (B x) \implies closed (∪ x \in A. B x)
  using closed-Union [of B \cdot A] by simp

lemma open-closed: open S \longleftrightarrow closed (− S)
  by (simp add: closed-def)

lemma closed-open: closed S \longleftrightarrow open (− S)
  by (rule closed-def)

lemma open-Diff [continuous-intros, intro]: open S \implies closed T \implies open (S − T)
  by (simp add: open-closed Diff-eq open-Int)

lemma closed-Diff [continuous-intros, intro]: closed S \implies open T \implies closed (S − T)
  by (simp add: open-closed Diff-eq closed-Int)

lemma open-Compl [continuous-intros, intro]: closed S \implies open (− S)
  by (simp add: closed-open)

lemma closed-Compl [continuous-intros, intro]: open S \implies closed (− S)
  by (simp add: open-closed)
lemma open-Collect-neg: closed \{x. P x\} \implies open \{x. \neg P x\}
unfolding Collect-neg-eq by (rule open-Compl)

lemma open-Collect-conj:
  assumes open \{x. P x\} open \{x. Q x\}
  shows open \{x. P x \land Q x\}
  using open-Int[OF assms] by (simp add: Int-def)

lemma open-Collect-disj:
  assumes open \{x. P x\} open \{x. Q x\}
  shows open \{x. P x \lor Q x\}
  using open-Un[OF assms] by (simp add: Un-def)

lemma open-Collect-ex: (\forall i. open \{x. P i x\}) \implies open \{x. \exists i. P i x\}
  using open-UN[of UNIV \lambda i. \{x. P i x\}] unfolding Collect-ex-eq by simp

lemma open-Collect-imp: closed \{x. P x\} \implies open \{x. Q x\} \implies open \{x. P x \rightarrow Q x\}
  unfolding imp-conv-disj by (intro open-Collect-disj open-Collect-neg)

lemma open-Collect-const: open \{x. P\}
  by (cases P) auto

lemma closed-Collect-neg: open \{x. P x\} \implies closed \{x. \neg P x\}
  unfolding Collect-neg-eq by (rule closed-Compl)

lemma closed-Collect-conj:
  assumes closed \{x. P x\} closed \{x. Q x\}
  shows closed \{x. P x \land Q x\}
  using closed-Int[OF assms] by (simp add: Int-def)

lemma closed-Collect-disj:
  assumes closed \{x. P x\} closed \{x. Q x\}
  shows closed \{x. P x \lor Q x\}
  using closed-Un[OF assms] by (simp add: Un-def)

lemma closed-Collect-all: (\forall i. closed \{x. P i x\}) \implies closed \{x. \forall i. P i x\}
  using closed-INT[of UNIV \lambda i. \{x. P i x\}] by (simp add: Collect-all-eq)

lemma closed-Collect-imp: open \{x. P x\} \implies closed \{x. Q x\} \implies closed \{x. P x \rightarrow Q x\}
  unfolding imp-conv-disj by (intro closed-Collect-disj closed-Collect-neg)

lemma closed-Collect-const: closed \{x. P\}
  by (cases P) auto

end
99.2 Hausdorff and other separation properties

**class** t0-space = topological-space +
\hspace{1em}**assumes** t0-space: \( x \neq y \implies \exists U. \text{open } U \land \neg (x \in U \iff y \in U) \)

**class** t1-space = topological-space +
\hspace{1em}**assumes** t1-space: \( x \neq y \implies \exists U. \text{open } U \land x \in U \land y \notin U \)

**instance** t1-space ⊆ t0-space
\hspace{1em}**by** standard (fast dest: t1-space)

**context** t1-space begin

**lemma** separation-t1: \( x \neq y \iff (\exists U. \text{open } U \land x \in U \land y \notin U) \)
\hspace{1em}**using** t1-space[of \( x \ y \)]\hspace{1em}**by** blast

**lemma** closed-singleton [iff]: closed \{a\}
**proof**
\hspace{1em}let \( ?T = \bigcup \{S. \text{open } S \land a \notin S\} \)
\hspace{1em}**have** open ?T\hspace{1em}**by** (simp add: open-Union)
\hspace{1em}also have \( ?T = - \{a\} \)
\hspace{1em}**by** (auto simp add: set-eq-iff separation-t1)
\hspace{1em}finally show closed \{a\}\hspace{1em}**by** (simp only: closed-def)
**qed**

**lemma** closed-insert [continuous-intros, simp]:
\hspace{1em}**assumes** closed S
\hspace{1em}**shows** closed (insert a S)
**proof**
\hspace{1em}**from** closed-singleton **assms** **have closed (\{a\} ∪ S)
\hspace{1em}**by** (rule closed-Un)
\hspace{1em}then show closed (insert a S)
\hspace{1em}**by** simp
**qed**

**lemma** finite-imp-closed: finite S \implies closed S
\hspace{1em}**by** (induct pred: finite) simp-all

**end**

T2 spaces are also known as Hausdorff spaces.

**class** t2-space = topological-space +
\hspace{1em}**assumes** hausdorff: \( x \neq y \implies \exists U \ V. \text{open } U \land \text{open } V \land x \in U \land y \in V \land U \cap V = \{\} \)

**instance** t2-space ⊆ t1-space
\hspace{1em}**by** standard (fast dest: hausdorff)
lemma (in t2-space) separation-t2: \( x \neq y \iff (\exists U \; \text{open} \; U \land \text{open} \; V \land x \in U \land y \in V \land U \cap V = \{\}) \)
using hausdorff [of x y] by blast

lemma (in t0-space) separation-t0: \( x \neq y \iff (\exists U \; \text{open} \; U \land \neg (x \in U \iff y \in U)) \)
using t0-space [of x y] by blast

A classical separation axiom for topological space, the T3 axiom – also called regularity: if a point is not in a closed set, then there are open sets separating them.

class t3-space = t2-space +
  assumes t3-space: closed \( S \implies y \notin S \implies \exists U \; \text{open} \; U \land \text{open} \; V \land y \in U \land S \subseteq V \land U \cap V = \{\} \)

A classical separation axiom for topological space, the T4 axiom – also called normality: if two closed sets are disjoint, then there are open sets separating them.

class t4-space = t2-space +
  assumes t4-space: closed \( S \implies \text{closed} \; T \implies S \cap T = \{\} \implies \exists U \; \text{open} \; U \land \text{open} \; V \land S \subseteq U \land T \subseteq V \land U \cap V = \{\} \)

T4 is stronger than T3, and weaker than metric.

instance t4-space \( \subseteq \) t3-space
proof
fix \( S \) and \( y::a \) assume closed \( S \) \( y \notin S \)
then show \( \exists U \; \text{open} \; U \land \text{open} \; V \land y \in U \land S \subseteq V \land U \cap V = \{\} \)
using t4-space[of \{y\} S] by auto
qed

A perfect space is a topological space with no isolated points.

class perfect-space = topological-space +
  assumes not-open-singleton: \( \neg \text{open} \; \{x\} \)

lemma (in perfect-space) UNIV-not-singleton: \( \text{UNIV} \neq \{x\} \)
for \( x::a \)
by (metis (no-types) open-UNIV not-open-singleton)

### 99.3 Generators for topologies

inductive generate-topology :: 'a set set \( \Rightarrow \) 'a set \( \Rightarrow \) bool for \( S :: \) 'a set set
where
UNIV: generate-topology \( S \) \( \text{UNIV} \)
| Int: generate-topology \( S \) (a \( \cap \) b) if generate-topology \( S \) a and generate-topology \( S \) b
| UN: generate-topology \( S \) (\( \bigcup \) K) if (\( \forall k. k \in K \implies \text{generate-topology} \; S \; k \))
| Basis: generate-topology \( S \) s if s \( \in S \)
THEORY “Topological-Spaces”

hide-fact (open) UNIV Int UN Basis

lemma generate-topology-Union:
  (∀k. k ∈ I ⟹ generate-topology S (K k)) ⟹ generate-topology S (⋃k∈I. K k)
  using generate-topology.UN [of K · I] by auto

lemma topological-space-generate-topology: class.topological-space (generate-topology S)
  by standard (auto intro: generate-topology.intros)

99.4 Order topologies

class order-topology = order + open +
  assumes open-generated-order: open = generate-topology (range (λa. {..< a}) ∪ range (λa. {a <..}))
begin

subclass topological-space
  unfolding open-generated-order
  by (rule topological-space-generate-topology)

lemma open-greaterThan [continuous-intros, simp]: open {..< a}
  unfolding open-generated-order by (auto intro: generate-topology.Basis)

lemma open-lessThan [continuous-intros, simp]: open {..< a}
  unfolding open-generated-order by (auto intro: generate-topology.Basis)

lemma open-greaterThanLessThan [continuous-intros, simp]: open {..<..< b}
  unfolding greaterThanLessThan-eq by (simp add: open-Int)

end

class linorder-topology = linorder + order-topology

lemma closed-atMost [continuous-intros, simp]: closed {..a}
  for a :: 'a::linorder-topology
  by (simp add: closed-open)

lemma closed-atLeast [continuous-intros, simp]: closed {..a}
  for a :: 'a::linorder-topology
  by (simp add: closed-open)

lemma closed-atLeastAtMost [continuous-intros, simp]: closed {a..b}
  for a b :: 'a::linorder-topology
  proof
    have {a .. b} = {a ..} ∩ {.. b}
      by auto
    then show ?thesis
      by (simp add: closed-Int)
lemma (in order) less-separate:
assumes $x < y$
shows $\exists a \ b. \ x \in \{..< a\} \land y \in \{b <..\} \land \{..< a\} \cap \{b <..\} = \{\}$
proof (cases $\exists z. \ x < z \land z < y$)
case True
then obtain $z$ where $x < z \land z < y$ ..
then have $x \in \{..< z\} \land y \in \{z <..\} \land \{..< z\} \cap \{z <..\} = \{\}$
by auto
then show ?thesis by blast
next
case False
with $\langle x < y \rangle$ have $x \in \{..< y\}, \ y \in \{x <..\} \land \{..< y\} = \{\}$
by auto
then show ?thesis by blast
qed

instance linorder-topology $\subseteq$ t2-space
proof
fix $x \ y :: 'a$
show $x \neq y \Rightarrow \exists U \ V, \ \text{open } U \land \text{open } V \land x \in U \land y \in V \land U \cap V = \{\}$
using less-separate [of $x \ y$] less-separate [of $y \ x$]
by (elim neqE; metis open-lessThan open-greaterThan Int-commute)
qed

lemma (in linorder-topology) open-right:
assumes open $S \ x \in S$
and $gt-ex: \ x < y$
shows $\exists b > x. \ \{x ..< b\} \subseteq S$
using assms unfolding open-generated-order
proof induct
case UNIV
then show ?case by blast
next
case (Int $A \ B$)
then obtain $a \ b$ where $a > x \ \{x ..< a\} \subseteq A \ \ b > x \ \{x ..< b\} \subseteq B$
by auto
then show ?case
by (auto intro!: exI[of - min $a \ b$])
next
case UN
then show ?case by blast
next
case Basis
then show ?case
by (fastforce intro: exI[of - $y$] $gt-ex$)
qed
lemma (in linorder-topology) open-left:
  assumes open S x ∈ S
  and lt-ex: y < x
  shows ∃ b < x. {b <.. x} ⊆ S
using assms unfolding open-generated-order
proof induction
  case UNIV
  then show ?case by blast
next
  case (Int A B)
  then obtain a b where a < x {a <.. x} ⊆ A  b < x {b <.. x} ⊆ B
    by auto
  then show ?case by (auto intro: exI[of - max a b])
next
  case UN
  then show ?case by blast
next
  case Basis
  then show ?case by (fastforce intro: exI[of - y] lt-ex)
qed

99.5  Setup some topologies

99.5.1 Boolean is an order topology

class discrete-topology = topological-space +
  assumes open-discrete: ∀A. open A

instance discrete-topology < t2-space
proof
  fix x y :: 'a
  assume x ≠ y
  then show ∃ U V. open U ∧ open V ∧ x ∈ U ∧ y ∈ V ∧ U ∩ V = {}
    by (intro exI[of - {}]) (auto intro: open-discrete)
qed

instantation bool :: linorder-topology
begin

definition open-bool :: bool set ⇒ bool
  where open-bool = generate-topology (range (λa. {..< a}) ∪ range (λa. {a <..})))

instance
  by standard (rule open-bool-def)

end

instance bool :: discrete-topology
proof
  fix A :: bool set
  have *: {False <..} = {True} {..< True} = {False}
    by auto
  have A = UNIV ∨ A = {} ∨ A = {False <..} ∨ A = {..< True}
    using subset-UNIV[of A] unfolding UNIV-bool * by blast
  then show open A
    by auto
qed

instantiation nat :: linorder-topology
begin

definition open-nat :: nat set ⇒ bool
  where open-nat = generate-topology (range (λa. {..< a}) ∪ range (λa. {a <..}))

instance
  by standard (rule open-nat-def)
end

instance nat :: discrete-topology
proof
  fix A :: nat set
  have open {n} for n :: nat
    proof (cases n)
    case 0
      moreover have {0} = {..<1::nat}
        by auto
      ultimately show ?thesis
        by auto
    next
    case (Suc n')
      then have {n} = {..<Suc n} ∩ {n' <..}
        by auto
      with Suc show ?thesis
        by (auto intro: open-lessThan open-greaterThan)
    qed
  then have open (⋃ a∈A. {a})
    by (intro open-UN) auto
  then show open A
    by simp
qed

instantiation int :: linorder-topology
begin

definition open-int :: int set ⇒ bool
  where open-int = generate-topology (range (λa. {..< a}) ∪ range (λa. {a <..}))
instance
  by standard (rule open-int-def)

end

instance int :: discrete-topology

proof
  fix A :: int set
  have {..<i + 1} ∩ {i - 1<..} = {i} for i :: int
    by auto
  then have open {i} for i :: int
    using open-Int[of open-lessThan[of i + 1] open-greaterThan[of i - 1]] by auto
  then have open (⋃a ∈ A. {a})
    by (intro open-UN) auto
  then show open A
    by simp

qed

99.5.2 Topological filters

definition (in topological-space) nhds :: 'a ⇒ 'a filter
  where nhds a = (INF S∈{S. open S ∧ a ∈ S}. principal S)

definition (in topological-space) at-within :: 'a ⇒ 'a set ⇒ 'a filter
  (at (-)/ within (-)[1000, 60] 60)
  where at a within s = inf (nhds a) (principal (s - {a}))

abbreviation (in topological-space) at :: 'a ⇒ 'a filter (at)
  where at x ≡ at x within (CONST UNIV)

abbreviation (in order-topology) at-right :: 'a ⇒ 'a filter
  where at-right x ≡ at x within {x <..}

abbreviation (in order-topology) at-left :: 'a ⇒ 'a filter
  where at-left x ≡ at x within {..< x}

lemma (in topological-space) nhds-generated-topology:
  open = generate-topology T ⇒ nhds x = (INF S∈{S∈T. x ∈ S}. principal S)

unfolding nhds-def

proof (safe intro: antisym INF-greatest)
  fix S
  assume generate-topology T S x ∈ S
  then show (INF S∈{S∈T. x ∈ S}. principal S) ≤ principal S
    by induct
      (auto intro: INF-lower order-trans simp: inf-principal[symmetric] simp del:
        inf-principal)

qed (auto intro!: INF-lower intro: generate-topology.intros)
lemma (in topological-space) eventually-nhds:
  eventually P (nhds a) ←→ (∃ S. open S ∧ a ∈ S ∧ (∀ x∈S. P x))
unfolding nhds-def by (subt eventually-INF-base) (auto simp: eventually-principal)

lemma eventually-eventually:
  eventually (λ y. eventually P (nhds y)) (nhds x) = eventually P (nhds x)
by (auto simp: eventually-nhds)

lemma (in topological-space) eventually-nhds-in-open:
  open s ⇒ x ∈ s ⇒ eventually (λ y ∈ s) (nhds x)
by (subst eventually-nhds) blast

lemma (in topological-space) eventually-nhds-x-imp-x:
  eventually P (nhds x) =⇒ P x
by (subst (asm) eventually-nhds) blast

lemma nhds-neq-bot [simp]: nhds a ≠ bot
by (simp add: trivial-limit-def eventually-nhds)

lemma (in t1-space) t1-space-nhds:
  x ≠ y =⇒ (∀ F x in nhds x. x ≠ y)
by (drule t1-space) (auto simp: eventually-nhds)

lemma (in topological-space) nhds-discrete-open:
  open {x} =⇒ nhds x = principal {x}
by (auto simp: nhds-def intro: antisym INF-greatest INF-lower2[of {x}])

lemma (in discrete-topology) nhds-discrete: nhds x = principal {x}
by (simp add: nhds-discrete-open open-discrete)

lemma (in discrete-topology) at-discrete:
  at x within S = bot
unfolding at-within-def nhds-discrete by simp

lemma (in discrete-topology) tendsto-discrete:
  filterlim (f :: 'b ⇒ 'a) (nhds y) F ←→ eventually (λ x. f x = y) F
by (auto simp: nhds-discrete filterlim-principal)

lemma (in topological-space) at-within-eq:
  at x within s = (INF S∈{S. open S ∧ x ∈ S}. principal (S ∩ s − {x})))
unfolding nhds-def at-within-def
by (subt INF-inf-const2[symmetric]) (auto simp: Diff-Int-distrib)

lemma (in topological-space) eventually-at-filter:
  eventually P (at a within s) ←→ eventually (λ x. x ≠ a =⇒ x ∈ s → P x) (nhds a)
by (simp add: at-within-def eventually-inf-principal imp-conjL[symmetric] conj-commute)

lemma (in topological-space) at-le: s ⊆ t =⇒ at x within s ≤ at x within t
unfolding at-within-def by (intro inf-monot) auto
lemma (in topological-space) eventually-at-topological:
  eventually P (at a within s) ↔ (∃ S. open S ∧ a ∈ S ∧ (∀ x ∈ S. x ≠ a → x ∈ s → P x))
  by (simp add: eventually-nhds eventually-at-filter)

lemma eventually-nhds-conv-at:
  eventually P (nhds x) ↔ eventually P (at x) ∧ P x
  unfolding eventually-at-topological eventually-nhds by fast

lemma eventually-at-in-open:
  assumes open A x ∈ A
  shows eventually (λ y. y ∈ A − {x}) (at x)
  using assms eventually-at-topological by blast

lemma eventually-at-in-open':
  assumes open A x ∈ A
  shows eventually (λ y. y ∈ A) (at x)
  using assms eventually-at-topological by blast

lemma (in topological-space) at-within-open: a ∈ S ⊆ open S ⊆ at a within S = at a
  unfolding filter-eq-iff eventually-at-topological by (metis open-Int Int-iff UNIV-I)

lemma (in topological-space) at-within-open-NO-MATCH:
  a ∈ s ⊆ open s ⊆ NO-MATCH UNIV s ⊆ at a within s = at a
  by simp only: at-within-open

lemma (in topological-space) at-within-open-subset:
  a ∈ S = open S ⊆ S ⊆ T ⊆ at a within T = at a
  by (metis at-le at-within-open dual-order.antisym subset-UNIV)

lemma (in topological-space) at-within-nhd:
  assumes x ∈ S open S T ∩ S − {x} = U ∩ S − {x}
  shows at x within T = at x within U
  unfolding filter-eq-iff eventually-at-filter
  proof (intro allI eventually-subst)
    have eventually (λ x. x ∈ S) (nhds x)
      using x ∈ S open S by (auto simp: eventually-nhds)
    then show ∀ P n in nhds x. (n ≠ x → n ∈ T → P n) = (n ≠ x → n ∈ U → P n) for P
      by eventually-elim (insert ‹ T ∩ S − {x} = U ∩ S − {x} ‚, blast)
  qed

lemma (in topological-space) at-within-empty simp: at a within {} = bot
  unfolding at-within-def by simp

lemma (in topological-space) at-within-union:
  at x within (S ∪ T) = sup (at x within S) (at x within T)
unfolding filter-eq-iff eventually-sup eventually-at-filter
by (auto elim!: eventually-rev-mp)

lemma (in topological-space) at-eq-bot-iff: at a = bot ↔ open {a}
unfolding trivial-limit-def eventually-at-topological
by (metis UNIV-I empty-iff is-singletonE is-singletonI' singleton-iff)

lemma (in t1-space) eventually-neq-at-within:
eventually (λw. w ≠ x) (at z within A)
by (smt (verit, ccfv-threshold) eventually-True eventually-at-topological separa-
tion-t1)

lemma (in perfect-space) at-neq-bot [simp]: at a ≠ bot
by (simp add: at-eq-bot-iff not-open-singleton)

lemma (in order-topology) nhds-order:
nhds x = inf (INF a∈{x <..}. principal {..< a}) (INF a∈{..< x}. principal {a <..})
proof (−)
    have 1: {S ∈ range lessThan ∪ range greaterThan. x ∈ S} =
        (λa. {..< a}) ' {..<} ∪ (λa. {a <..}) ' {..< x}
    by auto
    show ?thesis
    by (simp only: nhds-generated-topology[OF open-generated-order] INF-union 1
INF-image comp-def)
qed

lemma (in topological-space) filterlim-at-within-If:
assumes filterlim f G (at x within (A ∩ {x. P x}))
and filterlim g G (at x within (A ∩ {x. ¬P x}))
shows filterlim (λx. if P x then f x else g x) G (at x within A)
proof (rule filterlim-If)
    note assms(1)
    also have at x within (A ∩ {x. P x}) = inf (nhds x) (principal (A ∩ Collect P
− {x}))
    by (simp add: at-within-def)
    also have A ∩ Collect P − {x} = (A − {x}) ∩ Collect P
    by blast
    also have inf (nhds x) (principal ...) = inf (at x within A) (principal (Collect
P))
    by (simp add: at-within-def inf-assoc)
    finally show filterlim f G (inf (at x within A) (principal (Collect P))) .
next
    note assms(2)
    also have at x within (A ∩ {x. ¬ P x}) = inf (nhds x) (principal (A ∩ {x. ¬
P x} − {x}))
    by (simp add: at-within-def)
    also have A ∩ {x. ¬ P x} − {x} = (A − {x}) ∩ {x. ¬ P x}
    by blast
also have \( \inf (\text{nhds } x) (\text{principal } \ldots) = \inf (\text{at } x \text{ within } A) (\text{principal } \{x. \sim P x\}) \)

by (simp add: at-within-def inf-assoc)

finally show \( \text{filterlim } G (\inf (\text{at } x \text{ within } A) (\text{principal } \{x. \sim P x\})) \).

qed

lemma (in topological-space) filterlim-at-If:
assumes \( \text{filterlim } f G (\text{at } x \text{ within } \{x. P x\}) \)
and \( \text{filterlim } g G (\text{at } x \text{ within } \{x. \sim P x\}) \)
shows \( \text{filterlim } (\lambda x. \text{if } P x \text{ then } f x \text{ else } g x) G (\text{at } x) \)
using assms by (intro filterlim-at-within-If simp-all)

lemma (in linorder-topology) at-within-order:
assumes \( UNIV \neq \{x\} \)
shows \( \text{at } x \text{ within } s = \inf (\text{INF } a \in \{x <..\}. \text{principal } \{\ldots a \cap s - \{x\}\})) \)
(\( \text{INF } a \in \{..< x\}. \text{principal } \{a <.. \cap s - \{x\}\}) \)
proof (cases \( \{x <..\} = \{\} \{..< x\} = \{\} \) rule: case-split [case-product case-split])

case True-True
have \( UNIV = \{..< x\} \cup \{x\} \cup \{x <..\} \)
by auto
with assms True-True show ?thesis
by auto
qed (auto simp del: inf-principal simp: at-within-def nhds-order Int-Diff
inf-principal[ symmetric] INF-inf-const2 inf-sup-aci[where \( a = a \text{ filter}]]\)

lemma (in linorder-topology) at-left-eq:
\( y < x \implies \text{at-left } x = (\text{INF } a \in \{..< x\}. \text{principal } \{a <..< x\}) \)
by (subst at-within-order)
(auto simp: greaterThan-Int-greaterThan greaterThanLessThan-eq[symmetric] min.absorb2 INF-constant
intro: INF-lower2 inf-absorb2)

lemma (in linorder-topology) eventually-at-left:
\( y < x \implies \text{eventually } P (\text{at-left } x) \leftrightarrow (\exists b < x. \forall y > b. y < x \longrightarrow P y) \)
unfolding at-left-ev
by (subst eventually-INF-base) (auto simp: eventually-principal Ball-def)

lemma (in linorder-topology) at-right-eq:
\( x < y \implies \text{at-right } x = (\text{INF } a \in \{x <..\}. \text{principal } \{x <.. a\}) \)
by (subst at-within-order)
(auto simp: lessThan-Int-lessThan greaterThanLessThan-eq[symmetric] max.absorb2 INF-constant Int-commute
intro: INF-lower2 inf-absorb1)

lemma (in linorder-topology) eventually-at-right:
\( x < y \implies \text{eventually } P (\text{at-right } x) \leftrightarrow (\exists b > x. \forall y > x. y < b \longrightarrow P y) \)
unfolding at-right-ev
by (subst eventually-INF-base) (auto simp: eventually-principal Ball-def)
lemma eventually-at-right-less: \( \forall F \ y \ in \ at-right \ (x::'a::\{linorder-topology, no-top\}). \ x < y \)
using \( gt-ex[of \ x] \) \( \)eventually-at-right\( [of \ x] \) \by \( \)auto

lemma trivial-limit-at-right-top: \( \)at-right \( \)\( (top::::\{order-top,linorder-topology\}) = \)
bot \by \( \)auto \( \)simp: \( \)filter-eq-iff \( \)eventually-at-topological

lemma trivial-limit-at-left-bot: \( \)at-left \( \)\( (bot::::\{order-bot,linorder-topology\}) = \)
bot \by \( \)auto \( \)simp: \( \)filter-eq-iff \( \)eventually-at-topological

lemma trivial-limit-at-left-real \( \)simp: \( \)\neg trivial-limit \( \)\( (at-left x) \)
for \( \)\( x::'a::\{no-bot,dense-order,linorder-topology\} \)
using \( \)lt-ex \( \)\( of \ x \)
by \( \)safe \( \)auto \( \)simp add: \( \)trivial-limit-def \( \)eventually-at-left \( \)dest: \( \)dense

lemma trivial-limit-at-right-real \( \)simp: \( \)\neg trivial-limit \( \)\( (at-right x) \)
for \( \)\( x::'a::\{no-top,dense-order,linorder-topology\} \)
using \( \)gt-ex \( \)\( of \ x \)
by \( \)safe \( \)auto \( \)simp add: \( \)trivial-limit-def \( \)eventually-at-right \( \)dest: \( \)dense

lemma \( \)in \( \)linorder-topology \( \)at-eq-sup-left-right: \( \)at x = sup \( \)\( (at-left x) \) \( \)\( (at-right x) \)
by \( \)auto \( \)simp: \( \)eventually-at-filter \( \)filter-eq-iff \( \)eventually-sup \( \)elim: \( \)eventually-elim2 \( \)eventually-mono

lemma \( \)in \( \)linorder-topology \( \)eventually-at-split:
\( \)assumes \( \)\( \wedge x. \ x \in \{a<..<b\} \rightarrow P \ a < b \)
\( \)shows \( \)eventually \( \)\( P \) \( \)\( (at-left b) \)
using \( \)assms \( \)unfolding \( \)eventually-at-topological \( \)by \( \)intro \( \)exI \( \)\( of - \{a<..\}\) auto

lemma \( \)in \( \)order-topology \( \)eventually-at-leftI:
\( \)assumes \( \)\( \forall x. \ x \in \{a<..<b\} \rightarrow P \ a < b \)
\( \)shows \( \)eventually \( \)\( P \) \( \)\( (at-right a) \)
using \( \)assms \( \)unfolding \( \)eventually-at-topological \( \)by \( \)intro \( \)exI \( \)\( of - \{..<b\}\) auto

lemma eventually-filtercomap-nhds:
\( \)eventually \( \)\( P \) \( \)\( (filtercomap \ f \ (nhds \ x)) \rightarrow (\exists S. \ open \ S \wedge x \in S \wedge (\forall x. \ f x \in S \rightarrow P x)) \)
\( \)unfolding \( \)eventually-filtercomap \( \)eventually-nhds \( \)by \( \)auto

lemma eventually-filtercomap-at-topological:
\( \)eventually \( \)\( P \) \( \)\( (filtercomap \ f \ (at A \ within \ B)) \rightarrow (\exists S. \ open \ S \wedge A \in S \wedge (\forall x. \ f x \in S \cap B \rightarrow \{A\} \rightarrow P x)) \) \( \)is \( \)lhs = \( \)rhs
\( \)unfolding \( \)at-within-def \( \)filtercomap-inf \( \)eventually-inf-principal \( \)filtercomap-principal
lemma eventually-at-right-field:
  eventually P (at-right x) \iff (\exists b > x. \forall y > b. y < b \rightarrow P y)
for x :: 'a::{linordered-field, linorder-topology}
using linordered-field-no-ub [rule-format, of x]
by (auto simp: eventually-at-right)

todo
Global-Theory.add-thms-dynamic (binding {tendsto-eq-intros},
  fn context =>
    Named-Theorems.get (Context.proof-of context) named-theorems {tendsto-intros}
  |> map-filter (try (fn thm => @{(thm tendsto-eq-rhs) OF [thm]})))

context topological-space begin

lemma tendsto-def:
  \[(f \to l) F \iff (\forall S. \text{open } S \to l \in S \to \text{eventually } (\lambda x. f x \in S) F)\]
unfolding nhds-def filterlim-INF filterlim-principal by auto

lemma tendsto-cong: \[(f \to c) F \iff (g \to c) F\]
  if \[(\lambda x. f x = g x) F\]
by (rule filterlim-cong [OF refl refl that])

lemma tendsto-mono: \[F \leq F' \implies (f \to l) F' \implies (f \to l) F\]
unfolding tendsto-def le-filter-def by fast

lemma tendsto-ident-at [tendsto-intros, simp, intro]: \[(\lambda x. x) \to a (at a within s)\]
by (auto simp: tendsto-def eventually-at-topological)

lemma tendsto-const [tendsto-intros, simp, intro]: \[(\lambda x. k) \to k (F)\]
by (simp add: tendsto-def)

lemma filterlim-at:
\[(\lim x F. f x :> at b within s) \iff \text{eventually } (\lambda x. f x \in s \land f x \neq b) F \land (f \to b) F\]
by (simp add: at-within-def filterlim-inf filterlim-principal conj-commute)

lemma (in −)
  assumes filterlim f (nhds L) F
shows tendsto-imp-filterlim-at-right:
  eventually \((\lambda x. f x > L) F \implies \text{filterlim } f (at-right L) F\)
  and tendsto-imp-filterlim-at-left:
  eventually \((\lambda x. f x < L) F \implies \text{filterlim } f (at-left L) F\)
using assms by (auto simp: filterlim-at elim: eventually-mono)

lemma filterlim-at-withinI:
  assumes filterlim f (nhds c) F
  assumes eventually \((\lambda x. f x \in A - \{c\}) F\)
shows \(\text{filterlim } f (at c within A) F\)
using assms by (simp add: filterlim-at)

lemma filterlim-atI:
  assumes filterlim f (nhds c) F
  assumes eventually \((\lambda x. f x \neq c) F\)
shows \(\text{filterlim } f (at c) F\)
using assms by (intro filterlim-at-withinI) simp-all

lemma topological-tendstoI:
  \((\forall S. \text{open } S \implies l \in S \implies \text{eventually } (\lambda x. f x \in S) \ F) \implies (f \to l) \ F\)
  by (auto simp: tendsto-def)

lemma topological-tendstoD:
  \((f \to l) \ F \implies \text{open } S \implies l \in S \implies \text{eventually } (\lambda x. f x \in S) \ F\)
  by (auto simp: tendsto-def)

lemma tendsto-bot [simp]: \((f \to a) \bot\)
  by (simp add: tendsto-def)

lemma tendsto-eventually:
  \(\text{eventually } (\lambda x. f x = l) \ F \implies (\lambda x. f x) \to l \ F\)
  by (rule topological-tendstoI) (auto elim: eventually-mono)

lemma tendsto-principal-singleton [simp]:
  shows \((f \to f x) \text{principal } \{x\}\)
  unfolding tendsto-def eventually-principal by simp

end

lemma (in topological-space) filterlim-within-subset:
  \(\text{filterlim } f l \ (at x within S) \implies T \subseteq S \implies \text{filterlim } f l \ (at x within T)\)
  by (blast intro: filterlim-mono at-le)

lemmas tendsto-within-subset = filterlim-within-subset

lemma (in order-topology) order-tendsto-iff:
  \((f \to x) \ F \iff (\forall l<x. \text{eventually } (\lambda x. l < f x) \ F) \land (\forall u>x. \text{eventually } (\lambda x. f x < u) \ F)\)
  by (auto simp: nhds-order filterlim-inf filterlim-INF filterlim-principal)

lemma (in order-topology) order-tendstoI:
  \((\forall a. a < y \implies \text{eventually } (\lambda x. a < f x) \ F) \implies (\forall a. y < a \implies \text{eventually } (\lambda x. f x < a) \ F) \implies (f \to y) \ F\)
  by (auto simp: order-tendsto-iff)

lemma (in order-topology) order-tendstoD:
  assumes \((f \to y) \ F\)
  shows \(a < y \implies \text{eventually } (\lambda x. a < f x) \ F\)
  and \(y < a \implies \text{eventually } (\lambda x. f x < a) \ F\)
  using assms by (auto simp: order-tendsto-iff)

lemma (in linorder-topology) tendsto-max [tendsto-intros]:
  assumes \(X: (X \to x) \ F\)
  and \(Y: (Y \to y) \ F\)
shows $((\lambda x. \max (X \cdot x) (Y \cdot x)) \rightarrow \max x y)$ net

proof (rule order-tendstoI)
  fix a
  assume $a < \max x y$
  then show eventually $(\lambda x. a < \max (X \cdot x) (Y \cdot x))$ net
    using order-tendstoD(1)[OF X, of a] order-tendstoD(1)[OF Y, of a]
    by (auto simp: less-max-iff-disj elim: eventually-mono)
next
  fix a
  assume $\max x y < a$
  then show eventually $(\lambda x. \max (X \cdot x) (Y \cdot x) < a)$ net
    using order-tendstoD(2)[OF X, of a] order-tendstoD(2)[OF Y, of a]
    by (auto simp: eventually-conj-iff)
qed

lemma (in linorder-topology) tendsto-min[tendsto-intros]:
  assumes $X: (X \rightarrow x)$ net
  and $Y: (Y \rightarrow y)$ net
  shows $((\lambda x. \min (X \cdot x) (Y \cdot x)) \rightarrow \min x y)$ net

proof (rule order-tendstoI)
  fix a
  assume $a < \min x y$
  then show eventually $(\lambda x. a < \min (X \cdot x) (Y \cdot x))$ net
    using order-tendstoD(1)[OF X, of a] order-tendstoD(1)[OF Y, of a]
    by (auto simp: eventually-conj-iff)
next
  fix a
  assume $\min x y < a$
  then show eventually $(\lambda x. \min (X \cdot x) (Y \cdot x) < a)$ net
    using order-tendstoD(2)[OF X, of a] order-tendstoD(2)[OF Y, of a]
    by (auto simp: min-less-iff-disj elim: eventually-mono)
qed

lemma (in order-topology) at-within-Icc-at: $a < b$
  assumes $a < b$
  shows at-within-Icc-at-right: at a within $\{a..b\}$ = at-right a
  and at-within-Icc-at-left: at b within $\{a..b\}$ = at-left b
  using order-tendstoD(2)[OF tendsto-ident-at assms, of $\{a..\}$]
  using order-tendstoD(1)[OF tendsto-ident-at assms, of $..<b\}$]
  by (auto intro: order-class.order-antisym filter-leI
    simp: eventually-at-filter less-le
    elim: eventually-elim2)

lemma (in order-topology) at-within-Icc-at: $a < x \implies x < b \implies at x within \{a..b\} = at x$
  by (rule at-within-open-subset[where $S=\{a..<b\}$]) auto

lemma (in t2-space) tendsto-unique:
  assumes $F \neq bot$
and \((f \rightarrow a)\) \(F\)
and \((f \rightarrow b)\) \(F\)
shows \(a = b\)

proof (rule ccontr)
assume \(a \neq b\)
obtain \(U \ V\) where open \(U\) open \(V\) \(a \in U\) \(b \in V\) \(U \cap V = {}\)
using hausdorff \([OF \: a \neq b]\) by fast
have eventually \((\lambda x. \ f \ x \in U)\) \(F\)
using \((f \rightarrow a)\) \(F\) \(\langle open \ U \rangle \: \langle a \in U \rangle\) by (rule topological-tendstoD)
moreover
have eventually \((\lambda x. \ f \ x \in V)\) \(F\)
using \((f \rightarrow b)\) \(F\) \(\langle open \ V \rangle \: \langle b \in V \rangle\) by (rule topological-tendstoD)
ultimately
have eventually \((\lambda x. \ False)\) \(F\)
proof eventually-elim
  case (elim \(x\))
  then have \(f \ x \in U \cap V\) by simp
  with \(U \cap V = {}\) show \(?case\) by simp
qed
with \(\neg\) trivial-limit \(F\) show False
by (simp add: trivial-limit-def)


lemma (in t2-space) tendsto-const-iff:
fixes \(a\) \(b\) :: 'a
assumes \(\neg\) trivial-limit \(F\)
shows \((\lambda x. \ a) \rightarrow b\) \(F\) \(\iff\) \(a = b\)
by (auto intro!: tendsto-unique [OF assms tendsto-const])

lemma (in t2-space) tendsto-unique':
assumes \(F \neq bot\)
shows \(\exists_{\leq 1} l. \ (f \rightarrow l)\) \(F\)
using Uniq-def assms local.tendsto-unique by fastforce

lemma Lim-in-closed-set:
assumes closed \(S\) eventually \((\lambda x. \ f(x) \in S)\) \(F\) \(\neq bot\) \((f \rightarrow l)\) \(F\)
shows \(l \in S\)
proof (rule ccontr)
assume \(l \notin S\)
with \(\langle closed \ S \rangle\) have \(\langle open \ (\neg \ S) \rangle \: l \in \neg S\)
  by (simp-all add: open-Compl)
with assms(4) have eventually \((\lambda x. \ f \ x \in \neg S)\) \(F\)
  by (rule topological-tendstoD)
with assms(2) have eventually \((\lambda x. \ False)\) \(F\)
  by (rule eventually-elim2) simp
with assms(3) show False
  by (simp add: eventually-False)
qed
lemma (in t3-space) nhds-closed:
  assumes x ∈ A and open A
  shows ∃ A', x ∈ A' ∧ closed A' ∧ A' ⊆ A ∧ eventually (λy. y ∈ A') (nhds x)
proof
  from assms have ∃ U V. open U ∧ open V ∧ x ∈ U ∧ − A ⊆ V ∧ U ∩ V = {}
    by (intro t3-space) auto
  then obtain U V where UV:
    open U open V x ∈ U − A ⊆ V U ∩ V = {}
    by auto
  have eventually (λy. y ∈ U) (nhds x)
    using ‹open U› and ‹x ∈ U› by (intro eventually-nhds-in-open)
  hence eventually (λy. y ∈ − V) (nhds x)
    by eventually-elim (use UV in auto)
  with UV show ?thesis by (intro exI [of − V]) auto
qed

lemma (in order-topology) increasing-tendsto:
  assumes bdd: eventually (λn. f n ≤ l) F
    and ev: ∀x. x < l ⇒ eventually (λn. x < f n) F
  shows (f −−→ l) F
  using assms by (intro order-tendstoI) (auto elim!: eventually-mono)

lemma (in order-topology) decreasing-tendsto:
  assumes bdd: eventually (λn. l ≤ f n) F
    and ev: ∀x. l < x ⇒ eventually (λn. f n < x) F
  shows (f −−→ l) F
  using assms by (intro order-tendstoI) (auto elim!: eventually-mono)

lemma (in order-topology) tendsto-sandwich:
  assumes ev: eventually (λn. f n ≤ g n) net eventually (λn. g n ≤ h n) net
  assumes lim: (f −→ c) net (h −→ c) net
  shows (g −→ c) net
proof (rule order-tendstoI)
  fix a
  show a < c ⇒ eventually (λx. a < g x) net
    using order-tendstoD[OF lim(1), of a] ev by (auto elim: eventually-elim2)
next
  fix a
  show c < a ⇒ eventually (λx. g x < a) net
    using order-tendstoD[OF lim(2), of a] ev by (auto elim: eventually-elim2)
qed

lemma (in t1-space) limit-frequently-eq:
  assumes F ≠ bot
    and frequently (λx. f x = c) F
    and (f −→ d) F
  shows d = c
proof (rule ccontr)
  assume d ≠ c
from t1-space[OF this] obtain U where open U d ∈ U c ∉ U
  by blast
with assms have eventually (λx. f x ∈ U) F
  unfolding tendsto-def by blast
then have eventually (λx. f x ≠ c) F
  by eventually-elim (insert ‹c ∉ U›, blast)
with assms(2) show False
  unfolding frequently-def by contradiction
qed

lemma (in t1-space) tendsto-imp-eventually-ne:
  assumes (f ----> c) F c ≠ c'
  shows eventually (λz. f z ≠ c') F
proof (cases F=bot)
  case True
  thus ?thesis by auto
next
  case False
  show ?thesis
proof (rule ccontr)
  assume ¬ eventually (λz. f z ≠ c') F
  then have frequently (λz. f z = c') F
    by (simp add: frequently-def)
  from limit-frequently-eq[OF False this ‹(f ----> c) F›] and ‹c ≠ c'› show False
    by contradiction
qed

lemma (in linorder-topology) tendsto-le:
  assumes F: ¬ trivial-limit F
  and x: (f ----> x) F
  and y: (g ----> y) F
  and ev: eventually (λx. g x ≤ f x) F
  shows y ≤ x
proof (rule ccontr)
  assume ¬ y ≤ x
  with less-separate[of x y] obtain a b where xy: x < a b < y {..<a} ∩ {b<..} = 
    {}
    by (auto simp: not-le)
  then have eventually (λx. f x < a) F eventually (λx. b < g x) F
    using x y by (auto intro: order-tendstoD)
  with ev have eventually (λx. False) F
    by eventually-elim (insert xy, fastforce)
  with F show False
    by (simp add: eventually-False)
qed

lemma (in linorder-topology) tendsto-lowerbound:
assumes \( x : (f \longrightarrow x) F \)
and \( ev : \text{eventually } \left( \lambda i. \ a \leq f i \right) F \)
and \( F : \neg \ \text{trivial-limit } F \)
shows \( a \leq x \)
using \( F \ x \ \text{tendsto-const } ev \ \text{by (rule tendsto-le)} \)

**lemma** (in linorder-topology) tendsto-upperbound:
assumes \( x : (f \longrightarrow x) F \)
and \( ev : \text{eventually } \left( \lambda i. \ a \geq f i \right) F \)
and \( F : \neg \ \text{trivial-limit } F \)
shows \( a \geq x \)
by (rule tendsto-le [\( OF F \ \text{tendsto-const } x \ ev \)])

**lemma** filterlim-at-within-not-equal:
fixes \( f :: 'a \Rightarrow 'b::t2-space \)
assumes \( \text{filterlim } f \ (\text{at } a \ \text{within } s) F \)
shows \( \text{eventually } \left( \lambda w. \ f w \in s \wedge f w \neq b \right) F \)
proof (cases \( a = b \))
case True
then show \( \text{?thesis } \) using assms by (simp add: filterlim-at)
next
case False
from hausdorff [OF this] obtain \( U \ V \) where \( UV : \text{open } U \ \text{open } V \ a \in U \ b \in V \)
\( U \cap V = \{\} \)
by auto
have \( (f \longrightarrow a) F \) using assms filterlim-at by auto
then have \( \forall F \ y \in \at \ x \ \text{within } X \. \ P \ (\text{Lim } (\at \ x \ \text{within } s) (\lambda x. \ x) = x) \)
by (simp add: tendsto-Lim)
ultimately show \( \text{?thesis } \)
apply eventually-elim
using \( UV \) by auto
qed

**99.5.4 Rules about \( \text{Lim} \)**

**lemma** tendsto-Lim: \( \neg \ \text{trivial-limit net } \rightarrow (f \longrightarrow l) \ \text{net } \rightarrow \text{Lim net } f = l \)
unfolding Lim-def using tendsto-unique [of net f] by auto

**lemma** Lim-ident-at: \( \neg \ \text{trivial-limit } (\at \ x \ \text{within } s) \rightarrow \text{Lim } (\at \ x \ \text{within } s) (\lambda x. \ x) = x \)
by (simp add: tendsto-Lim)

**lemma** Lim-cong:
assumes \( \forall F \ x \in F. \ f x = g x \ F = G \)
shows \( \text{Lim } F f = \text{Lim } F g \)
unfolding t2-space-class.Lim-def using tendsto-cong assms by fastforce

**lemma** eventually-Lim-ident-at:
\((\forall F \ y \in \at \ x \ \text{within } X. \ P \ (\text{Lim } (\at \ x \ \text{within } X) (\lambda x. \ x)) \ y) \leftrightarrow \)
(∀ y in at x within X. P x y) for x::'a::t2-space
by (cases at x within X = bot) (auto simp: Lim-ident-at)

lemma filterlim-at-bot-at-right:
  fixes f :: 'a::linorder-topology ⇒ 'b::linorder
  assumes mono: ∀ x y. Q x ⇒ Q y ⇒ x ≤ y ⇒ f x ≤ f y
      and bij: ∀ x. P x ⇒ f (g x) = x
      and Q: eventually Q (at-right a)
      and bound: ∃ b. Q b ⇒ a < b
      and P: eventually P at-bot
  shows filterlim f at-bot (at-right a)
proof – 
  from P obtain x where x: ∃ y. y ≤ x ⇒ P y
    unfolding eventually-at-bot-linorder by auto
  show ?thesis
proof (intro filterlim-at-bot-le[THEN iffD2] allI impI)
  fix z
  assume z ≤ x
  with x have P z by auto
  have eventually (λ x. x ≤ g z) (at-right a)
    using bound[OF bij[OF ‹P z›]]
    unfolding eventually-at-right[OF bound[OF bij[OF ‹P z›]]]
    by (auto intro: exI[of - g z])
  with Q show eventually (λ x. f x ≤ z) (at-right a)
    by eventually-elim (metis bij ‹P z› mono)
  qed
qed

lemma filterlim-at-top-at-left:
  fixes f :: 'a::linorder-topology ⇒ 'b::linorder
  assumes mono: ∀ x y. Q x ⇒ Q y ⇒ x ≤ y ⇒ f x ≤ f y
      and bij: ∀ x. P x ⇒ f (g x) = x
      and Q: eventually Q (at-left a)
      and bound: ∃ b. Q b ⇒ b < a
      and P: eventually P at-top
  shows filterlim f at-top (at-left a)
proof – 
  from P obtain x where x: ∃ y. x ≤ y ⇒ P y
    unfolding eventually-at-top-linorder by auto
  show ?thesis
proof (intro filterlim-at-top-ge[THEN iffD2] allI impI)
  fix z
  assume x ≤ z
  with x have P z by auto
  have eventually (λ x. z ≤ f x) (at-left a)
    using bound[OF bij[OF ‹P z›]]
    unfolding eventually-at-left[OF bound[OF bij[OF ‹P z›]]]
    by (auto intro: exI[of - g z])
  with Q show eventually (λ x. z ≤ f x) (at-left a)
by eventually-elim (metis bij \langle P z \rangle \ mono)
qed

lemma filterlim-split-at:
filterlim f F (at-left x) \implies filterlim f F (at-right x) \implies
filterlim f F (at x)
for x :: 'a::linorder-topology
by (subst at-eq-sup-left-right) (rule filterlim-sup)

lemma filterlim-at-split:
filterlim f F (at x) \iff filterlim f F (at-left x) \land filterlim f F (at-right x)
for x :: 'a::linorder-topology
by (subst at-eq-sup-left-right) (simp add: filterlim-def filtermap-sup)

lemma eventually-nhds-top:
fixes P :: 'a :: {order-top,linorder-topology} \Rightarrow \bool
and b :: 'a
assumes b < top
shows eventually P (nhds top) \iff (\exists b<top. (\forall z. b < z \rightarrow P z))
unfolding eventually-nhds
proof safe
fix S :: 'a set
assume open S top \in S
note open-left[OF this \langle b < top \rangle]
moreover assume \forall s\in S. P s
ultimately show \exists b<top. \forall z>b. P z
  by (auto simp: subset-eq Ball-def)
next
fix b
assume b < top \forall z>b. P z
then show \exists S. open S \land top \in S \land (\forall xa\in S. P xa)
  by (intro exI[of \{ b <.. \}]) auto
qed

lemma tendsto-at-within-iff-tendsto-nhds:
(g \longrightarrow g l) (at l within S) \iff (g \longrightarrow g l) (\inf (nhds l) (principal S))
unfolding tendsto-def eventually-at-filter eventually-inf-principal
by (intro ext all-cong imp-cong) (auto elim!: eventually-mono)

99.6 Limits on sequences

abbreviation (in topological-space)
LIMSEQ :: [nat \Rightarrow 'a, 'a] \Rightarrow \bool (((-)/ \longrightarrow (-)) [60, 60] 60)
where X \longrightarrow L \equiv (X \longrightarrow L) sequentially

abbreviation (in t2-space) lim :: (nat \Rightarrow 'a) \Rightarrow 'a
where lim X \equiv Lim sequentially X
THEORY “Topological-Spaces”

definition (in topological-space) convergent :: (nat ⇒ ’a) ⇒ bool
  where convergent X = (∃ L. X −→ L)

lemma lim-def: lim X = (THE L. X −→ L)
  unfolding Lim-def ..

lemma lim-explicit:
  f −→ f0 ⇔ (∀ S. open S −→ f0 ∈ S −→ (∃ N. ∀ n≥N. f n ∈ S))
  unfolding tendsto-def eventually-sequentially by auto

lemma closed-sequentially:
  assumes closed S and ⋀ n. f n ∈ S and f −→ l
  shows l ∈ S by (metis Lim-in-closed-set assms eventually-sequentially trivial-limit-sequentially)

99.7 Monotone sequences and subsequences

Definition of monotonicity. The use of disjunction here complicates proofs considerably. One alternative is to add a Boolean argument to indicate the direction. Another is to develop the notions of increasing and decreasing first.

definition monoseq :: (nat ⇒ ’a::order) ⇒ bool
  where monoseq X ⇔ (∀ m. ∀ n≥m. X m ≤ X n) ∨ (∀ m. ∀ n≥m. X n ≤ X m)

abbreviation incseq :: (nat ⇒ ’a::order) ⇒ bool
  where incseq X ≡ mono X

lemma incseq-def: incseq X ⇔ (∀ m. ∀ n≥m. X n ≥ X m)
  unfolding mono-def ..

abbreviation decseq :: (nat ⇒ ’a::order) ⇒ bool
  where decseq X ≡ antimono X

lemma decseq-def: decseq X ⇔ (∀ m. ∀ n≥m. X n ≤ X m)
  unfolding antimono-def ..

99.7.1 Definition of subsequence.

lemma strict-mono-leD: strict-mono r ⇒ m ≤ n ⇒ r m ≤ r n
  by (erule (1) monoD [OF strict-mono-monotone])

lemma strict-mono-id: strict-mono id
  by (simp add: strict-mono-def)

lemma incseq-SucI: (∀ n. X n ≤ X (Suc n)) ⇒ incseq X
  using lift-Suc-monotone-le[of X] by (auto simp: incseq-def)

lemma incseqD: incseq f ⇒ i ≤ j ⇒ f i ≤ f j
  by (auto simp: incseq-def)
lemma incseq-SucD: incseq A \Longrightarrow A \leq A \ (\text{Suc} \ i)
using incseqD[of A i Suc i] by auto

lemma incseq-Suc-iff: incseq f \iff (\forall n. f \ n \ \leq f \ (\text{Suc} \ n))
by (auto intro: incseq-SucI dest: incseq-SucD)

lemma incseq-const[simp, intro]: incseq (\lambda x. k)
unfolding incseq-def by auto

lemma decseq-SucI: (\forall n. X \ (\text{Suc} \ n) \ \leq \ X \ n) \Longrightarrow decseq X
using order.lift-Suc-mono-le[OF dual-order, of X] by (auto simp: decseq-def)

lemma decseqD: decseq f \Longrightarrow i \leq j \Longrightarrow f j \leq f i
by (auto simp: decseq-def)

lemma decseq-SucD: decseq A \Longrightarrow A \ (\text{Suc} \ i) \ \leq \ A \ i
using decseqD[of A i Suc i] by auto

lemma decseq-Suc-iff: decseq f \iff (\forall n. f \ (\text{Suc} \ n) \ \leq \ f \ n)
by (auto intro: decseq-SucI dest: decseq-SucD)

lemma decseq-const[simp, intro]: decseq (\lambda x. k)
unfolding decseq-def by auto

lemma monoseq-iff: monoseq X \iff incseq X \lor decseq X
unfolding monoseq-def incseq-def decseq-def ..

lemma monoseq-Suc: monoseq X \iff (\forall n. X \ n \ \leq \ X \ (\text{Suc} \ n)) \lor (\forall n. X \ (\text{Suc} \ n) \ \leq \ X \ n)
unfolding monoseq-iff incseq-Suc-iff decseq-Suc-iff ..

lemma monoI1: \forall m. \forall n \geq m. X \ m \ \leq \ X \ n \ \Longrightarrow \ monoseq X
by (simp add: monoseq-def)

lemma monoI2: \forall m. \forall n \geq m. X \ n \ \leq \ X \ m \ \Longrightarrow \ monoseq X
by (simp add: monoseq-def)

lemma mono-SucI1: \forall n. X \ n \ \leq \ X \ (\text{Suc} \ n) \ \Longrightarrow \ monoseq X
by (simp add: monoseq-Suc)

lemma mono-SucI2: \forall n. X \ (\text{Suc} \ n) \ \leq \ X \ n \ \Longrightarrow \ monoseq X
by (simp add: monoseq-Suc)

lemma monoseq-minus:
fixes a :: nat \Rightarrow 'a::ordered-ab-group-add
assumes monoseq a
shows monoseq (\lambda n. - a \ n)
proof (cases \forall m. \forall n \geq m. a \ m \ \leq \ a \ n)
THEORY "Topological-Spaces"

case True
then have \( \forall m. \forall n \geq m. -a n \leq -a m \) by auto
then show \(?thesis\) by (rule monoI2)
next
case False
then have \( \forall m. \forall n \geq m. -a m \leq -a n \)
using \(<\text{monoseq } a[\text{unfolded monoseq-def}]\) by auto
then show \(?thesis\) by (rule monoI1)
qed

99.7.2 Subsequence (alternative definition, (e.g. Hoskins)

For any sequence, there is a monotonic subsequence.

lemma seq-monosub:
\[\text{fixes } s :: \text{nat} \Rightarrow 'a::linorder \]
shows \( \exists f. \text{strict-mono } f \land \text{monoseq} (\lambda n. (s \ f n)) \)
proof (cases \( \forall n. \exists p > n. \forall m \geq p. s m \leq s p \))
case True
then have \( \exists f. \forall n. (\forall m \geq f n. s m \leq s (f n)) \land f n < f (\text{Suc } n) \)
by (intro dependent-nat-choice) (auto simp: conj-commute)
then obtain \( f :: \text{nat} \Rightarrow \text{nat} \)
where \( f: \text{strict-mono } f \) and \( \text{mono: } \land n m. f n \leq m \Rightarrow s m \leq s (f n) \)
by (auto simp: strict-mono-Suc-iff)
then have incseq \( f \)
unfolding strict-mono-Suc-iff incseq-Suc-iff by (auto intro: less-imp-le)
then have monoseq \( (\lambda n. s (f n)) \)
by (auto simp add: incseq-def intro!: mono_monoI2)
with \( f \) show \(?thesis\)
by auto
next
case False
then obtain \( N \) where \( N: p > N \Rightarrow \exists m > p. s p < s m \) for \( p \)
by (force simp: not-le le-less)
have \( \exists f. \forall n. N < f n \land f n < f (\text{Suc } n) \land s (f n) \leq s (f (\text{Suc } n)) \)
proof (intro dependent-nat-choice)
fix \( x \)
assume \( N < x \) with \( N[\text{of } x] \)
show \( \exists y > N. x < y \land s x \leq s y \)
by (auto intro: less-trans)
qed auto
then show \(?thesis\)
by (auto simp: monoseq-iff incseq-Suc-iff strict-mono-Suc-iff)
qed

lemma seq-suble:
assumes \( sf: \text{strict-mono } (f :: \text{nat} \Rightarrow \text{nat}) \)
shows \( n \leq f n \)
proof (induct \( n \))
case \( 0 \)
show ?case by simp
next
  case (Suc n)
  with sf [unfolded strict-mono-Suc-iff, rule-format, of n] have n < f (Suc n)
  by arith
  then show ?case by arith
qed

lemma eventually-subseq:
strict-mono r \Longrightarrow eventually P sequentially \Longrightarrow eventually (\lambda n. P (r n)) sequentially
unfolding eventually-sequentially by (metis seq-suble le-trans)

lemma not-eventually-sequentiallyD:
assumes \neg eventually P sequentially
shows \exists r :: nat \Rightarrow nat. strict-mono r \land (\forall n. \neg P (r n))
proof –
  from assms have \forall n. \exists m \geq n. \neg P m
  unfolding eventually-sequentially by (simp add: not-less)
  then obtain r where \land n. r n \geq n \land n. \neg P (r n)
  by (auto simp: choice-iff)
  then show \?thesis
  by (auto intro!: exI [of - \lambda n. r (((Suc o r) ^^ Suc n) 0)]
      simp: less-eq-Suc-le strict-mono-Suc-iff)
qed

lemma sequentially-offset:
assumes eventually (\lambda i. P i) sequentially
shows \forall n \land n. \neg P (r n)
using assms by (rule eventually-sequentially-seg [THEN iffD2])

lemma seq-offset-neg:
(f \longrightarrow l) sequentially \Longrightarrow ((\lambda i. f(i - k)) \longrightarrow l) sequentially
apply (erule filterlim-compose)
apply (simp add: filterlim-def le-sequentially eventually-filtermap eventually-sequentially, arith)
done

lemma filterlim-subseq: strict-mono f \Longrightarrow filterlim f sequentially sequentially
unfolding filterlim-iff by (metis eventually-subseq)

lemma strict-mono-o: strict-mono r \Longrightarrow strict-mono s \Longrightarrow strict-mono (r o s)
unfolding strict-mono-def by simp

lemma strict-mono-compose: strict-mono r \Longrightarrow strict-mono s \Longrightarrow strict-mono (\lambda x. r (s x))
using strict-mono-o[of r s] by (simp add: o-def)

lemma incseq-imp-monoseq: incseq X \Longrightarrow monoseq X
by (simp add: incseq-def monoseq-def)

lemma decseq-imp-monoseq: decseq $X \rightarrow$ monoseq $X$
  by (simp add: decseq-def monoseq-def)

lemma decseq-eq-incseq: decseq $X = \text{incseq} (\lambda n. - X n)$
  for $X :: \text{nat}$ : 'a::ordered-ab-group-add
  by (simp add: decseq-def incseq-def)

lemma INT-decseq-offset:
  assumes decseq $F$
  shows $\bigcap i. F i = (\bigcap i\in\{n..\}. F i)$
proof safe
  fix $x i$
  assume $x: x \in (\bigcap i\in\{n..\}. F i)$
  show $x \in F i$
  proof cases
    from $x$ have $x: x \in F n$ by auto
    also assume $i: i \leq n$ with $\langle \text{decseq } F \rangle$
    have $F n \subseteq F i$ unfolding decseq-def by simp
    finally show $\text{thesis}$.
  qed (insert $x$, simp)
qed auto

lemma LIMSEQ-const-iff: $$(\lambda n. k) \rightarrow l \iff k = l$$
  for $k l :: \text{t2-space}$
  using trivial-limit-sequentially by (rule tendsto-const-iff)

lemma LIMSEQ-SUP: incseq $X \rightarrow X \rightarrow (\text{SUP } i. X i :: 'a::\{complete-linorder,linorder-topology\})$
  by (intro increasing-tendsto)
  (auto simp: SUP-upper less-SUP-iff incseq-def eventually-sequentially intro: less-le-trans)

lemma LIMSEQ-INF: decseq $X \rightarrow X \rightarrow (\text{INF } i. X i :: 'a::\{complete-linorder,linorder-topology\})$
  by (intro decreasing-tendsto)
  (auto simp: INF-lower INF-less-iff decseq-def eventually-sequentially intro: le-less-trans)

lemma LIMSEQ-ignore-initial-segment: $f \rightarrow a \Rightarrow (\lambda n. f (n + k)) \rightarrow a$
  unfolding tendsto-def by (subst eventually-sequentially-seg[where $k=k$])

lemma LIMSEQ-offset: $(\lambda n. f (n + k)) \rightarrow a \Rightarrow f \rightarrow a$
  unfolding tendsto-def
  by (subst (asm) eventually-sequentially-seg[where $k=k$])

lemma LIMSEQ-Suc: $f \rightarrow l \Rightarrow (\lambda n. f (Suc n)) \rightarrow l$
  by (drule LIMSEQ-ignore-initial-segment [where $k=Suc 0$]) simp

lemma LIMSEQ-imp-Suc: $(\lambda n. f (Suc n)) \rightarrow l \Rightarrow f \rightarrow l$
by (rule LIMSEQ-offset [where k=Suc 0]) simp

lemma LIMSEQ-lessThan-iff-atMost:
  shows (λn. f {..<n}) → x ↔ (λn. f {..n}) →→ x
apply (sub goal filterlim-sequentially-Suc [symmetric])
apply (simp only: lessThan-Suc-atMost)
done

lemma (in t2-space) LIMSEQ-Uniq: ∃≤1 l. X →→ l
by (simp add: tendsto-unique')

lemma (in t2-space) LIMSEQ-unique: X →→ a = X →→ b = a = b
using trivial-limit-sequentially by (rule tendsto-unique)

lemma LIMSEQ-le-const: X →→ x = ⇒ ∃N. ∀n≥N. a ≤ X n = a ≤ x
  for a x :: 'a::linorder-topology
by (simp add: eventually-at-top-linorder tendsto-lowerbound)

lemma LIMSEQ-le: X →→ x = Y →→ y = ⇒ ∃N. ∀n≥N. X n ≤ Y n
⇒ x ≤ y
  for x y :: 'a::linorder-topology
using tendsto-le[of sequentially Y y X x] by (simp add: eventually-sequentially)

lemma LIMSEQ-le-const2: X →→ x = ⇒ ∃N. ∀n≥N. X n ≤ a = x ≤ a
  for a x :: 'a::linorder-topology
by (rule LIMSEQ-le[of X x λn. a]) auto

lemma Lim-bounded: f →→ l = ⇒ ∀n≥M. f n ≤ C = l ≤ C
  for l :: 'a::linorder-topology
by (intro LIMSEQ-le-const2) auto

lemma Lim-bounded2:
  fixes X Y :: nat ⇒ 'a::linorder-topology
  assumes lim: f →→ l and ge: ∀n≥N. f n ≥ C
  shows l ≥ C
  using ge
  by (intro tendsto-le[OF trivial-limit-sequentially lim tendsto-const])
(auto simp: eventually-sequentially)

lemma lim-mono:
  fixes Y :: nat ⇒ 'a::linorder-topology
  assumes f: f n ≤ Y n
and X: X →→ x
  shows x ≤ y
  using assms(1) by (intro LIMSEQ-le[OF assms(2,3)]) auto

lemma Sup-lim:
  fixes a :: 'a::{complete-linorder, linorder-topology}
assumes $\bigwedge n. \ b \ n \in s$
and \( b \longrightarrow a \)
shows $a \leq \text{Sup} \ s$
by (metis Lim-bounded assms complete-lattice-class.Sup-upper)

lemma Inf-lim:
fixes a :: 'a::{complete-linorder,linorder-topology}
assumes $\bigwedge n. \ b \ n \in s$
and \( b \longrightarrow a \)
shows $\text{Inf} \ s \leq a$
by (metis Lim-bounded2 assms complete-lattice-class.Inf-lower)

lemma SUP-Lim:
fixes X :: nat $\Rightarrow$ 'a::{complete-linorder,linorder-topology}
assumes inc: incseq X
and \( l \)
shows $(\text{SUP} \ n. \ X \ n) = l$
using LIMSEQ-SUP[OF inc tendsto-unique[OF trivial-limit-sequentially l]]
by simp

lemma INF-Lim:
fixes X :: nat $\Rightarrow$ 'a::{complete-linorder,linorder-topology}
assumes dec: decseq X
and \( l \)
shows $(\text{INF} \ n. \ X \ n) = l$
using LIMSEQ-INF[OF dec tendsto-unique[OF trivial-limit-sequentially l]]
by simp

lemma convergentD: convergent X $\implies$ $\exists L. \ X \longrightarrow L$
by (simp add: convergent-def)

lemma convergentI: X $\longrightarrow$ L $\implies$ convergent X
by (auto simp add: convergent-def)

lemma convergent-LIMSEQ-iff: convergent X $\iff$ X $\longrightarrow$ lim X
by (auto intro: theI LIMSEQ-unique simp add: convergent-def lim-def)

lemma convergent-const: convergent ($\lambda n. \ c$)
by (rule convergentI) (rule tendsto-const)

lemma monoseq-le:
monoseq a $\longrightarrow$ a $\longrightarrow$ x $\implies$
$\forall n. \ a \ n \leq x \wedge (\forall m. \ \forall n\geq m. \ a \ m \leq a \ n) \vee$
$\forall n. \ x \leq a \ n \wedge (\forall m. \ \forall n\geq m. \ a \ n \leq a \ m)$
for x :: 'a::linorder-topology
by (metis LIMSEQ-le-const LIMSEQ-le-const2 decseq-def incseq-def monoseq-iff)

lemma LIMSEQ-subseq-LIMSEQ: X $\longrightarrow$ L $\implies$ strict mono f $\longrightarrow$ $(X \circ f)$ $\longrightarrow$ L
unfolding comp-def by (rule filterlim-compose [of X, OF - filterlim-subseq])

lemma convergent-subseq-convergent: convergent X \implies strict-mono f \implies convergent (X \circ f)
  by (auto simp: convergent-def intro: LIMSEQ-subseq-LIMSEQ)

lemma limI: X ----> L \implies \lim X = L
  by (rule tendsto-Lim) (rule trivial-limit-sequentially)

lemma lim-le: convergent f \implies (\forall n. f n \leq x) \implies \lim f \leq x
  for x :: 'a::linorder_topology
  using LIMSEQ-le-const \[of f \lim f x\]
  by (simp add: convergent-LIMSEQ-iff)

lemma lim-const [simp]: \lim (\lambda m. a) = a
  by (simp add: limI)

99.7.3 Increasing and Decreasing Series

lemma incseq-le: incseq X \implies X ----> L \implies X n \leq L
  for L :: 'a::linorder_topology
  by (metis incseq-def LIMSEQ-le-const)

lemma decseq-ge: decseq X \implies X ----> L \implies L \leq X n
  for L :: 'a::linorder_topology
  by (metis decseq-def LIMSEQ-le-const2)

99.8 First countable topologies

class first-countable-topology = topological-space +
  assumes first-countable-basis:
    \exists A::nat \Rightarrow 'a set. (\forall i. x \in A \ i \wedge open (A \ i)) \wedge (\forall S. open S \wedge x \in S \Rightarrow (\exists i. A \ i \subseteq S))

lemma (in first-countable-topology) countable-basis-at-decseq:
  obtains A :: nat \Rightarrow 'a set where
  \forall i. open (A \ i) \wedge i \in \{A \ i\}
  \forall S. open S \Rightarrow x \in S \Rightarrow eventually (\lambda i. A \ i \subseteq S) sequentally
  proof atomize-elim
  from first-countable-basis[of x] obtain A :: nat \Rightarrow 'a set
    where nhds: \forall i. open (A \ i) \wedge i \in \{A \ i\}
      and incl: \forall S. open S \Rightarrow x \in S \Rightarrow \exists i. A \ i \subseteq S
    by auto
  define F where F n = (\bigcap\{i_n \leq n. A \ i\}) for n
  show \exists A. (\forall i. open (A \ i)) \wedge (\forall i. x \in A \ i) \wedge
    (\forall S. open S \Rightarrow x \in S \Rightarrow eventually (\lambda i. A \ i \subseteq S) sequentally)
  proof (safe intro!: exI[of - F])
    fix i
    show open (F \ i)
      using nhds(1) by (auto simp: F-def)
    show x \in F \ i
next
  fix S
  assume open S x ∈ S
  from incl[OF this] obtain i where F i ⊆ S
  unfolding F-def by (auto simp:F-def)
moreover have ∀. i ≤ j ⇒ F j ⊆ F i
  by (simp add: Inf-superset-mono F-def image-mono)
ultimately show eventually (λi. F i ⊆ S) sequentially
  by (auto simp: eventually-sequentially)
qed
qed

lemma (in first-countable-topology) nhds-countable:
  obtains X :: nat ⇒ 'a set
  where decseq X "A n. open (X i) ∧ x ∈ X n nhds x = (INF n. principal (X i))"
proof
  from first-countable-basis obtain A :: nat ⇒ 'a set
    where *: ∀. x ∈ A n ∧ A i ≤ n. open (A i) ∧ S. open S ⇒ x ∈ S ⇒ ∃i. x ∈ A i ≤ S
    by metis
  show thesis
  proof
    show decseq (λn. i ≤ n. A i)
      by (simp add: antimono-iff-le-Suc atMost-Suc)
    show x ∈ (∪ i ≤ n. A i) ∧ A n. open (∪ i ≤ n. A i) for n
      using * by auto
    with * show nhds x = (INF n. principal (∪ i ≤ n. A i))
    unfolding nhds-def
    apply (intro INF-eq)
    apply fastforce
    apply blast
    done
  qed
qed

lemma (in first-countable-topology) countable-basis:
  obtains A :: nat ⇒ 'a set where
    ∀ i. open (A i) ∩ i ≤ n. A i
    ∀ F. (∀ n. F n ∈ A i) ⇒ F x
proof atomize-elim
  obtain A :: nat ⇒ 'a set where *:
    ∀ i. open (A i)
    ∀ i. x ∈ A i
    ∀ S. open S ⇒ x ∈ S ⇒ eventually (λi. A i ⊆ S) sequentially
    by (rule countable-basis-at-decseq)
  have eventually (λn. F n ∈ S) sequentially
    if ∀ n. F n ∈ A n open S x ∈ S for F S
THEORY "Topological-Spaces"

using *(3) of S that by (auto elim: eventually-mono simp: subset-eq)
with * show \(\exists A. (\forall i. \text{open } (A i)) \land (\forall i. x \in A i) \land (\forall F. (\forall n. F n \in A n) \rightarrow F \rightarrow x)\)
by (intro exI[of - A]) (auto simp: tendsto-def)

qed

lemma (in first-countable-topology) sequentially-imp-eventually-nhds-within:
assumes \(\forall f. (\forall n. f n \in s) \land f \rightarrow a \rightarrow \text{eventually } (\lambda n. P (f n))\) sequentially
shows eventually P (inf (nhds a) (principal s))

proof (rule ccontr)
obtain A :: nat \Rightarrow 'a set where *:
\(\land i. \text{open } (A i)\)
\(\land i. a \in A i\)
\(\land F. \forall n. F n \in A n \rightarrow F \rightarrow a\)
by (rule countable-basis) blast
assume \(\neg\ \text{thesis}\)
with * have \(\exists F. \forall n. F n \in s \land F n \in A n \land \neg P (F n)\)
unfolding eventually-inf-principal eventually-nhds
by (intro choice) fastforce
then obtain F where F: \(\forall n. F n \in s \land F n \in A n \land F': \forall n. \neg P (F n)\)
(F n)
by blast
with * have F \rightarrow a
by auto
then have eventually (\(\lambda n. P (F n)\)) sequentially
using assms F by simp
then show False
by (simp add: F')

qed

lemma (in first-countable-topology) eventually-nhds-iff-sequentially:
eventually P (inf (nhds a) (principal s)) \longleftrightarrow
(\(\forall f. (\forall n. f n \in s) \land f \rightarrow a \rightarrow \text{eventually } (\lambda n. P (f n))\) sequentially)

proof (safe intro!: sequentially-imp-eventually-nhds-within)
assume eventually P (inf (nhds a) (principal s))
then obtain S where open S a \in S \forall x \in S. x \in s \rightarrow P x
by (auto simp: eventually-inf-principal eventually-nhds)
moreover
fix f
assume \(\forall n. f n \in s f \rightarrow a\)
ultimately show eventually (\(\lambda n. P (f n)\)) sequentially
by (auto dest!: topological-tendstoD elim: eventually-mono)

qed

lemma (in first-countable-topology) eventually-nhds-iff-sequentially:
eventually P (nhds a) \longleftrightarrow (\(\forall f. f \rightarrow a \rightarrow \text{eventually } (\lambda n. P (f n))\) sequentially)
using eventually-nhds-within-iff-sequentially[of P a UNIV] by simp
lemma Inf-as-limit:
  fixes A :: 'a::{linorder_topology, first-countable_topology, complete_linorder} set
  assumes A ≠ {}
  shows ∃ u. (∀ n. u n ∈ A) ∧ u ----> Inf A
proof (cases Inf A ∈ A)
  case True
  show ?thesis
  by (rule exI [of - λ n. Inf A], auto simp add: True)
next
  case False
  obtain y where y ∈ A using assms by auto
  then have Inf A < y using False Inf-lower less_le by auto
  obtain F :: nat ⇒ 'a set where F :: ( ∀ i. open (F n) ∧ i. Inf A ∈ F i ) 
    ∧ u. (∀ n. u n ∈ F n) ----> u ----> Inf A
  by (metis first-countable_topology-class.countable_basis)
  define u where u = ( λ n. SOME z. z ∈ F n ∧ z ∈ A )
  have ∃ z. z ∈ U ∧ z ∈ A if Inf A ∈ U open U for U
proof
  obtain b where b > Inf A { Inf A ..< b } ⊆ U
  using open-right {OF open U, Inf A ∈ U, Inf A < y} by auto
  obtain z where z < b z ∈ A
  using Inf A < b Inf-less-iff by auto
  then have z ∈ { Inf A ..< b } by (simp add: Inf-lower)
  then show ?thesis using ( z ∈ A ) ( { Inf A ..< b } ⊆ U ) by auto
qed
then have u ----> Inf A using F(3) by simp
then show ?thesis using * by auto
qed

lemma tendsto-at-iff-sequentially:
( f ----> a ) ( at x within s ) <-> ( ∀ X. ( ∀ i. X i ∈ s - {x} ) ----> X ----> x ----> ( f o X ) ----> a )
  for f :: 'a::{first-countable-topology} ⇒ -
  unfolding filterlim_def[of - nhds a] le_filter_def eventually_filtermap
  at_within_def eventually_nhds_within_iff_sequentially comp_def
by metis

lemma approx-from-above-dense-linorder:
  fixes x :: 'a::{dense_linorder, linorder_topology, first-countable_topology}
  assumes x < y
  shows ∃ u. (∀ n. u n > x) ∧ ( u ----> x )
proof
  obtain A :: nat ⇒ 'a set where A :: ( ∀ i. open (A i) ∧ i. x ∈ A i ) 
    ∧ F. ( ∀ n. F n ∈ A n ) ----> F ----> x
by (metis first-countable-topology-class.countable-basis)
define u where u = (λn. SOME z. z ∈ A n ∧ z > x)
have ∃z. z ∈ U ∧ x < z if x ∈ U open U for U
  using open-right[OF ⟨open U⟩ ⟨x ∈ U⟩ ⟨x < y⟩]
  by (meson atLeastLessThan-iff dense less-imp-le subset-eq)
then have *: u n ∈ A n ∧ x < u n for n
  using ⟨x ∈ A n⟩ ∨ open (A n) unfolding u-def by (metis (no-types, lifting)
somel-ex)
then have u −→ x using A(3) by simp
then show ?thesis using * by auto
qed

lemma approx-from-below-dense-linorder:
  fixes x::'a::{dense-linorder, linorder-topology, first-countable-topology}
  assumes x > y
  shows ∃u. (∀n. u n < x) ∧ (u −→ x)
proof –
  obtain A :: nat ⇒ 'a set where A: ∀i. open (A i) \[i\]. x ∈ A i
              \[F. (∀n. F n ∈ A n) ⇒ F −→ x
                by (metis first-countable-topology-class.countable-basis)
define u where u = (λn. SOME z. z ∈ A n ∧ z < x)
have ∃z. z ∈ U ∧ x < z if x ∈ U open U for U
  using open-left[OF ⟨open U⟩ ⟨x ∈ U⟩ ⟨x > y⟩]
  by (meson dense greaterThanAtMost-iff dense less-imp-le subset-eq)
then have *: u n ∈ A n ∧ u n < x for n
  using ⟨x ∈ A n⟩ ∨ open (A n) unfolding u-def by (metis (no-types, lifting)
somel-ex)
then have u −→ x using A(3) by simp
then show ?thesis using * by auto
qed

99.9 Function limit at a point

abbreviation LIM :: (′a::topological-space ⇒ ′b::topological-space) ⇒ ′a ⇒ ′b ⇒ bool
  ([(−)/ (−)→ (−)]) [60, 0, 60] 60)
where f − a→ L ⇔ (f −→ L) (at a)

lemma tendsto-within-open: a ∈ S ⇒ open S ⇒ (f −→ l) (at a within S)
⇔ (f − a→ l)
  by (simp add: tendsto-def at-within-open[where S = S])

lemma tendsto-within-open-NO-MATCH:
  a ∈ S ⇒ NO-MATCH UNIV S ⇒ open S ⇒ (f −→ l)(at a within S) ⇔
  (f −→ l)(at a)
  for f :: ′a::topological-space ⇒ ′b::topological-space
  using tendsto-within-open by blast

lemma LIM-const-not-eq[tendsto-intros]: k ≠ L ⇒ (λx. k) − a→ L
for a :: 'a::perfect-space and k L :: 'b::t2-space
by (simp add: tendsto-const-iff)

lemmas LIM-not-zero = LIM-const-not-eq [where L = 0]

lemma LIM-const-eq: (λx. k) −−→ l =⇒ k = L
for a :: 'a::perfect-space and k L :: 'b::t2-space
by (simp add: tendsto-const-iff)

lemma LIM-unique: f −−→ l =⇒ f −−→ m =⇒ l = m
for a :: 'a::perfect-space and L M :: 'b::t2-space
using at-neq-bot by (rule tendsto-unique)

lemma LIM-Uniq: ∃ ≤ 1 L :: 'b::t2-space. f −−→ l
for a :: 'a::perfect-space
by (auto simp add: Uniq-def LIM-unique)

lemma LIM-equal: ∀ x. x ≠ a =⇒ f x = g x =⇒ (f −−→ l) =⇒ (g −−→ l)
by (simp add: tendsto-def eventually-at-topological)

lemma LIM-cong: a = b =⇒ (∀ x. x ≠ b =⇒ f x = g x) =⇒ l = m =⇒ (f −−→ l) =⇒ (g −−→ m)
by (simp add: LIM-equal)

lemma tendsto-cong-limit: (f −−→ l) F =⇒ k = l =⇒ (f −−→ k) F
by simp

lemma tendsto-at-iff-tendsto-nhds: g −−→ l =⇒ (λx. f x = g x) =⇒ l = m =⇒ (f −−→ l) =⇒ (g −−→ m)
unfolding tendsto-def eventually-at-filter
by (intro ext all-cong imp-cong) (auto elim!: eventually- mono)

lemma tendsto-compose: g −−→ l =⇒ (f −−→ l) F =⇒ ((λx. g (f x)) −−→ g l) F
unfolding tendsto-at-iff-tendsto-nhds by (rule filterlim-compose[of g])

lemma tendsto-compose-eventually:
  g −−→ m =⇒ (f −−→ l) F =⇒ eventually (λx. f x ≠ b) =⇒ (f −−→ m) F
by (rule filterlim-compose[of g - at l]) (auto simp add: filterlim-at)

lemma LIM-compose-eventually:
  assumes f −−→ b
  and g −−→ c
  and eventually (λx. f x ≠ b) (at a)
  shows (λx. g (f x)) −−→ c
using assms(2,1,3) by (rule tendsto-compose-eventually)

lemma tendsto-compose-filtermap: ((g o f) −−→ T) F =⇒ (g −−→ T) (filtermap
lemma tendsto-compose-at:
  assumes \( f : (f \longrightarrow y) \) \( F \) and \( g : (g \longrightarrow z) \) (at \( y \)) and \( f g \); eventually \( (\lambda w. f w = y \longrightarrow g y = z) \) \( F \)
  shows \((g \circ f) \longrightarrow z\) \( F \)
proof -
  have \((\forall F a \in F. f a \neq y) \vee g y = z\) using \( f g \) by force
  moreover have \((g \longrightarrow z) \) \( (\text{filtermap } f \ F) \) \( \vee \neg (\forall F a \in F. f a \neq y)\)
  ultimately show \( ?\text{thesis} \)
  by \( \text{(metis (no-types) filterlim-atI filterlim-def tendsto-mono } f \ g ) \)
ultimately show \( ?\text{thesis} \)
  by \( \text{(metis (no-types) tendsto-compose filterlim-filtermap } g \text{ tendsto-at-iff-tendsto-nhds tendsto-compose-filtermap) } \)
qed

lemma tendsto-nhds-iff: \( (f \longrightarrow (c :: 'a :: t1-space)) \) \( (\text{nhds } x) \) \( \Longleftrightarrow f \ -x \rightarrow c \land f \ISTRIBUTION x = c \)
proof safe
  assume \( \text{lim} : (f \longrightarrow c) \) \( (\text{nhds } x) \)
  show \( f \ x = c \)
  proof (rule econtr)
    assume \( f \ x \neq c \)
    hence \( c \neq f \ x \)
    by \( \text{auto} \)
    then obtain \( A \) where \( A: \text{open } A \ c \in A \ f \ x \notin A \)
    by \( \text{(subst (asm) separation-t1) auto} \)
    with \( \text{lim} \) obtain \( B \) where \( \text{open } B \ x \in B \ \land x. x \in B \Longrightarrow f x \in A \)
    unfolding tendsto-def eventually-nhds by \( \text{metis} \)
    with \( \langle f \ x \notin A \rangle \) show \( \text{False} \)
    by \( \text{blast} \)
  qed
  show \( f \longrightarrow c \) \( (\text{at } x) \)
  using \( \text{lim} \) by \( \text{(rule filterlim-mono) (auto simp: at-within-def) } \)
next
  assume \( f \ -x \rightarrow f \ x \ c = f \ x \)
  thus \( f \longrightarrow f \ x \) \( (\text{nhds } x) \)
  unfolding tendsto-def eventually-at-filter by \( \text{(fast elim: eventually-mono) } \)
qed

99.9.1 Relation of \( \text{LIM} \) and \( \text{LIMSEQ} \)

lemma (in \first-countable-topology) sequentially-imp-eventually-within:
  \((\forall f. (\forall n. f n \in s \Rightarrow f n \neq a) \land f \longrightarrow a) \longrightarrow \text{eventually } (\forall n. P (f n)) \) sequentially \( \longrightarrow \)
  eventually \( P \) \( (\text{at } a \text{ within } s) \)
  unfolding at-within-def
  by \( \text{(intro sequentially-imp-eventually-nhds-within) auto} \)
lemma (in first-countable-topology) sequentially-imp-eventually-at:
\[(\forall f. (\forall n. f n \neq a) \land f \longrightarrow a \longrightarrow \text{eventually } (\lambda n. P (f n)) \text{ sequentially}) \implies \text{eventually } P (at a)\]
using sequentially-imp-eventually-within [where s=UNIV] by simp

lemma LIMSEQ-SEQ-cone:
\[(\forall S. (\forall n. S n \neq a) \land S \longrightarrow a \longrightarrow (\lambda n. X (S n)) \longrightarrow L) \iff X \longrightarrow L (is ?lhs=?rhs)\]
for a :: 'a::first-countable-topology and L :: 'b::topological-space
proof
assume ?lhs then show ?rhs
by (simp add: sequentially-imp-eventually-within tendsto-def)
next
assume ?rhs then show ?lhs
using tendsto-compose-eventually eventuallyI by blast
qed

lemma sequentially-imp-eventually-at-left:
fixes a :: 'a::{linorder-topology,first-countable-topology}
assumes b [simp]: b < a
and \*: \(\forall f. (\forall n. b < f n) \implies (\forall n. f n < a) \implies \text{incseq } f \implies f \longrightarrow a \implies \text{eventually } (\lambda n. P (f n)) \text{ sequentially}\)
shows eventually P (at-left a)
proof (safe intro!: sequentially-imp-eventually-within)
fix X
assume X: \(\forall n. X n \in \{..< a\} \land X n \neq a \implies X n \longrightarrow a\)
show \((\lambda n. P (X n)) \text{ sequentially}\)
proof (rule contr)
assume neg: \(\neg ?thesis\)
have \(\exists s. \forall n. (\neg P (X (s n)) \land b < X (s n)) \land (X (s n) \leq X (s (Suc n)) \land Suc (s n) \leq s (Suc n))\)
(is \(\exists s. ?P s\))
proof (rule dependent-nat-choice)
have \(\neg (\lambda n. Suc x \leq n \implies b < X n \implies X x < X n \implies P (X n)) \text{ sequentially}\)
by (intro not-eventually-impl neg order-tendstoD [OF X(2) b])
then show \(\exists x. \neg P (X (x)) \land b < X x\)
by (auto dest!: not-eventuallyD)
next
fix x n
have \(\neg (\lambda n. Suc x \leq n \implies b < X n \implies X x < X n \implies P (X n)) \text{ sequentially}\)
using X
by (intro not-eventually-impl order-tendstoD [OF X(2)] eventually-ge-at-top neg) auto
then show \(\exists n. (\neg P (X n) \land b < X n) \land (X x \leq X n \land Suc x \leq n)\)
by (auto dest!: not-eventuallyD)
qed
then obtain s where ?P s ..
with \( X \) have \( b < X \) (\( s \ n \))

and \( X \) (\( s \ n \)) < \( a \)

and incseq (\( \lambda n \). \( X \) (\( s \ n \)))

and \( (\lambda n \. \ X \ (s \ n)) \relbarbar\rightarrow a \)

and \( \sim P (X \ (s \ n)) \)

for \( n \) by (auto simp: strict-mono-Suc-iff Suc-le-eq incseq-Suc-iff

intro: LIMSEQ-subseq-LIMSEQ[\{OF \ X\relbarbar\rightarrow a, unfolded comp-def\}]

from \( *[\{OF this(1, 2, 3, 4)\}]\) this(5) show False by auto

qed

qed

lemma tendsto-at-left-sequentially:

fixes \( a \ b :: \'b::\{linorder-topology, first-countable-topology\} \)

assumes \( b < a \)

assumes \( *:: \\( \forall S. (\\\forall n. S \ n < a) \implies (\\\forall n. b < S \ n) \implies incseq S \implies S \relbarbar\rightarrow a \) \implies \( (\\\forall n. X \ (s \ n)) \relbarbar\rightarrow L \)

shows \( (X \relbarbar\rightarrow L) \) (at-left \( a \))

using assms by (simp add: tendsto-def[where l=L] sequentially-imp-eventually-at-left)

lemma sequentially-imp-eventually-at-right:

fixes \( a \ b :: \'a::\{linorder-topology, first-countable-topology\} \)

assumes \( b::\{\text{simp}: a < b \} \)

assumes \( *:: \\( \forall f. (\\\forall n. a < f \ n) \implies (\\\forall n. f \ n < b) \implies decseq f \implies f \relbarbar\rightarrow a \)

\implies \( \text{eventually} (\lambda n. P (f \ n)) \) sequentially

shows \( \text{eventually} P \) (at-right \( a \))

proof (safe intro: sequentially-imp-eventually-within)

fix \( X \)

assume \( X:: \forall n. \ X \ n \in \{ a <.. \} \ \land \ X \ n \neq a \ X \relbarbar\rightarrow a \)

show \( \text{eventually} (\lambda n. P (X \ n)) \) sequentially

proof (rule ccontr)

assume \( \neg: \sim \? \text{thesis} \)

have \( \exists s. \ \forall n. \ (\sim P (X \ (s \ n)) \ \land \ X \ (s \ n) < b) \ \land \ (X \ (s \ (Suc \ n)) \leq X \ (s \ n) \ \land \ Suc \ (s \ n) \leq s \ (Suc \ n)) \)

(is \( \exists s. \ ? P s \) )

proof (rule dependent-nat-choice)

have \( \sim \text{eventually} (\lambda n. X \ n < b \implies P (X \ n)) \) sequentially

by (intro not-eventually-impl neg order-tendstoD[2] [OF \ X(2)] b])

then show \( \exists x. \sim P (X \ x) \ \land \ X \ x < b \)

by (auto dest!: not-eventuallyD)

next

fix \( x \ n \)

have \( \sim \text{eventually} (\lambda n. Suc \ x \leq n \implies X \ n < b \implies X \ n < X \ x \implies P (X \ n)) \) sequentially

using \( X \)

by (intro not-eventually-impl order-tendstoD[2][OF \ X(2)] eventually-ge-at-top)
neg) auto

then show \(\exists n. (\neg P (X n) \land X n < b) \land (X n \leq X x \land Suc x \leq n)\)
  by (auto dest!: not-eventuallyD)

qed

then obtain \(s\) where \(?P s\)

with \(X\) have \(a < X (s n)\)
  and \(X (s n) < b\)
  and \(\text{decseq} (\lambda n. X (s n))\)
  and \(\lambda n. X (s n) \longrightarrow a\)
  and \(\neg P (X (s n))\)
  for \(n\)
by (auto simp: strict-mono-Suc-iff Suc-le-eq decseq-Suc-iff
  intro!: LIMSEQ-subseq-LIMSEQ[OF \(\langle X \longrightarrow a \rangle\), unfolded comp-def])

from \(\ast\)[OF this \(\langle 1, 2, 3, 4 \rangle\)] this \(\langle 5 \rangle\) show False
by auto

qed

lemma tendsto-at-right-sequentially:
fixes \(a :: \cdot :: \{\text{linorder-topology, first-countable-topology}\}\)
assumes \(a < b\)
and \(\ast:: \bigwedge S. (\bigwedge n. a < S n) \implies (\bigwedge n. S n < b) \implies \text{decseq} S \implies S \longrightarrow a\)
shows \((X \longrightarrow L)\) (at-right \(a\))
using assms by (simp add: tendsto-def[where \(l=L\)] sequentially-imp-eventually-at-right)

99.10 Continuity
99.10.1 Continuity on a set

definition continuous-on :: 'a set \Rightarrow ('a::topological-space \Rightarrow 'b::topological-space)
  \Rightarrow bool
where continuous-on s f \iff (\forall x\in s. (f \longrightarrow f x) \text{ (at } x \text{ within } s))

lemma continuous-on-cong [cong]:
  \(s = t \implies (\bigwedge x. x \in t \implies f x = g x) \implies \text{continuous-on } s f \iff \text{continuous-on } t g\)
unfolding continuous-on-def
by (intro ball-cong filterlim-cong) (auto simp: eventually-at-filter)

lemma continuous-on-cong-simp:
  \(s = t \implies (\bigwedge x. x \in t =\text{simp=> } f x = g x) \implies \text{continuous-on } s f \iff \text{continuous-on } t g\)
unfolding simp-implies-def by (rule continuous-on-cong)

lemma continuous-on-topological:
  continuous-on s f \iff
  (\forall x\in s. \forall B. \text{open } B \implies f x \in B \implies (\exists A. \text{open } A \land x \in A \land (\forall y\in s. y \in A \implies f y \in B))))
unfolding continuous-on-def tendsto-def eventually-at-topological by metis

lemma continuous-on-open-invariant:
continuous-on s f \iff (\\forall B. open B \longrightarrow (\exists A. open A \land A \cap s = f^{-1} B \cap s))

proof safe
fix B :: 'b set
assume continuous-on s f open B
then have \forall x\in f^{-1} B \cap s. (\exists A. open A \land x \in A \land s \subseteq f^{-1} B)
by (auto simp: continuous-on-topological subset-eq Ball-def imp-conjL)
then obtain A where \forall x\in f^{-1} B \cap s. open (A x) \land x \in A x \land s \subseteq f^{-1} B
unfolding bchoice-iff ..
then show \exists A. open A \land A \cap s = f^{-1} B \cap s
by (intro exI[of - \bigcup x\in f^{-1} B \cap s. A x]) auto

next
assume B: \forall B. open B \longrightarrow (\exists A. open A \land A \cap s = f^{-1} B \cap s)
show continuous-on s f
unfolding continuous-on-topological
proof safe
fix x B
assume x \in s open B f x \in B
with B obtain A where A: open A \land s = f^{-1} B \cap s
by auto
with \langle x \in s, f x \in B, show \exists A. open A \land x \in A \land (\forall y\in s, y \in A \longrightarrow f y \in B)
by (intro exI[of - A]) auto
qed

lemma continuous-on-open-vimage:
open s \Longrightarrow continuous-on s f \iff (\forall B. open B \longrightarrow open (f^{-1} B \cap s))
unfolding continuous-on-open-invariant
by (metis open-Int Int-absorb Int-commute[of s] Int-assoc[of - - s])

corollary continuous-imp-open-vimage:
assumes continuous-on s f open s open B f^{-1} B \subseteq s
shows open (f^{-1} B)
by (metis assms continuous-on-open-vimage le-iff-inf)

corollary open-vimage[continuous-intros]:
assumes open s
and continuous-on UNIV f
shows open (f^{-1} s)
using assms by (simp add: continuous-on-open-vimage [OF open-UNIV])

lemma continuous-on-closed-invariant:
continuous-on s f \iff (\forall B. closed B \longrightarrow (\exists A. closed A \land A \cap s = f^{-1} B \cap s))

proof
have *: \langle A. P A \iff Q (\neg A) \rangle \longrightarrow (\forall A. P A) \iff (\forall A. Q A)
for P Q :: 'b set ⇒ bool
by (metis double-compl)
show ´thesis
  unfolding continuous-on-open-invariant
  by (intro *) (auto simp: open-closed[symmetric])
qed

lemma continuous-on-closed-vimage:
closed s =⇒ continuous-on s f ←→ (∀ B. closed B =⇒ closed (f -' B ∩ s))
  unfolding continuous-on-closed-invariant
  by (metis closed-Int Int-absorb Int-commute[of s] Int-assoc[of - s - s])
corollary closed-vimage-Int[continuous-intros]:
  assumes closed s
  and continuous-on t f
  and t: closed t
  shows closed (f -' s ∩ t)
  using assms by (simp add: continuous-on-closed-vimage [OF t])
corollary closed-vimage[continuous-intros]:
  assumes closed s
  and continuous-on UNIV f
  shows closed (f -' s)
  using closed-vimage-Int [OF assms] by simp
lemma continuous-on-empty [simp]: continuous-on {} f
  by (simp add: continuous-on-def)
lemma continuous-on-sing [simp]: continuous-on {x} f
  by (simp add: continuous-on-def at-within-def)
lemma continuous-on-open-Union:
  (∀s. s ∈ S =⇒ open s) =⇒ (∀s. s ∈ S =⇒ continuous-on s f) =⇒ continuous-on
  (∪S) f
  unfolding continuous-on-def
  by safe (metis open-Union at-within-open UnionI)
lemma continuous-on-open-UN:
  (∀s. s ∈ S =⇒ open (A s)) =⇒ (∀s. s ∈ S =⇒ continuous-on (A s) f) =⇒
  continuous-on (∪s∈S. A s) f
  by (rule continuous-on-open-Union) auto
lemma continuous-on-open-Un:
  open s =⇒ open t =⇒ continuous-on s f =⇒ continuous-on t f =⇒ continuous-on
  (s ∪ t) f
  using continuous-on-open-Union [of {s,t}] by auto
lemma continuous-on-closed-Un:
closed s =⇒ closed t =⇒ continuous-on s f =⇒ continuous-on t f =⇒ continu-
ous-on \((s \cup t) f\)

by (auto simp add: continuous-on-closed-vimage closed-Un Int-Un-distrib)

lemma continuous-on-closed-Union:
assumes finite I
\[ \forall i. \ i \in I \Longrightarrow \text{closed } (U i) \]
\[ \forall i. \ i \in I \Longrightarrow \text{continuous-on } (U i) f \]
shows continuous-on \((\bigcup i \in I. \ U i) f\)
using assms
by (induction I) (auto intro: continuous-on-closed-Un)

lemma continuous-on-If:
assumes closed: closed s closed t
and cont: continuous-on s f continuous-on t g
and \(P\):
\[ \forall x. \ x \in s \Longrightarrow \neg P x \Longrightarrow f x = g x \]
\[ \forall x. \ x \in t \Longrightarrow P x \Longrightarrow f x = g x \]
s shows continuous-on \((s \cup t) \ (\lambda x. \text{if } P x \text{ then } f x \text{ else } g x)\)
(is continuous-on - ?h)
proof -
from P have \(\forall x \in s. \ f x = ?h x \ \forall x \in t. \ g x = ?h x\)
by auto
with cont have continuous-on s ?h continuous-on t ?h
by simp-all
with closed show ?thesis
by (rule continuous-on-closed-Un)
qed

lemma continuous-on-cases:
closed s \(\Longrightarrow\) closed t \(\Longrightarrow\) continuous-on s f \(\Longrightarrow\) continuous-on t g \(\Longrightarrow\)
\[ \forall x. \ (x \in s \land \neg P x) \lor (x \in t \land P x) \Longrightarrow f x = g x \]
continuous-on \((s \cup t) \ (\lambda x. \text{if } P x \text{ then } f x \text{ else } g x)\)
by (rule continuous-on-If) auto

lemma continuous-on-id[continuous-intros,simp]: continuous-on s \((\lambda x. \ x)\)
unfolding continuous-on-def by fast

lemma continuous-on-id'[continuous-intros,simp]: continuous-on s id
unfolding continuous-on-id-def by fast

lemma continuous-on-const[continuous-intros,simp]: continuous-on s \((\lambda x. \ c)\)
unfolding continuous-on-def by auto

lemma continuous-on-subset: continuous-on s f \(\Longrightarrow\) t \(\subseteq\) s \(\Longrightarrow\) continuous-on t f
unfolding continuous-on-def
by (metis subset-eq tendsto-within-subset)

lemma continuous-on-compose[continuous-intros]:
continuous-on s f \(\Longrightarrow\) continuous-on \((f \circ s) g\) \(\Longrightarrow\) continuous-on s \((g \circ f)\)
unfolding continuous-on-topological by simp metis
**THEORY “Topological-Spaces”**

**lemma** continuous-on-compose2:
\[\text{continuous-on } t \ g \implies \text{continuous-on } s \ f \implies f^{-1} s \subseteq t \implies \text{continuous-on } s \ (\lambda x. g \ (f \ x))\]

using continuous-on-compose \([of \ s \ g] \text{ continuous-on-subset by (force simp add: comp-def)}\)

**lemma** continuous-on-generate-topology:
assumes \(*\): \text{open } = \text{generate-topology } X
and \(**\): \(\forall A. B \subseteq X \implies \exists C. \text{open } C \cap A = f^{-1} B \cap A\)
shows \text{continuous-on } A f

**proof** safe
fix \(B :: \text{a set}\)
assume \text{open } \(B\)
then show \(\exists C. \text{open } C \land C \cap A = f^{-1} B \cap A\)
unfolding \(*\)
**proof** induct
case (UN \(K\))
then obtain \(C\) where \(\forall k. k \in K \implies \text{open } C_k\)
by (metis)
then show \("case\"
by (intro exI \[, of - \(\bigcup k \in K. C_k\)\])

**qed** (auto intro: \(**\))

**qed**

**lemma** continuous-onI-mono:
fixes \(f :: \text{linorder-topology} \implies \text{linorder-topology} \{dense-order, linorder-topology\}\)
assumes \text{open } \(f' A\)
and \text{mono } \(\forall x y. x \in A \implies y \in A \implies x \leq y \implies f x \leq f y\)
shows \text{continuous-on } A f

**proof** (rule continuous-on-generate-topology \([OF \text{open-generated-order}, safe]\)
have \text{monoD } \(\forall x y. x \in A \implies y \in A \implies f x < f y \implies x < y\)
by (auto simp \[\text{not-le [symmetric]}\] \text{mono})
have \(\exists x. x \in A \land f x < b \land a < x\) if \(a \in A\) and \(fa < b\) for \(a b\)
**proof**
  obtain \(y\) where \(fa < y \{fa..<y\} \subseteq f' A\)
  using open-right \([OF \text{open } (f' A)], of fa b\) \(fa\)
  by auto
  obtain \(z\) where \(z : f a < z \ z < \min b y\)
  using dense \([of \text{fa min b y}] \{fa < y\} \{fa < b\}\) by auto
  then obtain \(c\) where \(z = f c c \in A\)
  using \(\{fa..<y\} \subseteq f' A\) \[\text{THEN subsetD, of z}\] by (auto simp \[\text{less-imp-le}\])
  with \(a z\) show \"thesis\"
  by (auto intro!: exI \[of - c\] simp: monoD)
**qed**
then show \(\exists C. \text{open } C \cap A = f^{-1} \{.. < b\} \cap A \text{ for } b\)
by (intro exI \[of \(\bigcup \{x \in A. f x < b\}. {..<x}\}\])
(auto intro: le-less-trans \[OF mono\] \text{less-imp-le})
have $\exists x. x \in A \land b < f x \land x < a$ if $a: a \in A$ and $fa: b < f a$ for $a \ b$

proof --

note $a \ fa$

moreover

obtain $y$ where $y < f a \ \{y <.. f a\} \subseteq f' A$

using $open\left(OF \ \cdot \ \cdot \ \cdot \ \left(f' A\right)\right)$, $of f a b$ $a \ fa$

by auto

then obtain $z$ where $z: \max b \ y \ z < f a$

using $\{y <.. f a\} \subseteq f'A$ \[THEN \ subsetD, of z\] by (auto simp: $less\_imp\_le$

with $a \ z$ show $?thesis$

by (auto intro $!$: exI $[of - c$ $simp$: monoD))

qed

lemma continuous-on-IccI:

\[(f \longrightarrow f a) \ (at-right a); \]

\[(f \longrightarrow f b) \ (at-left b); \]

\[\langle \forall x. a < x \Longrightarrow x < b \Longrightarrow f \ -x \rightarrow f x; a < b \rangle \Longrightarrow \]

continuous-on $\{a \ .. \ b\} \ f$

for $a: a::linorder\_topology$

using $at\_\_within\_\_open[of $ - \{a <.. \ b\}$]

by (auto simp: continuous-on-def at-within-Icc-at-right at-within-Icc-at-left $le\_less$

at-within-Icc-at)

lemma fixes $a \ b::a::linorder\_topology$

assumes continuous-on $\{a \ .. \ b\} \ f \ a < b$

shows continuous-on-Icc-at-rightD: $(f \longrightarrow f a) \ (at-right a)$

and continuous-on-Icc-at-leftD: $(f \longrightarrow f b) \ (at-left b)$

using assms

by (auto simp: at-within-Icc-at-at within-Icc-at-at continuous-on-def

dest: bspec\[where $x=a$] bspec\[where $x=b$])

lemma continuous-on-discrete $[simp]$:

\[\text{continuous-on} \ A \ (f :: a :: discrete\_topology} \Rightarrow \)

by (auto simp: continuous-on-def at-discrete)

lemma continuous-on-of-nat $[continuous\_intros]$:

assumes continuous-on $A \ f$

shows continuous-on $A \ (\lambda n. \ of\_nat \ (f n))$

using continuous-on-compose\[OF \ \text{assms \ continuous-on\_discrete[of - of\_nat]]\]

by (simp add: a-def)
lemma continuous-on-of-int [continuous-intros]:
assumes continuous-on A f
shows  continuous-on A (λn. of-int (f n))
using continuous-on-compose[OF assms continuous-on-discrete[of - of-int]]
by (simp add: o-def)

99.10.2 Continuity at a point

definition continuous :: 'a::t2-space filter ⇒ ('a ⇒ 'b::topological-space) ⇒ bool
where  continuous F f = (f → f (Lim F (λx. x))) F

lemma continuous-bot [continuous-intros, simp]: continuous bot f
unfolding continuous-def by auto

lemma continuous-trivial-limit: trivial-limit net ⇒ continuous net f
by simp

lemma continuous-within: continuous (at x within s) f =⇒ (f → f x) (at x within s)
by (cases trivial-limit (at x within s)) (auto simp add: Lim-ident-at continuous-def)

lemma continuous-within-topological:
continuous (at x within s) f =⇒
(∀ B. open B ⇒ f x ∈ B ⇒ (∃ A. open A ∧ x ∈ A ∧ (∀ y∈s. y ∈ A → f y ∈ B)))
unfolding continuous-within tendsto-to-topological by metis

lemma continuous-within-compose [continuous-intros]:
continuous (at x within s) f =⇒ continuous (at (f x) within f ' s) g =⇒
continuous (at x within s) (g o f)
by (simp add: continuous-within-topological) metis

lemma continuous-within-compose2:
continuous (at x within s) f =⇒ continuous (at (f x) within f ' s) g =⇒
continuous (at x within s) (λx. g (f x))
using continuous-within-compose[of x s f g] by (simp add: comp-def)

lemma continuous-at: continuous (at x) f =⇒ f → f x
using continuous-within[of x UNIV f] by simp

lemma continuous-ident [continuous-intros, simp]: continuous (at x within S) (λx. x)
unfolding continuous-within by (rule tendsto-ident-at)

lemma continuous-id [continuous-intros, simp]: continuous (at x within S) id
by (simp add: id-def)

lemma continuous-const [continuous-intros, simp]: continuous F (λx. c)
unfolding continuous-def by (rule tendsto-const)

lemma continuous-on-eq-continuous-within:
  continuous-on s f = (\forall x \in s. continuous (at x within s) f)
unfolding continuous-on-def continuous-within ..

lemma continuous-discrete [simp]:
  continuous (at x within A) (f::'a::discrete-topology ⇒ -)
  by (auto simp: continuous-def at-discrete)

abbreviation isCont :: ('a::t2-space ⇒ 'b::topological-space) ⇒ 'a ⇒ bool
where
  isCont f a = continuous (at a) f

lemma isCont-def: isCont f a = f − a → f a
  by (rule continuous-at)

lemma isContD: isCont f x =⇒ f − x → f x
  by (simp add: isCont-def)

lemma isCont-cong:
  assumes eventually (λx. f x = g x) (nhds x)
  shows isCont f x =⇒ isCont g x
proof −
  from assms have [simp]: f x = g x
    by (rule eventually-nhds-x-imp-x)
  from assms have eventually (λx. f x = g x) (at x)
    by (auto simp: eventually-at-filter elim!: eventually-mono)
  with assms have isCont f x =⇒ isCont g x unfolding isCont-def
    by (intro filterlim-cong) (auto elim!: eventually-mono)
  with assms show ?thesis by simp
qed

lemma continuous-at-imp-continuous-at-within: isCont f x =⇒ continuous (at x within s) f
  by (auto intro: tendsto-mono at-le simp: continuous-on-eq-continuous-within)

lemma continuous-on-eq-continuous-at: open s =⇒ continuous-on s f = (\forall x \in s. isCont f x)
  by (simp add: continuous-on-def continuous-at at-within-open[of - s])

lemma continuous-within-open: a ∈ A =⇒ open A =⇒ continuous (at a within A) f =⇒ isCont f a
  by (simp add: at-within-open-NO-MATCH)

lemma continuous-at-imp-continuous-on: ∀ x ∈ s. isCont f x =⇒ continuous-on s f
  by (auto intro: continuous-at-imp-continuous-at-within simp: continuous-on-eq-continuous-within)

lemma isCont-o2: isCont f a =⇒ isCont g (f a) =⇒ isCont (λx. g (f x)) a
  unfolding isCont-def by (rule tendsto-compose)
lemma continuous-at-compose[continuous-intros]: isCont f a  \implies  isCont (g o f) a
  unfolding o-def by (rule isCont-o2)

lemma isCont-tendsto-compose: isCont g l  \implies  (f \longrightarrow l) F  \implies  ((\lambda x. g (f x)) \longrightarrow g l) F
  unfolding isCont-def by (rule tendsto-compose)

lemma continuous-on-tendsto-compose:
  assumes f-cont: continuous-on s f
  and g: (g \longrightarrow l) F
  and l: l \in s
  and ev: \forall F x in F. g x \in s
  shows ((\lambda x. f (g x)) \longrightarrow f l) F
  proof
  - from f-cont l have f: (f \longrightarrow f l) (at l within s)
    by (simp add: continuous-on-def)
  have i: ((\lambda x. if g x = l then f l else f (g x)) \longrightarrow f l) F
    by (rule filterlim-If)
    (auto intro! : filterlim-compose[OF f] eventually-conj tendsto-mono[OF - g]
      simp: filterlim-at eventually-inf-principal eventually-mono[OF ev])
  show ?thesis
    by (rule filterlim-cong[THEN iffD1[OF - i]]) auto
  qed

lemma continuous-within-compose3:
  isCont g (f x)  \implies  continuous (at x within s) f  \implies  continuous (at x within s) (\lambda x. g (f x))
  using continuous-at-imp-continuous-at-within continuous-within-compose2 by blast

lemma at-within-isCont-imp-nhds:
  fixes f:: 'a:: {t2-space,perfect-space} \Rightarrow 'b:: t2-space
  assumes \forall F w in at z. f w = g w isCont f z isCont g z
  shows \forall F w in nhds z. f w = g w
  proof
  - have g -z\longrightarrow f z
    using assms isContD tendsto-cong by blast
  moreover have g -z\longrightarrow g z using isCont g z using isCont-def by blast
  ultimately have f z=z g z using LIM-unique by auto
  moreover have \forall F x in nhds z. x \neq z \longrightarrow f x = g x
    using assms unfolding eventually-at-filter by auto
  ultimately show ?thesis
    by (auto elim: eventually-mono)
  qed

lemma filtermap-nhds-open-map4:
  assumes cont: isCont f a
and open $A$ $a \in A$
and open-map: $\forall S. \text{open } S \implies S \subseteq A \implies \text{open } (f \circ S)$
shows filtermap $f$ (nhds $a$) = nhds ($f$ $a$)
unfolding filter-eq-iff
proof safe
  fix $P$
  assume eventually $P$ (filtermap $f$ (nhds $a$))
  then obtain $S$ where $S$: open $S$ $a \in S \forall x\in S. P (f x)$
  by (auto simp: eventually-filtermap eventually-nhds)
show eventually $P$ (nhds ($f$ $a$))
  unfolding eventually-nhds
proof (rule exI [of -$f' (A \cap S)$], safe)
  show open ($f' (A \cap S)$)
    using $S$ by (intro open-Int assms auto)
  show $f a \in f' (A \cap S)$
    using assms $S$ by auto
  show $P (f x)$ if $x \in A$ $x \in S$ for $x$
    using $S$ that by auto
qed
qed (metis filterlim-iff tendsto-at-iff-tendsto-nhds isCont-def eventually-filtermap cont)

lemma filtermap-nhds-open-map:
  assumes cont: isCont $f$ $a$
  and open-map: $\forall S. \text{open } S \implies \text{open } (f' S)$
shows filtermap $f$ (nhds $a$) = nhds ($f$ $a$)
using cont filtermap-nhds-open-map
by blast

lemma continuous-at-split:
  continuous (at $x$) $f$ $\iff$ continuous (at-left $x$) $f$ $\land$ continuous (at-right $x$) $f$
for $x :: 'a::linorder-topology$
by (simp add: continuous-within filterlim-at-split)

lemma continuous-on-max [continuous-intros]:
  fixes $f$ $g :: 'a::topological-space $\Rightarrow$ 'b::linorder-topology
shows continuous-on $A$ $f$ $\implies$ continuous-on $A$ $g$ $\implies$ continuous-on $A$ ($\lambda x. \max (f x) (g x)$)
by (auto simp: continuous-on-def intro!: tendsto-max)

lemma continuous-on-min [continuous-intros]:
  fixes $f$ $g :: 'a::topological-space $\Rightarrow$ 'b::linorder-topology
shows continuous-on $A$ $f$ $\implies$ continuous-on $A$ $g$ $\implies$ continuous-on $A$ ($\lambda x. \min (f x) (g x)$)
by (auto simp: continuous-on-def intro!: tendsto-min)

lemma continuous-max [continuous-intros]:
  fixes $f :: 'a::t2-space $\Rightarrow$ 'b::linorder-topology
shows [continuous $F$ $f$; continuous $F$ $g$] $\implies$ continuous $F$ ($\lambda x. \max (f x) (g x)$)
THEORY “Topological-Spaces”

by (simp add: tendsto-max continuous-def)

lemma continuous-min [continuous-intros]:
  fixes f :: 'a::t2-space ⇒ 'b::linorder-topology
  shows [continuous F f; continuous F g] ⇒ continuous F (λx. (min (f x) (g x)))
  by (simp add: tendsto-min continuous-def)

The following open/closed Collect lemmas are ported from Sébastien Gouëzel’s Ergodic-Theory.

lemma open-Collect-neq:
  fixes f g :: 'a::topological-space ⇒ 'b::t2-space
  assumes f: continuous-on UNIV f and g: continuous-on UNIV g
  shows open {x. f x ≠ g x}
  proof (rule openI)
    fix t
    assume t: t ∈ {x. f x ≠ g x}
    then obtain U V where *
      open U open V f t ∈ U g t ∈ V U ∩ V = {}
    by (auto simp add: separation-t2)
    with open-vimage[OF ‹open U› f]
    open-vimage[OF ‹open V› g]
    show ∃ T. open T ∧ t ∈ T ∧ T ⊆ {x. f x ≠ g x}
    by (intro exI[OF - f − ' U ∩ g − ' V]) auto
  qed

lemma closed-Collect-eq:
  fixes f g :: 'a::topological-space ⇒ 'b::t2-space
  assumes f: continuous-on UNIV f and g: continuous-on UNIV g
  shows closed {x. f x = g x}
  using open-Collect-neq[OF f g]
  by (simp add: closed-def Collect-neg-eq)

lemma open-Collect-less:
  fixes f g :: 'a::topological-space ⇒ 'b::linorder-topology
  assumes f: continuous-on UNIV f and g: continuous-on UNIV g
  shows open {x. f x < g x}
  proof (rule openI)
    fix t
    assume t: t ∈ {x. f x < g x}
    show ∃ z where f t < z ∧ z < g t by blast
    then show ?thesis
    using open-vimage[OF - f, of {..< z}]
    open-vimage[OF - g, of {z..<}]
    by (intro exI[of - f − ' {..< z} ∩ g − ' {z..<}]) auto
  next
  case False
  then have *: {g t ..} = {f t <..} {..< g t} = {.. f t}
  using t by (auto intro: leI)
  show ?thesis
  using open-vimage[OF - f, of {..< g t}]
  open-vimage[OF - g, of {f t <..}]

apply (intro exI[of - f ‑ '{..< g t} ∩ g ‑ '{f t..<}])
apply (simp add: open-Int)
apply (auto simp add: *)
done
qed

lemma closed-Collect-le:
  fixes f g :: 'a :: topological-space ⇒ 'b::linorder-topology
  assumes f: continuous-on UNIV f
    and g: continuous-on UNIV g
  shows closed {x. f x ≤ g x}
  using open-Collect-less [OF g f]
  by (simp add: closed-def Collect-neg-eq [symmetric] not-le)

99.10.3 Open-cover compactness

context topological-space
begin

definition compact :: 'a set ⇒ bool where
  compact-eq-Heine-Borel:
  compact S ←→ (∀C. (∀c∈C. open c) ∧ S ⊆ ⋃C → (∃D⊆C. finite D ∧ S ⊆ ⋃D))

lemma compactI:
  assumes ‚∀C. ∀t∈C. open t ⇒ s ⊆ ⋃C ⇒ ∃C′ ⊆ C ∧ finite C′ ∧ s ⊆ ⋃C′
  shows compact s
  unfolding compact-eq-Heine-Borel using assms by metis

lemma compact-empty[simp]: compact {}
  by (auto intro!: compactI)

lemma compactE:
  assumes compact S S ⊆ ⋃T ∧ B. B ∈ T ⇒ open B
  obtains T′ where T′ ⊆ T finite T′ ∧ S ⊆ ⋃T′
  by (meson assms compact-eq-Heine-Borel)

lemma compactE-image:
  assumes compact S
    and opn: ∃T. T ∈ C ⇒ open (f T)
    and S: S ⊆ (∪c∈C. f c)
  obtains C′ where C′ ⊆ C and finite C′ and S ⊆ (⋃c∈C′. f c)
  apply (rule compactE[OF ‚compact S ‚S])
  using opn apply force
  by (metis finite-subset-image)

lemma compact-Int-closed [intro]:
assumes compact $S$
and closed $T$
shows compact $(S \cap T)$

proof (rule compactI)
fix $C$
assume $C : \forall c \in C. \text{open } c$
assume cover: $S \cap T \subseteq \bigcup C$
from $C$ (closed $T$) have $\forall c \in C \cup \{-T\}. \text{open } c$
by auto
moreover from cover have $S \subseteq \bigcup (C \cup \{-T\})$
by auto
ultimately have $\exists D \subseteq C \cup \{-T\}. \text{finite } D \wedge S \subseteq \bigcup D$
using (compact $S$) unfolding compact-eq-Heine-Borel by auto
then obtain $D$ where $D \subseteq C \cup \{-T\} \wedge \text{finite } D \wedge S \cap T \subseteq \bigcup D$
by (intro exI[of $D - \{-T\}$]) auto
qed

lemma compact-diff: $\text{compact } S; \text{open } T \Rightarrow \text{compact } (S - T)$
by (simp add: Diff-eq compact-Int-closed open-closed)

lemma inj-setminus: inj-on uminus $(A::'a \text{ set set})$
by (auto simp: inj-on-def)

99.11 Finite intersection property

lemma compact-fip:
$\text{compact } U \iff$
$(\forall A. (\forall a \in A. \text{closed } a) \longrightarrow (\forall B \subseteq A. \text{finite } B \longrightarrow U \cap \bigcap B \neq \{\})) \longrightarrow U \cap \bigcap A \neq \{\}$
(is -$\leftarrow -$ $\Rightarrow$ $\exists R$)

proof (safe intro!: compact-eq-Heine-Borel[THEN iffD2])
fix $A$
assume compact $U$
assume $A : \forall a \in A. \text{closed } a \wedge A \cap \bigcap A \neq \{\}$
assume fin: $\forall B \subseteq A. \text{finite } B \longrightarrow U \cap \bigcap B \neq \{\}$
from $A$ have $(\forall a \in \text{uminus}A. \text{open } a) \wedge U \subseteq \bigcup (\text{uminus}A)$
by auto
with (compact $U$) obtain $B$ where $B \subseteq A \text{ finite } (\text{uminus}B) \wedge U \subseteq \bigcup (\text{uminus}B)$
unfolding compact-eq-Heine-Borel by (metis subset-image-iff)
with fin[THEN spec. of $B$] show False
by (auto dest: finite-imageD intro: inj-setminus)
next
fix $A$
assume $\exists R$
assume $\forall a \in A. \text{open } a \wedge U \subseteq \bigcup A$
then have $U \cap \bigcap (\text{uminus}A) = \{\} \forall a \in \text{uminus}A. \text{closed } a$
by auto
with $\exists R$ obtain $B$ where $B \subseteq A \text{ finite } (\text{uminus}B) \wedge U \cap \bigcap (\text{uminus}B) = \{\}$
by (metis subset-image-iff)
then show \( \exists T \subseteq A. \text{finite } T \land U \subseteq \bigcup T \)
  by (auto intro!: exI[of - B] inj-setminus dest: finite-imageD)
qed

lemma compact-imp-fip:
  assumes compact S
  and \( \bigwedge T. T \in F \Longrightarrow \text{closed } T \)
  and \( \bigwedge F'. \text{finite } F' \Longrightarrow F' \subseteq F \Longrightarrow S \cap (\bigcap F') \neq \{\} \)
  shows \( S \cap (\bigcap F) \neq \{\} \)
using assms unfolding compact-fip by auto

lemma compact-imp-fip-image:
  assumes compact s
  and P: \( \bigwedge i. i \in I \Longrightarrow \text{closed } (f i) \)
  and Q: \( \bigwedge I'. \text{finite } I' \Longrightarrow I' \subseteq I \Longrightarrow (s \cap (\bigcap i \in I'). f i) \neq \{\} \)
  shows \( s \cap (\bigcap f \cdot I) \neq \{\} \)
proof -
  from P have \( \forall i \in f \cdot I. \text{closed } i \)
    by blast
  moreover have \( \forall A. \text{finite } A \land A \subseteq f \cdot I \Longrightarrow (s \cap (\bigcap A) \neq \{\}) \)
    by (metis Q finite-subset-image)
  ultimately show \( s \cap (\bigcap f \cdot I) \neq \{\} \)
    by (metis (compact s) compact-imp-fip)
qed

end

lemma (in t2-space) compact-imp-closed:
  assumes compact s
  shows closed s
unfolding closed-def proof (rule openI)
fix y
assume y \( \in - s \)
let \(?C = \bigcup x \in s. \{u. \text{open } u \land x \in u \land \text{eventually } (\lambda y. y \notin u) \ (\text{nhds } y)\}\)
have s \( \subseteq \bigcup ?C \)
proof
  fix x
  assume x \( \in s \)
  with \( y \in - s \) have \( x \neq y \) by clarsimp
  then have \( \exists u v. \text{open } u \land \text{open } v \land x \in u \land y \in v \land u \cap v = \{\} \)
    by (rule hausdorff)
  with \( x \in s \) show \( x \in \bigcup ?C \)
    unfolding eventually-nhds by auto
qed

then obtain D where D \( \subseteq ?C \) and finite D and s \( \subseteq \bigcup D \)
  by (rule compactE [OF (compact s)] auto)
from \( D \subseteq ?C \) have \( \forall x \in D. \text{eventually } (\lambda y. y \notin x) \ (\text{nhds } y) \)

by auto
with \( \langle \text{finite } D \rangle \) have eventually \( (\lambda y. y \notin \bigcup D) \) (nhds y)
  by (simp add: eventually-ball-finite)
with \( s \subseteq \bigcup D \) have eventually \( (\lambda y. y \notin s) \) (nhds y)
  by (auto elim!: eventually-mono)
then show \( \exists t. \text{open } t \land y \in t \land t \subseteq - s \)
  by (simp add: eventually-nhds subset-eq)
qed

lemma compact-continuous-image:
  assumes \( f \colon \text{continuous-on } s f \)
  and \( s \colon \text{compact } s \)
  shows \( \text{compact } (f' s) \)
proof (rule compactI)
  fix \( C \)
  assume \( \forall c \in C. \text{open } c \) and cover: \( f's \subseteq \bigcup C \)
  with \( f \) have \( \forall c \in C. \exists A. \text{open } A \land A \cap s = f' c \cap s \)
  unfolding continuous-on-open-invariant by blast
  then obtain \( A \) where \( \forall c \in C. \text{open } A \land A \cap s = f' c \cap s \)
  unfolding bchoice-iff ..
  with cover have \( \forall c \in C \Longrightarrow \text{open } (A c) \land A c \cap s = f' c \cap s \)
  by (fastforce simp add: subset-eq set-eq-iff)
  from compactE-image[OF s this] obtain \( D \) where \( D \subseteq C \) finite \( D s \subseteq (\bigcup c \in C. A c) \).
  with \( A \) show \( \exists D \subseteq C. \text{finite } D \land f's \subseteq \bigcup D \)
  by (intro exI[of - D]) (fastforce simp add: subset-eq set-eq-iff)
qed

lemma continuous-on-inv:
  fixes \( \cdot a \colon \text{topological-space} \Rightarrow \cdot b \colon \text{t2-space} \)
  assumes \( \text{continuous-on } s f \)
  and \( s \colon \text{compact } s \)
  and \( \forall x \in s. g(f x) = x \)
  shows \( \text{continuous-on } (f' s) g \)
unfolding continuous-on-topological
proof (clarsimp simp add: assms(3))
  fix \( x \colon \cdot a \) and \( B \colon \cdot a \text{ set} \)
  assume \( x \in s \) and \( \text{open } B \) and \( x \in B \)
  have \( \forall x \in s. f x \in f' (s - B) \iff x \in s - B \)
    using assms(3) by (auto,metis)
  have \( \text{continuous-on } (s - B) f \)
    using \( \text{continuous-on } s f \) Diff-subset
    by (rule continuous-on-subset)
  moreover have \( \text{compact } (s - B) \)
    using \( \text{open } B \) and \( \text{compact } s \)
    unfolding Diff-eq by (intro compact-Int-closed closed-Compl)
  ultimately have \( \text{compact } (f' (s - B)) \)
    by (rule compact-continuous-image)
  then have \( \text{closed } (f' (s - B)) \)
by (rule compact-imp-closed)
then have open \((- f' (s - B))\)
by (rule open-Compl)
moreover have \(f x \in - f' (s - B)\)
  using \(\langle x \in s \rangle\) and \(\langle x \in B \rangle\) by (simp add: 1)
moreover have \(\forall y \in s. f y \in - f' (s - B) \rightarrow y \in B\)
  by (simp add: 1)
ultimately show \(\exists A. \text{open } A \land f x \in A \land (\forall y \in s. f y \in A \rightarrow y \in B)\)
by fast
qed

lemma continuous-on-inv-into:
fixes \(f :: 'a::topological-space \Rightarrow 'b::t2-space\)
assumes \(s\): continuous-on \(s\) \(f\) compact \(s\)
and \(f\): inj-on \(f\) \(s\)
shows continuous-on \((f' s)\) \((\text{the-inv-into } s\ f)\)
by (rule continuous-on-inv[OF \(s\)])(auto simp: the-inv-into-f-f[OF \(f\)])

lemma (in linorder-topology) compact-attains-sup:
assumes compact \(S\) \(S\neq\{\}\)
shows \(\exists s \in S. \forall t \in S. t \leq s\)
proof (rule classical)
assume \(\neg (\exists s \in S. \forall t \in S. t \leq s)\)
then obtain \(t\) where \(t\): \(\forall s \in S. t s \in S\) and \(\forall s \in S. s \leq t s\)
  by (metis not-le)
then have \(\forall s. s \in S \Longrightarrow \text{open } \{..< t s\} S \subseteq (\bigcup s \in S. \{..< t s\})\)
  by auto
with compact \(S\) obtain \(C\) where \(C\subseteq S\) finite \(C\) and \(C:: S \subseteq (\bigcup s \in C. \{..< t s\})\)
  by (metis compactE-image)
with \(\{s \neq \{\}\}\) have Max: Max \((t'C)\) \(\in t'C\) and \(\forall s \in t'C. s \leq \text{Max } (t'C)\)
  by (auto intro!: Max-in)
with \(C\) have \(S \subseteq \{..< \text{Max } (t'C)\}\)
  by (auto intro!: less-le-trans simp: subset-eq)
with \(t\) Max \(\langle C \subseteq S\rangle\) show ?thesis
  by fastforce
qed

lemma (in linorder-topology) compact-attains-inf:
assumes compact \(S\) \(S\neq\{\}\)
shows \(\exists s \in S. \forall t \in S. s \leq t\)
proof (rule classical)
assume \(\neg (\exists s \in S. \forall t \in S. s \leq t)\)
then obtain \(t\) where \(t\): \(\forall s \in S. t s \in S\) and \(\forall s \in S. t s \leq s\)
  by (metis not-le)
then have \(\forall s. s \in S \Longrightarrow \text{open } \{t s <..<\} S \subseteq (\bigcup s \in S. \{t s <..<\})\)
  by auto
with compact \(S\) obtain \(C\) where \(C\subseteq S\) finite \(C\) and \(C:: S \subseteq (\bigcup s \in C. \{t s <..<\})\)
by (metis compactE-image)

with \( S \neq \{\} \) have Min: Min \((t'C) \in t'C \land \forall s\in t'C. \text{Min}(t'C) \leq s \)
by (auto intro: Min-in)

with C have S \subseteq \{\text{Min}(t'C) <..\}
by (auto intro: le-less-trans simp: subset-eq)

with t Min \( t \subseteq S \); show ?thesis
by fastforce

qed

lemma continuous-attains-sup:
fixes f :: 'a::topological-space \Rightarrow 'b::linorder-topology
shows compact s \implies s \neq \{\} \implies continuous-on s f \implies \( \exists x\in S. \forall y\in S. f y \leq f x \)
using compact-attains-sup[of f ' s] compact-continuous-image[of s f] by auto

lemma continuous-attains-inf:
fixes f :: 'a::topological-space \Rightarrow 'b::linorder-topology
shows compact s \implies s \neq \{\} \implies continuous-on s f \implies \( \exists x\in S. \forall y\in S. f x \leq f y \)
using compact-attains-inf[of f ' s] compact-continuous-image[of s f] by auto

99.12 Connectedness

context topological-space
begin

definition connected S \leftrightarrow
\neg (\exists A B. open A \land open B \land S \subseteq A \cup B \land A \cap B \cap S = \{\} \land A \cap S \neq \{\} \land B \cap S \neq \{\})

lemma connectedI:
(\forall A B. open A \implies open B \implies A \cap U \neq \{\} \implies B \cap U \neq \{\} \implies A \cap B \cap U = \{\} \implies U \subseteq A \cup B \implies False)
\implies connected U
by (auto simp: connected-def)

lemma connected-empty [simp]: connected \{\}
by (auto intro!: connectedI)

lemma connected-sing [simp]: connected \{x\}
by (auto intro!: connectedI)

lemma connectedD:
connected A \implies open U \implies open V \implies U \cap V \cap A = \{\} \implies A \subseteq U \cup V
\implies U \cap A = \{\} \lor V \cap A = \{\}
by (auto simp: connected-def)
end

lemma connected-closed:
connected $s \iff$
\[
\neg \exists A B. \text{closed } A \land \text{closed } B \land s \subseteq A \cup B \land A \cap B \cap s = \{\} \land A \cap s \neq \{\}
\land B \cap s \neq \{\}
\]
apply (simp add: connected-def del: ex-simps, safe)
apply (drule-tac $x=-A$ in spec)
apply (drule-tac $x=-B$ in spec)
apply (fastforce simp add: closed-def [symmetric])
apply (drule-tac $x=-A$ in spec)
apply (drule-tac $x=-B$ in spec)
apply (fastforce simp add: open-closed [symmetric])
done

lemma connected-closedD:
\[
\begin{align*}
\{\text{connected } s; A \cap B \cap s = \{\}; s \subseteq A \cup B; \text{closed } A; \text{closed } B\} \implies A \cap s = \{\} \\
\lor B \cap s = \{\}
\end{align*}
\]
by (simp add: connected-closed)

lemma connected-Union:
assumes cs: $\forall s. s \in S \implies \text{connected } s$
and ne: $\bigcap S \neq \{\}$
shows connected($\bigcup S$)
proof (rule connectedI)
fix $A B$
assume $A$: open $A$ and $B$: open $B$ and Alap: $A \cap \bigcup S \neq \{\}$ and Blap: $B \cap \bigcup S \neq \{\}$
and disj: $A \cap B \cap \bigcup S = \{\}$ and cover: $\bigcup S \subseteq A \cup B$
have disjs: $\forall s. s \in S \implies A \cap B \cap s = \{\}$
using disj by auto
obtain $sa$ where $sa$: $sa \in S$ $A \cap sa \neq \{\}$
using Alap by auto
obtain $sb$ where $sb$: $sb \in S$ $B \cap sb \neq \{\}$
using Blap by auto
obtain $x$ where $x$: $\forall s. s \in S \implies x \in s$
using ne by auto
then have $x \in \bigcup S$
using $\langle sa \in S \rangle$ by blast
then have $x \in A \lor x \in B$
using cover by auto
then show False
using cs [unfolded connected-def]
by (metis A B IntI Sup-upper sa sb disjs x cover empty-iff subset-trans)
qed

lemma connected-Un: connected $s \implies$ connected $t \implies s \cap t \neq \{\} \implies$ connected ($s \cup t$)
using connected-Union [of $\{s,t\}$] by auto

lemma connected-diff-open-from-closed:
assumes st: $s \subseteq t$
and $t$: $t \subseteq u$
and $s$: open $s$
and $t$: closed $t$
and $u$: connected $u$
and $ts$: connected $(t - s)$
shows connected $(u - s)$
proof (rule connectedI)
fix $A B$
assume $A B$: open $A$ open $B$ $A \cap (u - s) \neq \{\} \quad B \cap (u - s) \neq \{\}$
and disj: $A \cap B \cap (u - s) = \{\}$
and cover: $u - s \subseteq A \cup B$
then consider $A \cap (t - s) = \{\} \quad B \cap (t - s) = \{\}$
using st $ts$ $tu$ connectedD [of $t - s \quad A \quad B$] by auto
then show False
proof cases
  case 1
  then have $(A - t) \cap (B \cup s) \cap u = \{\}$
  using disj $st$ by auto
  moreover have $u \subseteq (A - t) \cup (B \cup s)$
  using $t$ cover by auto
  ultimately show False
  using connectedD [of $u \quad A - t \quad B \cup s$] $AB \quad s \quad t \quad u$ by auto
  next
  case 2
  then have $(A \cup s) \cap (B - t) \cap u = \{\}$
  using disj $st$ by auto
  moreover have $u \subseteq (A \cup s) \cup (B - t)$
  using $2$ cover by auto
  ultimately show False
  using connectedD [of $u \quad A \cup s \quad B - t$] $AB \quad s \quad t \cup 2 \quad u$ by auto
  qed
  qed

lemma connected-iff-const:
  fixes $S \::\: \text{a::topological-space}$
  shows connected $S \iff (\forall P \::\: \text{a \Rightarrow bool}. \quad \text{continuous-on} \quad S \quad P \quad \rightarrow \quad (\exists c. \quad \forall s \in S. \quad P \quad s = \quad c))$
proof safe
  fix $P \::\: \text{a \Rightarrow bool}$
  assume connected $S$ continuous-on $S \quad P$
  then have $\bigwedge b. \quad \exists A. \quad \text{open} \quad A \quad \land \quad A \cap S = \quad P \quad - \quad \{b\} \quad \cap \quad S$
  unfolding continuous-on-open-invariant by (simp add: open-discrete)
  from this[of True] this[of False]
  obtain $t \quad f \quad$ where open $t$ open $f$ and $s$: $f \cap S = \quad P \quad - \quad \{\text{False}\} \quad \cap \quad S \quad t \cap S = \quad P$
  $- \quad \{\text{True}\} \quad \cap \quad S$
  by meson
  then have $t \cap S = \quad \{\} \quad \land \quad f \cap S = \quad \{\}$
  by (intro connectedD[of $\text{OF connected } S$]) auto
  then show $\exists c. \quad \forall s \in S. \quad P \quad s = \quad c$
proof (rule disjE)
  assume \( t \cap S = \{ \} \)
  then show ?thesis
    unfolding \(*\) by (intro exI[of - False]) auto
next
  assume \( f \cap S = \{ \} \)
  then show ?thesis
    unfolding \(*\) by (intro exI[of - True]) auto
qed

next
assume \( P : \forall P :: 'a \Rightarrow \text{bool} \). continuous-on S P \( \rightarrow \exists c. \forall s \in S. \ P s = c \)
show connected S
proof (rule connectedI)
  fix A B
  assume \(*\): open A open B \( A \cap S \neq \{ \} \) \( B \cap S \neq \{ \} \) \( A \cap B \cap S = \{ \} \) \( S \subseteq A \cup B \)
  have continuous-on-S \((\lambda x. x \in A )\)
    unfolding continuous-on-open-invariant
  proof safe
    fix C :: bool set
    have C = UNIV \( \lor \) C = \{ True \} \( \lor \) C = \{ False \} \( \lor \) C = \{ \}
    using subset-UNIV[of C] unfolding UNIV-bool by auto
    with \(*\) show \( \exists T. \ open T \land T \cap S = (\lambda x. x \in A ) - ' C \cap S \)
      by (intro exI[of - (\{ True \} \( \cup \) \{ False \} \( \cup \) \{ \} \) \( \{ \}))]) auto
    qed
    from P[rule-format, OF this] obtain c where \( \forall s \in S \Rightarrow (s \in A ) = c \)
      by blast
    with \(*\) show False
      by (cases c) auto
    qed
  qed

lemma connectedD-const: connected S \( \Rightarrow \) continuous-on S P \( \Rightarrow \exists c. \forall s \in S. \ P s = c \)
for \( P :: 'a::\text{topological-space} \Rightarrow \text{bool} \)
by (auto simp: connected-iff-const)

lemma connectedI-const:
  \( (\forall P :: 'a::\text{topological-space} \Rightarrow \text{bool} \). continuous-on S P \( \Rightarrow \exists c. \forall s \in S. \ P s = c \) \) \( \Rightarrow \) connected S
by (auto simp: connected-iff-const)

lemma connected-local-const:
  assumes connected A a \( \in \) A b \( \in \) A
  and \(*\): \( \forall a \in A. \ eventually (\lambda b. f a = f b) \) (at a within A)
  shows f a = f b
proof
  obtain S where S: \( \lambda a. a \in A \Rightarrow a \in S a \lambda a. a \in A \Rightarrow open (S a) \)
\( \forall a. \ a \in A \implies x \in S \implies x \in A \implies f a = f x \)

using * unfolding eventually-at-topological by metis

let \(?P = \bigcup b \in \{ b \in A. \ f a = f b \} .\ S b \) and \(?N = \bigcup b \in \{ b \in A. \ f a \neq f b \} .\ S b \)

have \(?P \cap A = \{ \} \lor \ ?N \cap A = \{ \} \)

using \( \langle \text{connected } A \rangle \ S \langle a \in A \rangle \)

by \((\text{intro connectedD}) \) \((\text{auto, metis})\)

then show \( f a = f b \)

proof

assume \(?N \cap A = \{ \} \)

then have \( \forall x \in A. \ f a = f x \)

using \((1)\) by \( \text{auto} \)

with \( \langle b \in A. \rangle \) show \( \langle \text{thesis} \rangle \) by \( \text{auto} \)

next

assume \(?P \cap A = \{ \} \) then show \( \langle \text{thesis} \rangle \)

using \( \langle a \in A. \ S(1)[\text{of } a] \rangle \) by \( \text{auto} \)

qed

qed

lemma \((\text{in linorder-topology}) \) connectedD-interval:

assumes \( \text{connected } U \)

and \( \text{xy} : x \in U \ y \in U \)

and \( x \leq z \ z \leq y \)

shows \( z \in U \)

proof –

have \( \text{eq: } \{..<z\} \cup \{z..<\} = - \{z\} \)

by \( \text{auto} \)

have \( \neg \text{connected } U \) if \( z \notin U \ x < z \ z < y \)

using \( \text{xy} \) that

apply \( (\text{simp only: connected-def simp-thms}) \)

apply \( (\text{rule-tac exI } \{.\ <z\})\)

apply \( (\text{rule-tac exI } \{z <.\})\)

apply \( (\text{auto simp add: eq}) \)

done

with \( \text{assms} \) show \( z \in U \)

by \( (\text{metis less-le}) \)

qed

lemma \((\text{in linorder-topology}) \) not-in-connected-cases:

assumes \( \text{conn: } \text{connected } S \)

assumes \( \text{nbdd: } x \notin S \)

assumes \( \text{ne: } S \neq \{ \} \)

obtains \( \text{bdd-above } S \land y. \ y \in S \implies x \geq y \mid \text{bdd-below } S \land y. \ y \in S \implies x \leq y \)

proof –

obtain \( s \) where \( s \in S \) using \( \text{ne} \) by \( \text{blast} \)

\{

assume \( s \leq x \)

have \( \langle \text{False if } x \leq y \ y \in S \rangle \)

using \( (\text{connectedD-interval})(\text{OF conn } \langle s \in S \rangle \langle y \in S \rangle \langle s \leq x \rangle \langle x \leq y \rangle \langle x \notin S \rangle) \)

by \( \text{simpl} \)

\}
then have wit: \( y \in S \implies x \geq y \) for \( y \)
using le-cases by blast
then have bdd-above \( S \)
by (rule local.bdd-aboveI)

note this wit

} moreover {
assume \( x \leq s \)
have False if \( x \geq y \) \( y \in S \) for \( y \)
using connectedD-interval[OF conn \( \langle y \in S \rangle \langle s \in S \rangle \langle x \geq y \rangle \langle s \geq x \rangle \) \( x \notin S \)],
by simp
then have wit: \( y \in S = \implies x \leq y \) for \( y \)
using le-cases by blast
then have bdd-below \( S \)
by (rule bdd-belowI)

note this wit

} ultimately show \( ?thesis \)
by (meson le-cases that)

qed

lemma connected-continuous-image:
assumes \( *: \) continuous-on \( s \) \( f \)
and connected \( s \)
shows connected \( (f \circ s) \)
proof (rule connectedI-const)
fix \( P \): \( 'b \Rightarrow \) bool
assume continuous-on \( (f \circ s) \) \( P \)
then have continuous-on \( s \) \( (P \circ f) \)
by (rule continuous-on-compose[OF \( * \)])
from connectedD-const[OF \( \langle \) connected \( s \rangle \langle \) this \( \rangle \)]
show \( \exists c. \forall x \in \) \( s \). \( P x = c \)
by auto

qed

lemma connected-Un-UN:
assumes connected \( A \backslash X. X \in B \implies \) connected \( X \backslash X. X \in B \implies A \cap X \neq \) \( \) 
shows connected \( (A \cup \bigcup B) \)
proof (rule connectedI-const)
fix \( f \): \( 'a \Rightarrow \) bool
assume \( f: \) continuous-on \( (A \cup \bigcup B) \) \( f \)
have connected \( A \) continuous-on \( A \) \( f \)
by (auto intro: assms continuous-on-subset[OF \( f \langle 1 \rangle \)])
from connectedD-const[OF \( \langle \) this \( \rangle \)]
obtain \( c \) where \( c: \forall x. x \in A \implies f x = c \)
by metis
have \( f x = c \) if \( x \in X \in B \) for \( x X \)
proof --
have connected \( X \) continuous-on \( X \) \( f \)
using that by (auto intro: assms continuous-on-subset[OF \( f \)])
from connectedD-const[OF \( \langle \) this \( \rangle \)]
obtain \( c' \) where \( c': \forall x. x \in X \implies f x = c' \)
by \textit{metis} \\
from \textit{assms} \(3\) and that obtain \(y\) where \(y \in A \cap X\) \\
by \textit{auto} \\
with \(c[\text{of } y] \ c''[\text{of } y] \ c'[\text{of } x]\) that show \(\varnothing\thesis\) \\
by \textit{auto} \\
\textbf{qed} \\
\textbf{with} \(c\) show \(\exists \ c. \ \forall x \in A \cup \bigcup B. \ fx = c\) \\
by (\textit{intro \(\exists\{[\text{of } - c]\})\textit{ auto} \\
\textbf{qed} \\

\section{100 Linear Continuum Topologies} \\

\textbf{class} \(\text{linear-continuum-topology} = \text{linorder-topology} + \text{linear-continuum}\) \\
\textbf{begin} \\

\textbf{lemma} \(\text{Inf-notin-open}:\) \\
\textbf{assumes} \(A: \text{open } A\) \\
and \(\text{bnd: } \forall a \in A. \ ax < a\) \\
\textbf{shows} \(\text{Inf } A \notin A\) \\
\textbf{proof} \\
assume \(\text{Inf } A \in A\) \\
then obtain \(b\) where \(b < \text{Inf } A \{b <. \ \text{Inf } A\} \subseteq A\) \\
using open-left[\text{of } A \text{ Inf } A x] \textit{assms} \textit{ by} \textit{auto} \\
with dense[\text{of } b \text{ Inf } A] \textit{obtain} \(c\) where \(c < \text{Inf } A \ c \in A\) \\
by (\textit{auto \text{ simp:} \text{subset-eq})} \\
then show \(\text{False}\) \\
using cInf-lower[\text{OF} \ \text{\(\exists\ c \in A\)}] \text{\textit{bnd}} \\
by (\textit{metis \text{not-le less-imp-le bdd-belowI})} \\
\textbf{qed} \\

\textbf{lemma} \(\text{Sup-notin-open}:\) \\
\textbf{assumes} \(A: \text{open } A\) \\
and \(\text{bnd: } \forall a \in A. \ ax < a\) \\
\textbf{shows} \(\text{Sup } A \notin A\) \\
\textbf{proof} \\
assume \(\text{Sup } A \in A\) \\
with \textit{assms} \textit{obtain} \(b\) where \(\text{Sup } A < b \ \{\text{Sup } A .. b\} \subseteq A\) \\
using open-right[\text{of } A \text{ Sup } A x] \textit{by} \textit{auto} \\
with dense[\text{of } \text{Sup } A \ b] \textit{obtain} \(c\) where \(\text{Sup } A < c \ c \in A\) \\
by (\textit{auto \text{ simp:} \text{subset-eq})} \\
then show \(\text{False}\) \\
using cSup-upper[\text{OF} \ \text{\(\exists\ c \in A\)}] \text{\textit{bnd}} \\
by (\textit{metis \text{less-imp-le not-le bdd-aboveI})} \\
\textbf{qed} \\
end \\
\textbf{instance} \(\text{linear-continuum-topology} \subseteq \text{perfect-space\textbf{proof}\textbf{ proof}}\)
THEORY "Topological-Spaces"

fix x :: 'a
obtain y where x < y ∨ y < x
  using ex-gt-or-lt [of x] ...
with Inf-notin-open[of {x} y] Sup-notin-open[of {x} y] show ¬ open {x}
  by auto
qed

lemma connectedI-interval:
  fixes U :: 'a :: linear-continuum-topology set
  assumes *: ∀ x y z. x ∈ U ⇒ y ∈ U ⇒ x ≤ z ⇒ z ≤ y ⇒ z ∈ U
  shows connected U
proof (rule connectedI)
  { fix A B
    assume open A open B A ∩ B ∩ U = {} U ⊆ A ∪ B
    fix x y
    assume x < y x ∈ A y ∈ B x ∈ U y ∈ U
    let ?z = Inf (B ∩ {x <..})
    have x ≤ ?z ?z ≤ y
      using ⟨y ∈ B⟩ ⟨x < y⟩ by (auto intro: cInf-lower cInf-greatest)
    with ⟨x ∈ U⟩ ⟨y ∈ U⟩ have ?z ∈ U
      by (rule *)
    moreover have ?z ∉ B ∩ {x <..}
      using ⟨open B⟩ by (intro Inf-notin-open) auto
    ultimately have ?z ∈ A
      using ⟨x ≤ ?z⟩ ⟨A ∩ B ∩ U = {}⟩ ⟨x ∈ A⟩ ⟨U ⊆ A ∪ B⟩ by auto
    have ∃b∈B. b ∈ A ∧ b ∈ U if ?z < y
      proof
        obtain a where ?z < a {?z ..< a} ⊆ A
          using open-right[of ⟨open A⟩ ⟨?z ∈ A⟩ ⟨?z < y⟩] by auto
        moreover obtain b where b ∈ B x < b b < min a y
          using cInf-less-iff[of B ∩ {x <..} min a y] ⟨?z < a⟩ ⟨?z < y⟩ ⟨x < y⟩ ⟨y ∈ B⟩
            by auto
        moreover have ?z ≤ b
          using ⟨b ∈ B⟩ ⟨x < b⟩
            by (intro cInf-lower) auto
        moreover have b ∈ U
          using ⟨x ≤ ?z⟩ ⟨?z ≤ b⟩ ⟨b < min a y⟩
            by (intro *[OF ⟨x ∈ U⟩ ⟨y ∈ U⟩]) (auto simp: less-imp-le)
        ultimately show ?thesis
          by (intro bexI[of - b]) auto
      qed
    then have False
      using ⟨?z ≤ y⟩ ⟨?z ∈ A⟩ ⟨y ∈ B⟩ ⟨y ∈ U⟩ ⟨A ∩ B ∩ U = {}⟩
        unfolding le-less by blast
  }
note not-disjoint = this

fix A B assume AB: open A open B U ⊆ A ∪ B A ∩ B ∩ U = {}
moreover assume A ∩ U ≠ {} then obtain x where x: x ∈ U x ∈ A by auto
moreover assume B ∩ U ≠ {} then obtain y where y: y ∈ U y ∈ B by auto
moreover note not-disjoint[of B A y x] not-disjoint[of A B x y]
ultimately show False
  by (cases x y rule: linorder-cases) auto
qed

lemma connected-iff-interval: connected U ←→ (∀ x∈U. ∀ y∈U. ∀ z. x ≤ z → z ≤ y → z ∈ U)
  for U :: 'a::linear-continuum-topology set
  by (auto intro: connectedI-interval dest: connectedD-interval)

lemma connected-UNIV[simp]: connected (UNIV::'a::linear-continuum-topology set)
  by (simp add: connected-iff-interval)

lemma connected-Ioi[simp]: connected {a<..}
  for a :: 'a::linear-continuum-topology
  by (auto simp: connected-iff-interval)

lemma connected-Ici[simp]: connected {a..}
  for a :: 'a::linear-continuum-topology
  by (auto simp: connected-iff-interval)

lemma connected-Iio[simp]: connected {...<a}
  for a :: 'a::linear-continuum-topology
  by (auto simp: connected-iff-interval)

lemma connected-Iic[simp]: connected {...a}
  for a :: 'a::linear-continuum-topology
  by (auto simp: connected-iff-interval)

lemma connected-Ioo[simp]: connected {a<..<b}
  for a b :: 'a::linear-continuum-topology
  unfolding connected-iff-interval by auto

lemma connected-Ioc[simp]: connected {a..<b}
  for a b :: 'a::linear-continuum-topology
  by (auto simp: connected-iff-interval)

lemma connected-Ico[simp]: connected {a..<b}
  for a b :: 'a::linear-continuum-topology
  by (auto simp: connected-iff-interval)

lemma connected-Icc[simp]: connected {a..b}
  for a b :: 'a::linear-continuum-topology
  by (auto simp: connected-iff-interval)
lemma connected-contains-Ioo:
fixes A :: 'a :: linorder-topology set
assumes connected A a ∈ A b ∈ A
shows {a..<b} ⊆ A
using connectedD-interval[OF assms] by (simp add: subset_eq Ball_def less_imp_le)

lemma connected-contains-Icc:
fixes A :: 'a :: linorder-topology set
assumes connected A a ∈ A b ∈ A
shows {a..b} ⊆ A
proof
  fix x assume x ∈ {a..b}
  then have x = a ∨ x = b ∨ x ∈ {a..<b}
  by auto
  then show x ∈ A
  using assms connected-contains-Ioo[of A a b] by auto
qed

100.1 Intermediate Value Theorem

lemma IVT':
fixes f :: 'a::linear-continuum-topology ⇒ 'b::linorder-topology
assumes y: f a ≤ y y ≤ f b a ≤ b
  and *: continuous-on {a .. b} f
shows ∃ x. a ≤ x ∧ x ≤ b ∧ f x = y
proof
  have connected {a..b}
  unfolding connected-iff-interval by auto
  from connected-continuous-image[OF * this, THEN connectedD-interval, of f a f b y] y
  show ?thesis
  by (auto simp add: atLeastAtMost_def atLeast_def atMost_def)
qed

lemma IVT2':
fixes f :: 'a::linear-continuum-topology ⇒ 'b::linorder-topology
assumes y: f b ≤ y y ≤ f a a ≤ b
  and *: continuous-on {a .. b} f
shows ∃ x. a ≤ x ∧ x ≤ b ∧ f x = y
proof
  have connected {a..b}
  unfolding connected-iff-interval by auto
  from connected-continuous-image[OF * this, THEN connectedD-interval, of f b f a y] y
  show ?thesis
  by (auto simp add: atLeastAtMost_def atLeast_def atMost_def)
qed

lemma IVT:
fixes $f : 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
shows $f \ a \leq y \Longrightarrow y \leq f \ b \Longrightarrow a \leq b \Longrightarrow (\forall x. \ a \leq x \land x \leq b \longrightarrow \text{isCont} \ f \ x)$

$\Longrightarrow \exists x. \ a \leq x \land x \leq b \land f \ x = y$
by (rule IVT'') (auto intro: continuous-at-imp-continuous-on)

lemma IVT2:
fixes $f : 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
shows $f \ b \leq y \Longrightarrow y \leq f \ a \Longrightarrow a \leq b \Longrightarrow (\forall x. \ a \leq x \land x \leq b \longrightarrow \text{isCont} \ f \ x)$

$\Longrightarrow \exists x. \ a \leq x \land x \leq b \land f \ x = y$
by (rule IVT'') (auto intro: continuous-at-imp-continuous-on)

lemma continuous-inj-imp-mono:
fixes $f : 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
assumes $x : a < x x < b$
and $\text{cont}: \text{continuous-on} \ \{a..b\} \ f$
and $\text{inj}: \text{inj-on} \ f \ \{a..b\}$
shows $(f \ a < f \ x \land f \ x < f \ b) \lor (f \ b < f \ x \land f \ x < f \ a)$

proof

note $I = \text{inj-on-eq-iff} \ [(\text{OF inj})]$

{ assume $f \ x < f \ a \ f \ x < f \ b$
then obtain $s t$ where $x \leq s s \leq b \ a \leq t t \leq x \ f \ s = f \ t \ f \ x < f \ s$
using IVT"[of $f \ x \min \ (f \ a)$ $(f \ b)$ $b$] IVT2"[of $f \ x \min \ (f \ a)$ $(f \ b)$ $a$] $x$
by (auto simp: continuous-on-subset[OF cont] less-imp-le)
with $x I$ have False by auto
}

moreover

{ assume $f \ a < f \ x \ f \ b < f \ x$
then obtain $s t$ where $x \leq s s \leq b \ a \leq t t \leq x \ f \ s = f \ t \ f \ s < f \ x$
using IVT"[of $f \ a \max \ (f \ a)$ $(f \ b)$ $x$] IVT2"[of $f \ b \max \ (f \ a)$ $(f \ b)$ $x$] $x$
by (auto simp: continuous-on-subset[OF cont] less-imp-le)
with $x I$ have False by auto
}

ultimately show $?thesis$
using $I[\text{of} \ a \ x]$ $I[\text{of} \ x \ b]$ $x$ less-trans[OF $x$]
by (auto simp add: le-less less-imp-neq neq-iff)

qed

lemma continuous-at-Sup-mono:
fixes $f : 'a::\{\text{linorder-topology}\text{,conditionally-complete-linorder}\} \Rightarrow \ 'b::\{\text{linorder-topology}\text{,conditionally-complete-linorder}\}$
assumes mono $f$
and $\text{cont}: \text{continuous} \ (\text{at-left} \ (\text{Sup} \ S)) \ f$
and $S: S \neq \\{\}$ $\text{bdd-above} \ S$
shows $f \ (\text{Sup} \ S) = (\text{SUP} \ s \in S. \ f \ s)$

proof (rule antisym)
have \( f : (f \rightarrow f (\text{Sup } S)) \) (at-left (Sup S))
  using cont unfolding continuous-within .
show \( f (\text{Sup } S) \leq (\text{SUP } s \in S. f s) \)
proof cases
  assume Sup S \in S
  then show \( \text{thesis} \)
    by (rule cSUP-upper) (auto intro: bdd-above-image-mono S \langle\text{mono } f\rangle)
next
  assume Sup S \notin S
  from \( S \neq \{\} \) obtain s where s \in S
  by auto
with \( \langle\text{Sup } S \notin S, S \rangle \) have s < Sup S
  unfolding less-le by (blast intro: cSup-upper)
show \( \text{thesis} \)
proof (rule ccontr)
  assume \( \neg \langle\text{thesis}\rangle \)
  with order-tendstoD(1)[\langle\text{OF } f, \text{of SUP } s \in S. f s\rangle] obtain b where b < Sup S
  and s: \( y. b < y \Rightarrow y < \text{Sup } S \Rightarrow (\text{SUP } s \in S. f s) < f y \)
  by (auto simp; not-le eventually-at-left[\langle\text{OF } s < \text{Sup } S\rangle])
  with \( S \neq \{\} \) obtain c where c \in S b < c
  using less-cSUPD[s b] by auto
with \( \langle\text{Sup } S \notin S, S \rangle \) have c < Sup S
  unfolding less-le by (blast intro: cSup-upper)
  from s[\langle\text{OF } b < c, c < \text{Sup } S\rangle] cSUP-upper[\langle\text{OF } c \in S, \text{bdd-above-image-mono[of f]}\rangle]
  show False
  by (auto simp; assms)
qed
qed

lemma continuous-at-Sup-antimono:
fixes f :: 'a::{linorder-topology,conditionally-complete-linorder} \Rightarrow 'b::{linorder-topology,conditionally-complete-linorder}
assumes antimono f
  and cont: continuous (at-left (\text{Sup } S)) f
  and S: \( S \neq \{\} \) bdd-above S
shows f (\text{Sup } S) = (\text{INF } s \in S. f s)
proof (rule antisym)
  have f: \( f \rightarrow f (\text{Sup } S) \) (at-left (\text{Sup } S))
    using cont unfolding continuous-within .
  show \( \langle\text{INF } s \in S. f s\rangle \leq f (\text{Sup } S) \)
    by (intro cINF-lower) (auto intro: bdd-below-image-antimono S \langle\text{antimono } f\rangle)
next
  assume S Sup S \notin S
  from \( S \neq \{\} \) obtain s where s \in S
THEORY “Topological-Spaces”

by auto
with ⟨Sup S /∈ S⟩ S have s < Sup S
unfolding less-le by (blast intro: cSup-upper)
show ?thesis
proof (rule ccontr)
assume ¬ ?thesis
with ⟨Sup S /∈ S⟩ S have s < Sup S
and 
<: ∀y. b < y ⇒ y < Sup S ⇒ f y < (INF s∈S. f s)
by (auto simp: not-le eventually-at-left[OF ⟨s < Sup S⟩])
with ⟨S ≠ ⟩ Obtained c where c ∈ S b < c
using less-cSupD[of S b] by auto
with ⟨Sup S /∈ S⟩ S have c < Sup S
unfolding less-le by (blast intro: cSup-upper)
from ⟨OF ⟨b < c⟩ c < Sup S⟩ cINF-lower[OF bdd-below-image-antimono, of f S c] ⟨c ∈ S⟩
show False
  by (auto simp: assms)
qed
qed

lemma continuous-at-Inf-mono:
  fixes f :: 'a::{linorder-topology,conditionally-complete-linorder} ⇒ 'b::{linorder-topology,conditionally-complete-linorder}
  assumes mono f
  and cont: continuous (at-right (Inf S)) f
  and S: S ≠ ⟩ bdd-below S
  shows f (Inf S) = (INF s∈S. f s)
proof (rule antisym)
have f: (f −−→ f (Inf S)) (at-right (Inf S))
  using cont unfolding continuous-within.
show (INF s∈S. f s) ≤ f (Inf S)
proof cases
  assume Inf S ∈ S
  then show ?thesis
  by (rule cINF-lower[rotated]) (auto intro: bdd-below-image-mono S ⟨mono f⟩)
next
  assume Inf S /∈ S
  from ⟨S ≠ ⟩ Obtained s where s ∈ S
  by auto
  with ⟨Inf S /∈ S⟩ S have Inf S < s
  unfolding less-le by (blast intro: cINF-upper)
  show ?thesis
  proof (rule ccontr)
    assume ¬ ?thesis
    with ⟨order-tendstoD(2)[OF f, of INF s∈S. f s]⟩ Obtained b where Inf S < b
    and 
    <: ∀y. Inf S < y ⇒ y < b ⇒ f y < (INF s∈S. f s)
    by (auto simp: not-le eventually-at-right[OF ⟨Inf S < s⟩])
    with ⟨S ≠ ⟩ Obtained c where c ∈ S c < b
THEORY “Topological-Spaces”

using cInf-lessD[of S b] by auto
with /Inf S ∉ S: S have Inf S < c
unfolding less-le by (blast intro: cInf-lower)
from *[OF :Inf S < c) ∨ c < b] cINF-lower[OF bdd-below-image-mono[of f]
⟨c ∈ S] show False
  by (auto simp: assms)
qed
qed
qed (intro cINF-greatest ⟨mono f|THEN monoD] cInf-lower ⟨bdd-below S⟩ ⟨S ≠
{}⟩)

lemma continuous-at-Inf-antimono:
  fixes f :: 'a::{linorder_topology,conditionally-complete-linorder} ⇒ '
b::{linorder-topology,conditionally-complete-linorder}
  assumes antimono f
  and cont: continuous (at-right (Inf S)) f
  and S: S ≠ {} bdd-below S
  shows f (Inf S) = (SUP s∈S. f s)
proof (rule antisym)
  have f: (f −−−→ f (Inf S)) (at-right (Inf S))
    using cont unfolding continuous-within .
  show f (Inf S) ≤ (SUP s∈S. f s)
    proof cases
    assume Inf S ∈ S
    then show ?thesis
      by (rule cSUP-upper) (auto intro: bdd-above-image-antimono S ⟨antimono f⟩)
    next
    assume Inf S ∉ S
    from ⟨S ≠ {}⟩ obtain s where s ∈ S
    by auto
    with Inf S ∉ S: S have Inf S < s
    unfolding less-le by (blast intro: cInf-lower)
    show ?thesis
    proof (rule ccontr)
      assume ¬ ?thesis
      with order-tendstoD(1)[OF f, of SUP s∈S. f s] obtain b where Inf S < b
      and : ∀ y. Inf S < y ⇒ y < b ⇒ (SUP s∈S. f s) < f y
      by (auto simp: not-le eventually-at-right[OF ⟨Inf S < s|])
      with ⟨S ≠ {}⟩ obtain c where c ∈ S c < b
      using cInf-lessD[of S b] by auto
      with Inf S ∉ S: S have Inf S < c
      unfolding less-le by (blast intro: cInf-lower)
      from *[OF :Inf S < c) ∨ c < b] cSUP-upper[OF ⟨c ∈ S⟩ bdd-above-image-antimono[of
f]] show False
        by (auto simp: assms)
    qed
    qed
THEORY "Topological-Spaces"

qed (intro cSUP-least 'antimono f:[THEN antimonoD] cInf-lower S)

100.2 Uniform spaces

class uniformity =
  fixes uniformity :: ('a × 'a) filter
begin

abbreviation uniformity-on :: 'a set ⇒ ('a × 'a) filter
  where uniformity-on s ≡ inf uniformity (principal (s×s))

end

lemma uniformity-Abort:
  uniformity = Filter.abstract-filter (λu. Code.abort (STR "uniformity is not executable") (λu. uniformity))
  by simp

class open-uniformity = open + uniformity +
  assumes open-uniformity:
    ⋀ U. open U ↔ (∀ x∈ U. eventually (λ(x', y). x' = x → y ∈ U) uniformity)
begin

subclass topological-space
  by standard (force elim: eventually-mono eventually-elim2 simp: split-beta' open-uniformity)+

end

class uniform-space = open-uniformity +
  assumes uniformity-refl: eventually E uniformity ⇒ E (x, x)
  and uniformity-sym: eventually E uniformity ⇒ eventually (λ(x, y). E (y, x)) uniformity
  and uniformity-trans:
    eventually E uniformity ⇒
    ∃ D. eventually D uniformity ∧ (∀ x y z. D (x, y) → D (y, z) → E (x, z))
begin

lemma uniformity-bot: uniformity ≠ bot
  using uniformity-refl by auto

lemma uniformity-trans':
  eventually E uniformity ⇒
  eventually (λ((x, y), (y', z)). y = y' → E (x, z)) (uniformity ×F uniformity)
  by (drule uniformity-trans) (auto simp add: eventually-prod-same)

lemma uniformity-transE:
  assumes eventually E uniformity
  obtains D where eventually D uniformity ⋀ x y z. D (x, y) → D (y, z) → E
THEORY “Topological-Spaces”

\[(x, z)\]
using uniformity-trans [OF assms] by auto

lemma eventually-nhds-uniformity:
  \[\text{eventually } P (\text{nhds } x) \longleftrightarrow (\lambda(x', y). \ x' = x \rightarrow P y) \text{ uniformity} \]

(is - \longleftrightarrow \ ?N P x)

unfolding eventually-nhds

proof safe
  assume \[\ast\]: \[\ast\]
  have \(?N \ (\ ?N P) \ x\) if \(?N P x\) for \(x\)
  proof -
    from that obtain \(D\) where ev: \[\text{eventually } D \text{ uniformity}\]
    and \(D: \ (a, b) \mapsto D (b, c) \mapsto \text{ case } (a, c) \text{ of } (x', y) \rightarrow x' = x \rightarrow P y\)
  for \(a \ b \ c\)
    by (rule uniformity-transE) simp
  from ev show \(?thesis\)
    by eventually-elim (insert ev D, force elim: eventually-mono split: prod.split)
  qed
  then have open \(\{x. \ ?N P x\}\)
    by (simp add: open-uniformity)
  then show \(\exists S. \ (\text{open } S \land x \in S \land (\forall x \in S. \ P x))\)
    by (intro exI[of - \{x. \ ?N P x\}] ) (auto dest: uniformity-refl simp: \[\ast\])
  qed (force simp add: open-uniformity elim: eventually-mono)

100.2.1 Totally bounded sets

definition totally-bounded :: 'a set \Rightarrow bool
  where totally-bounded \(S\) \longleftrightarrow
  \((\forall E. \ \text{eventually } E \text{ uniformity} \rightarrow (\exists X. \ \text{finite } X \land (\forall s \in S. \ \exists x \in X. \ E(x, s))))\)

lemma totally-bounded-empty[iff]: totally-bounded \(\{}\)
  by (auto simp add: totally-bounded-def)

lemma totally-bounded-subset: totally-bounded \(S\) \Rightarrow \(T \subseteq S \Rightarrow \text{totally-bounded } T\)
  by (fastforce simp add: totally-bounded-def)

lemma totally-bounded-Union[intro]:
  assumes \(M: \text{finite } M \land S. \ S \in M \Rightarrow \text{totally-bounded } S\)
  shows \(\text{totally-bounded } (\bigcup M)\)

unfolding totally-bounded-def

proof safe
  fix \(E\)
  assume \(\text{eventually } E \text{ uniformity}\)
  with \(M\) obtain \(X\) where \(\forall S \in M. \ \text{finite } (X S) \land (\forall s \in S. \ \exists x \in X S. \ E(x, s))\)
    by (metis totally-bounded-def)
  with \(\text{finite } M\) show \(\exists X. \ \text{finite } X \land (\forall s \in \bigcup M. \ \exists x \in X. \ E(x, s))\)
    by (intro exI[of - \bigcup S \in M. X S]) force
  qed
100.2.2 Cauchy filter

**Definition** cauchy-filter : 'a filter ⇒ bool
where cauchy-filter F ‒› F × F F ≤ uniformity

**Definition** Cauchy : (nat ⇒ 'a) ⇒ bool
where Cauchy-uniform : Cauchy X = cauchy-filter (filtermap X sequentially)

**Lemma** Cauchy-uniform-iff:
Cauchy X ‒› (∀ P. eventually P uniformity → (∃ N. ∀ n≥N. ∀ m≥N. P (X n, X m)))

**Proof** safe
let ?U = λ P. eventually P uniformity
{ fix P
  assume ?U P ∀ P. ?U P → (∃ N. ∀ n≥N. ∀ m≥N. P (X n, X m))
  then obtain Q N where \( \forall n \geq N \rightarrow Q (X n) \) \( \forall x y. Q x \rightarrow Q y \rightarrow P (x, y) \)
  by metis
  then show \( \exists N. \forall n\geq N. \forall m\geq N. P (X n, X m) \)
  by blast
next
fix P
assume ?U P and P: ∀ P. ?U P → (∃ N. ∀ n≥N. ∀ m≥N. P (X n, X m))
then obtain Q where ?U Q and Q: \( \forall x y z. Q (x, y) \rightarrow Q (y, z) \rightarrow P (x, z) \)
by (auto elim: uniformity-transE)
then have ?U (\( \lambda x. Q x \land (\lambda (x, y). Q (y, x)) \)) x
unfolding eventually-conj-iff by (simp add: uniformity-sym)
from P[rule-format, OF this]
obtain N where N: \( \forall n \geq N \rightarrow m \geq N \rightarrow Q (X n, X m) \land Q (X m, X n) \)
by auto
show \( \exists Q. (\exists N. \forall n \geq N. Q (X n)) \land (\forall x y. Q x \rightarrow Q y \rightarrow P (x, y)) \)
proof (safe intro!: excl[of - λx. ∀ n≥N. Q (x, X n) ∧ Q (X n, x)] excl[of - N] N)
fix x y
assume \( \forall n \geq N. Q (x, X n) \land Q (X n, x) \land Q (Y n, x) \land Q (X n, y) \)
then have Q (x, X N) Q (X N, y) by auto
then show P (x, y)
  by (rule Q)
qed
}

**Lemma** nhds-imp-cauchy-filter:
assumes "*: F ≤ nhds x"
show\ cauchy-filter\ F
proof
  have\ F \times F \leq\ nhds\ x \times_F\ nhds\ x
    by\ \textit{intro}\ prod-filter-mono\ *
  also\ have\ \ldots\ \leq\ uniformity
    unfolding\ le-filter-def\ eventually-nhds-uniformity\ eventually-prod-same
proof\ safe
  fix\ P
  assume\ eventually\ P\ uniformity
  then\ obtain\ Ql\ where\ \textit{ev}:\ eventually\ Ql\ uniformity
    and\ Ql\ (x,\ y) \Rightarrow Ql\ (y,\ z) \Rightarrow P\ (x,\ z)\ \textit{for}\ x,\ y,\ z
    by\ \textit{rule uniformity-transE} simp
  with\ ev[\textit{THEN}\ uniformity-sgm]
  show\ \exists\ Q.\ eventually\ (\lambda(x',\ y).\ x' = x \rightarrow Q\ y)\ uniformity\ \wedge
    (\forall x,\ y.\ Q\ x \rightarrow Q\ y \rightarrow P\ (x,\ y))
    by\ \textit{rule-tac}\ \textit{exI} [of \lambda y.\ Ql\ (y,\ x) \wedge Ql\ (x,\ y)]\ (fastforce\ elim: eventually-elim\ 2)
qed
finally\ show\ \textit{thesis}
  by\ \textit{simp}\ add: cauchy-filter-def)
qed

\textbf{lemma\ LIMSEQ-imp-Cauchy:}\ \textit{X} \xrightarrow{\rightarrow} x \Rightarrow Cauchy\ \textit{X}
\textit{unfolding}\ Cauchy-uniform\ filterlim-def\ by\ \textit{intro}\ nhds-imp-cauchy-filter

\textbf{lemma\ Cauchy-subseq-Cauchy:}
\textit{assumes}\ Cauchy\ \textit{X}\ \textit{strict-mono}\ \textit{f}
\textit{shows}\ Cauchy\ \textit{(X} \circ\ \textit{f})
\textit{unfolding}\ Cauchy-uniform\ \textit{comp-def}\ \textit{filtermap-filtermap}[symmetric]\ \textit{cauchy-filter-def}
\textit{by}\ \textit{rule}\ \textit{order-trans} [\textit{OF} \cdot\ \textit{Cauchy}\ \textit{X}[unfolded\ Cauchy-uniform\ cauchy-filter-def]]
  \textit{(intro}\ \textit{prod-filter-mono}\ \textit{filtermap-mono}\ \textit{filterlim-subseq}[\textit{OF} \langle\textit{strict-mono}\ \textit{f}\rangle,\ unfolding\ \textit{filterlim-def}]\)

\textbf{lemma\ convergent-Cauchy:}\ \textit{convergent}\ \textit{X} \Rightarrow Cauchy\ \textit{X}
\textit{unfolding}\ \textit{convergent-def}\ by\ \textit{erule}\ \textit{exE},\ \textit{erule}\ \textit{LIMSEQ-imp-Cauchy}

\textbf{definition\ complete::\ '{a}\ set\ \Rightarrow bool}
\textit{where}\ complete-uniform: complete\ S \longleftrightarrow
  (\forall\ F\ \leq\ \textit{principal}\ S.\ F \neq\ \textit{bot} \rightarrow cauchy-filter\ F \rightarrow (\exists x\in S.\ F\ \leq\ nhds\ x))

\textbf{lemma\ (in\ uniform-space)\ cauchy-filter-complete-converges:}
\textit{assumes}\ cauchy-filter\ \textit{F}\ \textit{complete}\ \textit{A}\ \textit{F}\ \leq\ \textit{principal}\ \textit{A}\ \textit{F}\ \neq\ \textit{bot}
\textit{shows}\ \exists\ c.\ \textit{F}\ \leq\ \textit{nhds}\ c
\textit{using}\ \textit{assms}\ \textit{unfolding}\ complete-uniform\ \textit{by}\ \textit{blast}

end
100.2.3 Uniformly continuous functions

definition uniformly-continuous-on :: 'a set ⇒ ('a::uniform-space ⇒ 'b::uniform-space) ⇒ bool
  where uniformly-continuous-on-uniformity: uniformly-continuous-on s f ←→
    (LIM (x, y) (uniformity-on s). (f x, f y) ⇒ uniformity)

lemma uniformly-continuous-onD:
  assumes uniformly-continuous-on s f ⇒ eventually E uniformity
  shows eventually (λ(x, y). x ∈ s ∧ y ∈ s ⇒ E (f x, f y)) uniformity
  by (simp add: uniformly-continuous-on-uniformity filterlim-iff
               eventually-inf-principal split-beta' mem-Times-iff imp-conjL)

lemma uniformly-continuous-on-const[continuous-intros]: uniformly-continuous-on
  s (λx. c)
  by (auto simp: uniformly-continuous-on-uniformity filterlim-iff uniformity-refl)

lemma uniformly-continuous-on-id[continuous-intros]: uniformly-continuous-on
  s (λx. x)
  by (auto simp: uniformly-continuous-on-uniformity filterlim-def)

lemma uniformly-continuous-on-compose:
  assumes uniformly-continuous-on s g ⇒ uniformly-continuous-on (g's) f ⇒
  uniformly-continuous-on s (λx. f (g x))
  using filterlim-compose[of (λ(x, y). f x, f y) uniformity
    (λ(x, y). g x, g y) uniformity-on s]
  by (simp add: split-beta' uniformly-continuous-on-uniformity
               filterlim-inf filterlim-principal eventually-inf-principal mem-Times-iff)

lemma uniformly-continuous-imp-continuous:
  assumes f: uniformly-continuous-on s f
  shows continuous-on s f
  by (auto simp: filterlim-iff eventually-at-filter eventually-nhds-uniformity continuous-on-def
               elim: eventually-mono dest!: uniformly-continuous-onD[OF f])

101 Product Topology

101.1 Product is a topological space

instantiation prod :: (topological-space, topological-space) topological-space
begin

definition open-prod-def[code del]:
  open (S :: ('a × 'b) set) ←→
  (∀ x∈S. ∃ A B. open A ∧ open B ∧ x ∈ A × B ∧ A × B ⊆ S)

lemma open-prod-elim:
  assumes open S and x ∈ S
obtains $A B$ where $\text{open } A$ and $\text{open } B$ and $x \in A \times B$ and $A \times B \subseteq S$

using assms unfolding open-prod-def by fast

lemma open-prod-intro:
assumes $\forall x. x \in S \rightarrow \exists A B. \text{open } A \land \text{open } B \land x \in A \times B \land A \times B \subseteq S$
shows $\text{open } S$
using assms unfolding open-prod-def by fast

instance
proof
show $\text{open } (\text{UNIV :: ('a} \times 'b) \text{ set})$
  unfolding open-prod-def by auto
next
fix $S T :: ('a \times 'b) \text{ set}$
assume $\text{open } S \text{ open } T$
show $\text{open } (S \cap T)$
proof (rule open-prod-intro)
  fix $x$
  assume $x :: x \in S \cap T$
  from $x$ have $x \in S$ by simp
  obtain $A B$ where $A :: \text{open } A \text{ open } B$ $x \in A \times B$ $A \times B \subseteq S$
    using $\text{open } S$ and $\langle x \in S \rangle$ by (rule open-prod-elim)
  from $x$ have $x \in T$ by simp
  obtain $A B$ where $B :: \text{open } T \text{ open } T$ $x \in T \times T$ $T \times T \subseteq T$
    using $\text{open } T$ and $\langle x \in T \rangle$ by (rule open-prod-elim)
  let $\langle A = \text{Sa} \cap \text{Tu} \rangle$ and $\langle B = \text{Sb} \cap \text{ Tb} \rangle$
  have $\text{open } \langle A \land \text{open } ?B \land x \in ?A \times ?B \land ?A \times ?B \subseteq S \cap T \rangle$
    using $\text{open } A \text{ by (auto simp add: open-Int)}$
  then show $\exists A B. \text{open } A \land \text{open } B \land x \in A \times B \land A \times B \subseteq S \cap T$
    by fast
qed
next
fix $K :: ('a \times 'b) \text{ set set}$
assume $\forall S \in K. \text{open } S$
then show $\text{open } (\bigcup K)$
  unfolding open-prod-def by fast
qed
end

declare [[code abort: open :: ('a::topological-space \times 'b::topological-space) set \Rightarrow bool]]

lemma open-Times: $\text{open } S \implies \text{open } T \implies \text{open } (S \times T)$
  unfolding open-prod-def by auto

lemma fst-vimage-eq-Times: $\text{fst} \cdot \text{vimage } S = S \times \text{UNIV}$
  by auto
lemma snd-vimage-eq-Times: snd − ':' S = UNIV × S
by auto

lemma open-vimage-fst: open S \implies open (fst − ':' S)
by (simp add: fst-vimage-eq-Times open-Times)

lemma open-vimage-snd: open S \implies open (snd − ':' S)
by (simp add: snd-vimage-eq-Times open-Times)

lemma closed-vimage-fst: closed S \implies closed (fst − ':' S)
unfolding closed-open vimage-Compl [symmetric]
by (rule open-vimage-fst)

lemma closed-vimage-snd: closed S \implies closed (snd − ':' S)
unfolding closed-open vimage-Compl [symmetric]
by (rule open-vimage-snd)

lemma closed-Times: closed S \implies closed T \implies closed (S × T)
proof –
have S × T = (fst − ':' S) ∩ (snd − ':' T)
  by auto
then show closed S \implies closed T \implies closed (S × T)
  by (simp add: closed-vimage-fst closed-vimage-snd closed-Int)
qed

lemma subset-fst-imageI: A × B ⊆ S \implies y ∈ B \implies A ⊆ fst − ':' S
unfolding image-def subset-eq by force

lemma subset-snd-imageI: A × B ⊆ S \implies x ∈ A \implies B ⊆ snd − ':' S
unfolding image-def subset-eq by force

lemma open-image-fst:
assumes open S
shows open (fst − ':' S)
proof (rule openI)
fix x
assume x ∈ fst − ':' S
then obtain y where (x, y) ∈ S
  by auto
then obtain A B where open A open B x ∈ A y ∈ B A × B ⊆ S
  using (open S) unfolding open-prod-def by auto
from (A × B ⊆ S) (y ∈ B) have A ⊆ fst − ':' S
  by (rule subset-fst-imageI)
with (open A) (x ∈ A) have open A ∧ x ∈ A ∧ A ⊆ fst − ':' S
  by simp
then show \exists T. open T ∧ x ∈ T ∧ T ⊆ fst − ':' S ..
qed

lemma open-image-snd:
assumes open S
shows open (snd' S)
proof (rule openI)
  fix y
  assume y ∈ snd' S
  then obtain x where (x, y) ∈ S by auto
  using (open S) unfolding open-prod-def by auto
  with (open B) y ∈ B have open B ∧ y ∈ B ∧ B ⊆ snd' S by simp
  then show ∃ T. open T ∧ y ∈ T ∧ T ⊆ snd' S.. qed

lemma nhds-prod: nhds (a, b) = nhds a ×F nhds b
unfolding nhds-def
proof (subst prod-filter-INF, auto intro: antisym INF-greatest simp: principal-prod-principal)
  fix S T
  assume open S a ∈ S open T b ∈ T
  then show (INF x ∈ {S. open S ∧ (a, b) ∈ S}. principal x) ≤ principal (S × T)
    by (intro INF-lower) (auto intro: open-Times)
next
  fix S'
  assume open S' (a, b) ∈ S'
  then obtain S T where open S a ∈ S open T b ∈ T S × T ⊆ S'
    by (auto elim: open-prod-elim)
  then show (INF x ∈ {S. open S ∧ a ∈ S}. INF y ∈ {S. open S ∧ b ∈ S}. principal (x × y)) ≤ principal S'
    by (auto intro: INF-lower2)
qed

101.1.1 Continuity of operations

lemma tendsto-fst [tendsto-intros]:
  assumes (f ⊢→ a) F
  shows ((λx. fst (f x)) ⊢→ fst a) F
proof (rule topological-tendstoI)
  fix S
  assume open S and fst a ∈ S
  then have open (fst' S) and a ∈ fst' S
    by (simp-all add: open-vimage-fst)
  with assms have eventually (λx. f x ∈ fst' S) F
    by (rule topological-tendstoD)
  then show eventually (λx. fst (f x) ∈ S) F
    by simp
  qed
lemma tendsto-snd [tendsto-intros]:
assumes \( (f \rightarrow a) \) \( F \)
shows \( ((\lambda x. \text{snd} \ (f \ x)) \rightarrow \text{snd} \ a) \) \( F \)
proof (rule topological-tendstoI)
fix \( S \)
assume open \( S \) and \( \text{snd} \ a \in S \)
then have open \( (\text{snd} \ '-' \ S) \) and \( a \in \text{snd} \ '-' \ S \)
by (simp-all add: open-vimage-snd)
with assms have eventually \( (\lambda x. \ f \ x \in \text{snd} \ '-' \ S) \) \( F \)
by (rule topological-tendstoD)
then show eventually \( (\lambda x. \ \text{snd} \ (f \ x) \in S) \) \( F \)
by simp
qed

lemma tendsto-Pair [tendsto-intros]:
assumes \( (f \rightarrow a) \) \( F \) and \( (g \rightarrow b) \) \( F \)
shows \( ((\lambda x. \ (f \ x, \ g \ x)) \rightarrow (a, \ b)) \) \( F \)
unfolding nhds-prod using assms by (rule filterlim-Pair)

lemma continuous-fst[continuous-intros]: continuous \( F \) \( f \) \( \Rightarrow \) continuous \( F \) \( (\lambda x. \ \text{fst} \ (f \ x)) \)
unfolding continuous-def by (rule tendsto-fst)

lemma continuous-snd[continuous-intros]: continuous \( F \) \( f \) \( \Rightarrow \) continuous \( F \) \( (\lambda x. \ \text{snd} \ (f \ x)) \)
unfolding continuous-def by (rule tendsto-snd)

lemma continuous-Pair[continuous-intros]:
continuous \( F \) \( f \) \( \Rightarrow \) continuous \( F \) \( g \) \( \Rightarrow \) continuous \( F \) \( (\lambda x. \ (f \ x, \ g \ x)) \)
unfolding continuous-def by (rule tendsto-Pair)

lemma continuous-on-fst[continuous-intros]:
continuous-on \( s \) \( f \) \( \Rightarrow \) continuous-on \( s \) \( (\lambda x. \ \text{fst} \ (f \ x)) \)
unfolding continuous-on-def by (auto intro: tendsto-fst)

lemma continuous-on-snd[continuous-intros]:
continuous-on \( s \) \( f \) \( \Rightarrow \) continuous-on \( s \) \( (\lambda x. \ \text{snd} \ (f \ x)) \)
unfolding continuous-on-def by (auto intro: tendsto-snd)

lemma continuous-on-Pair[continuous-intros]:
continuous-on \( s \) \( f \) \( \Rightarrow \) continuous-on \( s \) \( g \) \( \Rightarrow \) continuous-on \( s \) \( (\lambda x. \ (f \ x, \ g \ x)) \)
unfolding continuous-on-def by (auto intro: tendsto-Pair)

lemma continuous-on-swap[continuous-intros]: continuous-on \( A \ prod.swap \)
by (simp add: prod.swap-def continuous-on-fst continuous-on-snd continuous-on-Pair continuous-on-id)

lemma continuous-on-swap-args:
assumes continuous-on \((A \times B)\) \((\lambda(x,y), d x y)\)
shows continuous-on \((B \times A)\) \((\lambda(x,y), d y x)\)
proof
have \((\lambda(x,y), d y x) = (\lambda(x,y), d x y) \circ prod.swap\)
  by force
then show ?thesis
  by (metis assms continuous-on-compose continuous-on-swap product-swap)
qed

lemma isCont-fst [simp]:
\(\text{isCont } f \ a \implies \text{isCont } (\lambda x. \text{fst } (f x)) \ a\)
by (fact continuous-fst)

lemma isCont-snd [simp]:
\(\text{isCont } f \ a \implies \text{isCont } (\lambda x. \text{snd } (f x)) \ a\)
by (fact continuous-snd)

lemma isCont-Pair [simp]: \([\text{isCont } f \ a; \text{isCont } g \ a]\) \(\implies \text{isCont } (\lambda x. (f x, g x)) \ a\)
by (fact continuous-Pair)

lemma continuous-on-compose-Pair:
assumes \(f: \text{continuous-on } (\Sigma \ A \ B)\) \((\lambda(a, b), f a b)\)
assumes \(g: \text{continuous-on } C\) \(g\)
assumes \(h: \text{continuous-on } C\) \(h\)
assumes subset: \(\forall c. c \in C \implies g \ c \in A \ \forall c. c \in C \implies h \ c \in B\ (g \ c)\)
shows continuous-on \((\lambda c. f (g \ c) (h \ c))\)
using continuous-on-compose2[OF \(f\) continuous-on-Pair[OF \(g\) \(h\)]] subset
by auto

101.1.2 Connectedness of products

proposition connected-Times:
assumes \(S: \text{connected } S\) \(\text{and } T: \text{connected } T\)
shows connected \((S \times T)\)
proof (rule connectedI-const)
  fix \(P:\'a \times \'b \Rightarrow \text{bool}\)
  assume \(P\)\[THEN continuous-on-compose2, continuous-intros]: \(\text{continuous-on } (S \times T)\) \(P\)
  have continuous-on \(S\) \((\lambda s. P (s, t))\) \(\text{if } t \in T \text{ for } t\)
    by (auto intro!: continuous-intros that)
  from connectedD-const[OF \(S\) this]
  obtain \(c1\) where \(c1: \forall t. t \in T \implies s \in S \implies P (s, t) = c1\) \(t\)
    by metis
  moreover
  have continuous-on \(T\) \((\lambda t. P (s, t))\) \(\text{if } s \in S \text{ for } s\)
    by (auto intro!: continuous-intros that)
  from connectedD-const[OF \(T\) this]
  obtain \(c2\) where \(\forall t. t \in T \implies s \in S \implies P (s, t) = c2\) \(s\)
    by metis
  ultimately show \(\exists c. \forall s \in S \times T. P s = c\)
    by auto
corollary connected-Times-eq [simp]:

connected \((S \times T)\) \iff S = \{\} \lor T = \{\} \lor connected S \land connected T \ (\text{is} \ \text{lhs} = \text{rhs})

proof
assume L: \text{lhs}
show \text{rhs}
proof cases
  assume S \neq \{\} \land T \neq \{\}
  moreover
  have connected (fst \ ((S \times T))) connected (snd \ ((S \times T)))
    using continuous-on-fst continuous-on-snd continuous-on-id
    by (blast intro: connected-continuous-image [OF - L])+
  ultimately show \text{thesis}
    by auto
  qed
qed (auto simp: connected-Times)

101.1.3 Separation axioms

instance prod :: (t0-space, t0-space) t0-space
proof
fix x y :: 'a \times 'b
assume x \neq y
then have fst x \neq fst y \lor snd x \neq snd y
  by (simp add: prod-eq-iff)
then show \exists U. open U \land (x \in U) \neq (y \in U)
  by (fast dest: t0-space elim: open-vimage-fst open-vimage-snd)
qed

instance prod :: (t1-space, t1-space) t1-space
proof
fix x y :: 'a \times 'b
assume x \neq y
then have fst x \neq fst y \lor snd x \neq snd y
  by (simp add: prod-eq-iff)
then show \exists U V. open U \land x \in U \land y \notin U
  by (fast dest: t1-space elim: open-vimage-fst open-vimage-snd)
qed

instance prod :: (t2-space, t2-space) t2-space
proof
fix x y :: 'a \times 'b
assume x \neq y
then have fst x \neq fst y \lor snd x \neq snd y
  by (simp add: prod-eq-iff)
then show \exists U V. open U \land open V \land x \in U \land y \in V \land U \cap V = \{\}
  by (fast dest: hausdorff elim: open-vimage-fst open-vimage-snd)
lemma isCont-swap[continuous-intros]: isCont prod.swap a
  using continuous-on-eq-continuous-within continuous-on-swap by blast

lemma open-diagonal-complement:
  open \{(x, y) | x \neq (y::'a::t2-space)\}
proof -
  have open \{(x, y), x \neq (y::'a)\}
    unfolding split-def by (intro open-Collect-neq continuous-intros)
  also have \{(x, y), x \neq (y::'a)\} = \{(x, y) | x y x \neq (y::'a)\}
    by auto
  finally show \?thesis .
qed

lemma closed-diagonal:
  closed \{y. \exists x::'a::t2-space. y = (x,x)\}
proof -
  have \{y. \exists x::'a. y = (x,x) = UNIV - \{(x,y) | x y x \neq y\}\} by auto
  then show \?thesis using open-diagonal-complement closed-Diff by auto
qed

lemma open-superdiagonal:
  open \{(x, y) | x y x > (y::'a::linorder-topology)\}
proof -
  have open \{(x, y), x > (y::'a)\}
    unfolding split-def by (intro open-Collect-less continuous-intros)
  also have \{(x, y), x > (y::'a)\} = \{(x, y) | x y x > (y::'a)\}
    by auto
  finally show \?thesis .
qed

lemma closed-subdiagonal:
  closed \{(x, y) | x y x \leq (y::'a::linorder-topology)\}
proof -
  have \{(x, y), x \leq (y::'a)\} = UNIV - \{(x,y) | x y x > (y::'a)\} by auto
  then show \?thesis using open-superdiagonal closed-Diff by auto
qed

lemma open-subdiagonal:
  open \{(x, y) | x y x < (y::'a::linorder-topology)\}
proof -
  have open \{(x, y), x < (y::'a)\}
    unfolding split-def by (intro open-Collect-less continuous-intros)
  also have \{(x, y), x < (y::'a)\} = \{(x, y) | x y x < (y::'a)\}
    by auto
  finally show \?thesis .
qed
lemma closed-superdiagonal:
closed \{ (x, y) | x \geq (y::'a::(linorder-topology)) \}
proof
  have \{ (x, y) | x \geq (y::'a) \} = UNIV - \{ (x, y) | x < y \}
  then show \thesis using open-subdiagonal closed-Diff
qed

theory Hull
  imports Main
begin

101.2 A generic notion of the convex, affine, conic hull, or closed "hull".

definition hull :: ('a set ⇒ bool) ⇒ 'a set ⇒ 'a set (infixl hull 75)
  where S hull s = ∩ \{ t. S t ∧ s ⊆ t \}

lemma hull-same: S s =⇒ S hull s = s
  unfolding hull-def by auto

lemma hull-in: (∀ T. Ball T S =⇒ S (\bigcap T)) =⇒ S (S hull s)
  unfolding hull-def Ball-def by auto

lemma hull-eq: (∀ T. Ball T S =⇒ S (\bigcap T)) =⇒ (S hull s) = s ⇔ S s
  using hull-same[of S s] hull-in[of S s] by metis

lemma hull-hull [simp]: S hull (S hull s) = S hull s
  unfolding hull-def by blast

lemma hull-subset[intro]: s ⊆ (S hull s)
  unfolding hull-def by blast

lemma hull-mono: s ⊆ t =⇒ (S hull s) ⊆ (S hull t)
  unfolding hull-def by blast

lemma hull-antimono: ∀ x. S x =⇒ T x =⇒ (T hull s) ⊆ (S hull s)
  unfolding hull-def by blast

lemma hull-minimal: s ⊆ t =⇒ S t =⇒ (S hull s) ⊆ t
  unfolding hull-def by blast

lemma subset-hull: S t =⇒ S hull s ⊆ t ⇔ s ⊆ t
  unfolding hull-def by blast

lemma hull-UNIV [simp]: S hull UNIV = UNIV
  unfolding hull-def by auto
lemma hull-unique: $s \subseteq t \implies S t \implies (\forall t'. s \subseteq t' \implies S t' \implies t \subseteq t') \implies (S \text{ hull } s = t)$
    unfolding hull-def by auto

lemma hull-induct: $\exists a \in Q \text{ hull } S; \forall x. x \in S \implies P x; Q \{x. P x\} \implies P a$
    using hull-minimal[of S \{x. P x\} Q]
    by (auto simp add: subset-eq)

lemma hull-inc: $x \in S \implies x \in P \text{ hull } S$
    by (metis hull-subset subset-eq)

lemma hull-Un-subset: $(S \text{ hull } s) \cup (S \text{ hull } t) \subseteq (S \text{ hull } (s \cup t))$
    unfolding Un-subset-iff by (metis hull-mono Un-upper1 Un-upper2)

lemma hull-Un: assumes $T: \exists T. {\text{Ball} T S} \implies S (\bigcap T)$
    shows $S \text{ hull } (s \cup t) = S \text{ hull } (S \text{ hull } s \cup S \text{ hull } t)$
    apply (rule equalityI)
    apply (meson hull-mono hull-subset sup mono)
    by (metis hull-Un-subset hull-hull hull-mono)

lemma hull-Un-left: $P \text{ hull } (S \cup T) = P \text{ hull } (P \text{ hull } S \cup T)$
    apply (rule equalityI)
    apply (simp add: Un-commute hull-mono hull-subset sup coboundedI2)
    by (metis Un-subset-iff hull-hull hull-mono hull-subset)

lemma hull-Un-right: $P \text{ hull } (S \cup T) = P \text{ hull } (S \cup P \text{ hull } T)$
    by (metis hull-Un-left sup commute)

lemma hull-insert: $P \text{ hull } (\text{insert } a S) = P \text{ hull } (\text{insert } a (P \text{ hull } S))$
    by (metis hull-Un-right insert-is-Un)

lemma hull-redundant-eq: $a \in (S \text{ hull } s) \iff S \text{ hull } (\text{insert } a s) = S \text{ hull } s$
    unfolding hull-def by blast

lemma hull-redundant: $a \in (S \text{ hull } s) \implies S \text{ hull } (\text{insert } a s) = S \text{ hull } s$
    by (metis hull-redundant-eq)

end

102 Modules

Bases of a linear algebra based on modules (i.e. vector spaces of rings).

theory Modules
    imports Hull
begin
102.1 Locale for additive functions

locale additive =  
  fixes f :: `'a::ab-group-add ⇒ 'b::ab-group-add 
  assumes add: f (x + y) = f x + f y 
begin

lemma zero: f 0 = 0
proof –
  have f 0 = f (0 + 0) by simp 
  also have . . . = f 0 + f 0 by (rule add)
  finally show f 0 = 0 by simp 
qed

lemma minus: f (− x) = − f x
proof –
  have f (− x) + f x = f (− x + x) by (rule add [symmetric])
  also have . . . = − f x + f x by (simp add: zero)
  finally show f (− x) = − f x by (rule add-right-imp-eq)
qed

lemma diff: f (x − y) = f x − f y 
  using add [of x − y] by (simp add: minus)

lemma sum: f (sum g A) = (∑ x∈A. f (g x))
  by (induct A rule: infinite-finite-induct) (simp-all add: zero add)
end

Modules form the central spaces in linear algebra. They are a generalization from vector spaces by replacing the scalar field by a scalar ring.

locale module =
  fixes scale :: `'a::comm-ring-1 ⇒ 'b::ab-group-add ⇒ 'b (infixr ∗ 75)
  assumes scale-right-distrib [algebra-simps, algebra-split-simps]:
    a ∗ s (x + y) = a ∗ s x + a ∗ s y 
  and scale-left-distrib [algebra-simps, algebra-split-simps]:
    (a + b) ∗ s x = a ∗ s x + b ∗ s x
  and scale-scale [simp]: a ∗ s (b ∗ s x) = (a ∗ b) ∗ s x
  and scale-one [simp]: 1 ∗ s x = x
begin

lemma scale-left-commute: a ∗ s (b ∗ s x) = b ∗ s (a ∗ s x)
  by (simp add: mult.commute)

lemma scale-zero-left [simp]: 0 ∗ s x = 0
  and scale-minus-left [simp]: (− a) ∗ s x = − (a ∗ s x)
  and scale-left-diff-distrib [algebra-simps, algebra-split-simps]:
    (a − b) ∗ s x = a ∗ s x − b ∗ s x
  and scale-sum-left: (sum f A) ∗ s x = (∑ a∈A. (f a) ∗ s x)
proof –
interpret s: additive λa. a *s x
  by standard (rule scale-left-distrib)
show 0 *s x = 0 by (rule s.zero)
show (− a) *s x = − (a *s x) by (rule s.minus)
show (a − b) *s x = a *s x − b *s x by (rule s.diff)
show (sum f A) *s x = (∑ a∈A. (f a) *s x) by (rule s.sum)
qed

lemma scale-zero-right [simp]; a *s 0 = 0
  and scale-minus-right [simp]; a *s (− x) = − (a *s x)
  and scale-right-diff-distrib [algebra-simps, algebra-split-simps]:
    a *s (x − y) = a *s x − a *s y
  and scale-sum-right: a *s (∑ f A) = (∑ x∈A. a *s (f x))
proof –
interpret s: additive λx. a *s x
  by standard (rule scale-right-distrib)
show a *s 0 = 0 by (rule s.zero)
show a *s (− x) = − (a *s x) by (rule s.minus)
show a *s (x − y) = a *s x − a *s y by (rule s.diff)
show a *s (∑ f A) = (∑ x∈A. a *s (f x)) by (rule s.sum)
qed

lemma sum-constant-scale: (∑ x∈A. y) = scale (of-nat (card A)) y
  by (induct A rule: infinite-finite-induct) (simp-all add: algebra-simps)
end

setup ⟨Sign.add-const-constraint (const-name divide), SOME typ (′a ⇒ ′a ⇒ ′a)⟩

context module
begin

lemma [field-simps, field-split-simps]:
  shows scale-left-distrib-NO-MATCH: NO-MATCH (x div y) c ⇒ (a + b) *s x
    = a *s x + b *s x
  and scale-right-distrib-NO-MATCH: NO-MATCH (x div y) a ⇒ a *s (x + y)
    = a *s x + a *s y
  and scale-left-diff-distrib-NO-MATCH: NO-MATCH (x div y) c ⇒ (a − b)
    *s x = a *s x − b *s x
  and scale-right-diff-distrib-NO-MATCH: NO-MATCH (x div y) a ⇒ a *s (x − y)
    = a *s x − a *s y
  by (rule scale-left-distrib scale-right-distrib scale-left-diff-distrib scale-right-diff-distrib)+
end

setup ⟨Sign.add-const-constraint (const-name divide), SOME typ (′a::divide ⇒ ′a ⇒ ′a)⟩


103 Subspace

definition subspace :: 'b set ⇒ bool
  where subspace S ←→ 0 ∈ S ∧ (∀ x∈S. ∀ y∈S. x + y ∈ S) ∧ (∀ c. ∀ x∈S. c * s x ∈ S)

lemma subspaceI:
  0 ∈ S ⇒ (∀ x y. x ∈ S ⇒ y ∈ S ⇒ x + y ∈ S) ⇒ (∀ c x ∈ S ⇒ c * s x ∈ S)
  by (auto simp: subspace-def)

lemma subspace-UNIV[simp]: subspace UNIV
  by (simp add: subspace-def)

lemma subspace-single-0[simp]: subspace {0}
  by (simp add: subspace-def)

lemma subspace-0: subspace S ⇒ 0 ∈ S
  by (metis subspace-def)

lemma subspace-add: subspace S ⇒ x ∈ S ⇒ y ∈ S ⇒ x + y ∈ S
  by (metis subspace-def)

lemma subspace-scale: subspace S ⇒ x ∈ S ⇒ c * s x ∈ S
  by (metis subspace-def)

lemma subspace-neg: subspace S ⇒ x ∈ S ⇒ − x ∈ S
  by (metis scale-minus-left scale-one subspace-scale)

lemma subspace-diff: subspace S ⇒ x ∈ S ⇒ y ∈ S ⇒ x − y ∈ S
  by (metis diff-conv-add-uminus subspace-add subspace-neg)

lemma subspace-sum: subspace A ⇒ (∀ x. x ∈ B ⇒ f x ∈ A) ⇒ sum f B ∈ A
  by (induct B rule: infinite-finite-induct) (auto simp add: subspace-add subspace-0)

lemma subspace-Int: (∀ i. i ∈ I ⇒ subspace (s i)) ⇒ subspace (∩ i∈I. s i)
  by (auto simp: subspace-def)

lemma subspace-Inter: ∀ s ∈ f. subspace s ⇒ subspace (∩ f)
  unfolding subspace-def by auto

lemma subspace-inter: subspace A ⇒ subspace B ⇒ subspace (A ∩ B)
  by (simp add: subspace-def)
104 Span: subspace generated by a set

definition span :: 'b set ⇒ 'b set
  where span-explicit: span b = {(∑ a∈t. r a * s a) | t r. finite t ∧ t ⊆ b}

lemma span-explicit':
  span b = {(∑ v | f v ≠ 0. f v * s v) | f. finite {v. f v ≠ 0} ∧ (∀ v. f v ≠ 0 → v ∈ b)}
  unfolding span-explicit
proof safe
  fix t r assume finite t t ⊆ b
  then show ∃ f. (∑ a∈t. r a * s a) = (∑ v | f v ≠ 0. f v * s v) ∧ finite {v. f v ≠ 0} ∧ (∀ v. f v ≠ 0 → v ∈ b)
    by (intro exI[of - λv. if v ∈ t then v else 0]) (auto intro!: sum_mono_neutral_cong_right)
next
  fix f :: 'b ⇒ 'a assume finite {v. f v ≠ 0} (∀ v. f v ≠ 0 → v ∈ b)
  then show ∃ t r. (∑ v | f v ≠ 0. f v * s v) = (∑ a∈t. r a * s a) ∧ finite t ∧ t ⊆ b
    by (intro exI[of - {v. f v ≠ 0}] exI[of - f]) auto
qed

lemma span-alt:
  span B = {(∑ x | f x ≠ 0. f x * s x) | f. {x. f x ≠ 0} ⊆ B ∧ finite {x. f x ≠ 0}}
  unfolding span-explicit' by auto

lemma span-finite:
  assumes fS: finite S
  shows span S = range (λu. ∑ v∈S. u v * s v)
  unfolding span-explicit
proof safe
  fix t r assume t ⊆ S then show (∑ a∈t. r a * s a) ∈ range (λu. ∑ v∈S. u v * s v)
    by (intro image_eqI[λv - λa. if a ∈ t then r a else 0])
      (auto simp: if_distrib[where t = r a, simplified] Int_absorb1)
next
  show ∃ t r. (∑ v∈S. u v * s v) = (∑ a∈t. r a * s a) ∧ finite t ∧ t ⊆ S for u
    by (intro exI[of - u] exI[of - S]) (auto intro: fS)
qed

lemma span-induct-alt [consumes 1, case_names base step, induct set: span]:
  assumes x: x ∈ span S
  assumes h0: h 0 and hS: (∀ x y. x ∈ S → h y → h (c * s x + y)
  shows h x
  using x unfolding span-explicit
proof safe
  fix t r assume finite t t ⊆ S then show h (∑ a∈t. r a * s a)
    by (induction t) (auto intro!: hS h0)
qed

lemma span-mono: A ⊆ B → span A ⊆ span B
by (auto simp: span-explicit)

lemma span-base: \( a \in S \implies a \in \text{span} S \)
by (auto simp: span-explicit intro: exI[of - \{a\}] exI[of - \lambda a. 1])

lemma span-superset: \( S \subseteq \text{span} S \)
by (auto simp: span-base)

lemma span-zero: \( 0 \in \text{span} S \)
by (auto simp: span-explicit intro: exI[of - {}])

lemma span-UNIV[simp]: \( \text{span} \ \text{UNIV} = \text{UNIV} \)
by (auto intro: span-base)

lemma span-add: \( x \in \text{span} S \implies y \in \text{span} S \implies x + y \in \text{span} S \)
unfolding span-explicit
proof safe
  fix tx ty rx ry
  assume *: finite tx finite ty tx \subseteq S ty \subseteq S
  have [simp]: \( (tx \cup ty) \cap tx = tx \) \( (tx \cup ty) \cap ty = ty \)
    by auto
  show \( \exists t. r. \big( \sum a \in tx. \ rx a \ast s a \big) + \big( \sum a \in ty. \ ry a \ast s a \big) = \big( \sum a \in t. \ r a \ast s a \big) \land \) finite t \land t \subseteq S
    apply (intro exI[of - tx \cup ty])
    apply (intro exI[of - \lambda a. \ (if a \in tx \ then rx a \ else 0) + (if a \in ty \ then ry a \ else 0)])
    apply (auto simp: * scale-left-distrib sum.distrib if-distrib [of \lambda r. r \ast s a for a] sum.If-cases)
  done
qed

lemma span-scale: \( x \in \text{span} S \implies c \ast s x \in \text{span} S \)
unfolding span-explicit
proof safe
  fix t r
  assume *: finite t t \subseteq S
  show \( \exists t'. r'. c \ast s (\sum a \in t. \ r a \ast s a) = (\sum a \in t'. \ r' a \ast s a) \land \) finite t' \land t' \subseteq S
    by (intro exI[of - t] exI[of - \lambda a. c \ast r a]) (auto simp: * scale-sum-right)
qed

lemma subspace-span [iff]: subspace (span S)
by (auto simp: subspace-def span-zero span-add span-scale)

lemma span-neg: \( x \in \text{span} S \implies - x \in \text{span} S \)
by (metis subspace-neg subspace-span)

lemma span-diff: \( x \in \text{span} S \implies y \in \text{span} S \implies x - y \in \text{span} S \)
by (metis subspace-span subspace-diff)

lemma span-sum: \( \bigwedge. x \in A \implies f x \in \text{span} S \implies \text{sum} f A \in \text{span} S \)
by (rule subspace-sum, rule subspace-span)
lemma span-minimal: $S \subseteq T \implies \text{subspace } T \implies \text{span } S \subseteq T$
  by (auto simp: span-explicit intro!: subspace-sum subspace-scale)

lemma span-def: $\text{span } S = \text{subspace hull } S$
  by (intro hull-unique|symmetric| span-superset subspace-span span-minimal)

lemma span-unique:
  $S \subseteq T \implies \text{subspace } T \implies (\forall T'. S \subseteq T' \implies \text{subspace } T' \implies T \subseteq T') \implies \text{span } S = T$
  unfolding span-def by (rule hull-unique)

lemma span-subspace-induct[consumes 2]:
  assumes $x: x \in \text{span } S$
   and $P$: subspace $P$
   and $SP$: $\forall x. x \in S \implies x \in P$
  shows $x \in P$
  proof
    from $SP$ have $SP': S \subseteq P$
      by (simp add: subset-eq)
    from $x$ hull-minimal[where $S=\text{subspace}$, OF $SP'$ $P$, unfolded span-def|symmetric]
    show $x \in P$
      by (metis subset-eq)
  qed

lemma (in module) span-induct[consumes 1, case-names base step, induct set: span]:
  assumes $x: x \in \text{span } S$
   and $P$: subspace $(\text{Collect } P)$
   and $SP$: $\forall x. x \in S \implies P x$
  shows $P x$
  using $P$ $SP$ span-subspace-induct $x$ by fastforce

lemma span-empty[simp]: $\text{span } \emptyset = \{0\}$
  by (rule span-unique) (auto simp add: subspace-def)

lemma span-subspace: $A \subseteq B \implies B \subseteq \text{span } A \implies \text{subspace } B \implies \text{span } A = B$
  by (metis order-antisym span-def hull-minimal)

lemma span-span: $\text{span } (\text{span } A) = \text{span } A$
  unfolding span-def hull-hull ..

lemma span-add-eq: assumes $x: x \in \text{span } S$ shows $x + y \in \text{span } S \iff y \in \text{span } S$
  proof
    assume $*: x + y \in \text{span } S$
    have $(x + y) - x \in \text{span } S$ using $*$ by (rule span-diff)
    then show $y \in \text{span } S$ by simp
THEORY "Modules"

qed (intro span-add \( x \))

lemma span-add-eq2: assumes \( y \in \text{span } S \) shows \( x + y \in \text{span } S \) ↔ \( x \in \text{span } S \)
using span-add-eq[of \( y \) \( S \) \( x \)] by (auto simp: ac-simps)

lemma span-singleton: \( \text{span } \{ x \} = \text{range } (\lambda k. k \ast s x) \)
by (auto simp: span-finite)

lemma span-Un: \( \text{span } (S \cup T) = \{ x + y \mid x, y \in \text{span } S \land y \in \text{span } T \} \)
proof safe
fix \( x \) assume \( x \in \text{span } (S \cup T) \)
then obtain \( t \) \( r \) where \( t \) : \( \text{finite } t \) \( t \subseteq S \cup T \) and \( x = (\sum_{a \in t} r a \ast s a) \)
by (auto simp: span-explicit)
moreover have \( t \cap S \cup (t - S) = t \) by auto
ultimately show \( \exists xa y. x = xa + y \land xa \in \text{span } S \land y \in \text{span } T \)
unfolding \( x \)
apply (rule-tac exI[of - \( \sum_{a \in t \cap S} r a \ast s a \)])
apply (rule-tac exI[of - \( \sum_{a \in t - S} r a \ast s a \)])
apply (subst sum.union-inter-neutral[symmetric])
apply (auto intro!: span-sum span-scale intro: span-base)
done

next
fix \( x \) \( y \) assume \( x \in \text{span } S \) \( y \in \text{span } T \) then show \( x + y \in \text{span } (S \cup T) \)
using span-mono[of \( S \cup T \)] span-mono[of \( T S \cup T \)]
by (auto intro!: span-add)

qed

lemma span-insert: \( \text{span } (\text{insert } a S) = \{ x. \exists k. (x - k \ast s a) \in \text{span } S \} \)
proof –
have \( \text{span } (\{ a \} \cup S) = \{ x. \exists k. (x - k \ast s a) \in \text{span } S \} \)
unfolding span-Un span-singleton
apply (auto simp add: set-eq_iff)
subgoal for \( y k \) by (auto intro!: exI[of - \( k \)])
subgoal for \( y k \) by (rule exI[of - \( k \ast s a \)], rule exI[of - \( y - k \ast s a \)]) auto
done
then show \( \exists \text{thesis by simp} \)

qed

lemma span-breakdown:
assumes \( bS \): \( b \in S \) and \( aS \): \( a \in \text{span } S \)
shows \( \exists k. a - k \ast s b \in \text{span } (S - \{ b \}) \)
using assms span-insert [of \( b S - \{ b \} \)]
by (simp add: insert-absorb)

lemma span-breakdown-eq: \( x \in \text{span } (\text{insert } a S) \) ↔ \( \exists k. x - k \ast s a \in \text{span } S \)
by (simp add: span-insert)
lemmas span-clauses = span-base span-zero span-add span-scale

lemma span-eq-iff[simp]: span s = s ≡ subspace s
  unfolding span-def by (rule hull-eq) (rule subspace-Inter)

lemma span-eq: span S = span T ≡ S ⊆ span T ∧ T ⊆ span S
  by (metis span-minimal span-subspace span-superset subspace-span)

lemma eq-span-insert-eq:
  assumes \((x - y) \in \text{span } S\)
  shows \(\text{span } (\text{insert } x S) = \text{span } (\text{insert } y S)\)
  proof
    have *: \(\text{span } (\text{insert } x S) \subseteq \text{span } (\text{insert } y S)\) if \((x - y) \in \text{span } S\) for \(x y\)
      proof
        have 1: \((r * s x - r * s y) \in \text{span } S\) for \(r\)
          by (metis scale-right-diff-distrib span-scale that)
        have 2: \((z - k * s y) - k * s (x - y) = z - k * s x\) for \(z k\)
          by (simp add: scale-right-diff-distrib)
        show ?thesis apply (clarsimp simp add: span-breakdown-eq)
          by (metis 1 2 diff-add-cancel scale-right-diff-distrib span-add-eq)
      qed
    qed
    show ?thesis apply (intro subset-antisym *: assms)
      using assms subspace-neg subspace-span minus-diff-eq
      by force
    qed

105 Dependent and independent sets

declaration dependent :: 'b set ⇒ bool
  where dependent-explicit: dependent s ≡ (∃t u. finite t ∧ t ⊆ s ∧ (∑v∈t. u v * s v) = 0 ∧ (∃v∈t. u v ≠ 0))

abbreviation independent s ≡ ¬ dependent s

lemma dependent-mono: dependent B =⇒ B ⊆ A =⇒ dependent A
  by (auto simp add: dependent-explicit)

lemma independent-mono: independent A =⇒ B ⊆ A =⇒ independent B
  by (auto intro: dependent-mono)

lemma dependent-zero: \(0 ∈ A =⇒ \text{dependent } A\)
  by (auto simp: dependent-explicit intro!: exI[of - λi. 1] exI[of : {0}])

lemma independent-empty[intro]: independent {}
  by (simp add: dependent-explicit)

lemma independent-explicit-module:
  independent s ≡ (∀t u v. finite t → t ⊆ s → (∑v∈t. u v * s v) = 0 → v)
unfolding dependent-explicit by auto

lemma independentD: independent s \implies finite t \implies t \subseteq s \implies (\sum_{v \in t} u \cdot v \ast s v) = 0 \implies v \in t \implies u \cdot v = 0

  by (simp add: independent-explicit-module)

lemma independent-Union-directed:
  assumes directed: \\\( \forall c \ d. \ c \in C \implies d \in C \implies c \subseteq d \lor d \subseteq c \) 
  assumes indep: \( \forall c. \ c \in C \implies independent c \) 
  shows independent (\bigcup C)
proof
  assume dependent (\bigcup C)
  then obtain u \ v \ S where S: finite S S \subseteq \bigcup C \ v \in S u \ v \neq 0 (\sum_{v \in S} u \cdot v \ast s v) = 0
  by (auto simp: dependent-explicit)

  have S \neq \{\}
    using \( \forall v \in S \) by auto
  have \( \exists c \in C. \ S \subseteq c \)
    using (finite S) (S \neq \{\}) (S \subseteq \bigcup C)
proof (induction rule: finite-ne-induct)
  case (insert i I)
  then obtain c d where cd: c \in C d \in C and iI: I \subseteq c i \in d
    by blast
  from directed[OF cd] cd have c \cup d \in C
    by (auto simp: sup.absorb1 sup.absorb2)
  with iI show ?case
    by (intro bexI[of - c \cup d]) auto
qed auto

then obtain c where c \in C S \subseteq c
  by auto

  have dependent c
    unfolding dependent-explicit
    by (intro exI[of - S] exI[of - u] bexI[of - v] conjI) fact+
  with indep[OF \( \forall c \in C \)] show False
    by auto
qed

lemma dependent-finite:
  assumes finite S
  shows dependent S \longleftrightarrow (\exists u. (\exists v \in S. \ u \cdot v \neq 0) \land (\sum_{v \in S} u \cdot v \ast s v) = 0)
(is \?lhs = \?rhs)
proof
  assume \?lhs
  then obtain T u v
    where finite T T \subseteq S v\in T u \cdot v \neq 0 (\sum_{v \in T} u \cdot v \ast s v) = 0
    by (force simp: dependent-explicit)
  with assms show \?rhs

apply (rule_tac x="λv. if v ∈ T then u v else 0" in exI)
apply (auto simp: sum_mono_neutral-right)
done

next
assume ?rhs with assms show ?lhs
  by (fastforce simp add: dependent_explicit)
qed

lemma dependent-alt:
 dependent B ⟷
 (∃X. finite {x. X x ≠ 0} ∧ {x. X x ≠ 0} ⊆ B ∧ (∑x|X x ≠ 0. X x * s x) =
  0 ∧ (∃x. X x ≠ 0))

unfolding dependent-explicit
apply safe

subgoal for S u v
apply (intro exI[of - "λx. if x ∈ S then u x else 0"])
apply (subst sum_mono_neutral_cong_left[where T=S])
apply (auto intro!: sum_mono_neutral_cong_right cong: rev_conj_cong)
done
apply auto
done

lemma independent-alt:
 independent B ⟷
 (∀X. finite {x. X x ≠ 0} −→ {x. X x ≠ 0} ⊆ B −→ (∑x|X x ≠ 0. X x * s x) =
  0 −→ (∀x. X x = 0))

unfolding dependent-alt by auto

lemma independentD-alt:
independent B ⟹ finite {x. X x ≠ 0} ⟹ {x. X x ≠ 0} ⊆ B ⟹ (∑x|X x ≠
  0. X x * s x) = 0 ⟹ X x = 0

unfolding independent-alt by blast

lemma independentD-unique:
assumes B: independent B
  and X: finite {x. X x ≠ 0} {x. X x ≠ 0} ⊆ B
  and Y: finite {x. Y x ≠ 0} {x. Y x ≠ 0} ⊆ B
  and (∑x|X x ≠ 0. X x * s x) = (∑x|Y x ≠ 0. Y x * s x)
shows X = Y

proof
  have X x - Y x = 0 for x
    using B
  proof (rule independentD-alt)
    have {x. X x - Y x ≠ 0} ⊆ {x. X x ≠ 0} ∪ {x. Y x ≠ 0}
      by auto
    then show finite {x. X x - Y x ≠ 0} {x. X x - Y x ≠ 0} ⊆ B
      using X Y by (auto dest: finite_subset)
    then have (∑x|X x - Y x ≠ 0. (X x - Y x) * s x) = (∑x∈{S. X S ≠ 0}
      ∪ {S. Y S ≠ 0}. (X v - Y v) * s v)
106 Representation of a vector on a specific basis

**Definition**: representation :: 'b set ⇒ 'b ⇒ 'a

**Where**: representation basis v =

  (if independent basis ∧ v ∈ span basis then
   SOME f. (∀ v. f v ≠ 0 → v ∈ basis) ∧ finite {v. f v ≠ 0} ∧ (∑ v∈{v. f v ≠ 0}. f v * s v) = v
   else (λb. 0))

**Lemma**: unique-representation:

**Assumes**: basis: independent basis

  and in-basis: ∀ v. f v ≠ 0 ⇒ v ∈ basis ∧ v. g v ≠ 0 ⇒ v ∈ basis

  and [simp]: finite {v. f v ≠ 0} ∧ finite {v. g v ≠ 0}

  and eq: (∑ v∈{v. f v ≠ 0}. f v * s v) = (∑ v∈{v. g v ≠ 0}. g v * s v)

**Shows**: f = g

**Proof** (rule ext, rule ccontr)

**Fix**: v assume ne: f v ≠ g v

**Have**: dependent basis

**Unfolding**: dependent-explicit

**Proof** (intro ex1 conj)

**Have**: ∃ v. f v - g v ≠ 0 ⊆ {v. f v ≠ 0} ∪ {v. g v ≠ 0}

  by auto

  show finite {v. f v - g v ≠ 0}

    by (rule finite-subset[OF *]) simp

  show ∃ v∈{v. f v - g v ≠ 0}. f v - g v ≠ 0

    by (rule bex1[of - v]) (auto simp: ne)

  have (∑ v | f v - g v ≠ 0). (f v - g v) * s v) =

    (∑ v∈{v. f v ≠ 0} ∪ {v. g v ≠ 0}. (f v - g v) * s v)

    by (intro sum_mono_neutral_cong_left *) auto

  also have ... =

    (∑ v∈{v. f v ≠ 0} ∪ {v. g v ≠ 0}. f v * s v) - (∑ v∈{v. f v ≠ 0} ∪ {v. g v ≠ 0}. g v * s v)
by (simp add: algebra-simps sum-subtractf)
also have \[ \ldots = (\sum v \mid f v \neq 0. f v * s v) - (\sum v \mid g v \neq 0. g v * s v) \]
by (intro arg-cong2[where f= (\sim)] sum.mono-neutral-cong-right) auto
finally show \[ (\sum v \mid f v - g v \neq 0. (f v - g v) * s v) = 0 \]
by (simp add: eq)
show \{v. f v - g v \neq 0\} \subseteq basis
using in-basis * by auto
qed

lemma
shows representation-ne-zero: \( \forall b. \) representation basis \( v \ b \neq 0 \implies b \in \text{basis} \)
and finite-representation: finite \( \{b. \) representation basis \( v \ b \neq 0\} \)
and sum-nonzero-representation-eq:
\( \text{independent basis} \implies v \in \text{span basis} \implies (\sum b \mid \text{representation basis} v \ b \neq 0. \text{representation basis} v \ b * s b) = v \)
proof –
\{ assume basis: independent basis and v: v \in \text{span basis} \\
define p where p f \leftarrow (\forall v. f v \neq 0 \implies v \in \text{basis}) \land \text{finite} \{v. f v \neq 0\} \land (\sum v \in \{v. f v \neq 0\}. f v * s v) = v \ for \ f \\
\begin{align*}
\text{obtain} \ t \ r \ where & : \text{finite} \ t \ t \subseteq \text{basis} \ (\sum \in \{r b \mid s b\} = v) \\
\begin{align*}
\text{using} \ & v \in \text{span basis} \ by (\text{auto simp: span-explicit}) \\
\text{define} \ f \ where & : f b = (\text{if} b \in t \ then \ r b \ else 0) \ for \ b \\
\text{have} \ & p f \\
\text{using} \ & (\text{auto simp: p-def f-def intro: sum.mono-neutral-cong-left}) \\
\text{have} \ & : \text{representation basis} v = \text{Eps p} \ by (\text{simp add: p-def[abs-def] representation-def basis v}) \\
\text{from} \ & \text{someI[of} p f, OF \ \text{OF} \ \text{OF}\} \ \text{have} \ p \ (\text{representation basis b}) \\
\text{unfolding} \ & * \ . \} \\
\text{note} \ & * = \text{this} \\
\begin{align*}
\text{show} \ & \text{representation basis} v \ b \neq 0 \implies b \in \text{basis for} \ b \\
\text{using} \ & * \ by (\text{cases independent basis} \land v \in \text{span basis}) (\text{auto simp: representation-def}) \\
\text{show} \ & \text{finite} \{b. \ text{representation basis} v \ b \neq 0\} \\
\text{using} \ & * \ by (\text{cases independent basis} \land v \in \text{span basis}) (\text{auto simp: representation-def}) \\
\text{show} \ & \text{independent basis} \implies v \in \text{span basis} \implies (\sum b \mid \text{representation basis} v \ b \neq 0. \text{representation basis} v \ b * s b) = v \\
\text{using} \ & * \ by \auto \\
\text{qed} \\
\text{lemma} \ & \text{sum-representation-eq:} \\
(\sum \in B. \text{representation basis} v \ b * s b) = v \\
\text{if} \ \text{independent basis} v \in \text{span basis} \ \text{finite} \ B \text{ basis} \subseteq B \}
proof  
  have \((\sum_{b \in B. \text{representation basis } v \ b \ast s \ b}) = \)  
  \((\sum_{b \mid \text{representation basis } v \ b \neq 0. \text{representation basis } v \ b \ast s \ b})\)  
  apply (rule sum.mono-neutral-cong)  
  apply (rule finite-representation)  
  apply fact  
  subgoal for \(b\)  
  using that representation-ne-zero[of basis v b]  
  by auto  
  subgoal by auto  
  subgoal by simp  
  done  
also have \(\ldots = v\)  
  by (rule sum-nonzero-representation-eq; fact)  
finally show \(?\text{thesis}\).  
qed 

lemma representation-eqI:  
assumes basis: independent basis and \(b: v \in \text{span basis}\)  
and ne-zero: \(\forall b. f b \neq 0 \implies b \in \text{basis}\)  
and finite: finite \(\{b. f b \neq 0\}\)  
and eq: \(\sum_{b \mid f b \neq 0. f b \ast s \ b} = v\)  
shows representation basis \(v = f\)  
by (rule unique-representation[OF basis])  
(auto simp: representation-ne-zero finite-representation sum-nonzero-representation-eq[OF basis b] ne-zero finite eq) 

lemma representation-basis:  
assumes basis: independent basis and \(b: b \in \text{basis}\)  
shows representation basis \(b = (\lambda v. \text{if } v = b \text{ then } 1 \text{ else } 0)\)  
proof (rule unique-representation[OF basis])  
show representation basis \(b v \neq 0 \implies v \in \text{basis for } v\)  
  using representation-ne-zero .  
show finite \(\{v. \text{representation basis } b v \neq 0\}\)  
  using finite-representation .  
show \(\text{if } v = b \text{ then } 1 \text{ else } 0 \neq 0 \implies v \in \text{basis for } v\)  
  by (cases v = b) (auto simp: b)  
have \(*: \{v. \text{if } v = b \text{ then } 1 \text{ else } 0 :: \text{'}a\} \neq 0\} = \{b\}\)  
  by auto  
show finite \(\{v. \text{if } v = b \text{ then } 1 \text{ else } 0 \neq 0\}\) unfolding * by auto  
show \((\sum v \mid \text{representation basis } b v \neq 0. \text{representation basis } b v \ast s \ v) = \)  
\((\sum v \mid (\text{if } v = b \text{ then } 1 \text{ else } 0::\text{'}a) \neq 0. (\text{if } v = b \text{ then } 1 \text{ else } 0) \ast s \ v)\)  
unfolding * sum-nonzero-representation-eq[OF basis span-base[OF b]] by auto  
qed 

lemma representation-zero: representation basis \(0 = (\lambda b. 0)\)  
proof cases  
assume basis: independent basis show \(?\text{thesis}\)  
  by (rule representation-eqI[OF basis span-zero]) auto
lemma representation-diff:
  assumes basis: independent basis and v: v ∈ span basis and w: u ∈ span basis
  shows representation basis (u − v) = (λb. representation basis u b − representation basis v b)
proof (rule representation-eqI[OF basis span-diff[OF u v]])
  let ?R = representation basis
  note finite-representation[simp] u[simp] v[simp]
  have *: {b. ?R u b − ?R v b ≠ 0} ⊆ {b. ?R u b ≠ 0} ∪ {b. ?R v b ≠ 0}
    by auto
  then show ?R u b − ?R v b ≠ 0 ⇒ b ∈ basis for b
    by (auto dest: representation-ne-zero)
  show finite {b. ?R u b − ?R v b ≠ 0}
    by (intro finite-subset[OF *]) simp-all
  have (∑ b ∈ {b. ?R u b − ?R v b ≠ 0}. ?R u b − ?R v b) * s b) =
    (∑ b ∈ {b. ?R u b ≠ 0}. ?R v b ≠ 0). ?R u b − ?R v b) * s b)
    by (intro sum_nonzero_representation_eq[OF basis])
  also have ... =
    (∑ b ∈ {b. ?R u b ≠ 0} ∪ {b. ?R v b ≠ 0}. ?R u b * s b) − (∑ b ∈ {b. ?R u b ≠ 0} ∪ {b. ?R v b ≠ 0}. ?R v b * s b)
    by (simp add: algebra_simps sum_subtractf)
  also have ... = (∑ b | ?R u b ≠ 0. ?R u b * s b) − (∑ b | ?R v b ≠ 0. ?R v b * s b)
    by (intro arg_cong2[where f = (−)] sum_mono_neutral_cong_right) auto
  finally show (sum b | ?R u b − ?R v b ≠ 0. (?R u b − ?R v b) * s b) = u − v
    by (simp add: sum_nonzero_representation_eq[OF basis])
qed

lemma representation-neg:
  independent basis = v ∈ span basis ⇒ representation basis (− v) = (λb. representation basis v b)
using representation-diff[of basis v 0] by (simp add: representation_zero span_zero)

lemma representation-add:
  independent basis = v ∈ span basis = u ∈ span basis =⇒
  representation basis (u + v) = (λb. representation basis u b + representation basis v b)
using representation-diff[of basis − v u] by (simp add: representation_neg representation_diff span_neg)

lemma representation-sum:
  independent basis = (∀i. i ∈ I =⇒ v i ∈ span basis) =⇒
  representation basis (sum v I) = (λb. ∑ i ∈ I. representation basis (v i) b)
by (induction I rule: infinite_finite_induct)
  (auto simp: representation_zero representation_add span_sum)

lemma representation-scale:
  assumes basis: independent basis and v: v ∈ span basis
shows representation basis \((r \ast s v) = (\lambda h. r \ast \text{representation basis } v b)\)

proof (rule representation-eqI[OF basis span-scale[OF v]])

let \(R = \text{representation basis}\)

note finite-representation[simp] v[simp]

have \(*: \{ b. r \ast ?R v b \neq 0 \} \subseteq \{ b. ?R v b \neq 0 \}\)

by auto

then show \(r \ast \text{representation basis } v b \neq 0 \implies b \in \text{basis for } b\)

using representation-ne-zero by auto

show finite \((\sum b | r \ast ?R v b \neq 0. (r \ast ?R v b) \ast s b) = (\sum b \in \{ b. ?R v b \neq 0 \}. (r \ast ?R v b) \ast s b)\)

by (intro finite-subset[OF *]) simp-all

also have \(\ldots = r \ast s (\sum b | ?R v b \neq 0. ?R v b \ast s b)\)

by (simp add: scale-scale[symmetric] scale-sum-right del; scale-scale)

finally show \((\sum b | r \ast ?R v b \neq 0. (r \ast ?R v b) \ast s b) = r \ast s v\)

by (simp add: sum-nonzero-representation-eq[OF basis])

qed

lemma representation-extend:

assumes basis: independent basis and \(v: v \in \text{span basis'} \text{ and basis'}: \text{basis'} \subseteq \text{basis}\)

shows representation basis \(v = \text{representation basis'} v\)

proof (rule representation-eqI[OF basis])

show \(v': v \in \text{span basis} \text{ using span-mono[OF basis']} v \text{ by auto}\)

have \(*: \text{independent basis'} \text{using basis'} \text{basis by (auto intro: dependent-mono)}\)

show representation basis' \(v b \neq 0 \implies b \in \text{basis for } b\)

using representation-ne-zero basis' by auto

show finite \((\sum b | \text{representation basis'} v b \neq 0)\)

using finite-representation .

show \((\sum b | \text{representation basis'} v b \neq 0. \text{representation basis'} v b \ast s b) = v\)

using sum-nonzero-representation-eq[OF * v] .

qed

The set \(B\) is the maximal independent set for \(\text{span } B\), or \(A\) is the minimal spanning set

lemma spanning-subset-independent:

assumes BA: \(B \subseteq A\)

and iA: independent A

and AsB: \(A \subseteq \text{span } B\)

shows \(A = B\)

proof (intro antisym[OF - BA] subsetI)

have iB: independent \(B\) using independent-mono[OF iA BA] .

fix \(v\) assume \(v \in A\)

with AsB have \(v \in \text{span } B\) by auto

let \(?RB = \text{representation } B\ v\) and \(?RA = \text{representation } A\ v\)

have \(?RB v = 1\)

unfolding representation-extend[OF iA \(v \in \text{span } B\), BA, symmetric] representation-basis[OF iA \(v \in A\)] by simp
then show \( v \in B \)
  using \texttt{representation-ne-zero[of B v v]} by auto
qed

end

A linear function is a mapping between two modules over the same ring.

locale module-hom = m1: module s1 + m2: module s2
  for s1 :: 'a::comm-ring-1 ⇒ 'b::ab-group-add ⇒ 'c (infixr ∗ 75)
  and s2 :: 'a::comm-ring-1 ⇒ 'c::ab-group-add ⇒ 'c (infixr ∗ 75) +
fixes f :: 'b ⇒ 'c
assumes add: \( f (b_1 + b_2) = f b_1 + f b_2 \)
  and scale: \( f (r ∗ a b) = r ∗ b f b \)
begin

lemma \texttt{zero[simp]}: \( f 0 = 0 \)
  using \texttt{scale[of 0 0]} by simp

lemma \texttt{neg}: \( f (- x) = - f x \)
  using \texttt{scale[where r = -1]} by (metis add add-eq-0-iff zero)

lemma \texttt{diff}: \( f (x - y) = f x - f y \)
  by (metis diff-conv-add-uminus add neg)

lemma \texttt{sum}: \( f (\texttt{sum g S}) = (\texttt{∑ a ∈ S. f (g a)}) \)
proof (induct S rule: infinite-finite-induct)
case (insert x F)
  have \( f (\texttt{sum g (insert x F)}) = f (g x + \texttt{sum g F}) \)
    using \texttt{insert.hyps by simp}
  also have \( \ldots = f (g x) + f (\texttt{sum g F}) \)
    using \texttt{add by simp}
  also have \( \ldots = (\texttt{∑ a ∈ insert x F. f (g a)}) \)
    using \texttt{insert.hyps by simp}
finally show \( \texttt{case} \).
qed simp-all

lemma \texttt{inj-on-iff-eq-0}:
  assumes s: \texttt{m1.subspace s}
  shows \texttt{inj-on f s} \( (\forall x \in s. f x = 0 \longrightarrow x = 0) \)
proof
  have \( \texttt{inj-on f s} \longleftrightarrow (\forall x \in s. f x = 0 \longrightarrow x = y = 0) \)
    by (simp add: \texttt{inj-on-def})
  also have \( \ldots \\longleftrightarrow (\forall x \in s. f (x - y) = 0 \longrightarrow x - y = 0) \)
    by (simp add: \texttt{diff})
  also have \( \ldots \\longleftrightarrow (\forall x \in s. f x = 0 \longrightarrow x = 0) \) (is \( \texttt{?l = ?r} \))
proof safe
  fix x assume \( \texttt{?l} \) assume \( x \in s \) \( f x = 0 \) with \( \texttt{?l}[\texttt{rule-format, of x 0}] \) s show \( x = 0 \)
    by (auto simp: \texttt{m1.subspace-0})
THEORY "Modules"

next
  fix \( x \) \( y \) assume \( ?r \) assume \( x \in s y \in s f (x - y) = 0 \)
  with \( (?r)\text{[rule-format, of } x - y\text{]} s \)
  show \( x - y = 0 \)
    by (auto simp: m1.subspace-diff)
qed

finally show \( ?\text{thesis} \)
  by auto

qed

lemma inj-if-iff-eq-0
  assumes \( \text{inj f} = (\forall x. f x = 0 \rightarrow x = 0) \)
  by (rule inj-on-iff-eq-0[OF m1.subspace-UNIV, unfolded ball-UNIV])

lemma subspace-image: assumes \( S : m1.\text{subspace} \) shows \( m2.\text{subspace} (f ' S) \)
  unfolding m2.subspace-def
proof safe
  show \( 0 \in f ' S \)
    by (rule image-eqI[of - - 0]) (auto simp: S m1.subspace-0)
  show \( x \in S \rightarrow y \in S \rightarrow f x + f y \in f ' S \) for \( x y \)
    by (rule image-eqI[of - - x + y]) (auto simp: m1.subspace-add add)
  show \( x \in S \rightarrow r \ast b f x \in f ' S \) for \( r x \)
    by (rule image-eqI[of - - r \ast a x]) (auto simp: S m1.subspace-scale scale)
qed

lemma subspace-vimage: \( m2.\text{subspace} S \rightarrow m1.\text{subspace} (f ^- ' S) \)
  by (simp add: vimage-def add scale m1.subspace-def m2.subspace-def m2.subspace-scale)

lemma subspace-kernel: \( m1.\text{subspace} \{ x. f x = 0 \} \)
  using subspace-vimage[OF m2.subspace-single-0] by (simp add: vimage-def)

lemma span-image: \( m2.\text{span} (f ' S) = f ' (m1.\text{span} S) \)
proof (rule m2.span-unique)
  show \( f ' S \subseteq f ' m1.\text{span} S \)
    by (rule image-monotone, rule m1.span-superset)
  show \( m2.\text{span} (f ' m1.\text{span} S) \)
    using m1.subspace-span by (rule subspace-image)
next
  fix \( T \) assume \( f ^- ' S \subseteq T \) and \( m2.\text{subspace} T \) then show \( f ^- ' m1.\text{span} S \subseteq T \)
  unfolding image-subset-iff-subset-vimage by (metis subspace-vimage m1.span-minimal)
qed

lemma dependent-inj-imageD:
  assumes \( d : m2.\text{dependent} (f ^- ' s) \) and \( i : \text{inj-on} f (m1.\text{span} s) \)
  shows \( m1.\text{dependent} s \)
proof
  have [intro]: \( \text{inj-on} f s \)
    using \( \text{inj-on} f (m1.\text{span} s) \) m1.span-superset by (rule inj-on-subset)
  from \( d \) obtain \( s' \) \( r v \) where \( * : \text{finite} s' s' \subseteq s (\sum v \in f ^- ' s'. r v \ast b v) = 0 v \in s' \)
\[
\text{r (f v)} \neq 0
\]
by \(\text{(auto simp: m2.dependent-explicit subset-image-iff dest!: finite-imageD intro: inj-on-subset)}\)
\[
\text{have f } (\sum \text{v} \in s'. \text{ r (f v) } * \text{a v}) = (\sum \text{v} \in s'. \text{ r (f v) } * \text{b f v})
\]
by \(\text{(simp add: sum scale)}\)
also have ... = (\sum \text{v} \in f \ ' s'. \text{ r v } * \text{b v})
using \(\text{(subst sum.reindex)}\) \(\text{(auto dest!: finite-imageD intro: inj-on-subset)}\)
finally have f (\sum \text{v} \in s'. \text{ r (f v) } * \text{a v}) = 0
by \(\text{(simp add: *)}\)
with \(\text{(subst sum.reindex)}\) \(\text{(auto simp add: *)}\)
\[
\text{r v} \neq 0 \Rightarrow \text{sum scale}
\]
\[
\text{have f (\sum \text{v} \in s'. \text{ r (f v) } * \text{a v}) = 0}
\]
by \(\text{(intro inj-on-D[OF \text{f}] m1.span-zero m1.span-sum m1.span-scale)}\) \(\text{(auto intro: m1.span-base)}\)
then show m1.dependent s
using \(\text{(subth-injective-f-f[f b] the-inv-into f-f \text{m2.span-explicit')}}\)
\[
\text{f v} \neq 0 \Rightarrow \text{f v} \neq 0 \Rightarrow \text{f v} \neq 0
\]
by \(\text{(force simp add: m1.dependent-explicit)}\)
qed

lemma eq-0-on-span:
assumes \(\text{f0: } \bigwedge \text{x. x } \in \text{b } \Rightarrow \text{f x = 0 and x } \in \text{m1.span b shows f x = 0} \)
using m1.span-induct[OF m1.span-base] \(\text{f0 by simp}\)

lemma independent-injective-image: m1.independent s \(\Rightarrow\) inj-on f (m1.span s)
\[\Rightarrow m2.independent (f ' s)\]
using dependent-injective-image[OF s] \(\text{by auto}\)

lemma inj-on-span-independent-image:
assumes \(\text{if B: m2.independent (f ' B) and f: inj-on f B shows inj-on f (m1.span B)}\)
\[
\text{unfolding inj-on-D[of f] inj-on-D[of s] unfolding m1.span-explicit'}\]
\[
\text{fix r assume fr: finite \{v. r v \neq 0\} and r: } \forall \text{x. x } \in \text{B } \Rightarrow \text{v } \in \text{B}
\]
and eq0 = \(\text{f (}\sum \text{v} | \text{r v } \neq 0 \Rightarrow \text{r v } * \text{a v}) = 0\)
\[
\text{have 0 = (}\sum \text{v} | \text{r v } \neq 0 \Rightarrow \text{r v } * \text{b f v})
\]
using eq0 by \(\text{(simp add: sum scale)}\)
also have ... = (\sum \text{v} \in f \ ' \{v. r v \neq 0\}. \text{r (the-inv-into f f) } * \text{b v})
using r by \(\text{(subst sum.reindex)}\) \(\text{(auto simp: the-inv-into-f-f[f OF f] intro: inj-on-subset[of f] sum.cong)}\)
finally have r v \neq 0 \Rightarrow (\text{the-inv-into f f (f v)}) = 0 \text{ for v}
using fr ifB[unfolded m2.independent-explicit-module, rule-format,
of f \ ' \{v. r v \neq 0\} \lambda v. \text{r (the-inv-into f f v)}]
by auto
then show r v = 0 \text{ for v}
using the-inv-into-f-f[f OF f] \(\text{r by auto}\)
then show (\sum \text{v} | \text{r v } \neq 0 \Rightarrow \text{r v } * \text{a v}) = 0 \text{ by auto}
qed

lemma inj-on-span-iff-independent-image: m2.independent (f ' B) \(\Rightarrow\) inj-on f (m1.span B) \(\Leftarrow\) inj-on f B
using inj-on-span-independent-image[OF B] \(\text{inj-on-subset[of f OF - m1.span-superset, of f B] by auto}\)
lemma subspace-linear-preimage: \( m2.\text{subspace} S \implies m1.\text{subspace} \{ x. \, f x \in S \} \)
by (simp add: add scale m1.\text{subspace-def} m2.\text{subspace-def})

lemma spans-image: \( V \subseteq m1.\text{span} B \implies f ' V \subseteq m2.\text{span} (f ' B) \)
by (metis image-mono span-image)

Relation between bases and injectivity/surjectivity of map.

lemma spanning-surjective-image:
assumes us: \( \text{UNIV} \subseteq m1.\text{span} S \)
and sf: surj f
shows \( \text{UNIV} \subseteq m2.\text{span} (f ' S) \)
proof
  have \( \text{UNIV} \subseteq f ' \text{UNIV} \)
  using sf by (auto simp add: surj-def)
  also have \( \ldots \subseteq m2.\text{span} (f ' S) \)
  using spans-image [OF us].
  finally show \( \?\text{thesis} \).
qed

lemmas independent-inj-on-image = independent-injective-image

lemma independent-inj-image:
\( m1.\text{independent} S \implies \text{inj} f \implies m2.\text{independent} (f ' S) \)
using independent-inj-on-image [of S] by (auto simp: subset-inj-on)
end

lemma module-hom-iff:
module-hom s1 s2 f \rightleftharpoons
module s1 \land module s2 \land
(\forall x y. f (x + y) = f x + f y) \land (\forall c x. f (s1 c x) = s2 c (f x))
by (simp add: module-hom-def module-hom-axioms-def)

locale module-pair = m1:: module s1 + m2:: module s2
for s1 :: 'a:: comm-ring-1 \Rightarrow 'b \Rightarrow 'b:: ab-group-add
and s2 :: 'a:: comm-ring-1 \Rightarrow 'c \Rightarrow 'c:: ab-group-add
begin

lemma module-hom-zero: module-hom s1 s2 (\lambda x. 0)
by (simp add: module-hom-iff m1.module-axioms m2.module-axioms)

lemma module-hom-add: module-hom s1 s2 f \rightleftharpoons module-hom s1 s2 g \rightleftharpoons module-hom s1 s2 (\lambda x. f x + g x)
by (simp add: module-hom-iff module.scale-right-distrib)

lemma module-hom-sub: module-hom s1 s2 f \rightleftharpoons module-hom s1 s2 g \rightleftharpoons module-hom s1 s2 (\lambda x. f x - g x)
by (simp add: module-hom-iff module.scale-right-diff-distrib)
lemma module-hom-neg: module-hom s1 s2 f \rightarrow module-hom s1 s2 (\lambda x. - f x) 
  by (simp add: module-hom-iff module.scale-minus-right)

lemma module-hom-scale: module-hom s1 s2 f \rightarrow module-hom s1 s2 (\lambda x. s2 c (f x)) 
  by (simp add: module-hom-iff module.scale-scale module.scale-right-distrib ac-simps)

lemma module-hom-compose-scale: 
  module-hom s1 s2 (\lambda x. s2 (f x) (c)) 
  if module-hom s1 (\*) f 
proof –
  interpret mh: module-hom s1 (\*) f by fact 
show ?thesis 
  by unfold-locales (simp-all add: mh.add mh.scale m2.scale-left-distrib)
  qed

lemma bij-module-hom-imp-inv-module-hom: module-hom scale1 scale2 f \rightarrow bij f 
  \rightarrow module-hom scale2 scale1 (inv f) 
  by (auto simp: module-hom-iff bij-is-surj bij-is-inj surj-f-inv-f intro: Hilbert-Choice.inv-f-eq)

lemma module-hom-sum: (\forall i. i \in I \rightarrow module-hom s1 s2 (f i)) \rightarrow (I = \{\}) \rightarrow module s1 \land module s2 \rightarrow module-hom s1 s2 (\lambda x. \sum i \in I. f i x) 
  apply (induction I rule: infinite-finite-induct) 
  apply (auto intro: module-hom-zero module-hom-add) 
  using m1.module-axioms m2.module-axioms by blast

lemma module-hom-eq-on-span: f x = g x 
  if module-hom s1 s2 f module-hom s1 s2 g 
  and ((\forall x. x \in B \rightarrow f x = g x) x \in m1.span B) 
proof –
  interpret module-hom s1 s2 \lambda x. f x - g x 
  by (rule module-hom-sub that)+ 
  from eq-0-on-span[OF - that(4)] that(3) show ?thesis by auto
  qed

end

context module begin

lemma module-hom-scale-self[simp]: 
  module-hom scale scale (\lambda x. scale c x) 
  using module-axioms module-hom-iff scale-left-commute scale-right-distrib by blast

lemma module-hom-scale-left[simp]: 
  module-hom (\*) scale (\lambda r. scale r x)
by unfold-locales (auto simp: algebra-simps)

lemma module-hom-id: module-hom scale scale id
  by (simp add: module-hom-iff module-axioms)

lemma module-hom-ident: module-hom scale scale (λx. x)
  by (simp add: module-hom-iff module-axioms)

lemma module-hom-uminus: module-hom scale scale uminus
  by (simp add: module-hom-iff module-axioms)

end

lemma module-hom-compose: module-hom s1 s2 f ⇒ module-hom s2 s3 g ⇒ module-hom s1 s3 (g o f)
  by (auto simp: module-hom-iff)

end

107 Vector Spaces

theory Vector-Spaces
  imports Modules
begin

lemma isomorphism-expand:
  f ◦ g = id ∧ g ◦ f = id ↔ (∀x. f (g x) = x) ∧ (∀x. g (f x) = x)
  by (simp add: fun-eq-iff o-def id-def)

lemma left-right-inverse-eq:
  assumes fg: f ◦ g = id and gh: g ◦ h = id
  shows f = h
proof
  have f = f ◦ (g ◦ h)
    unfolding gh by simp
  also have ... = (f ◦ g) ◦ h
    by (simp add: o-assoc)
  finally show f = h
    unfolding fg by simp
qed

lemma ordLeq3-finite-infinite:
  assumes A: finite A and B: infinite B
  shows ordLeq3 (card-of A) (card-of B)
proof
  have ordLeq3 (card-of A) (card-of B) ∨ ordLeq3 (card-of B) (card-of A)
    by (intro ordLeq-total card-of-Well-order)
  moreover have ¬ ordLeq3 (card-of B) (card-of A)
    using B A card-of-ordLeq-finite[of B A] by auto
ultimately show \( ?\text{thesis by auto} \)

qed

locale vector-space =
  fixes scale :: 'a::field \Rightarrow 'b::{ab-group-add} \Rightarrow 'b (infixr * 75)
  assumes vector-space-assms:— re-stating the assumptions of module instead of extending module allows us to rewrite in the sublocale.
  \( a * (x + y) = a * x + a * y \)
  \( (a + b) * x = a * x + b * x \)
  \( a * (b * x) = (a * b) * x \)
  \( 1 * x = x \)

lemma module-iff-vector-space: \( \text{module s} \iff \text{vector-space s} \)
fold unfolding module-def vector-space-def ..

locale linear = vs1::vector-space s1 + vs2::vector-space s2 + module-hom s1 s2 f
for s1 :: 'a::field \Rightarrow 'b::{ab-group-add} \Rightarrow 'b (infixr * a 75)
and s2 :: 'a::field \Rightarrow 'c::{ab-group-add} \Rightarrow 'c (infixr * b 75)
and f :: 'b \Rightarrow 'c

lemma module-hom-iff-linear: \( \text{module-hom s1 s2 f} \iff \text{linear s1 s2 f} \)
fold unfolding module-hom-def linear-def module-iff-vector-space by auto

lemmas module-hom-eq-linear = module-hom-iff-linear[abs-def, THEN meta-eq-to-obj-eq]
lemmas linear-module-homI = module-hom-iff-linear[THEN iffD1]
and module-hom-linearI = module-hom-iff-linear[THEN iffD2]

context vector-space begin

sublocale module scale rewrites module-hom = linear
by unfold-locales (fact vector-space-assms module-hom-eq-linear)

lemmas— from module
  linear-id = module-hom-id
and linear-ident = module-hom-ident
and linear-scale-self = module-hom-scale-self
and linear-scale-left = module-hom-scale-left
and linear-uminus = module-hom-uminus

lemma linear-imp-scale:
  fixes D::'a \Rightarrow 'b
  assumes linear (*) scale D
  obtains d where D = (\lambda x. scale x d)
  proof —
  interpret linear (*) scale D by fact
  show ?thesis
  by (metis mult.commute mult.left-neutral scale that)
  qed
lemma \textit{scale-eq-0-iff} [simp]: \textit{scale} a \ x = 0 \iff a = 0 \lor \ x = 0  
by \textit{(metis scale-left-commute right-inverse scale-one scale-scale scale-zero-left)}

lemma \textit{scale-left-imp-eq};  
\begin{itemize}  
\item \textbf{assumes nonzero:} a \neq 0  
\item \textbf{and scale:} \textit{scale} a \ x = \textit{scale} a \ y  
\item \textbf{shows} \ x = \ y  
\end{itemize}  
\textbf{proof} \ 
\begin{itemize}  
\item \textbf{from} \textit{scale} have \textit{scale} a \ (x - y) = 0  
\item \textbf{with nonzero} have \ x - y = 0 \textbf{ by simp}  
\item \textbf{then show} \ x = \ y \textbf{ by (simp only: right-minus-eq)}  
\end{itemize}  
\textbf{qed}

lemma \textit{scale-right-imp-eq};  
\begin{itemize}  
\item \textbf{assumes nonzero:} x \neq 0  
\item \textbf{and scale:} \textit{scale} a \ x = \textit{scale} b \ x  
\item \textbf{shows} \ a = b  
\end{itemize}  
\textbf{proof} \ 
\begin{itemize}  
\item \textbf{from} \textit{scale} have \textit{scale} (a - b) \ x = 0  
\item \textbf{with nonzero} have \ a - b = 0 \textbf{ by simp}  
\item \textbf{then show} \ a = b \textbf{ by (simp only: right-minus-eq)}  
\end{itemize}  
\textbf{qed}

lemma \textit{scale-cancel-left} [simp]; \textit{scale} a \ x = \textit{scale} a \ y \iff \ x = \ y \lor a = 0  
by \textit{(auto intro: scale-left-imp-eq)}

lemma \textit{scale-cancel-right} [simp]; \textit{scale} a \ x = \textit{scale} b \ x \iff a = b \lor \ x = 0  
by \textit{(auto intro: scale-right-imp-eq)}

lemma \textit{injective-scale}: c \neq 0 \Rightarrow \textit{inj} (\lambda x. \textit{scale} c \ x)  
by \textit{(simp add: inj-on-def)}

lemma \textit{dependent-def}: dependent P \iff (\exists a \in P. a \in \textit{span} (P - \{a\}))  
\textbf{unfolding dependent-explicit}  
\textbf{proof} \textit{safe}  
\begin{itemize}  
\item fix a assume \textit{aP}: a \in P \textbf{ and } a \in \textit{span} (P - \{a\})  
\item then obtain a S u  
\item where \textit{aP}: a \in P \textbf{ and } \textit{fS}: \textit{finite} S \textbf{ and } \textit{SP}: S \subseteq P \ a \notin S \textbf{ and } \textit{ua}: (\sum v \in S. u v * s v) = a  
\item unfolding \textit{span-explicit} \textbf{ by blast}  
\item let \textit{?S} = \textit{insert} a S  
\item let \textit{?u} = \textit{\lambda y. if y = a then -1 else u y}  
\item from \textit{fS SP have} (\sum v \in S. \textit{?u v * s v}) = 0  
\item by \textit{(simp add: if-distrib[of \lambda r. r * s a ] sum.If-cases field-simps Diff-eq[ symmetric ] ua)}  
\item moreover have \textit{finite} \textit{?S} \textit{?S} \subseteq P \ a \in \textit{?S} \ ?u a \neq 0  
\item using \textit{fS SP aP} \textbf{ by auto}  
\end{itemize}
ultimately show \( \exists t u. \) finite \( t \subseteq P \) \& \( (\sum v \in t. \ u \ v \cdot s \ v) = 0 \) \& \( (\exists v \in t. \ u \ v \neq 0) \) by fast

next

fix \( S u v \)
assume \( fS: \) finite \( S \) and \( SP: \) \( S \subseteq P \) and \( vS: \) \( v \in S \)
and \( wv: \) \( u \ v \neq 0 \) and \( w: (\sum v \in S. \ u \ v \cdot s \ v) = 0 \)
let \( ?a = v \)
let \( ?S = S - \{v\} \)
let \( ?u = \lambda i. \ (u \ i) / \ u \ v \)
have \( \text{th0: } ?a \in P \) finite \( ?S \) \( ?S \subseteq P \)
using \( fS \ SP \ vS \) by auto
have \( (\sum v \in ?S. \ ?u \ v \cdot s \ v) = (\sum v \in S. \ (\text{inverse} \ (u \ ?a))) \cdot s \ (u \ v \cdot s \ v) - ?u \ v \cdot s \ v \)
using \( fS \ vS \) by (simp add: sum-diff1 field-simps)
also have \( ... = ?a \)
unfolding \( \text{scale-sum-right[symmetric]} \) \( u \) using \( wv \) by simp
finally have \( (\sum v \in S. \ ?u \ v \cdot s \ v) = ?a \).
with \( \text{th0} \) show \( \exists a \in P. \ a \in \text{span} \ (P - \{a\}) \)
unfolding \( \text{span-explicit} \) by (auto intro!: bexI[where \( x=?a \)] exI[where \( x=?S \)] exI[where \( x=?a \)]
qed

lemma \( \text{dependent-single[simp]}: \) dependent \( \{x\} \longleftrightarrow x = 0 \)
unfolding \( \text{dependent-def} \) by auto

lemma \( \text{in-span-insert}: \)
assumes \( a: \) \( a \in \text{span} \ (\text{insert} \ b \ S) \)
and \( na: \) \( a \notin \text{span} \ S \)
shows \( b \in \text{span} \ (\text{insert} \ a \ S) \)
proof –
from \( \text{span-breakdown[of } b \ \text{insert} \ b \ S \ a, \ OF \ \text{insertI1 } a] \)
obtain \( k \) where \( k: \) \( a - k \cdot s \ b \in \text{span} \ (S - \{b\}) \) by auto
have \( k \neq 0 \)
proof
assume \( k = 0 \)
with \( k \cdot \text{span-mono[of } S - \{b\} \ S \) have \( a \in \text{span} \ S \) by auto
with \( na \) show False by blast
qed
then have \( eq: \) \( b = (1/k) \cdot s \ a - (1/k) \cdot s \ (a - k \cdot s \ b) \)
by (simp add: algebra-simps)
from \( k \) have \( (1/k) \cdot s \ (a - k \cdot s \ b) \in \text{span} \ (S - \{b\}) \)
by (rule \( \text{span-scale} \))
also have \( ... \subseteq \text{span} \ (\text{insert} \ a \ S) \)
by (rule \( \text{span-mono} \) auto)
finally show \( \text{?thesis} \)
using \( k \) by (subst \( eq \) ) (blast intro: \( \text{span-diff} \ \text{span-scale} \ \text{span-base} \))
qed
**THEORY “Vector-Spaces”**

**lemma dependent-insertD**: assumes a: a ∉ span S and S: dependent (insert a S) shows dependent S

**proof** –

have a ∉ S using a by (auto dest: span-base)

obtain b where b: b = a ∨ b ∈ S b ∈ span (insert a S − {b})

using S unfolding dependent-def by blast

have b ≠ a b ∈ S

using b (a ∉ S) a by auto

with b have *: b ∈ span (insert a (S − {b})) by (rule in-span-insert)

with a show ?thesis by (auto simp: insert-absorb)

qed

**lemma independent-insertI**: a ∉ span S ⇒ independent S ⇒ independent (insert a S)

by (auto dest: dependent-insertD)

**lemma independent-insert**: independent (insert a S) ←→ (if a ∈ S then independent S else independent S ∧ a ∉ span S)

**proof** –

have a ∉ S ⇒ a ∈ span S ⇒ dependent (insert a S)

by (auto simp: dependent-def)

then show ?thesis by (auto intro: dependent-mono simp: independent-insertI)

qed

**lemma maximal-independent-subset-extend**: assumes S ⊆ V independent S

obtains B where S ⊆ B B ⊆ V independent B V ⊆ span B

**proof** –

let ?C = {B. S ⊆ B ∧ independent B ∧ B ⊆ V}

have ∃M ∈ ?C. ∀X ∈ ?C. M ⊆ X ⇒ X = M

**proof** (rule subset-Zorn)

fix C :: 'b set set assume subset.chain ?C C

then have C: ∧c. c ∈ C ⇒ c ⊆ V ∧c. c ∈ C ⇒ S ⊆ c ∧c. c ∈ C ⇒ independent c

∧c d. c ∈ C ⇒ d ∈ C ⇒ c ⊆ d ∨ d ⊆ c

unfolding subset.chain-def by blast+
show \( \exists U \subseteq C. \forall X \subseteq C. X \subseteq U \)

proof cases 
  assume \( C = \{\} \) with assms show \( \text{thesis} \)
  by (auto intro: exI[of - S])

next 
  assume \( C \neq \{\} \)
  with \( C(2) \) have \( S \subseteq \bigcup C \)
  by auto 
  moreover have independent \( (\bigcup C) \)
  by (intro independent-Union-directed C)
  moreover have \( \bigcup C \subseteq V \)
  using \( C \) by auto 
  ultimately show \( \text{thesis} \)
  by auto 
qed 

then obtain \( B \) where \( B \subseteq V \) independent \( B \) \( V \subseteq B \)
and max: \( \forall S. \) independent \( S \) \( \implies S \subseteq V \implies B \subseteq S \implies S = B \)
by auto
moreover 
\{
  assume \( \neg V \subseteq \text{span } B \)
  then obtain \( v \) where \( v \in V \) \( v \notin \text{span } B \)
  by auto 
  with \( B \) have independent \( (\text{insert } v \ B) \)
  by (auto intro: dependent-insertD)
  from \( \text{max[OF this]} \) \( \langle v \in V \rangle \langle B \subseteq V \rangle \)
  have \( v \in B \)
  by auto 
  with \( v \notin \text{span } B \) have False
  by (auto intro: span-base) 
\}
ultimately show \( \text{thesis} \)
by (meson that)

qed 

lemma maximal-independent-subset:
obtains \( B \) where \( B \subseteq V \) independent \( B \) \( V \subseteq \text{span } B \)
by (metis maximal-independent-subset-extend[of \{\}] empty-subsetI independent-empty)

Extends a basis from \( B \) to a basis of the entire space.

definition extend-basis :: \('b set \Rightarrow 'b set\)
where extend-basis \( B \) = (SOME \( B' \). \( B \subseteq B' \wedge \text{independent } B' \wedge \text{span } B' = \text{UNIV} )

lemma 
assumes \( B \): independent \( B \)
shows extend-basis-superset: \( B \subseteq \text{extend-basis } B \)
and independent-extend-basis: independent \( (\text{extend-basis } B) \)
and span-extend-basis(simp): \( \text{span } (\text{extend-basis } B) = \text{UNIV} \)
proof —
define \( p \) where \( p \ B' \equiv B \subseteq B' \wedge \text{independent } B' \wedge \text{span } B' = \text{UNIV} \) for \( B' \)
obtain $B'$ where $p$ $B'$
  using maximal-independent-subset-extend[OF subset-UNIV $B$]
  by (metis top.extremum-uniqueI p-def)
then have $p$ (extend-basis $B$)
  unfolding extend-basis-def p-def[symmetric] by (rule someI)
then show $B \subseteq$ extend-basis $B$ independent (extend-basis $B$) span (extend-basis $B$) = UNIV
  by (auto simp: p-def)
qed

lemma in-span-delete:
  assumes $a$: $a$ $\in$ span $S$ and $na$: $a$ $\notin$ span ($S - \{b\}$)
  shows $b$ $\in$ span ($\text{insert } a (S - \{b\})$)
  by (metis Diff-empty Diff-insert0 a in-span-insert insert-Diff $na$)

lemma span-redundant: $x$ $\in$ span $S$ $\implies$ span ($\text{insert } x S$) = span $S$
  unfolding span-def by (rule hull-redundant)

lemma span-trans: $x$ $\in$ span $S$ $\implies$ $y$ $\in$ span ($\text{insert } x S$) $\implies$ $y$ $\in$ span $S$
  by (simp only: span-redundant)

lemma span-insert-0 [simp]: span ($\text{insert } 0 S$) = span $S$
  by (metis span-zero span-redundant)

lemma span-delete-0 [simp]: span ($S - \{0\}$) = span $S$
  proof
    show span ($S - \{0\}$) $\subseteq$ span $S$
      by (blast intro!: span-mono)
    next
      have span $S$ $\subseteq$ span($\text{insert } 0 (S - \{0\})$)
        by (blast intro!: span-mono)
      also have ... $\subseteq$ span($S - \{0\}$)
        using span-insert-0 by blast
      finally show span $S$ $\subseteq$ span ($S - \{0\}$).
  qed

lemma span-image-scale:
  assumes finite $S$ and nz: $\forall x. x \in S \Longrightarrow c x \neq 0$
  shows span (($\lambda x. c x * s x$) '$ S) = span $S$
  using assms
  proof (induction $S$ arbitrary: $c$)
    case (empty $c$) show ?case by simp
  next
    case (insert $x F$ $c$)
    show ?case
    proof (intro set-eqI iffI)
      fix $y$
      assume $y$ $\in$ span (($\lambda x. c x * s x$) '$ insert $x F$)
      then show $y$ $\in$ span ($\text{insert } x F$)
    qed
  qed
using insert by (force simp: span-breakdown-eq)

next

fix y

assume y ∈ span (insert x F)

then show y ∈ span (λx. c x * s x) ‘ insert x F

using insert

apply (clarsimp simp: span-breakdown-eq)

apply (rule-tac x=k / c x in ext)

by simp

qed

lemma exchange-lemma:

assumes f: finite T

and i: independent S

and sp: S ⊆ span T

shows ∃ t’. card t’ = card T ∧ finite t’ ∧ S ⊆ t’ ∧ t’ ⊆ S ∪ T ∧ S ⊆ span t’

using f i sp

proof (induct card (T − S) arbitrary: S T rule: less-induct)

case less

note ft = (finite T) and S = (independent S) and sp = (S ⊆ span T)

let ?P = λt’. card t’ = card T ∧ finite t’ ∧ S ⊆ t’ ∧ t’ ⊆ S ∪ T ∧ S ⊆ span t’

show ?case

proof (cases S ⊆ T ∨ T ⊆ S)

case True

then show ?thesis

proof

assume S ⊆ T then show ?thesis

by (metis ft Un-commute sp sup-ge1)

next

assume T ⊆ S then show ?thesis

by (metis Un-absorb sp spanning-subset-independent[OF - S sp] ft)

qed

next

case False

then have st: ¬ S ⊆ T − T ⊆ S

by auto

from st(2) obtain b where b: b ∈ T b /∈ S

by blast

from b have T − {b} − S ⊆ T − S

by blast

then have cardlt: card (T − {b} − S) < card (T − S)

using ft by (auto intro: psubset-card_mono)

from b ft have c0: card T ≠ 0

by auto

show ?thesis

proof (cases S ⊆ span (T − {b}))

case True

from ft have ftb: finite (T − {b})
by auto
from less(1)[OF cardlt ftb S True]
obtain U where U: card U = card (T - {b}) S ⊆ U U ⊆ S ∪ (T - {b})
S ⊆ span U
  and fu: finite U by blast
let ?w = insert b U
have th0: S ⊆ insert b U
  using U by blast
have th1: insert b U ⊆ S ∪ T
  using U b by blast
have bu: b /∈ U
  using b U by blast
from U(1) ft b have card U = (card T - 1)
  by auto
then have th2: card (insert b U) = card T
  using card-insert-disjoint[OF fu bu] ct0 by auto
from U(4) have S ⊆ span U.
also have ... ⊆ span (insert b U)
  by (rule span-mono) blast
finally have th3: S ⊆ span (insert b U).
from th0 th1 th2 th3 fu have th: ?P ?w
  by blast
from th show ?thesis by blast
next
case False
then obtain a where a: a ∈ S a /∈ span (T - {b})
  by blast
have ab: a ≠ b
  using a b by blast
have at: a /∈ T
  using a ab span-base[of a T - {b}] by auto
have mlt: card ((insert a (T - {b})) - S) < card (T - S)
  using cardlt ft a b by auto
have ft': finite (insert a (T - {b}))
  using ft by auto
have sp': S ⊆ span (insert a (T - {b}))
proof
fix x
assume xs: x ∈ S
have T: T ⊆ insert b (insert a (T - {b}))
  using b by auto
have bs: b ∈ span (insert a (T - {b}))
  by (rule in-span-delete) (use a sp in auto)
from xs sp have x ∈ span T
  by blast
with span-mono[OF T] have x: x ∈ span (insert b (insert a (T - {b}))) ..
from span-trans[OF bs x] show x ∈ span (insert a (T - {b})).
qed
from less(1)[OF mlt ft' S sp'] obtain U where U:
THEORY "Vector-Spaces"

card U = card (insert a (T - {b}))
finite U S ⊆ U U ⊆ S ∪ insert a (T - {b})
S ⊆ span U by blast

from U a b ft at ct0 have ?P U
by auto
then show ?thesis by blast
qed

lemma independent-span-bound:
assumes f: finite T
and i: independent S
and sp: S ⊆ span T
shows finite S ∧ card S ≤ card T
by (metis exchange-lemma[OF f i sp] finite-subset card-mono)

lemma independent-explicit-finite-subsets:
independent A ←→ (∀ S ⊆ A. finite S → (∀ u. (∑ v∈S. u v * s v) = 0 → (∀ v∈S. u v = 0)))

lemma independent-if-scalars-zero:
assumes fin-A: finite A
and sum: ⋀ f x. (∑ x∈A. f x * s x) = 0 ⇒ x ∈ A ⇒ f x = 0
shows independent A
proof (unfold independent-explicit-finite-subsets, clarify)
fix S v and u :: 'b ⇒ 'a
assume S: S ⊆ A and v: v ∈ S
let ?g = λx. if x ∈ S then u x else 0
have (∑ v∈A. ?g v * s v) = (∑ v∈S. u v * s v)
  using S fin-A by (auto intro!: sum.mono-neutral-cong-right)
also assume (∑ v∈S. u v * s v) = 0
finally have ?g v = 0 using v S sum by force
thus u v = 0 unfolding if-P[OF v].
qed

lemma bij-if-span-eq-span-bases:
assumes B: independent B and C: independent C
and eq: span B = span C
shows ∃ f. bij-betw f B C
proof cases
assume finite B ∨ finite C
then have finite B ∧ finite C ∧ card C = card B
  span-superset[of B] span-superset[of C]
  by auto
then show ?thesis
  by (auto intro!: finiteSameCardBij)
qed
next
assume \( \neg (\text{finite } B \lor \text{finite } C) \)
then have infinite \( B \text{ } \) infinite \( C \) by auto
\{ fix \( B \) assume \( B: \text{ independent } B \) and \( C: \text{ independent } C \) and infinite \( B \)
infinite \( C \) and eq: span \( B = \text{span } C \)
\let \( ?R = \text{representation } B \) and \( ?R' = \text{representation } C \) \let \( ?U = \lambda c. \{ v. \ ?R c \ v \neq 0 \} \)
have in-span-C [simp, intro]: \( b \in B \implies b \in \text{span } C \) \for b unfolding eq[symmetric] by (rule span-base)
have in-span-B [simp, intro]: \( c \in C \implies c \in \text{span } B \) \for c unfolding eq by (rule span-base)
have \( B \subseteq (\bigcup \{ c \in C. \ ?U c \}) \)
proof
fix \( b \) assume \( \langle b \in B \rangle \)
have \( \langle b \in \text{span } C \rangle \)
using \( \langle b \in B \rangle \) unfolding eq[symmetric] by (rule span-base)
have \( \langle (\sum v \ | \ ?R' b v \neq 0. \sum w \ | \ ?R v w \neq 0. (\ ?R' b v * ?R v w) * s w) = \rangle \)
\( \langle (\sum v \ | \ ?R' b v \neq 0. \ ?R' b v * (\sum w \ | \ ?R v w \neq 0. ?R v w * s w)) \rangle \)
by (simp add: scale-sum-right)
also have \( \langle \ldots = (\sum v \ | \ ?R' b v \neq 0. \ ?R' b v * s v) \rangle \)
by (auto simp: sum-nonzero-representation-eq B eq span-base representation-ne-zero)
also have \( \langle \ldots = b \rangle \)
by (rule sum-nonzero-representation-eq[OF C \( \langle b \in \text{span } C \rangle \)])
finally have \( ?R b b = ?R (\sum v \ | \ ?R' b v \neq 0. \sum w \ | \ ?R v w \neq 0. (\ ?R' b v * ?R v w) * s w) \rangle \)
by simp
also have \( \langle \ldots = (\sum i \in \{ v. \ ?R' b v \neq 0 \}, \ ?R (\sum w \ | \ ?R i w \neq 0. (\ ?R' b i * ?R i w) * s w) b) \rangle \)
by (subsum representation-sum[OF B]) \( \langle \text{auto intro: span-sum span-scale span-base representation-ne-zero} \rangle \)
also have \( \langle \ldots = (\sum i \in \{ v. \ ?R' b v \neq 0 \}, \ ?R (\sum w \ | \ ?R' b v \neq 0. \ ?R' b v * ?R v w) * ?R w b) \rangle \)
using \( B \langle b \in B \rangle \) by (simp add: representation-scale[OF B] span-base representation-ne-zero)
finally have \( \langle (\sum v \ | \ ?R' b v \neq 0. \sum w \ | \ ?R v w \neq 0. ?R' b v * ?R v w * ?R w b) \neq 0 \rangle \)
using representation-basis[OF B \( \langle b \in B \rangle \) by auto
then obtain \( v w \) where \( \langle b: \ ?R' b v \neq 0 \) and \( vv: ?R v w \neq 0 \) and \( ?R' b v \)
\( ?R v w * ?R w b \neq 0 \)
by (blast elim: sum.not-neutral-contains-not-neutral)
with representation-basis[OF B, of \( w \)] vv[THEN representation-ne-zero]
have \( \langle ?R' b v \neq 0 \rangle \)
\( \langle ?R v w \neq 0 \rangle \) by (auto split: if-splits)
then show \( \langle b \in (\bigcup \{ c \in C. \ ?U c \}) \rangle \)
by (auto dest: representation-ne-zero)
qed
then have B-eq: \( B = (\bigcup c \in C. ?U c) \)
  by (auto intro: span-base representation-ne-zero eq)
have ordLeq3 (card-of B) (card-of C)
proof (subst B-eq, rule card-of-UNION-ordLeq-infinit[ OF \langle infinite C \rangle ])
  show ordLeq3 (card-of C) (card-of C)
    by (intro ordLeq-refl card-of-Card-order)
  show \( \forall c \in C. \ ordLeq3 \ (\text{card-of} \ \{ v. \ ?R c v \neq 0 \}) \) (card-of C)
    by (intro ballI ordLeq3-finite-infinite \langle infinite C \rangle finite-representation)
qed }
from this [ of B C ] this [ of C B ] B C eq \langle infinite C \rangle \langle infinite B \rangle
show \text{thesis} by (auto simp add: ordIso-iff-ordLeq card-of-ordIso)
qed

definition dim :: 'b set \Rightarrow \text{nat}
where dim V = (if \exists b. \ independent b \land \ span b = \ span V then card (SOME b. \ independent b \land \ span b = \ span V) else 0)

lemma dim-eq-card:
  assumes BV: \( \text{span} B = \text{span} V \) and B: \text{independent} B
  shows dim V = card B
proof –
  define p where p b \equiv \ independent b \land \ span b = \ span V for b
  have p (SOME B. p B)
    using assms by (intro someI[p B]) (auto simp: p-def)
  then have \( \exists f. \ bij-betw f B \ (\text{SOME} B. p B) \)
    by (subst \langle asm\rangle p-def, intro bij-if-span-eq-span-bases[OF B]) (simp-all add: BV)
  then have card B = card (SOME B. p B)
    by (auto intro: bij-betw-same-card)
  then show \text{thesis}
    using BV B
    by (auto simp add: dim-def p-def)
qed

lemma basis-card-eq-dim: \( B \subseteq V \Rightarrow V \subseteq \text{span} B \Rightarrow \text{independent} B \Rightarrow \text{card} B = \text{dim} V \)
  by auto

lemma basis-exists:
  obtains B where B \subseteq V \ independent B V \subseteq \text{span} B \ card B = \text{dim} V
  by (meson basis-card-eq-dim empty-subsetI independent-empty maximal-independent-subset-extend)

lemma dim-eq-card-independent: \text{independent} B \Rightarrow \text{dim} B = \text{card} B
  by (rule dim-eq-card[OF refl])

lemma dim-span[simp]: \text{dim} (\text{span} S) = \text{dim} S
  by (auto simp add: dim-def span-span)
lemma dim-span-eq-card-independent: independent $B \implies \dim(\text{span } B) = \text{card } B$
by (simp add: dim-eq-card)

lemma dim-le-card; assumes $V \subseteq \text{span } W$ finite $W$
shows $\dim V \leq \text{card } W$
proof
obtain $A$ where independent $A$ $A \subseteq V$ $V \subseteq \text{span } A$
using maximal-independent-subset[of $V$] by force
with assms independent-span-bound[of $W$ $A$]
basis-card-eq-dim[of $A$ $V$]
show \text{thesis by auto}
qued

lemma span-eq-dim: $\text{span } S = \text{span } T \implies \dim S = \dim T$
by (metis dim-span)

corollary dim-le-card':
finite $s \implies \dim s \leq \text{card } s$
by (metis basis-exists card-mono)

lemma span-card-ge-dim:
$B \subseteq V \implies V \subseteq \text{span } B \implies \text{finite } B \implies \dim V \leq \text{card } B$
by (simp add: dim-le-card)

lemma dim-unique:
$B \subseteq V \implies V \subseteq \text{span } B \implies \text{independent } B \implies \text{card } B = n \implies \dim V = n$
by (metis basis-card-eq-dim)

lemma subspace-sums: [subspace $S$; subspace $T$] \implies subspace $\{x + y|x, y. x \in S \land y \in T\}$
apply (simp add: subspace-def)
apply (intro conjI implI allI; clarsimp simp: algebra-simps)
using add.left-neutral apply blast
apply (metis add.assoc)
using scale-right-distrib by blast

end

lemma linear-iff: linear $s1$ $s2$ $f$ \iff
(vector-space $s1$ \land vector-space $s2$ \land ($\forall x, y. f (x + y) = f x + f y$) \land ($\forall c. f (s1 c x) = s2 c (f x)$))
unfolding linear-def module-hom-iff vector-space-def module-def by auto

context begin
qualified lemma linear-compose: linear $s1$ $s2$ $f$ \implies linear $s2$ $s3$ $g$ \implies linear $s1$
$s3$ $(g \circ f)$
unfolding module-hom-iff-linear[symmetric]
by (rule module-hom-compose)
end
locale vector-space-pair = vs1: vector-space s1 + vs2: vector-space s2 
for s1 :: 'a::field ⇒ 'b::ab-group-add ⇒ 'b (infixr *a 75) 
and s2 :: 'a::field ⇒ 'c::ab-group-add ⇒ 'c (infixr *b 75) 
begin

context fixes f assumes linear s1 s2 f begin 
interpretation linear s1 s2 f by fact 
lemmas— from locale module-hom 
  linear-0 = zero 
  and linear-add = add 
  and linear-scale = scale 
  and linear-neg = neg 
  and linear-diff = diff 
  and linear-sum = sum 
  and linear-inj-on-iff-eq-0 = inj-on-iff-eq-0 
  and linear-inj-iff-eq-0 = inj-iff-eq-0 
  and linear-subspace-image = subspace-image 
  and linear-subspace-aimage = subspace-vimage 
  and linear-subspace-kernel = subspace-kernel 
  and linear-span-image = span-image 
  and linear-dependent-inj-imageD = dependent-inj-imageD 
  and linear-eq-0-on-span = eq-0-on-span 
  and linear-independent-injective-image = independent-injective-image 
  and linear-inj-on-span-independent-image = inj-on-span-independent-image 
  and linear-inj-on-span-iff-independent-image = inj-on-span-iff-independent-image 
  and linear-subspace-linear-preimage = subspace-linear-preimage 
  and linear-spans-image = spans-image 
  and linear-spanning-surjective-image = spanning-surjective-image 
end 

sublocale module-pair 
  rewrites module-hom = linear 
  by unfold-locales (fact module-hom-eq-linear) 

lemmas— from locale module-pair 
  linear-eq-0-on-span = module-hom-eq-0-on-span 
  and linear-compose-scale-right = module-hom-scale 
  and linear-compose-add = module-hom-add 
  and linear-zero = module-hom-zero 
  and linear-compose-sub = module-hom-sub 
  and linear-compose-neg = module-hom-neg 
  and linear-compose-scale = module-hom-compose-scale 

lemma linear-indep-image-lemma: 
assumes lf: linear s1 s2 f 
  and fB: finite B 
  and ifB: vs2.independent (f ' B) 
  and fi: inj-on f B 
  and xsB: x ∈ vs1.span B
and $f x = 0$

shows $x = 0$

using $f x$ if $f x$

proof (induction $B$ arbitrary; $x$ rule: finite-induct)

  case empty
  then show $\vdash ?case$ by auto

next

  case $(\text{insert } a \ b \ x)$
  have $\vdash \text{th0: } f \cdot b \subseteq f \cdot (\text{insert } a \ b)$
    by (simp add: subset-insertI)
  have $\vdash \text{ifb: vs2.independent } (f \cdot b)$
    using $\vdash \text{vs2.independent-mono insert.prems(1) th0}$ by blast
  have $f b$ \text{inj-on } $f b$
    using $\vdash \text{insert.prems(2) by blast}$
  from $\vdash \text{vs1.span-breakdown[of a insert a b, simplified, OF insert.prems(3)]}$
  obtain $k$ where $\vdash k: x - k \cdot a \ a \in \text{vs1.span } (b - \{a\})$
    by blast
  have $f (x - k \cdot a \ a) \in \text{vs2.span } (f \cdot b)$
    unfolding $\vdash \text{linear-span-image[OF lf]}$
    using $\vdash \text{insert.prems(2) k by auto}$
  then have $\vdash \text{f x - k \cdot f a \in vs2.span } (f \cdot b)$
    by (simp add: linear-diff linear-scale lf)
  then have $\vdash \text{th: -k \cdot f a \in vs2.span } (f \cdot b)$
    using $\vdash \text{insert.prems(4) by simp}$
  have $\vdash \text{xsb: x \in vs1.span b}$
  proof (cases $k = 0$

    case True
    with $\vdash k$ have $\vdash \text{x \in vs1.span } (b - \{a\})$ by simp
    then show $\vdash ?thesis$ using $\vdash \text{vs1.span-mono[of b - \{a\} b]}$
      by blast

    next

    case False
    from $\vdash \text{inj-on-image-set-diff[OF insert.prems(2), OF insert a b \{a\}, symmetric]}$
    have $\vdash f \cdot \text{insert a b - f \cdot \{a\} = f \cdot (insert a b - \{a\})}$ by blast
    then have $\vdash f a \notin \text{vs2.span } (f \cdot b)$
      using $\vdash \text{vs2.dependent-def insert.prems(2) insert.prems(1) by fastforce}$
    moreover have $\vdash f a \in \text{vs2.span } (f \cdot b)$
      using $\vdash \text{False vs2.span-scale[OF th, OF -1/ k]}$ by auto
    ultimately have $\vdash \text{False}$
      by blast
    then show $\vdash ?thesis$ by blast
  qed

  qed

lemma linear-eq-on:
  assumes $\vdash \text{l: linear s1 s2 f linear s1 s2 g}$
  assumes $\vdash x: x \in \text{vs1.span } B$ and $\vdash eq: \forall b. b \in B \Longrightarrow f b = g b$
  qed
shows $f \ x = g \ x$

proof -
interpet d: linear s1 s2 $\lambda x. f \ x - g \ x$
using l by (intro linear-compose-sub) (auto simp: module-hom-iff-linear)
have $f \ x - g \ x = 0$
by (rule d.eq-0-on-span[OF - x]) (auto simp: eq)
then show ?thesis by auto
qed

definition construct :: $'b$ set $\Rightarrow$ $'b$ $\Rightarrow$ $'b$
where construct B g v = $(\sum b | \ vs1.\representation \ (\ vs1.\extend-basis \ B) \ v \ b \neq 0) \ vs1.\representation \ (\ vs1.\extend-basis \ B) \ v \ b \ * \ b$ (if $b \in B$ then $g \ b$ else 0))

lemma construct-cong: $(\forall b. b \in B \Longrightarrow f \ b = g \ b) \Longrightarrow$ construct $B \ f =$ construct $B \ g$
unfolding construct-def by (rule ext, auto intro!: sum.cong)

lemma linear-construct:
assumes $B[\simp]: \ vs1.\independent \ B$
shows linear s1 s2 (construct $B \ f$
unfolding module-hom-iff-linear linear-iff
proof safe
have $eB[\simp]: \ vs1.\independent \ (\ vs1.\extend-basis \ B)$
using $\ vs1.\independent-extend-basis[OF \ B]$ .
let $?R = \ vs1.\representation \ (\ vs1.\extend-basis \ B)$
fix c x y
have construct $B \ f \ (x + y) = (\sum b \in \{b. ?R \ x \ b \neq 0\} \cup \{b. ?R \ y \ b \neq 0\}. \ ?R \ (x + y) \ b \ * \ b$ (if $b \in B$ then $f \ b$ else 0))
by (auto intro!: sum.mono-neutral-cong-left simp: $\ vs1.\finite-representation \ vs1.\representation-add \ construct-def$)
also have ... = construct $B \ f \ x +$ construct $B \ f \ y$
by (auto simp: construct-def $\ vs1.\representation-add \ vs2.\scale-left-distrib \ sum.\distrib \ intro!: \ arg-cong2[where $f= (+)] \ sum.\mono-neutral-cong-right \ vs1.\finite-representation$)
finally show construct $B \ f \ (x + y) =$ construct $B \ f \ x +$ construct $B \ f \ y$ .

show construct $B \ f \ (c * a \ x) =$ $c * b$ construct $B \ f \ x$
by (auto simp del: $\ vs2.\scale-scale \ intro!: \ sum.\mono-neutral-cong-left \ vs1.\finite-representation \ simp \ add: \ construct-def \ vs2.\scale-scale[\ symmetric] \ vs1.\representation-scale \ vs2.\scale-sum-right$)
qed intro-locales

lemma construct-basis:
assumes $B[\simp]: \ vs1.\independent \ B \ and \ b: b \in B$
shows construct $B \ f \ b = f \ b$
proof -
have *: $\ vs1.\representation \ (\ vs1.\extend-basis \ B) \ b = (\lambda v. \ if \ v = b \ then \ 1 \ else \ 0)$
using $\ vs1.\extend-basis-superset[OF \ B] \ b$
by (intro vs1.representation-basis vs1.independent-extend-basis) auto
then have \{ v. vs1.representation (vs1.extend-basis B) b v \neq 0 \} = \{ b \}
  by auto
then show \( ? \)thesis
  unfolding construct-def by (simp add: \* b)
qed

lemma construct-outside:
  assumes B: vs1.independent B and v: v \in vs1.span (vs1.extend-basis B - B)
  shows construct B f v = 0
  unfolding construct-def
proof (clarsimp intro!:
  sum.neutral simp del:
  vs2.scale-eq-0-iff)
fix b assume b \in B
then have vs1.representation (vs1.extend-basis B - B) v b = 0
  using vs1.representation-ne-zero[of vs1.extend-basis B - B v b] by auto
moreover have vs1.representation (vs1.extend-basis B) v = vs1.representation
  (vs1.extend-basis B - B) v
  using vs1.representation-extend[OF vs1.independent-extend-basis[of B] v] by auto
ultimately show vs1.representation (vs1.extend-basis B) v b \* b f b = 0
  by simp
qed

lemma construct-add:
  assumes B[simp]: vs1.independent B
  shows construct (λx. f x + g x) v = construct B f v + construct B g v
proof (rule linear-eq-on)
  show v \in vs1.span (vs1.extend-basis B) by simp
  show b \in vs1.extend-basis B \implies construct B (λx. f x + g x) b = construct B f
    b + construct B g b for b
    using construct-outside[OF B vs1.span-base, of b] by (cases b \in B) (auto simp:
    construct-basis)
qed (intro linear-compose-add linear-construct B)+

lemma construct-scale:
  assumes B[simp]: vs1.independent B
  shows construct (λx. c \* b f x) v = c \* b construct B f v
proof (rule linear-eq-on)
  show v \in vs1.span (vs1.extend-basis B) by simp
  show b \in vs1.extend-basis B \implies construct B (λx. c \* b f x) b = c \* b construct
    B f b for b
    using construct-outside[OF B vs1.span-base, of b] by (cases b \in B) (auto simp:
    construct-basis)
qed (intro linear-construct module-hom-scale B)+

lemma construct-in-span:
  assumes B[simp]: vs1.independent B
  shows construct B f v \in vs2.span (f ' B)
proof –
interpret \( c \colon \text{linear } s1 \ s2 \) construct \( B \ f \) by (rule linear-construct) fact
let \( ?R = vs1.\text{representation } B \)
have \( v \in vs1.\text{span } ((vs1.\text{extend-basis } B - B) \cup B) \)
  by (auto simp: \Un-absorb2 vs1.\text{extend-basis-superset})
then obtain \( x \ y \) where \( v = x + y \ x \in vs1.\text{span } (vs1.\text{extend-basis } B - B) \ y \in vs1.\text{span } B \)
  unfolding vs1.\text{span-Un}
moreover have \( \text{construct } B \ f \ (\sum b \ | \ ?R y b \neq 0. \ ?R y b \ast a b) \in vs2.\text{span } (f \ ' B) \)
  by (auto simp add: \Un-absorb2 vs1.\text{extend-basis-superset} vs2.\text{span-extend-basis} vs2.\text{span-base})
ultimately show \( \text{construct } B \ f \ v \in vs2.\text{span } (f \ ' B) \)
qed

lemma \( \text{linear-compose-sum} \):
  assumes \( \forall a \in S. \text{linear } s1 \ s2 \ (f \ a) \)
  shows \( \text{linear } s1 \ s2 \ (\lambda x. \text{sum } (\lambda a. \text{f } a \ x) \ S) \)
proof (cases finite S)
case True
  then show \( ?\text{thesis} \) using \( \text{lS} \) by (induct simp-all add: \text{linear-zero} \text{linear-compose-add})
next
case False
  then show \( ?\text{thesis} \)
    by (simp add: \Un-absorb2)
qed

lemma \( \text{in-span-in-range-construct} \):
  \( x \in \text{range } (\text{construct } B \ f) \) if \( i \colon \text{vs1.\text{independent } B \ \text{and } x \in vs2.\text{span } (f \ ' B)} \)
proof –
interpret linear \((\ast \ a) \ (\ast \ b)\) construct \( B \ f \)
  using \( i \) by (rule linear-construct)
obtain \( \bb :: (\forall c \Rightarrow c) = \forall c \Rightarrow cb \Rightarrow cb \ \text{where} \)
  \( \forall x0 \ x1 \ x2. \exists x1. \ x1 \ x2 \in x2 \land x1 x1 \neq x0 x1 \) = \( \forall x0 \ x1 \ x2. \exists x1. \ x1 \ x2 \in x2 \land x1 (\bb x0 x1 x2) \neq x0 (\bb x0 x1 x2) \)
  by (moua)
then have \( f2 \colon \forall B Ba f fa. \ (B \neq Ba \lor \bb f fa f Ba \in Ba \land f (bb f fa f Ba) \neq fa (bb f fa f Ba)) \lor f ' B = fa ' Ba \)
  by (meson \text{image-cong})
have \( vs1.\text{span } B \subseteq vs1.\text{span } (vs1.\text{extend-basis } B) \)
  by (simp add: \text{vs1.\text{extend-basis-superset} vs1.\text{span-mono}})
then show \( x \in \text{range } (\text{construct } B \ f) \)
  using \( f2 x \ y \) by (metis \( \text{no-types} \) \text{construct-basis} \text{vs1.\text{span-mono}})
qed

lemma \( \text{range-construct-eq-span} \):
  \( \text{range } (\text{construct } B \ f) = vs2.\text{span } (f \ ' B) \)
THEORY "Vector-Spaces"

if vs1.independent B
  by (auto simp: that construct-in-span in-span-in-range-construct)

lemma linear-independent-extend-subspace:
  — legacy: use construct instead
  assumes vs1.independent B
  shows \exists g. linear s1 s2 g \land (\forall x \in B. g x = f x) \land range g = vs2.span (f'B)
  by (rule exI[where x=construct B f])
    (auto simp: linear-construct assms construct-basis range-construct-eq-span)

lemma linear-independent-extend:
  vs1.independent B \implies \exists g. linear s1 s2 g \land (\forall x \in B. g x = f x)
  using linear-independent-extend-subspace[of B f]
  by auto

lemma linear-exists-left-inverse-on:
  assumes lf: linear s1 s2 f
  assumes V: vs1.subspace V and f: inj-on f V
  shows \exists g. g ' UNIV \subseteq V \land linear s2 s1 g \land (\forall v \in V. g (f v) = v)
  proof
    interpret linear s1 s2 f by fact
    obtain B where V-eq: V = vs1.span B and B: vs1.independent B
      using vs1.maximal-independent-subset[of V] vs1.span-minimal[OF ‹vs1.subspace V›]
      by (metis antisym-conv)
    have fB: vs2.independent (f ' B)
      using independent-injective-image[OF B f].
    let ?g = p.construct (f ' B) (the-inv-into B f)
    show linear (?g ◦ f) for V-eq .
      using f unfolding V-eq.
    show ?thesis
      proof
        interpret p: vector-space-pair s2 s1 by unfold-locales
        have fB: vs2.independent (f ' B)
          using independent-injective-image[of B f].
        let ?g = p.construct (f ' B) (the-inv-into B f)
        show linear (?g ◦ b) for ?g
          by (rule p.linear-construct[of fB])
        have ?g b \in vs1.span (the-inv-into B f ' f ' B) for b
          by (intro p.construct-in-span fB)
        moreover have the-inv-into B f ' f ' B = B
          by (auto simp: image-comp comp-def the-inv-into-f f inj-on-subset[of f V]
            vs1.span-superset)
        ultimately show ?g ◦ UNIV \subseteq V
          by (auto simp: V-eq)
        have (?g ◦ f) v = id v if v \in vs1.span B for v
          proof
            (rule vector-space-pair.linear-eq-on[where x=v])
            show vector-space-pair (?a) (?a) by unfold-locales
            show linear (?a) (?a) (?g ◦ f)
              proof
                (rule Vector-Spaces.linear-compose[of - (?b)])
                show linear (?a) (?b) f
                  by unfold-locales
lemma linear-exists-right-inverse-on:
  assumes lf: linear s1 s2 f
  assumes vs1: vs1.subspace V
  shows \exists g. g' UNIV \subseteq V \land linear s2 s1 g \land (\forall v \in f' V. f (g v) = v)
proof
  obtain B where V-eq: V = vs1.span B and B: vs1.independent B
    using vs1.maximal-independent-subset[of V] vs1.span-minimal[of V - \vs1.subspace V]
    by (metis antisym-conv)
  obtain C where C: vs2.independent C and fB-C: f' B \subseteq \vs2.span C C \subseteq f' B
    using vs2.maximal-independent-subset[of f' B] by metis
  then have \forall v \in C. \exists b \in B. v = f b by auto
  then obtain g where g: \\forall v. v \in C \Longrightarrow g v \in B \\& v \in C \Longrightarrow f (g v) = v
    by metis
  show \?thesis
  proof (intro exI ballI conjI)
    interpret p: vector-space-pair s2 s1 by unfold-locales
    let \?g = p.construct C g
    show linear (\*b) (\*a) \?g
      by (rule p.linear-construct[of C])
    have \?g v \in vs1.span (g' C) for v
      by (rule p.construct-in-span[of C])
    also have \ldots \subseteq V unfolding V-eq using g by (intro vs1.span-mono) auto
    finally show \?g' UNIV \subseteq V by auto
    have (f o \?g) v = id v if v: v \in f' V for v
      proof (rule vector-space-pair.linear-eq-on[where x=v])
        show vector-space-pair (\*b) (\*b) by unfold-locales
        show linear (\*b) (\*b) (f o \?g)
          by (rule Vector-Spaces.linear-compose[of - (\*a)]) fact+
        show linear (\*b) (\*b) id by (rule vs2.linear-id)
        have vs2.span (f' B) = vs2.span C
          using fB-C vs2.span-mono[of f' B] vs2.span-minimal[of f'B vs2.span C]
          by auto
        then show v \in vs2.span C
          using v linear-span-image[of f, of B] by (simp add: V-eq)
        show (f o p.construct C g) b = id b if b: b \in C for b
          by (auto simp: p.construct-basis g C b)
qed
then show $v \in f ' V \implies f (\exists g. v) = v$ for $v$ by (auto simp: comp-def id-def)
qed

lemma linear-inj-on-left-inverse:
  assumes lf: linear s1 s2 f
  assumes fi: inj-on f (vs1.span S)
  shows $\exists g. \text{range } g \subseteq \text{vs1.span } S$ \land linear s2 s1 g \land (\forall x \in \text{vs1.span } S. g (f x) = x)
  using linear-exists-left-inverse-on[OF lf vs1.span fi]
  by (auto simp: linear-iff-module-hom)

lemma linear-injective-left-inverse: linear s1 s2 f \implies inj f \implies $\exists g. \text{linear } s2 s1 g \land g \circ f = id$
  using linear-inj-on-left-inverse[of f UNIV]
  by force

lemma linear-surj-right-inverse:
  assumes lf: linear s1 s2 f
  assumes sf: vs2.span T \subseteq f'vs1.span S
  shows $\exists g. \text{range } g \subseteq \text{vs1.span } S$ \land linear s2 s1 g \land (\forall x \in \text{vs2.span } T. f (g x) = x)
  using linear-exists-right-inverse-on[OF lf vs1.span, of S] sf
  by (force simp: linear-iff-module-hom)

lemma linear-surjective-right-inverse: linear s1 s2 f \implies surj f \implies $\exists g. \text{linear } s2 s1 g \land g \circ f = id$
  using linear-surj-right-inverse[of f UNIV UNIV]
  by (auto simp: fun-eq_iff)

lemma finite-basis-to-basis-subspace-isomorphism:
  assumes s: vs1.span S
  and t: vs2.span T
  and d: vs1.dim S = vs2.dim T
  and fB: finite B
  and B: B \subseteq S \subseteq vs1.span B \land card B = vs1.dim S
  and fC: finite C
  and C: C \subseteq T \subseteq vs2.span C \land card C = vs2.dim T
  shows $\exists f. \text{linear } s1 s2 f \land f ' B = C \land f ' S = T \land inj-on f S$
proof –
  from B(4) C(4) \ card-le-inj[of B C] d obtain f where
  f: f ' B \subseteq C \inj-on f B using (finite B) (finite C) by auto
  from linear-independent-extend[OF B(2)] obtain g where
  g: linear s1 s2 g \forall x \in B. g x = f x by blast
  interpret g: linear s1 s2 g by fact
  from \inj-on-[ff-eq-card[OF fB, of f] f(2)] have \card (f ' B) = card B by simp
  with B(4) C(4) have ceq: \card (f ' B) = card C using d
by simp
have \( g \cdot B = f \cdot B \) using \( g(2) \)
  by (auto simp add: image-iff)
also have \( \ldots = C \) using card-subset-eq[OF \( fC \) \( f(1) \) \( ceq \) ] .
finally have \( gBC: g \cdot B = C \) .
have \( gi: \text{inj-on} \ g \ B \) using f(2) \( g(2) \)
  by (auto simp add: inj-on-def)
note \( g0 = \text{linear-indep-image-lemma}[OF g(1) \ fB, \text{unfolded} \ gBC, \ OF \ C(2) \ gi] \)
{  
  fix \( x \ y \)
  assume \( x: x \in S \) and \( y: y \in S \) and \( gxy: g \ x = g \ y \)
  from \( B(3) \) \( x \ y \) have \( x': x \in vs1.span \ B \) and \( y': y \in vs1.span \ B \)
  by blast+
  from \( gxy \) have \( \text{th0}: g \ (x - y) = 0 \)
  by (simp add: g.diff)
  have \( \text{th1}: x - y \in vs1.span \ B \) using \( x' \ y' \)
  by (metis vs1.span-diff)
  have \( x = y \) using \( g0[OF \text{th1} \ \text{th0}] \) by simp
}
th(0) have \( giS: \text{inj-on} \ g \ S \) unfolding inj-on-def by blast
from \( vs1.span-subspace[OF B(1,3) \ s] \)
have \( g \cdot S = vs2.span (g \cdot B) \)
  by (simp add: g.span-image)
also have \( \ldots = T \)
  using \( vs2.span-subspace[OF C(1,3) \ t] \) .
finally have \( gS: g \cdot S = T \) .
from \( g(1) \ gS \ gS \ gBC \) show \( \text{thesis} \)
  by blast
qed

locale finite-dimensional-vector-space = vector-space +
  fixes Basis :: `'b set
  assumes finite-Basis: finite Basis
  and independent-Basis: independent Basis
  and span-Basis: span Basis = UNIV
begin

definition dimension = card Basis

lemma finitel-independent: independent B \imp finite B
  using independent-span-bound[OF finite-Basis, of B] by (auto simp: span-Basis)

lemma dim-empty [simp]: dim \{\} = 0
  by (rule dim-unique[OF order-refl]) (auto simp: dependent-def)
lemma dim-insert:
dim (insert x S) = (if x \in span S then dim S else dim S + 1)
proof 
  show \?thesis
  proof (cases x \in span S)
    case True then show \?thesis
      by (metis dim-span span-redundant)
  next
case False
obtain B where B: B \subseteq span S independent B span S \subseteq span B card B = dim (span S)
  using basis-exists [of span S] by blast
  have dim (span (insert x S)) = Suc (dim S)
  proof (rule dim-unique)
    show insert x B \subseteq span (insert x S)
      by (meson B(1) insertI1 insert-subset order-trans span-base span-mono subset-insertI)
    show span (insert x S) \subseteq span (insert x B)
      by (metis B(1 - 3) independent-insert span-subspace span-subspace False)
    show card (insert x B) = Suc (dim S)
      using B False finiteI-independent by force
    qed
then show \?thesis
  by (metis False Suc-eq-plus1 dim-span)
  qed
  qed

lemma dim-singleton [simp]: dim\{x\} = (if x = 0 then 0 else 1)
by (simp add: dim-insert)

proposition choose-subspace-of-subspace:
assumes n \leq dim S
obtains T where subspace T T \subseteq span S dim T = n
proof 
  have \exists T. subspace T \land T \subseteq span S \land dim T = n
  using assms
proof (induction n)
case 0 then show \?case by (auto intro: exI[where x=\{0\}] span-zero)
next
case (Suc n)
then obtain T where subspace T T \subseteq span S dim T = n
  by force
then show \?case
proof (cases span S \subseteq span T)
case True

have span $T \subseteq \text{span } S$
by (simp add: $T \subseteq \text{span } S \cdot \text{span-minimal})$
then have $\dim S = \dim T$
by (rule span-eq-dim [OF subset-antisym [OF True]])
then show ?thesis
using Suc.prems ($\dim T = n$) by linarith
next
case False
then obtain $y$ where $y \in S \setminus T$
by (meson span-mono subsetI)
then have span $(\text{insert } y \ T) \subseteq \text{span } S$
by (metis (no-types) ($T \subseteq \text{span } S$) subsetD insert-subset span-superset span-mono span-span)
with ($\dim T = n$) ($\text{subspace } T$) y show ?thesis
apply (rule-tac $x = \text{span}(\text{insert } y \ T)$ in exI)
using span-eq-iff by (fastforce simp: dim-insert)
qed qed
with that show ?thesis by blast
qed

lemma basis-subspace-exists:
assumes $\text{subspace } S$
obtains $B$ where finite $B$ $B \subseteq S$ independent $B$ $\text{span } B = S$ $\text{card } B = \dim S$
by (metis assms span-subspace basis-exists finiteI-independent)

lemma dim-mono: assumes $V \subseteq \text{span } W$ shows $\dim V \leq \dim W$
proof –
oindent obtain $B$ where independent $B$ $B \subseteq W$ $W \subseteq \text{span } B$
using maximal-independent-subset[of $W$] by force
by (auto simp: finite-Basis span-Basis)
qed

lemma dim-subset: $S \subseteq T$ $\rightarrow \dim S \leq \dim T$
using dim-mono[of $S$ $T$] by (auto intro: span-base)

lemma dim-eq-0 [simp]:
$\dim S = 0 \iff S \subseteq \{0\}$
by (metis basis-exists card-eq-0-iff dim-span finiteI-independent span-empty subset-empty subset-singletonD)

lemma dim-UNIV [simp]: $\dim \text{UNIV} = \text{card Basis}$
using dim-eq-card[of Basis $\text{UNIV}$] by (simp add: independent-Basis span-Basis)

lemma independent-card-le-dim: assumes $B \subseteq V$ and independent $B$ shows $\text{card}$
B ≤ dim V
  by (subst dim-eq-card[symmetric, OF refl independent B]) (rule dim-subset[OF B ⊆ V])

lemma dim-subset-UNIV: dim S ≤ dimension
  by (metis dim-subset subset-UNIV dim-UNIV dimension-def)

lemma card-ge-dim-independent:
  assumes BV: B ⊆ V
  and iB: independent B
  and dVB: dim V ≤ card B
  shows V ⊆ span B
proof
  fix a
  assume aV: a ∈ V
  { assume aB: a ∉ span B
    then have iaB: independent (insert a B)
      using iB aV BV by (simp add: independent-insert)
    from aV BV have th0: insert a B ⊆ V
      by blast
    from aB have a ∉ B
      by (auto simp add: span-base)
    with independent-card-le-dim[OF th0 iaB] dVB finiteI-independent[OF iB]
    have False by auto
  }
  then show a ∈ span B by blast
qed

lemma card-le-dim-spanning:
  assumes BV: B ⊆ V
  and VB: V ⊆ span B
  and fB: finite B
  and dVB: dim V ≥ card B
  shows independent B
proof
  { fix a
    assume a: a ∈ B a ∈ span (B − {a})
    from a fB have c0: card B ≠ 0
      by auto
    from a fB have cb: card (B − {a}) = card B − 1
      by auto
    { fix x
      assume x: x ∈ V
      from a have eq: insert a (B − {a}) = B
        by blast
      from x VB have x': x ∈ span B
  qed
THEORY "Vector-Spaces"

by blast
from span-trans[OF a(2), unfolded eq, OF x]
have x ∈ span (B - {a}) .
}
then have th1: V ⊆ span (B - {a})
  by blast
have th2: finite (B - {a})
  using fB by auto
from dim-le-card[OF th1 th2]
have c: dim V ≤ card (B - {a}) .
from c c0 dVB cb have False by simp
}
then show ?thesis
  unfolding dependent-def by blast
qed

lemma card-eq-dim: B ⊆ V ⇒ card B = dim V ⇒ finite B ⇒ independent B
  ⇔ V ⊆ span B
by (metis order-eq-iff card-le-dim-spanning card-ge-dim-independent)

lemma subspace-dim-equal:
  assumes subspace S
  and subspace T
  and S ⊆ T
  and dim S ≥ dim T
  shows S = T
proof –
  obtain B where B: B ≤ S independent B ∧ S ⊆ span B card B = dim S
    using basis-exists[of S] by metis
  then have span B ⊆ S
    using span-monono[of B S] span-eq-iff[of S] assms by metis
  then have span B = S
    using B by auto
  have dim S = dim T
    using assms dim-subset[of S T] by auto
  then have T ⊆ span B
    using card-eq-dim[of B T] B finiteI-independent assms by auto
  then show ?thesis
    using assms ‹span B = S› by auto
qed

corollary dim-eq-span:
  shows [S ⊆ T; dim T ≤ dim S] ⇒ span S = span T
by (simp add: span-monono subspace-dim-equal)

lemma dim-psubset:
  span S ⊆ span T ⇒ dim S < dim T
by (metis order-eq-iff dim-span less-le not-le subspace-dim-equal subspace-span)
The document contains a proof of the theorem that the dimension of a vector space is equal to its span, denoted as `dim-eq-full`. It also includes lemmas for independent sets and their size bounds, as well as propositions and proofs for the dimensions of sums and intersections of subspaces.
show $D \subseteq \textit{?ST}$
using $\text{span-zero span-minimal} \ [\text{OF - (subspace S)}] \ (D \subseteq T)$ by force
qed

have $a \cdot v = 0$ if $0 \cdot (\sum v \in C. \ a \ v \ * s \ v) + (\sum v \in D - C. \ a \ v \ * s \ v) = 0$
and $v \cdot v \in C \cup (D - C)$ for $a \cdot v$

proof

have $\text{CsS}: \bigwedge x. \ x \in C \implies a \cdot x \ * s \ x \in S$
using $C \subseteq S$, (subspace $S$) subspace-scale by auto
have $\text{eq}: (\sum v \in D - C. \ a \ v \ * s \ v) = - (\sum v \in C. \ a \ v \ * s \ v)$
using that add-eq-0-iff by blast
have $(\sum v \in D - C. \ a \ v \ * s \ v) \in S$
by (simp add: eq CsS (subspace $S$) subspace-neg subspace-sum)
moreover have $(\sum v \in D - C. \ a \ v \ * s \ v) \in T$
apply (rule subspace-sum [OF (subspace $T$)])
by (meson DiffD1 (D \subseteq T) (subspace $T$) subset-eq subspace-def)
ultimately have $(\sum v \in D - C. \ a \ v \ * s \ v) \in \text{span} \ B$
using $B$ by blast
then obtain $e$ where $e: (\sum v \in B. \ e \ v \ * s \ v) = (\sum v \in D - C. \ a \ v \ * s \ v)$
using $\text{span-finite} \ [\text{OF (finite B)}]$ by force
have $\bigwedge c. \ (\sum v \in C. \ c \ v \ * s \ v) = 0; \ v \in C \implies c \ v = 0$
using (finite $C$) (independent $C$) independentD by blast
define $\text{cc}$ where $\text{cc} x = (\text{if} \ x \in B \text{ then} \ a \ x + e \ x \text{ else} a \ x)$ for $x$
have $\text{[simp]}: C \cap B = B \cap B = B \cap B = C - D \cap B = (D - C)$ = {}
using $B \subseteq C$, $B \subseteq D$, $\text{Beq}$ by blast+
have $f2: (\sum v \in C \cap D. \ e \ v \ * s \ v) = (\sum v \in D - C. \ a \ v \ * s \ v)$
using $\text{Beq} e$ by presburger
have $f3: (\sum v \in C \cup D. \ a \ v \ * s \ v) = (\sum v \in C - D. \ a \ v \ * s \ v) + (\sum v \in D - C. \ a \ v \ * s \ v) + (\sum v \in C \cap D. \ a \ v \ * s \ v)$
using (finite $C$) (finite $D$) sum.union-diff2 by blast
have $f4: (\sum v \in C \cup (D - C). \ a \ v \ * s \ v) = (\sum v \in C. \ a \ v \ * s \ v) + (\sum v \in D - C. \ a \ v \ * s \ v)$
by (meson Diff-disjoint (finite $C$) (finite $D$) finite-Diff sum.union-disjoint)
have $(\sum v \in C. \ c \ v \ * s \ v) = 0$
using $0 \ f2 \ f3 \ f4$
apply (simp add: cc-def Beq (finite $C$) sum.If-cases algebra-simps sum.distrib
if-distrib if-distribR)
apply (simp add: add.commute add.left-commute diff-eq)
done
then have $\bigwedge v. \ v \in C \implies c \ v = 0$
using independent-explicit (independent $C$) (finite $C$) by blast
then have $\text{C0}: \bigwedge v. \ v \in C - B \implies a \ v = 0$
by (simp add: cc-def Beq) meson
then have $\text{[simp]}: (\sum x \in C - B. \ a \ x \ * s \ x) = 0$
by simp
have $(\sum x \in C. \ a \ x \ * s \ x) = (\sum x \in B. \ a \ x \ * s \ x)$
proof

have $C - D = C - B$
using $\text{Beq}$ by blast
then show $\text{?thesis}$
using Beq \((\sum_{x \in C} a \cdot s \cdot x) = 0\) \(f_3 f_4\) by auto

qed

with \(\theta\) have Dc0: \((\sum_{v \in D} a \cdot s \cdot v) = 0\)
by (subst Dc0) (simp add: finite B finite D sum-Un)

then have D0: \(\forall v. v \in D \Longrightarrow a \cdot v = 0\)
  using independent-explicit \(\langle\text{independent D}\rangle\) finite D by blast

show \(?\text{thesis}\)
using v C0 D0 Beq by blast

qed

then have independent \((C \cup (D - C))\)
  unfolding independent-explicit
  using independent-explicit
  by (simp add: finite C finite D sum-Un del: Un-Diff-cancel)

then have indCUD: independent \((C \cup D)\) by simp

have dim \((S \cap T) = \text{card } B\)
  by (rule dim-unique [OF B indB refl])

moreover have dim \(S = \text{card } C\)
  by (metis \(\langle\text{independent C}\rangle\) \(\langle S \subseteq \text{span } C\rangle\) basis-card-eq-dim)

moreover have dim \(T = \text{card } D\)
  by (metis \(\langle\text{independent D}\rangle\) \(\langle T \subseteq \text{span } D\rangle\) basis-card-eq-dim)

moreover have dim \(?ST = \text{card } (C \cup D)\)
proof
  have \(*: \\{x, y. \[x \in S; y \in T\] \Longrightarrow x + y \in \text{span } (C \cup D)\)
    by (meson \(\langle S \subseteq \text{span } C\rangle\) \(\langle T \subseteq \text{span } D\rangle\) span-add span-mono subsetCE sup.cobounded1 sup.cobounded2)

  show \(?\text{thesis}\)
  by (auto intro: \(\ast\) dim-unique [OF CUD indCUD refl])

qed

ultimately show \(?\text{thesis}\)
  using \(\langle B = C \cap D\rangle\) [symmetric]
  by (simp add: \(\langle\text{independent C}\rangle\) \(\langle\text{independent D}\rangle\) card-Un-Int finiteI-independent)

qed

lemma dependent-bigger-set-general:
  \((\text{finite } S \Longrightarrow \text{card } S > \text{dim } S) \Longrightarrow \text{dependent } S\)
using independent-bound-general[of S] by (metis linorder-not-le)

lemma subset-le-dim:
  \(S \subseteq \text{span } T \Longrightarrow \text{dim } S \leq \text{dim } T\)
by (metis dim-span dim-subset)

lemma linear-inj-imp-surj:
  assumes \(lf\): linear scale scale \(f\)
  and \(fi\): inj \(f\)
  shows surj \(f\)

proof
  interpret \(lf\): linear scale scale \(f\) by fact
from basis-exists[of UNIV] obtain \(B\)
  where \(B: B \subseteq \text{UNIV independent } B \text{ UNIV} \subseteq \text{span } B\) card \(B = \text{dim UNIV}\)
by blast
from $B(4)$ have $d$: dim $\text{UNIV} = \text{card } B$
by simp
have $\text{UNIV} \subseteq \text{span } (f \cdot B)$
proof (rule card-ge-dim-independent)
  show independent $(f \cdot B)$
    by (simp add: $B(2)$ fi if. independent-inj-image)
  have $\text{card } (f \cdot B) = \text{dim } \text{UNIV}$
    by (metis $B(1)$ card-image $d$ fi inj-on-subset)
  then show $\text{dim } \text{UNIV} \leq \text{card } (f \cdot B)$
by simp
qed blast
then show $?\text{thesis}$
  unfolding $lf$. span-image surj-def
  using $B(3)$ by blast
qed

locale finite-dimensional-vector-space-pair-1 =
vs1: finite-dimensional-vector-space s1 $B1$ + vs2: vector-space s2
for s1 :: 'a::field ⇒ 'b::ab-group-add ⇒ 'b (infixr "*" 75)
and $B1$ :: 'b set
and s2 :: 'a::field ⇒ 'c::ab-group-add ⇒ 'c (infixr "*" 75)
begin
sublocale vector-space-pair s1 s2 by unfold-locales

lemma dim-image-eq:
  assumes $lf$: linear s1 s2 $f$
    and $fi$: inj-on f (vs1.span $S$)
  shows vs2.dim $(f \cdot S) = vs1.dim S$
proof –
  interpret $lf$: linear by fact
  obtain $B$ where $B$: $B \subseteq S$ vs1.independent $B$ $S \subseteq$ vs1.span $B$ card $B = vs1.dim S$
    using vs1.basis-exists[of $S$] by auto
  then have vs1.span $S = vs1.span B$
  moreover have card $(f \cdot B) = \text{card } B$
    by auto
  moreover have $(f \cdot B) \subseteq (f \cdot S)$
    using $B$ by auto
  ultimately show $?\text{thesis}$
    by (metis $B(2)$ $B(4)$ fi if. dependent-inj-imageD $lf$. span-image vs2.dim-eq-card-independent
      vs2.dim-span)
qed
lemma dim-image-le:
  assumes lf: linear s1 s2 f
  shows vs2.\dim (f ' S) \leq vs1.\dim (S)
proof -
  from vs1.basis-exists[of S] obtain B where
    B: B \subseteq S vs1.independent B S \subseteq vs1.span B card B = vs1.dim S by blast
  from B have fB: finite B card B = vs1.dim S using vs1.independent-bound-general by blast+
  have vs2.dim (f ' S) \leq \card (f ' B) apply (rule vs2.span-card-ge-dim)
  using lf fB apply (auto simp add: module-hom.span-image module-hom.spans-image subset-image-iff linear-iff-module-hom)
  done
  also have \ldots \leq vs1.dim S using card-image-le[OF fB(1)] fB by simp
  finally show \?thesis . qed

locale finite-dimensional-vector-space-pair =
  vs1: finite-dimensional-vector-space s1 B1 + vs2: finite-dimensional-vector-space s2 B2
  for s1 :: 'a::field \Rightarrow 'b::ab-group-add \Rightarrow 'b (infixr *a 75)
  and B1 :: 'b set
  and s2 :: 'a::field \Rightarrow 'c::ab-group-add \Rightarrow 'c (infixr *b 75)
  and B2 :: 'c set
begin

sublocale finite-dimensional-vector-space-pair-1 ..

lemma linear-surjective-imp-injective:
  assumes lf: linear s1 s2 f and sf: surj f and eq: vs2.\dim UNIV = vs1.\dim UNIV
  shows inj f
proof -
  interpret linear s1 s2 f by fact
  have *: \card (f ' B1) \leq vs2.\dim UNIV using vs1.finite-Basis vs1.dim-eq-card[of B1 UNIV] sf
    by (auto simp: vs1.span-Basis vs1.independent-Basis eq simp del: vs2.dim-UNIV intro!: card-image-le)
  have indep-fB: vs2.independent (f ' B1) using vs1.finite-Basis vs1.dim-eq-card[of B1 UNIV] sf *
    by (intro vs2.card-le-dim-spanning[of f ' B1 UNIV]) (auto simp: span-image vs1.span-Basis)
have vs2.dim UNIV ≤ card (f ` B1)
  unfolding eq sf [symmetric] vs2.dim-span-card-independent [symmetric, OF indep-fB]
  vs2.dim-span
  by (intro vs2.dim-mono) (auto simp: span-image vs1.span-Basis)
with * have card (f ` B1) = vs2.dim UNIV by auto
also have ... = card B1
  unfolding eq vs1.dim-UNIV ..
finally have inj-on f B1
  by (subst inj-on-iff-eq-card[OF vs1.finite-Basis])
then show inj f
  using inj-on-span-iff-independent-image[OF indep-fB] vs1.span-Basis by auto
qed

lemma linear-injective-imp-surjective:
assumes lf: linear s1 s2 f and sf: inj f and eq: vs2.dim UNIV = vs1.dim UNIV
shows surj f
proof –
interpret linear s1 s2 f by fact
have *: False if b: b /∈ vs2.span (f ` B1) for b
proof –
  have *: vs2.independent (f ` B1)
    using vs1.independent-Basis by (intro independent-injective-image inj-on-subset[OF sf]) auto
  have **: vs2.independent (insert b (f ` B1))
    using b * by (rule vs2.independent-insert1)
  then have Suc (card B1) = card (insert b (f ` B1))
    using sf[TTHEN inj-on-subset, of B1] by (subst card.insert-remove) (auto intro: vs1.finite-Basis simp: card-image)
also have ... = vs2.dim (insert b (f ` B1))
  using vs2.dim-eq-card-independent[OF **] by simp
also have vs2.dim (insert b (f ` B1)) ≤ vs2.dim B2
  by (rule vs2.dim-mono) (auto simp: vs2.span-Basis)
also have ... = card B1
  vs1.dim-eq-card-independent[OF vs1.independent-Basis] by simp
finally show False by simp
qed

have f ` UNIV = f ` vs1.span B1 unfolding vs1.span-Basis ..
also have ... = vs2.span (f ` B1) unfolding span-image ..
also have ... = UNIV using * by blast
finally show ?thesis .
qed

lemma linear-injective-isomorphism:
assumes lf: linear s1 s2 f
and \( \text{fi}: \text{inj} f \)
and \( \text{dims}: \text{vs2}.\dim \text{UNIV} = \text{vs1}.\dim \text{UNIV} \)
shows \( \exists f'. \text{linear} s2 s1 f' \land (\forall x. f'(f x) = x) \land (\forall x. f(f' x) = x) \)
unfolding isomorphism-expand\[\text{symmetric}\]
using linear-injective-imp-surjective\[\text{OF} \ \text{fi} \ \text{dims}\]
using fi left-right-inverse-eq lf linear-injective-left-inverse linear-surjective-right-inverse
by blast

lemma linear-surjective-isomorphism:
assumes \( \text{lf}: \text{linear} s1 s2 f \)
and \( \text{sf}: \text{surj} f \)
and \( \text{dims}: \text{vs2}.\dim \text{UNIV} = \text{vs1}.\dim \text{UNIV} \)
shows \( \exists f'. \text{linear} s2 s1 f' \land (\forall x. f'(f x) = x) \land (\forall x. f(f' x) = x) \)
using linear-surjective-imp-injective\[\text{OF} \ \text{lf} \ \text{sf} \ \text{dims}\]
sf linear-exists-right-inverse-on \[\text{OF} \ \text{lf} \ \text{vs1}.\subspace-\text{UNIV}\]
sf linear-exists-left-inverse-on \[\text{OF} \ \text{lf} \ \text{vs1}.\subspace-\text{UNIV}\]
dims \( \text{lf} \ \text{linear-injective-isomorphism} \) by auto

lemma basis-to-basis-subspace-isomorphism:
assumes \( s: \text{vs1}.\subspace S \)
and \( t: \text{vs2}.\subspace T \)
and \( d: \text{vs1}.\dim S = \text{vs2}.\dim T \)
and \( B: B \subseteq S \ \text{vs1}.\text{independent} B S \subseteq \text{vs1}.\span B \ \text{card} B = \text{vs1}.\dim S \)
and \( C: C \subseteq T \ \text{vs2}.\text{independent} C T \subseteq \text{vs2}.\span C \ \text{card} C = \text{vs2}.\dim T \)
shows \( \exists f. \text{linear} s1 s2 f \land f' B = C \land f' S = T \land \text{inj-on} f S \)
proof –
from \( B \) have \( \text{fB}: \text{finite} B \)
  by (simp add: vs1.finiteI-independent)
from \( C \) have \( \text{fC}: \text{finite} C \)
  by (simp add: vs2.finiteI-independent)
from \[\text{finite-basis-to-basis-subspace-isomorphism}\[\text{OF} \ s \ t \ d \ \text{fB} \ \text{fC} \ \text{C} \] show \( ?\text{thesis} \)
qed

end
context finite-dimensional-vector-space begin
lemma linear-surj-imp-inj:
assumes \( \text{lf}: \text{linear} \ \text{scale} \ \text{scale} f \) and \( \text{sf}: \text{surj} f \)
shows \( \text{inj} f \)
proof –
interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-locales
let \( ?U = \text{UNIV} :: \, 'b \text{ set} \)
from \[\text{basis-exists}[\text{of} \ ?U]\] obtain \( B \)
  where \( B: B \subseteq ?U \ \text{independent} B \ ?U \subseteq \text{span} B \) and \( d: \text{card} B = \text{dim} ?U \)
  by blast
}\
fix x
assume x: x ∈ span B and fx: f x = 0
from B(2) have fB: finite B
  using finiteI-independent by auto
have Uspan: UNIV ⊆ span (f · B)
  by (simp add: B(3) if linear-spanning-surjective-image sf)
have fBi: independent (f · B)
proof (rule card-le-dim-spanning)
  show card (f · B) ≤ dim ?U
    using card-image-le d fB by fastforce
qed (use fB Uspan in auto)
have th0: dim ?U ≤ card (f · B)
  by (rule span-card-ge-dim) (use Uspan fB in auto)
moreover have card (f · B) ≤ card B
  by (rule card-image-le, rule fB)
ultimately have th1: card B = card (f · B)
  unfolding d by arith
have fB: inj-on f B
  by (simp add: eq-card-imp-inj-on fB th1)
from linear-indep-image-lemma[OF lf fB fBi fiB x] fx
have x = 0 by blast
} then show ?thesis
  unfolding linear-inj-iff-eq-0[OF lf] using B(3) by blast
qed

lemma linear-inverse-left:
  assumes lf: linear scale scale f
  and lf': linear scale scale f'
  shows f ◦ f' = id ↔ f' ◦ f = id
proof –
  { fix f f': 'b ⇒ 'b
    assume lf: linear scale scale f linear scale scale f'
    assume f: f ◦ f' = id
    from f have sf: surj f
      by (auto simp add: o-def id-def surj-def metis)
    interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-local
    from linear-surjective-isomorphism[OF lf(1) sf] lf f
    have f' ◦ f = id
      unfolding fun-eq iff o-def id-def by metis
  }
then show ?thesis
  using lf' by metis
qed

lemma left-inverse-linear:
  assumes lf: linear scale scale f
and \( gf \): \( g \circ f = id \)
shows linear scale scale \( g \)

proof -
from \( gf \) have \( fi \): \( inj f \)
by (auto simp add: inj-on-def o-def id-def fun-eq-iff) metis
interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-locales
from linear-injective-isomorphism[OF \( if \) \( fi \)]
obtain \( h \): \( 'b \Rightarrow 'b \) where linear scale scale \( h \) and \( h: \forall x. h \( f x \) = x \forall x. f \( h x \) = x \)
by blast
have \( h = g \)
by (metis \( gf \) \( h \) isomorphism-expand left-right-inverse-eq)
with \( \langle \text{linear scale scale } h \rangle \) show \?thesis by blast
qed

lemma inj-linear-imp-inv-linear:
assumes linear scale scale \( f \) \( inj f \) shows linear scale scale \( (\text{inv } f) \)
using assms inj-iff left-inverse-linear by blast

lemma right-inverse-linear:
assumes \( \text{if } \): linear scale scale \( f \)
and \( gf \): \( f \circ g = id \)
shows linear scale scale \( g \)

proof -
from \( gf \) have \( fi \): \( surj f \)
by (auto simp add: surj-def o-def id-def) metis
interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-locales
from linear-surjective-isomorphism[OF \( if \) \( fi \)]
obtain \( h \): \( 'b \Rightarrow 'b \) where \( h \): linear scale scale \( h \) \( \forall x. h \( f x \) = x \forall x. f \( h x \) = x \)
by blast
then have \( h = g \)
by (metis \( gf \) isomorphism-expand left-right-inverse-eq)
with \( h(1) \) show \?thesis by blast
qed

end

context finite-dimensional-vector-space-pair begin

lemma subspace-isomorphism:
assumes \( s \): \( \text{vs1.subspace } S \)
and \( t \): \( \text{vs2.subspace } T \)
and \( d \): \( \text{vs1.dim } S = \text{vs2.dim } T \)
shows \( \exists f. \text{linear } s1 \( s2 \) \( f \) \( S = T \wedge inj-on f S \)

proof -
from \( \text{vs1.basis-exists[of } S \) \( \text{vs1.finiteI-independent} \)
obtain \( B \) where \( B \): \( B \subseteq S \) \( \text{vs1.independent } B \) \( S \subseteq \text{vs1.span } B \) \( \text{card } B = \text{vs1.dim } \)

end
S and fB: finite B

by metis
from vs2.basis-exists[of T] vs2.finiteI-independent
obtain C where C: C ⊆ T vs2.independent C T ⊆ vs2.span C card C = vs2.dim T
and fC: finite C
  by metis
from B(4) C(4) card-le-inj[of B C] d
obtain f where f: f ' B ⊆ C inj-on f B using ⟨finite B⟩ ⟨finite C⟩
  by auto
from linear-independent-extend[OF B(2)]
obtain g where g: linear s1 s2 g ∀x ∈ B. g x = f x
  by blast
interpret g: linear s1 s2 g by fact
from inj-on-iff-eq-card[OF fB, of f]
have ceq: card (f ' B) = card B
by simp
also have . . . = C using card-subset-eq[OF fC, of f]
finally have gBC: g ' B = C.

have gi: inj-on g B
  using f(2) g(2) by (auto simp add: inj-on-def)
note g0 = linear-indep-image-lemma[OF g(1) fB, unfolded gBC, OF C(2) gi]
{
  fix x y
  assume x: x ∈ S and y: y ∈ S and gxy: g x = g y
  from B(3) x y have x': x ∈ vs1.span B and y': y ∈ vs1.span B
    by blast
  from gxy have th0: g (x - y) = 0
    by (simp add: linear-diff g)
  have th1: x - y ∈ vs1.span B
    using x' y' by (metis vs1.span-diff)
  have x = y
    using g0[OF th1 th0]
  have gS: inj-on g S
    unfolding inj-on-def by simp
  from vs1.span-subspace[OF B(1,3) s] have g : S = vs2.span (g ' B)
    by (simp add: module-hom.span-image[OF g(1)[unfolded linear-iff-module-hom]])
  also have . . . = vs2.span C unfolding gBC ..
  also have . . . = T using vs2.span-subspace[OF C(1,3) t] .
  finally have g5: g ' S = T .
  from g(1) g5 giS have ?thesis
    by blast
qed

end
hide-const (open) linear

108 Vector Spaces and Algebras over the Reals

theory Real-Vector-Spaces
  imports Real Topological-Spaces Vector-Spaces
begin

108.1 Real vector spaces

class scaleR =
  fixes scaleR :: real ⇒ 'a ⇒ 'a (infixr " * " 75)
begin

abbreviation divideR :: 'a ⇒ real ⇒ 'a (infixl " / " 70)
where x / R r ≡ inverse r * R x

end

class real-vector = scaleR + ab-group-add +
  assumes scaleR-add-right: a * R (x + y) = a * R x + a * R y
  and scaleR-add-left: (a + b) * R x = a * R x + b * R x
  and scaleR-scaleR: a * R b * R x = (a * b) * R x
  and scaleR-one: 1 * R x = x

class real-algebra = real-vector + ring +
  assumes mult-scaleR-left [simp]: a * R x * y = a * R (x * y)
  and mult-scaleR-right [simp]: x * a * R y = a * R (x * y)

class real-algebra-1 = real-algebra + ring-1

class real-div-algebra = real-algebra-1 + division-ring

class real-field = real-div-algebra + field

instantiation real :: real-field
begin

definition real-scaleR-def [simp]: scaleR a x = a * x

instance
  by standard (simp add: algebra-simps)

end

locale linear = Vector-Spaces.linear scaleR::⇒⇒ 'a::real-vector scaleR::⇒⇒ 'b::real-vector
begin
lemmas $\text{scaleR} = \text{scale}$

end

global-interpretation $\text{real-vector?} :: \text{vector-space scaleR} :: \text{real} \Rightarrow 'a \Rightarrow 'a :: \text{real-vector}$
rewrites $\text{Vector-Spaces.linear} ((* R) (* R)) = \text{linear}$
and $\text{Vector-Spaces.linear} (\text{(*)}) (* R) = \text{linear}$
defines dependent-raw-def: dependent = $\text{real-vector.dependent}$
and representation-raw-def: representation = $\text{real-vector.representation}$
and subspace-raw-def: subspace = $\text{real-vector.subspace}$
and span-raw-def: span = $\text{real-vector.span}$
and extend-basis-raw-def: extend-basis = $\text{real-vector.extend-basis}$
and dim-raw-def: dim = $\text{real-vector.dim}$
proof unfold-locales
  show $\text{Vector-Spaces.linear} ((* R) (* R)) = \text{linear}$ $\text{Vector-Spaces.linear} (\text{(*)}) (* R) = \text{linear}$
  by (force simp: linear-def real-scaleR-def[abs-def])+
qed (use scaleR-add-right scaleR-add-left scaleR-scaleR scaleR-one in auto)

hide-const (open) — locale constants
  $\text{real-vector.dependent}$
  $\text{real-vector.independent}$
  $\text{real-vector.representation}$
  $\text{real-vector.subspace}$
  $\text{real-vector.span}$
  $\text{real-vector.extend-basis}$
  $\text{real-vector.dim}$

abbreviation $\text{independent} x \equiv \neg \text{dependent} x$

global-interpretation $\text{real-vector?} :: \text{vector-space-pair scaleR} :: \Rightarrow \Rightarrow 'a::\text{real-vector}$
and $\text{scaleR} :: \Rightarrow \Rightarrow 'b::\text{real-vector}$
rewrites $\text{Vector-Spaces.linear} ((* R) (* R)) = \text{linear}$
and $\text{Vector-Spaces.linear} (\text{(*)}) (* R) = \text{linear}$
defines construct-raw-def: construct = $\text{real-vector.construct}$
proof unfold-locales
  show $\text{Vector-Spaces.linear} (\text{(*)}) (* R) = \text{linear}$
  unfolding linear-def real-scaleR-def by auto
qed (auto simp: linear-def)

hide-const (open) — locale constants
  $\text{real-vector.construct}$

lemma $\text{linear-compose}: \text{linear} f \Longrightarrow \text{linear} g \Longrightarrow \text{linear} (g \circ f)$
unfolding linear-def by (rule Vector-Spaces.linear-compose)

Recover original theorem names
lemmas $\text{scaleR-left-commute} = \text{real-vector.scale-left-commute}$
lemmas scaleR-zero-left = real-vector.scale-zero-left
lemmas scaleR-minus-left = real-vector.scale-minus-left
lemmas scaleR-diff-left = real-vector.scale-left-diff-distrib
lemmas scaleR-sum-left = real-vector.scale-sum-left
lemmas scaleR-zero-right = real-vector.scale-zero-right
lemmas scaleR-minus-right = real-vector.scale-minus-right
lemmas scaleR-diff-right = real-vector.scale-right-diff-distrib
lemmas scaleR-sum-right = real-vector.scale-sum-right
lemmas scaleR-eq-0-iff = real-vector.scale-eq-0-iff
lemmas scaleR-left-imp-eq = real-vector.scale-left-imp-eq
lemmas scaleR-right-imp-eq = real-vector.scale-right-imp-eq
lemmas scaleR-cancel-left = real-vector.scale-cancel-left
lemmas scaleR-cancel-right = real-vector.scale-cancel-right

lemma [field-simps]:
c ≠ 0 ⇒ a = b / R c ⇐= c *R a = b
c ≠ 0 ⇒ b / R c = a ⇐= b = c *R a
c ≠ 0 ⇒ a + b / R c = (c *R a + b) / R c
c ≠ 0 ⇒ a / R c + b = (a + c *R b) / R c
c ≠ 0 ⇒ a - b / R c = (c *R a - b) / R c
c ≠ 0 ⇒ a / R c - b = (a - c *R b) / R c
c ≠ 0 ⇒ - (a / R c) + b = (- a + c *R b) / R c
c ≠ 0 ⇒ - (a / R c) - b = (- a - c *R b) / R c
for a b :: 'a :: real-vector
by (auto simp add: scaleR-add-right scaleR-add-left scaleR-diff-right scaleR-diff-left)

Legacy names
lemmas scaleR-left-distrib = scaleR-add-left
lemmas scaleR-right-distrib = scaleR-add-right
lemmas scaleR-left-diff-distrib = scaleR-diff-left
lemmas scaleR-right-diff-distrib = scaleR-diff-right

lemmas linear-injective-0 = linear-inj-iff-eq-0
and linear-injective-on-subspace-0 = linear-inj-on-iff-eq-0
and linear-cmul = linear-scale
and linear-scaleR = linear-scale-self
and subspace-mul = subspace-scale
and span-linear-image = linear-span-image
and span-0 = span-zero
and span-mul = span-scale
and injective-scaleR = injective-scale

lemma scaleR-minus1-left [simp]: scaleR (-1) x = - x
for x :: 'a::real-vector
by simp

lemma scaleR-2:
fixes x :: 'a::real-vector
shows scaleR 2 x = x + x
unfolding one-add-one [symmetric] scaleR-left-distrib by simp

lemma scaleR-half-double [simp]:
fixes a :: 'a::real-vector
shows \((1/2) \cdot_R (a + a) = a\)
proof
  have \(\forall r. \ r \cdot_R (a + a) = (r \cdot 2) \cdot_R a\)
  by (metis scaleR-2 scaleR-scaleR)
  then show \(?thesis\)
  by simp
qed

lemma shift-zero-ident [simp]:
fixes f :: 'a => 'b::real-vector
shows \((+0) \circ f = f\)
by force

lemma linear-scale-real:
fixes r :: real
shows linear f \(\Rightarrow\) \(f (r \cdot_B b) = r \cdot_B f b\)
using linear-scale by fastforce

interpretation scaleR-left: additive \((\lambda a. \cdot_R a) x :: 'a::real-vector)\)
by standard (rule scaleR-left-distrib)

interpretation scaleR-right: additive \((\lambda x. \cdot_R a) x :: 'a::real-vector)\)
by standard (rule scaleR-right-distrib)

lemma nonzero-inverse-scaleR-distrib:
\(a \neq 0 \Rightarrow x \neq 0 \Rightarrow inverse (\cdot_R a) x = \cdot_R (inverse a) (inverse x)\)
for x :: 'a::real-div-algebra
by (rule inverse-unique) simp

lemma inverse-scaleR-distrib: inverse \(\cdot_R a) x = \cdot_R (inverse a) (inverse x)\)
for x :: 'a::{real-div-algebra,division-ring}
by (metis inverse-zero nonzero-inverse-scaleR-distrib scale-eq-0-iff)

lemmas sum-constant-scaleR = real-vector.sum-constant-scale — legacy name

named-theorems vector-add-divide-simps to simplify sums of scaled vectors

lemma [vector-add-divide-simps]:
v + (b / z) \cdot_R w = (if z = 0 then v else (z \cdot_R v + b \cdot_R w) / _R z)
a \cdot_R v + (b / z) \cdot_R w = (if z = 0 then a \cdot_R v else ((a * z) \cdot_R v + b \cdot_R w) / _R z)
(a / z) \cdot_R v + w = (if z = 0 then w else (a \cdot_R v + z \cdot_R w) / _R z)
(a / z) \cdot_R v + b \cdot_R w = (if z = 0 then b \cdot_R w else (a \cdot_R v + (b * z) \cdot_R w) / _R z)
v - (b / z) \cdot_R w = (if z = 0 then v else (z \cdot_R v - b \cdot_R w) / _R z)
THEORY "Real-Vector-Spaces"

\[
a *_R v - (b / z) *_R w = (if z = 0 then a *_R v else ((a * z) *_R v - b *R w) / R z)
\]
\[
(a / z) *_R v - w = (if z = 0 then -w else (a *_R v - z *_R w) / R z)
\]
\[
(a / z) *_R v - b *_R w = (if z = 0 then -b *_R w else (a *_R v - (b * z) *_R w) / R z)
\]

for \(v :: 'a :: real-vector\)
by (simp-all add: divide-inverse-commute scaleR-add-right scaleR-diff-right)

lemma eq-vector-fraction-iff [vector-add-divide-simps]:
fixes \(x :: 'a :: real-vector\)
shows \((x = (u / v) *_R a) \iff (if v=0 then x = 0 else v *_R x = u *_R a)\)
by auto (metis (no-types) divide-eq-1-iff divide-inverse-commute scaleR-one scaleR-scaleR)

lemma vector-fraction-eq-iff [vector-add-divide-simps]:
fixes \(x :: 'a :: real-vector\)
shows \((u / v) *_R a = x \iff (if v=0 then x = 0 else u *_R a = v *_R x)\)
by (metis eq-vector-fraction-iff)

lemma real-vector-affinity-eq:
fixes \(x :: 'a :: real-vector\)
assumes \(m0: m \neq 0\)
shows \(m *_R x + c = y \iff x = inverse m *_R y - (inverse m * R c)\)
(is \(\text{lhs} \iff \text{rhs}\))
proof
  assume \(\text{lhs}\)
  then have \(m *_R x = y - c\) by (simp add: field-simps)
  then have \(inverse m *_R (m *_R x) = inverse m *_R (y - c)\) by simp
  then show \(x = inverse m *_R y - (inverse m *_R c)\)
    using \(m0\)
  by (simp add: scaleR-diff-right)
next
  assume \(\text{rhs}\)
  with \(m0\) show \(m *_R x + c = y\)
    by (simp add: scaleR-diff-right)
qed

lemma real-vector-eq-affinity: \(m \neq 0 \implies y = m *_R x + c \iff inverse m *_R y - (inverse m *_R c) = x\)
  for \(x :: 'a::real-vector\)
using real-vector-affinity-eq[where \(m=m\) and \(x=x\) and \(y=y\) and \(c=c\)]
by metis

lemma scaleR-eq-iff [simp]: \(b + u *_R a = a + u *_R b \iff a = b \lor u = 1\)
  for \(a :: 'a::real-vector\)
proof (cases \(u = 1\))
  case True
  then show \(\text{thesis}\) by auto
next
case False
have a = b if b + u *R a = a + u *R b
proof -
  from that have (u - 1) *R a = (u - 1) *R b
  by (simp add: algebra-simps)
with False show ?thesis
  by auto
qed
then show ?thesis by auto
qed

lemma scaleR-collapse [simp]: (1 - u) *R a + u *R a = a
  for a :: 'a::real-vector
  by (simp add: algebra-simps)

108.2 Embedding of the Reals into any real-algebra-1: of-real
definition of-real :: real ⇒ 'a::real-algebra-1
  where of-real r = scaleR r 1

lemma scaleR-conv-of-real: scaleR r x = of-real r * x
  by (simp add: of-real-def)

lemma of-real-0 [simp]: of-real 0 = 0
  by (simp add: of-real-def)

lemma of-real-1 [simp]: of-real 1 = 1
  by (simp add: of-real-def)

lemma of-real-add [simp]: of-real (x + y) = of-real x + of-real y
  by (simp add: of-real-def scaleR-left-distrib)

lemma of-real-minus [simp]: of-real (−x) = −of-real x
  by (simp add: of-real-def)

lemma of-real-diff [simp]: of-real (x - y) = of-real x - of-real y
  by (simp add: of-real-def scaleR-left-diff-distrib)

lemma of-real-mult [simp]: of-real (x * y) = of-real x * of-real y
  by (simp add: of-real-def)

lemma of-real-sum[simp]: of-real (sum f s) = (∑x∈s. of-real (f x))
  by (induct s rule: infinite-finite-induct) auto

lemma of-real-prod[simp]: of-real (prod f s) = (∏x∈s. of-real (f x))
  by (induct s rule: infinite-finite-induct) auto

lemma nonzero-of-real-inverse:
  x ≠ 0 ☨ of-real (inverse x) = inverse (of-real x :: 'a::real-div-algebra)
by (simp add: of-real-def nonzero-inverse-scaleR-distrib)

**lemma** of-real-inverse [simp]:

\(\text{of-real } (\text{inverse } x) = \text{inverse } (\text{of-real } x :: 'a::\{\text{real-div-algebra},\text{division-ring}\})\)

by (simp add: of-real-def inverse-scaleR-distrib)

**lemma** nonzero-of-real-divide:

\(y \neq 0 \implies \text{of-real } (x / y) = (\text{of-real } x / \text{of-real } y :: 'a::\text{real-field})\)

by (simp add: divide-inverse nonzero-of-real-inverse)

**lemma** of-real-divide [simp]:

\(\text{of-real } (x / y) = (\text{of-real } x / \text{of-real } y :: 'a::\text{real-div-algebra})\)

by (simp add: divide-inverse)

**lemma** of-real-power [simp]:

\(\text{of-real } (x ^ n) = (\text{of-real } x :: 'a::\{\text{real-algebra-1}\}) ^ n\)

by (induct n) simp-all

**lemma** of-real-power-int [simp]:

\(\text{of-real } (\text{power-int } x n) = \text{power-int } (\text{of-real } x :: 'a::\{\text{real-div-algebra},\text{division-ring}\}) n\)

by (auto simp: power-int-def)

**lemma** of-real-eq-iff [simp]: \(\text{of-real } x = \text{of-real } y \iff x = y\)

by (simp add: of-real-def)

**lemma** inj-of-real: \(\text{inj } \text{of-real}\)

by (auto intro: injI)

**lemmas** of-real-eq-0-iff [simp] = of-real-eq-iff [of 0, simplified]

**lemmas** of-real-eq-1-iff [simp] = of-real-eq-iff [of 1, simplified]

**lemma** minus-of-real-eq-of-real-iff [simp]: \(\text{of-real } x = \text{of-real } y \iff -x = y\)

using of-real-eq-iff[of \(-x\) \(y\)] by (simp only: of-real-minus)

**lemma** of-real-eq-minus-of-real-iff [simp]: \(\text{of-real } x = -\text{of-real } y \iff x = -y\)

using of-real-eq-iff[of \(x\) \(-y\)] by (simp only: of-real-minus)

**lemma** of-real-eq-id [simp]: \(\text{of-real } = (\text{id } :: \text{real } \Rightarrow \text{real})\)

by (rule ext) (simp add: of-real-def)

Collapse nested embeddings.

**lemma** of-real-of-nat-eq [simp]: \(\text{of-real } (\text{of-nat } n) = \text{of-nat } n\)

by (induct n) auto

**lemma** of-real-of-int-eq [simp]: \(\text{of-real } (\text{of-int } z) = \text{of-int } z\)

by (cases \(z\) rule: int-diff-cases) simp

**lemma** of-real-numeral [simp]: \(\text{of-real } (\text{numeral } w) = \text{numeral } w\)
using of-real-of-int-eq [of numeral w] by simp

lemma of-real-neg-numeral [simp]: of-real (− numeral w) = − numeral w
using of-real-of-int-eq [of − numeral w] by simp

lemma numeral-power-int-eq-of-real-cancel-iff [simp]:
power-int (numeral x) n = (of-real y :: real-div-algebra, division-ring) ←→
power-int (numeral x) n = y
proof
have power-int (numeral x) n = (of-real (power-int (numeral x) n) :: a)
by simp
also have . . . = of-real y ←→ power-int (numeral x) n = y
by (subst of-real-eq-iff) auto
finally show ?thesis .
qed

lemma of-real-eq-numeral-power-int-cancel-iff [simp]:
(of-real y :: a :: real-div-algebra, division-ring) = power-int (numeral x) n ←→
y = power-int (numeral x) n
by (subst (1 2) eq-commute) simp

lemma of-real-eq-of-real-power-int-cancel-iff [simp]:
power-int (of-real b :: a :: real-div-algebra, division-ring) w = of-real x ←→
power-int b w = x
by (metis of-real-power-int of-real-eq-iff)

lemma of-real-in-Ints-iff [simp]: of-real x ∈ ℤ ←→ x ∈ ℤ
proof safe
fix x assume (of-real x :: a) ∈ ℤ
then obtain n where (of-real x :: a) = of-int n
by (auto simp: Ints-def)
also have of-int n = of-real (real-of-int n)
by simp
finally have x = real-of-int n
by (subst (asm) of-real-eq-iff)
thus x ∈ ℤ.
by auto
qed (auto simp: Ints-def)

lemma Ints-of-real [intro]: x ∈ ℤ ⇒ of-real x ∈ ℤ
by simp

Every real algebra has characteristic zero.

instance real-algebra-1 < ring-char-0
proof
from inj-of-real inj-of-nat have inj (of-real ∘ of-nat)
by (rule inj-compose)
then show inj (of-real :: nat ⇒ a)
by (simp add: comp-def)
lemmas

108.3 The Set of Real Numbers

definition Reals :: 'a::real-algebra_1 set (R)
  where R = range of-real

lemmas Reals-of-real [simp]: of-real r \in R
  by (simp add: Reals-def)

lemmas Reals-of-int [simp]: of-int z \in R
  by (subst of-real-of-int-eq [symmetric], rule Reals-of-real)

lemmas Reals-of-nat [simp]: of-nat n \in R
  by (subst of-real-of-nat-eq [symmetric], rule Reals-of-real)

lemmas Reals-numeral [simp]: numeral w \in R
  by (subst of-real-numeral [symmetric], rule Reals-of-real)

lemmas Reals-0 [simp]: 0 \in R and Reals-1 [simp]: 1 \in R
  by (simp-all add: Reals-def)

lemmas Reals-add [simp]: a \in R \Longrightarrow b \in R \Longrightarrow a + b \in R
  by (metis (no-types, opaque-lifting) Reals-def Reals-of-real imageE of-real-add)

lemmas Reals-minus [simp]: a \in R \Longrightarrow - a \in R
  by (auto simp: Reals-def)

lemmas Reals-minus_iff [simp]: - a \in R \iff a \in R
  using Reals-minus by fastforce
lemma Reals-diff [simp]: \( a \in \mathbb{R} \implies b \in \mathbb{R} \implies a - b \in \mathbb{R} \)
by (metis Reals-add Reals-minus-iff add-uminus-conv-diff)

lemma Reals-mult [simp]: \( a \in \mathbb{R} \implies b \in \mathbb{R} \implies a \times b \in \mathbb{R} \)
by (metis (no-types, lifting) Reals-def Reals-of-real imageE of-real-mult)

lemma nonzero-Reals-inverse: \( a \in \mathbb{R} \implies a \neq 0 \implies \text{inverse } a \in \mathbb{R} \)
for \( a :: 'a::real-div-algebra \)
by (metis Reals-def Reals-of-real imageE of-real-inverse)

lemma Reals-inverse [simp]: \( a \in \mathbb{R} \implies \text{inverse } a \in \mathbb{R} \)
for \( a :: 'a::\{real-div-algebra,division-ring\} \)
using nonzero-Reals-inverse by fastforce

lemma Reals-inverse-iff [simp]: \( \text{inverse } x \in \mathbb{R} \iff x \in \mathbb{R} \)
for \( x :: 'a::\{real-div-algebra,division-ring\} \)
by (metis Reals-inverse inverse-inverse-eq)

lemma nonzero-Reals-divide: \( a \in \mathbb{R} \implies b \in \mathbb{R} \implies b \neq 0 \implies a / b \in \mathbb{R} \)
for \( a \ b :: 'a::real-field \)
by (simp add: divide-inverse)

lemma Reals-divide [simp]: \( a \in \mathbb{R} \implies b \in \mathbb{R} \implies a / b \in \mathbb{R} \)
for \( a \ b :: 'a::\{real-field,field\} \)
using nonzero-Reals-divide by fastforce

lemma Reals-power [simp]: \( a \in \mathbb{R} \implies a ^ n \in \mathbb{R} \)
for \( a :: 'a::real-algebra-1 \)
by (metis Reals-def Reals-of-real imageE of-real-power)

lemma Reals-cases [cases set: Reals]:
assumes \( q \in \mathbb{R} \)
obtains (of-real) \( r \) where \( q = \text{of-real } r \)
unfolding Reals-def
proof –
from \( q \in \mathbb{R} \) have \( q \in \text{range of-real unfolding Reals-def} \).
then obtain \( r \) where \( q = \text{of-real } r \).
then show thesis ..
qed

lemma sum-in-Reals [intro,simp]: \( \bigwedge i. i \in s \implies f i \in \mathbb{R} \implies \text{sum } f s \in \mathbb{R} \)
proof (induct s rule: infinite-finite-induct)
  case infinite
  then show \( ?\text{case} \) by (metis Reals-0 sum.infinite)
qed simp-all

lemma prod-in-Reals [intro,simp]: \( \bigwedge i. i \in s \implies f i \in \mathbb{R} \implies \text{prod } f s \in \mathbb{R} \)
proof (induct s rule: infinite-finite-induct)
case infinite
then show \(?\)case by (metis Reals-1 prod.infinite)
qed simp-all

lemma Reals-induct [case-names of-real, induct set: Reals]:
\(q \in \mathbb{R} \implies (\forall r. P (\text{of-real } r)) \implies P q\)
by (rule Reals-cases) auto

108.4 Ordered real vector spaces

class ordered-real-vector = real-vector + ordered-ab-group-add +
  assumes scaleR-left-mono: \(x \leq y \implies 0 \leq a \implies a \cdot_R x \leq a \cdot_R y\)
  and scaleR-right-mono: \(a \leq b \implies 0 \leq x \implies a \cdot_R x \leq b \cdot_R x\)
begin

lemma scaleR-mono:
\(a \leq b \implies x \leq y \implies 0 \leq b \implies 0 \leq c \implies a \cdot_R x \leq b \cdot_R y\)
by (meson order-trans scaleR-left-mono scaleR-right-mono)

lemma scaleR-mono':
\(a \leq b \implies c \leq d \implies 0 \leq a \implies 0 \leq c \implies a \cdot_R c \leq b \cdot_R d\)
by (rule scaleR-mono) (auto intro: order.trans)

lemma pos-le-divideR-eq [field-simps]:
\(a \leq b /_R c \iff c \cdot_R a \leq b\) if \(0 < c\)
proof
  assume \(?P\)
  with scaleR-left-mono that have \(c \cdot_R a \leq c \cdot_R (b /_R c)\)
  by simp
  with that show \(?Q\)
  by (simp add: scaleR-one scaleR-scaleR inverse-eq-divide)
next
  assume \(?Q\)
  with scaleR-left-mono that have \(c \cdot_R a /_R c \leq b /_R c\)
  by simp
  with that show \(?P\)
  by (simp add: scaleR-one scaleR-scaleR inverse-eq-divide)
qed

lemma pos-less-divideR-eq [field-simps]:
\(a < b /_R c \iff c \cdot_R a < b\) if \(c > 0\)
using that pos-le-divideR-eq [of \(c a b\)]
by (auto simp add: le-less scaleR-scaleR scaleR-one)

lemma pos-divideR-le-eq [field-simps]:
\(b /_R c \leq a \iff b \leq c \cdot_R a\) if \(c > 0\)
using that pos-le-divideR-eq [of inverse \(c b a\)] by simp

lemma pos-divideR-less-eq [field-simps]:
\[ \begin{align*}
\text{THEORY } \text{“Real-Vector-Spaces”} & \\
\text{if } c > 0 \quad \text{and } a < b < c \quad \text{then } b / R c < a ≤ b < c * R a & \\
\text{using that pos-less-divideR-eq } [\text{of } \text{inverse } c \ b \ a]\quad \text{by } \text{simp} & \\
\text{lemma } \text{pos-le-minus-divideR-eq } [\text{field-simps}]: \\
a ≤ -(b / R c) ←→ c * R a ≤ - b \quad \text{if } c > 0 & \\
\text{using that by } (\text{metis } \text{add-minus-cancel } \text{diff-0 } \text{left-minus } \text{minus-minus } \text{neg-le-iff-le} \\
& \text{scaleR-add-right } \text{uminus-add-conv-diff } \text{pos-le-divideR-eq}) & \\
\text{lemma } \text{pos-less-minus-divideR-eq } [\text{field-simps}]: \\
a < -(b / R c) ←→ c * R a < - b \quad \text{if } c > 0 & \\
\text{using that by } (\text{metis } \text{le-less } \text{less-le-not-le } \text{pos-divideR-le-eq} \\
& \text{pos-divideR-less-eq } \text{pos-le-minus-divideR-eq}) & \\
\text{lemma } \text{pos-minus-divideR-le-eq } [\text{field-simps}]: \\
-(b / R c) ≤ a ←→ - b ≤ c * R a \quad \text{if } c > 0 & \\
\text{using that by } (\text{metis } \text{pos-divideR-le-eq } \text{pos-less-divideR-eq } \text{inverse-positive-iff-positive} \\
& \text{le-imp-neg-le } \text{minus-minus}) & \\
\text{lemma } \text{pos-minus-divideR-less-eq } [\text{field-simps}]: \\
-(b / R c) < a ←→ - b < c * R a \quad \text{if } c > 0 & \\
\text{using that by } (\text{simp } \text{add}: \text{less-le-not-le } \text{pos-less-divideR-eq } \text{pos-minus-divideR-le-eq}) & \\
\text{lemma } \text{scaleR-image-atLeastAtMost}: c > 0 \implies \text{scaleR } c \ \{x..y\} = \{c * R x..c * R y\} & \\
\text{apply } (\text{auto } \text{intro: } \text{scaleR-left-mono } \text{simp: } \text{image-iff } \text{Bex-def}) & \\
\text{using } \text{pos-divideR-le-eq } [\text{of } c \ a \ b] & \\
\text{apply } (\text{meson } \text{local.order-eq-iff}) & \\
\text{done} & \\
\text{end} & \\
\text{lemma } \text{neg-le-divideR-eq } [\text{field-simps}]: \\
a ≤ b / R c ←→ b ≤ c * R a \quad \text{if } c < 0 & \\
\text{for } a \ b :: \text{`a :: ordered-real-vector} & \\
\text{using that } \text{pos-le-divideR-eq } [\text{of } - c \ a \ b] \text{ by } \text{simp} & \\
\text{lemma } \text{neg-less-divideR-eq } [\text{field-simps}]: \\
a < b / R c ←→ b < c * R a \quad \text{if } c < 0 & \\
\text{for } a \ b :: \text{`a :: ordered-real-vector} & \\
\text{using that } \text{neg-le-divideR-eq } [\text{of } c \ a \ b] \text{ by } (\text{auto } \text{simp add: le-less}) & \\
\text{lemma } \text{neg-divideR-le-eq } [\text{field-simps}]: \\
b / R c ≤ a ←→ c * R a ≤ b \quad \text{if } c < 0 & \\
\text{for } a \ b :: \text{`a :: ordered-real-vector} & \\
\text{using that } \text{pos-divideR-le-eq } [\text{of } - c \ b \ a] \text{ by } \text{simp} & \\
\text{lemma } \text{neg-divideR-less-eq } [\text{field-simps}]: \\
b / R c < a ←→ c * R a < b \quad \text{if } c < 0 & \\
\text{by } \text{simp} & \\
\end{align*}\]
THEORY "Real-Vector-Spaces"

for a b :: 'a :: ordered-real-vector
using that neg-divideR-le-eq [of c b a] by (auto simp add: le-less)

lemma neg-le-minus-divideR-eq [field-simps]:
a ≤ −(b / R c) −→ − b ≤ c ∗ R a if c < 0
for a b :: 'a :: ordered-real-vector
using that pos-le-minus-divideR-eq [of − c a − b] by (simp add: minus-le-iff)

lemma neg-less-minus-divideR-eq [field-simps]:
a < −(b / R c) −→ − b < c ∗ R a if c < 0
for a b :: 'a :: ordered-real-vector
proof −
have *: − b = c ∗ R a −→ b = −(c ∗ R a)
by (metis add.inverse-inverse)
from that neg-le-minus-divideR-eq [of c a b]
show thesis by (auto simp add: le-less *)
qed

lemma neg-minus-divideR-le-eq [field-simps]:
−(b / R c) ≤ a −→ c ∗ R a ≤ − b if c < 0
for a b :: 'a :: ordered-real-vector
using that pos-minus-divideR-eq [of − c b a] by (simp add: le-minus-iff)

lemma neg-minus-divideR-less-eq [field-simps]:
−(b / R c) < a −→ c ∗ R a < − b if c < 0
for a b :: 'a :: ordered-real-vector
using that by (simp add: less-le-not-le neg-le-minus-divideR-eq neg-minus-divideR-le-eq)

lemma [field-split-simps]:
a = b / R c −→ (if c = 0 then a = 0 else c ∗ R a = b)
b / R c = a −→ (if c = 0 then a = 0 else b = c ∗ R a)
a + b / R c = (if c = 0 then a else (c ∗ R a + b) / R c)
a ∗ R c + b = (if c = 0 then b else (a + c ∗ R b) / R c)
a − b / R c = (if c = 0 then a else (c ∗ R a − b) / R c)
a ∗ R c − b = (if c = 0 then b else (a − c ∗ R b) / R c)
−(a / R c) + b = (if c = 0 then b else (− a + c ∗ R b) / R c)
−(a / R c) − b = (if c = 0 then − b else (− a − c ∗ R b) / R c)
for a b :: 'a :: real-vector
by (auto simp add: field-simps)

lemma [field-split-simps]:
0 < c −→ a ≤ b / R c −→ (if c > 0 then c ∗ R a ≤ b else if c < 0 then b ≤ c ∗ R a else a ≤ 0)
0 < c −→ a < b / R c −→ (if c > 0 then c ∗ R a < b else if c < 0 then b < c ∗ R a else a < 0)
0 < c −→ b / R c ≤ a −→ (if c > 0 then b ≤ c ∗ R a else if c < 0 then c ∗ R a ≤ b else a ≥ 0)
0 < c −→ b / R c < a −→ (if c > 0 then b < c ∗ R a else if c < 0 then c ∗ R a < b else a > 0)
\[ 0 < c \implies a \leq - (b \cdot _R c) \iff (if \ c > 0 \ then \ c \cdot _R a \leq - b \ else \ if \ c < 0 \ then \ - b \leq c \cdot _R a \ else \ a \leq 0) \]
\[ 0 < c \implies a < - (b \cdot _R c) \iff (if \ c > 0 \ then \ c \cdot _R a < - b \ else \ if \ c < 0 \ then \ - b < c \cdot _R a \ else \ a < 0) \]
\[ 0 < c \implies - (b \cdot _R c) \leq a \iff (if \ c > 0 \ then \ - b \leq c \cdot _R a \ else \ if \ c < 0 \ then \ c \cdot _R a \leq - b \ else \ a \geq 0) \]
\[ 0 < c \implies - (b \cdot _R c) < a \iff (if \ c > 0 \ then \ - b < c \cdot _R a \ else \ if \ c < 0 \ then \ c \cdot _R a < - b \ else \ a > 0) \]

for \( a, b :: 'a :: \text{ordered-real-vector} \)
by (clarsimp intro!: field-simps)

lemma \( \text{scaleR-nonneg-nonneg} \): \( 0 \leq a \implies 0 \leq x \implies 0 \leq a \cdot _R x \)
for \( x :: 'a::\text{ordered-real-vector} \)
using \( \text{scaleR-left-mono [of 0 x a]} \) by simp

lemma \( \text{scaleR-nonneg-nonpos} \): \( 0 \leq a \implies x \leq 0 \implies a \cdot _R x \leq 0 \)
for \( x :: 'a::\text{ordered-real-vector} \)
using \( \text{scaleR-left-mono [of x 0 a]} \) by simp

lemma \( \text{scaleR-nonpos-nonneg} \): \( a \leq 0 \implies 0 \leq x \implies a \cdot _R x \leq 0 \)
for \( x :: 'a::\text{ordered-real-vector} \)
using \( \text{scaleR-right-mono [of a 0 x]} \) by simp

lemma \( \text{split-scaleR-neg-le} \): \( (0 \leq a \land x \leq 0) \lor (a \leq 0 \land 0 \leq x) \implies a \cdot _R x \leq 0 \)
for \( x :: 'a::\text{ordered-real-vector} \)
by (auto simp: scaleR-nonneg-nonpos scaleR-nonpos-nonneg)

lemma \( \text{le-add-iff1} \): \( a \cdot _R e + c \leq b \cdot _R e + d \iff (a - b) \cdot _R e + c \leq d \)
for \( c, d :: 'a::\text{ordered-real-vector} \)
by (simp add: algebra-simps)

lemma \( \text{le-add-iff2} \): \( a \cdot _R e + c \leq b \cdot _R e + d \iff c \leq (b - a) \cdot _R e + d \)
for \( c, d :: 'a::\text{ordered-real-vector} \)
by (simp add: algebra-simps)

lemma \( \text{scaleR-left-mono-neg} \): \( b \leq a \implies c \leq 0 \implies c \cdot _R a \leq c \cdot _R b \)
for \( a, b :: 'a::\text{ordered-real-vector} \)
by (drule scaleR-left-mono [of - - c], simp-all)

lemma \( \text{scaleR-right-mono-neg} \): \( b \leq a \implies c \leq 0 \implies a \cdot _R c \leq b \cdot _R c \)
for \( c :: 'a::\text{ordered-real-vector} \)
by (drule scaleR-right-mono [of - - c], simp-all)

lemma \( \text{scaleR-nonpos-nonpos} \): \( a \leq 0 \implies b \leq 0 \implies 0 \leq a \cdot _R b \)
for \( b :: 'a::\text{ordered-real-vector} \)
using \( \text{scaleR-right-mono-neg [of a 0 b]} \) by simp

lemma \( \text{split-scaleR-pos-le} \): \( 0 \leq a \land 0 \leq b \lor (a \leq 0 \land b \leq 0) \implies 0 \leq a \cdot _R b \)
for \( b :: 'a::\text{ordered-real-vector} \)
by (auto simp: scaleR-nonneg-nonneg scaleR-nonpos-nonpos)

lemma zero-le-scaleR-iff:
  fixes b :: 'a::ordered-real-vector
  shows 0 ≤ a * R b ←→ 0 < a ∧ 0 ≤ b ∨ a < 0 ∧ b ≤ 0 ∨ a = 0
  (is {?lhs = ?rhs})
  proof (cases a = 0)
    case True
    then show ?thesis by simp
  next
    case False
    show ?thesis
      proof
        assume ?lhs
        from \( a \neq 0 \) consider a > 0 | a < 0 by arith
        then show ?rhs
          proof
            cases
            case 1
            with (?lhs) have inverse a * R 0 ≤ inverse a * R (a * R b)
            by (intro scaleR-mono) auto
            with 1 show ?thesis
            by simp
          next
            case 2
            with (?lhs) have − inverse a * R 0 ≤ − inverse a * R (a * R b)
            by (intro scaleR-mono) auto
            with 2 show ?thesis
            by simp
          qed
        next
        assume ?rhs
        then show ?lhs
          by (auto simp: not-le \( a \neq 0 \) intro: split-scaleR-pos-le)
      qed
  qed

lemma scaleR-le-0-iff: a * R b ≤ 0 ←→ 0 < a ∧ b ≤ 0 ∨ a < 0 ∧ b ≤ 0 ∨ a = 0
  for b::'a::ordered-real-vector
  by (insert zero-le-scaleR-iff [of − a b]) force

lemma scaleR-le-cancel-left: c * R a ≤ c * R b ←→ (0 < c → a ≤ b) ∧ (c < 0 → b ≤ a)
  for b::'a::ordered-real-vector
  by (auto simp: neq-iff scaleR-left-mono scaleR-left-mono-neg
dest: scaleR-left-mono[where a=inverse c] scaleR-left-mono-neg[where c=inverse c])

lemma scaleR-le-cancel-left-pos: 0 < c → c * R a ≤ c * R b ←→ a ≤ b
  for b::'a::ordered-real-vector
```
by (auto simp: scaleR-le-cancel-left)

lemma scaleR-le-cancel-left-neg: \( c < 0 \rightleftharpoons c * a \leq c * b \leftrightarrow b \leq a \)
  for b :: 'a::ordered-real-vector
  by (auto simp: scaleR-le-cancel-left)

lemma scaleR-left-le-one-le: \( 0 \leq x \rightleftharpoons a \leq 1 \rightleftharpoons a * x \leq x \)
  for x :: 'a::ordered-real-vector and a :: real
  using scaleR-right-mono[of a 1 x] by simp

108.5 Real normed vector spaces

class dist =
  fixes dist :: 'a ⇒ 'a ⇒ real

class norm =
  fixes norm :: 'a ⇒ real

class sgn-div-norm = scaleR + norm + sgn +
  assumes sgn-div-norm: sgn x = x /\ R norm x

class dist-norm = dist + norm + minus +
  assumes dist-norm: dist x y = norm (x - y)

class uniformity-dist = dist + uniformity +
  assumes uniformity-dist: uniformity = (INF e∈{0 <..}. principal {(x, y). dist x y < e})
begin

lemma eventually-uniformity-metric:
  eventually P uniformity ←→ (∃ e>0. ∀ x y. dist x y < e → P (x, y))
  unfolding uniformity-dist
  by (subst eventually-INF-base)
    (auto simp: eventually-principal subset-eq intro: bexI[of - min - -])
end

class real-normed-vector = real-vector + sgn-div-norm + dist-norm + uniformity-dist + open-uniformity +
  assumes norm-eq-zero [simp]: norm x = 0 ↔ x = 0
  and norm-triangle-ineq: norm (x + y) ≤ norm x + norm y
  and norm-scaleR [simp]: norm (scaleR a x) = |a| * norm x
begin

lemma norm-ge-zero [simp]: \( 0 \leq norm x \)
proof –
  have \( \emptyset = norm \ (x + -1 *\ R x) \)
    using scaleR-add-left[of 1 -1 x] norm-scaleR[of 0 x] by (simp add: scaleR-one)
  also have \( \ldots \leq norm x + norm \ (-1 *\ R x) \)
    by (rule norm-triangle-ineq)
```
finally show ?thesis by simp
qed

lemma bdd-below-norm-image: bdd-below (norm ' A)
  by (meson bdd-belowI2 norm-ge-zero)
end

class real-normed-algebra = real-algebra + real-normed-vector +
  assumes norm-mult-ineq: norm (x * y) ≤ norm x * norm y

class real-normed-algebra-1 = real-algebra-1 + real-normed-algebra +
  assumes norm-one [simp]: norm 1 = 1

lemma (in real-normed-algebra-1) scaleR-power [simp]: (scaleR x y) ^ n = scaleR (x^n) (y^n)
  by (induct n) (simp-all add: scaleR-one scaleR-scaleR mult-ac)

class real-normed-div-algebra = real-div-algebra + real-normed-vector +
  assumes norm-mult: norm (x * y) = norm x * norm y

lemma divideR-right:
  fixes x y :: 'a::real-normed-vector
  shows r ≠ 0 ==> y = x / R r <-> r *R y = x
  by auto

class real-normed-field = real-field + real-normed-div-algebra

instance real-normed-div-algebra < real-normed-algebra-1
  proof
    show norm (x * y) ≤ norm x * norm y for x y :: 'a
      by (simp add: norm-mult)

next
  have norm (1 * 1::'a) = norm (1::'a) * norm (1::'a)
    by (rule norm-mult)
then show norm (1::'a) = 1 by simp
qed

context real-normed-vector begin

lemma norm-zero [simp]: norm (0::'a) = 0
  by simp

lemma zero-less-norm-iff [simp]: norm x > 0 <-> x ≠ 0
  by (simp add: order-less-le)

lemma norm-not-less-zero [simp]: ~ norm x < 0
  by (simp add: linorder-not-less)
lemma norm-le-zero-iff [simp]: norm x ≤ 0 ↔ x = 0
  by (simp add: order-le-less)

lemma norm-minus-cancel [simp]: norm (− x) = norm x
proof −
  have − 1 *R x = − (1 *R x)
    unfolding add-eq-0-iff2[symmetric] scaleR-add-left[symmetric]
    using norm-eq-zero
    by fastforce
  then have norm (− x) = norm (scaleR (− 1) x)
    by (simp only: scaleR-one)
  also have . . . = |− 1| * norm x
    by (rule norm-scaleR)
  finally show ?thesis by simp
qed

lemma norm-minus-commute: norm (a − b) = norm (b − a)
proof −
  have norm (− (b − a)) = norm (b − a)
    by (rule norm-minus-cancel)
  then show ?thesis by simp
qed

lemma dist-add-cancel [simp]: dist (a + b) (a + c) = dist b c
  by (simp add: dist-norm)

lemma dist-add-cancel2 [simp]: dist (b + a) (c + a) = dist b c
  by (simp add: dist-norm)

lemma norm-uminus-minus: norm (− x − y) = norm (x + y)
  by (subst (2) norm-minus-cancel[symmetric], subst minus-add-distrib) simp

lemma norm-triangle-ineq2: norm a − norm b ≤ norm (a − b)
proof −
  have norm (a − b + b) ≤ norm (a − b) + norm b
    by (rule norm-triangle-ineq)
  then show ?thesis by simp
qed

lemma norm-triangle-ineq3: |norm a − norm b| ≤ norm (a − b)
proof −
  have norm a − norm b ≤ norm (a − b)
    by (simp add: norm-triangle-ineq2)
  moreover have norm b − norm a ≤ norm (a − b)
    by (metis norm-minus-commute norm-triangle-ineq2)
  ultimately show ?thesis
    by (simp add: abs-le-iff)
qed
lemma norm-triangle-ineq4: norm (a - b) ≤ norm a + norm b
proof
  have norm (a + - b) ≤ norm a + norm (- b)
    by (rule norm-triangle-ineq)
  then show ?thesis by simp
qed

lemma norm-triangle-le-diff: norm x + norm y ≤ e =⇒ norm (x - y) ≤ e
  by (meson norm-triangle-ineq4 order-trans)

lemma norm-diff-ineq: norm a - norm b ≤ norm (a + b)
proof
  have norm a - norm (- b) ≤ norm (a - - b)
    by (rule norm-triangle-ineq2)
  then show ?thesis by simp
qed

lemma norm-triangle-sub: norm x ≤ norm y + norm (x - y)
  using norm-triangle-ineq[of y x - y] by (simp add: field-simps)

lemma norm-triangle-le: norm x + norm y ≤ e =⇒ norm (x + y) ≤ e
  by (rule norm-triangle-ineq [THEN order-trans])

lemma norm-triangle-lt: norm x + norm y < e =⇒ norm (x + y) < e
  by (rule norm-triangle-ineq [THEN le-less-trans])

lemma norm-add-leD: norm (a + b) ≤ c =⇒ norm b ≤ norm a + c
  by (metis ab-semigroup-add-class.add.commute add-commute diff-le-eq norm-diff-ineq order-trans)

lemma norm-diff-triangle-ineq: norm (((a + b) - (c + d))) ≤ norm (a - c) + norm (b - d)
proof
  have norm ((a + b) - (c + d)) = norm ((a - c) + (b - d))
    by (simp add: algebra-simps)
  also have ... ≤ norm (a - c) + norm (b - d)
    by (rule norm-triangle-ineq)
  finally show ?thesis .
qed

lemma norm-diff-triangle-le: norm (x - z) ≤ e1 + e2
if norm (x - y) ≤ e1 norm (y - z) ≤ e2
proof
  have norm (x - (y + z - y)) ≤ norm (x - y) + norm (y - z)
    using norm-diff-triangle-ineq that diff-diff-eq2 by presburger
  with that show ?thesis by simp
qed

lemma norm-diff-triangle-less: norm (x - z) < e1 + e2
if \( \|x - y\| < e1 \) \( \|y - z\| < e2 \)

proof –

have \( \|x - z\| \leq \|x - y\| + \|y - z\| \)

by (metis norm-diff-triangle-ineq add-diff-cancel-left' diff-diff-eq2)

with that show \( \text{thesis} \) by auto

qed

lemma norm-triangle-mono:
\( \|a\| \leq r \implies \|b\| \leq s \implies \|a + b\| \leq r + s \)

by (metis (mono-tags) add-mono-thms-linordered-semiring (1) norm-triangle-ineq order.trans)

lemma norm-sum: \( \|\sum f A\| \leq \sum_{i \in A} \|f i\| \)

for \( f: 'b \Rightarrow 'a \)

by (induct A rule: infinite-finite-induct) (auto intro: norm-triangle-mono)

lemma sum-norm-le: \( \|\sum f S\| \leq \sum g S \)

if \( \forall x. x \in S \implies \|f x\| \leq g x \)

for \( f: 'b \Rightarrow 'a \)

by (rule order-trans [OF norm-sum sum-mono]) (simp add: that)

lemma abs-norm-cancel [simp]: \( |\|a\|| = \|a\| \)

by (rule abs-of-nonneg [OF norm-ge-zero])

lemma sum-norm-bound:
\( \|\sum f S\| \leq \text{of-nat} \ (\text{card} S) \cdot K \)

if \( \forall x. x \in S \implies \|f x\| \leq K \)

for \( f: 'b \Rightarrow 'a \)

using sum-norm-le[OF that] sum-constant[symmetric]

by simp

lemma norm-add-less: \( \|x\| < r \implies \|y\| < s \implies \|x + y\| < r + s \)

by (rule order-le-less-trans [OF norm-triangle-ineq add-strict-mono])

end

lemma dist-scaleR [simp]: \( \text{dist} (x \cdot R a) (y \cdot R a) = |x - y| \cdot \|a\| \)

for \( a: \text{real-normed-vector} \)

by (metis dist-norm norm-scaleR scaleR-left.diff)

lemma norm-mult-less: \( \|x\| < r \implies \|y\| < s \implies \|x \cdot y\| < r \cdot s \)

for \( x y: 'a::\text{real-normed-algebra} \)

by (rule order-le-less-trans [OF norm-mult-ineq]) (simp add: mult-strict-mono')

lemma norm-of-real [simp]: \( \|\text{of-real} r:: 'a::\text{real-normed-algebra-1}\| = |r| \)

by (simp add: of-real-def)

lemma norm-numeral [simp]: \( \|\text{numeral w:: 'a::\text{real-normed-algebra-1}}\| = \text{numeral w} \)
by (subst of-real-numeral [symmetric], subst norm-of-real, simp)

lemma norm-neg-numeral [simp]: norm (− numeral w :: 'a::real-normed-algebra-1) = numeral w 
  by (subst of-real-neg-numeral [symmetric], subst norm-of-real, simp)

lemma norm-of-real-add1 [simp]: norm (of-real x + 1 :: 'a::real-normed-div-algebra) = |x + 1|
  by (metis norm-of-real of-real-1 of-real-add)

lemma norm-of-real-addn [simp]: norm (of-real x + numeral b :: 'a::real-normed-div-algebra) = |x + numeral b|
  by (metis norm-of-real of-real-add of-real-numeral)

lemma norm-of-int [simp]: norm (of-int z :: 'a::real-normed-algebra-1) = |of-int z|
  by (subst of-real-of-int-eq [symmetric], rule norm-of-real)

lemma nonzero-norm-inverse: a ≠ 0 ==> norm (inverse a) = inverse (norm a) 
  for a :: 'a::real-normed-div-algebra 
  by (metis inverse-unique norm-mult nonzero-norm-inverse)

lemma nonzero-norm-divide: b ≠ 0 ==> norm (a / b) = norm a / norm b 
  for a b :: 'a::real-normed-field 
  by (simp add: divide-inverse norm-mult nonzero-norm-inverse)

lemma norm-power-ineq: norm (x ^ n) ≤ norm x ^ n 
  for x :: 'a::real-normed-algebra-1 
  proof (induct n)
    case 0 

show norm (x ^ 0) ≤ norm x ^ 0 by simp
next
  case (Suc n)
  have norm (x * x ^ n) ≤ norm x * norm (x ^ n)
    by (rule norm-mult-ineq)
  also from Suc have ... ≤ norm x * norm x ^ n
    using norm-ge-zero by (rule mult-left-mono)
  finally show norm (x ^ Suc n) ≤ norm x ^ Suc n
    by simp
qed

lemma norm-power: norm (x ^ n) = norm x ^ n
  for x :: 'a::real-normed-div-algebra
  by (induct n) (simp-all add: norm-mult)

lemma norm-power-int: norm (power-int x n) = power-int (norm x) n
  for x :: 'a::real-normed-div-algebra
  by (cases n rule: int-cases4) (auto simp: norm-power power-int-minus norm-inverse)

lemma power-eq-imp-eq-norm:
  fixes w :: 'a::real-normed-div-algebra
  assumes eq: w ^ n = z ^ n and n > 0
  shows norm w = norm z
proof -
  have norm w ^ n = norm z ^ n
    by (metis (no-types) eq norm-power)
  then show ?thesis
    using assms by (force intro: power-eq-imp-eq-base)
qed

lemma power-eq-1-iff:
  fixes w :: 'a::real-normed-div-algebra
  shows w ^ n = 1 =⇒ norm w = 1 ∨ n = 0
  by (metis norm-one power-0-left power-eq-0-iff power-eq-imp-eq-norm power-one)

lemma norm-mult-numeral1 [simp]: norm (numeral w * a) = numeral w * norm a
  for a b :: 'a::real-normed-field_field
  by (simp add: norm-mult)

lemma norm-mult-numeral2 [simp]: norm (a * numeral w) = norm a * numeral w
  for a b :: 'a::real-normed-field_field
  by (simp add: norm-mult)

lemma norm-divide-numeral [simp]: norm (a / numeral w) = norm a / numeral w
  for a b :: 'a::real-normed-field_field
  by (simp add: norm-divide)
lemma norm-of-real-diff [simp]:
  norm (of-real b - of-real a :: 'a::real-normed-algebra-1) ≤ |b - a|
by (metis norm-of-real of-real-diff order-refl)

Despite a superficial resemblance, norm-eq-1 is not relevant.

lemma square-norm-one:
  fixes x :: 'a::real-normed-div-algebra
  assumes x^2 = 1
  shows norm x = 1
by (metis assms norm-minus-cancel norm-one power2-eq-1-iff)

lemma norm-less-p1:
  norm x < norm (of-real (norm x) + 1 :: 'a::real-normed-algebra-1)
proof -
  have norm x < norm (of-real (norm x + 1) :: 'a)
    by (simp add: of-real-def)
  then show ?thesis
    by simp
qed

lemma prod-norm:
  prod (λx. norm (f x)) A = norm (prod f A)
for f :: 'a⇒'b::{comm-semiring-1,real-normed-div-algebra}
by (induct A rule: infinite-finite-induct) (auto simp: norm-mult)

lemma norm-prod-le:
  norm (prod f A) ≤ (∏a∈A. norm (f a :: 'a::real-normed-algebra-1,comm-monoid-mult)))
proof (induct A rule: infinite-finite-induct)
  case empty
  then show ?case by simp
next
  case (insert a A)
  then have norm (prod f (insert a A)) ≤ norm (f a) * norm (prod f A)
    by (simp add: norm-mult-ineq)
  also have norm (prod f A) ≤ (∏a∈A. norm (f a))
    by (rule insert)
  finally show ?case
    by (simp add: insert mult-left-mono)
next
  case infinite
  then show ?case by simp
qed

lemma norm-prod-diff:
  fixes z w :: 'i ⇒ 'a::{real-normed-algebra-1, comm-monoid-mult}
  shows (∏i. i ∈ I ⇒ norm (z i) ≤ 1) ⇒ (∏i. i ∈ I ⇒ norm (w i) ≤ 1) ⇒
    norm ((((∏i∈I. z i) - (∏i∈I. w i)) ≤ (∑i∈I. norm (z i - w i))
proof (induction I rule: infinite-finite-induct)
  case empty
then show ?case by simp

next

  case (insert i I)
  note insert.hyps[simp]

  have norm ((∏ i∈insert i I. z i) - (∏ i∈insert i I. w i)) =
    norm((∏ i∈I. z i) * (z i - w i) + ((∏ i∈I. z i) - (∏ i∈I. w i)) * w i)
    (is - = norm (?t1 + ?t2))
    by (auto simp: field-simps)
  also have ... ≤ norm ?t1 + norm ?t2
    by (rule norm-triangle-ineq)
  also have norm ?t1 ≤ norm (∏ i∈I. z i) * norm (z i - w i)
    by (rule norm-mult-ineq)
  also have ... ≤ (∏ i∈I. norm (z i)) * norm(z i - w i)
    by (rule mult-right-mono) (auto intro: norm-prod-le)
  also have (∏ i∈I. norm (z i)) ≤ (∏ i∈I. 1)
    by (intro prod-mult-ineq) (auto intro: insert)
  also have norm ?t2 ≤ norm ((∏ i∈I. z i) - (∏ i∈I. w i)) * norm (w i)
    by (rule norm-mult-ineq)
  also have norm (w i) ≤ 1
    by (auto intro: insert)
  also have norm ((∏ i∈I. z i) - (∏ i∈I. w i)) ≤ (∑ i∈I. norm (z i - w i))
    using insert by auto
  finally show ?thesis .

next

  case infinite
  then show ?case by simp

qed

lemma norm-power-diff:
  fixes z w :: 'a::{real-normed-algebra-1, comm-monoid-mult}
  assumes norm z ≤ 1 norm w ≤ 1
  shows norm (z ^ m - w ^ m) ≤ m * norm (z - w)
proof -
  have norm (z ^ m - w ^ m) = norm (∏ i< m. z) - (∏ i< m. w)
    by simp
  also have ... ≤ (∑ i<m. norm (z - w))
    by (intro norm-prod-diff) (auto simp: assms)
  also have ... = m * norm (z - w)
    by simp
  finally show ?thesis .

qed

108.6 Metric spaces

class metric-space = uniformity-dist + open-uniformity +
  assumes dist-eq-0-iff [simp]: dist x y = 0 ↔ x = y
  and dist-triangle2: dist x y ≤ dist x z + dist y z
begin

lemma dist-self [simp]: dist x x = 0 by simp

lemma zero-le-dist [simp]: 0 ≤ dist x y using dist-triangle2 [of x x y] by simp

lemma zero-less-dist-iff: 0 < dist x y if and only if x ≠ y by (simp add: less-le)

lemma dist-not-less-zero [simp]: ¬ dist x y < 0 by (simp add: not-less)

lemma dist-le-zero-iff [simp]: dist x y ≤ 0 if and only if x = y by (simp add: le-less)

lemma dist-commute: dist x y = dist y x
proof (rule order-antisym)
  show dist x y ≤ dist y x using dist-triangle2 [of x y x] by simp
  show dist y x ≤ dist x y using dist-triangle2 [of y x y] by simp
qed

lemma dist-commute-lessI: dist y x < e if and only if dist x y < e by (simp add: dist-commute)

lemma dist-triangle: dist x z ≤ dist x y + dist y z using dist-triangle2 [of x z y] by (simp add: dist-commute)

lemma dist-triangle3: dist x y ≤ dist a x + dist a y using dist-triangle2 [of x a y] by (simp add: dist-commute)

lemma abs-dist-diff-le: |dist a b - dist b c| ≤ dist a c using dist-triangle3 [of b c a] dist-triangle2 [of a b c] by simp

lemma dist-pos-lt: x ≠ y if and only if 0 < dist x y by (simp add: zero-less-dist-iff)

lemma dist-nz: x ≠ y if and only if 0 < dist x y by (simp add: zero-less-dist-iff)

declare dist-nz [symmetric, simp]

lemma dist-triangle-le: dist x z + dist y z ≤ e if and only if dist x y ≤ e by (rule order-trans [OF dist-triangle2])

lemma dist-triangle-lt: dist x z + dist y z < e if and only if dist x y < e
by (rule le-less-trans [OF dist-triangle2])

lemma dist-triangle-less-add: dist x1 y < e1 ⇒ dist x2 y < e2 ⇒ dist x1 x2 < e1 + e2
  by (rule dist-triangle-lt [where z=y]) simp

lemma dist-triangle-half-l: dist x1 y < e / 2 ⇒ dist x2 y < e / 2 ⇒ dist x1 x2 < e
  by (rule dist-triangle-half-l) (simp-all add: dist-commute)

lemma dist-triangle-half-r: dist y x1 < e / 2 ⇒ dist y x2 < e / 2 ⇒ dist x1 x2 < e
  by (rule dist-triangle-half-l) (simp-all add: dist-commute)

lemma dist-triangle-third:
  assumes dist x1 x2 < e / 3 dist x2 x3 < e / 3 dist x3 x4 < e / 3
  shows dist x1 x4 < e
proof −
  have dist x1 x3 < e / 3 + e / 3
    by (metis assms(1) assms(2) dist-commute dist-triangle-less-add)
  then have dist x1 x4 < (e / 3 + e / 3) + e / 3
    by (metis assms(3) dist-commute dist-triangle-less-add)
  then show ?thesis
    by simp
qed

subclass uniform-space
proof
  fix E x
  assume eventually E uniformity
  then obtain e where E: 0 < e ∧ x y. dist x y < e ⇒ E (x, y)
    by (auto simp: eventually-uniformity-metric)
  then show E (x, x) ∀ F (x, y) in uniformity. E (y, x)
    by (auto simp: eventually-uniformity-metric dist-commute)
  show ∃ D. eventually D uniformity ∧ (∀ x y z. D (x, y) −→ D (y, z) −→ E (x, z))
    using E dist-triangle-half-l [where e=e] unfolding eventually-uniformity-metric
    by (intro exI[of - λ(x, y). dist x y < e / 2] exI[of - e/2] conjI)
      (auto simp: dist-commute)
qed

lemma open-dist: open S −→ (∀ x ∈ S. Ex e>0. ∀ y. dist x y < e −→ y ∈ S)
  by (simp add: dist-commute open-uniformity eventually-uniformity-metric)

lemma open-ball: open { y. dist x y < d}
unfolding open-dist
proof (intro ballI)
  fix y
assume ∀y. dist x y < d
then show ∃e>0. ∀z. dist z y < e → z ∈ {y. dist x y < d}
  by (auto intro!: exI[of - d - dist x y] simp: field-simps dist-triangle-lt)
qed

subclass first-countable-topology
proof
  fix x
  show ∃A::nat ⇒ ′a set. (∀i. x ∈ A i ∧ open (A i)) ∧ (∀S. open S ∧ x ∈ S → (∃i. A i ⊆ S))
    proof (safe intro!: exI[of λn. {y. dist x y < inverse (Suc n)}])
      fix S
      assume open S x ∈ S
      then obtain e where e: 0 < e and {y. dist x y < e} ⊆ S by (auto simp: open-dist subset-eq dist-commute)
      moreover from e obtain i where inverse (Suc i) < e by (auto dest!: reals-Archimedean)
      then have {y. dist x y < inverse (Suc i)} ⊆ {y. dist x y < e} by auto
      ultimately show ∃i. {y. dist x y < inverse (Suc i)} ⊆ S by blast
    qed (auto intro: open-ball)
  qed
end

instance metric-space ⊆ t2-space
proof
  fix x y z :: ′a::metric-space
  assume xy: x ≠ y
  let ?U = {y'. dist x y' < dist x y / 2}
  let ?V = {x'. dist y x' < dist x y / 2}
  have ∀d. d x z ≤ d x y + d y z = d z y = ∨ (d x y * 2 < d x z ∧ d z y * 2 < d x z)
    for d :: ′a ⇒ ′a ⇒ real and x y z :: ′a
      by arith
    using open-ball[of - dist x y / 2] by auto
  then show ∃U V. open U ∧ open V ∧ x ∈ U ∧ y ∈ V ∧ U ∩ V = {} by blast
  qed

Every normed vector space is a metric space.

instance real-normed-vector < metric-space
proof
  fix x y z :: ′a
  show dist x y = 0 ↔ x = y
by (simp add: dist-norm)
show \( \text{dist } x \ y \leq \text{dist } x \ z + \text{dist } y \ z \) 
using \text{norm-triangle-ineq} \ [\text{of } x - z \ y - z] \ by \ (\text{simp add: dist-norm)}
qed

108.7 Class instances for real numbers

instantiation real :: real-normed-field
begin

definition dist-real-def: \( \text{dist } x \ y = \lvert x - y \rvert \)
definition uniformity-real-def [code del]:
  \( (\text{uniformity} :: (\text{real} \times \text{real}) \text{ filter}) = (\text{INF } e \in \{0 <\ldots\}. \text{principal } \{(x, y). \text{dist } x \ y < e\}) \)
definition open-real-def [code del]:
  \( \text{open } (U :: \text{real set}) \rightleftharpoons (\forall x \in U. \text{ eventually } (\lambda(x', y). x' = x \longrightarrow y \in U) \text{ uniformity}) \)
definition real-norm-def [simp]: \( \text{norm } r = \lvert r \rvert \)
instance 
  by \ (\text{intro-classes } (\text{auto simp: abs-mult open-def dist-real-def sgn-real-def uniformity-real-def})
end

declare uniformity-Abort[\text{where } 'a=\text{real}, \text{code}]

lemma dist-of-real [simp]: \( \text{dist } (\text{of-real } x :: 'a) (\text{of-real } y) = \text{dist } x \ y \)
for \( a :: 'a::\text{real-normed-div-algebra} \)
by (metis \text{dist-norm norm-of-real of-real-diff real-norm-def})
declare [[\text{code abort: open :: real set } \Rightarrow \text{bool}]]
instance real :: linorder-topology
proof
  show \( \text{open :: real set } \Rightarrow \text{bool} \rightleftharpoons \text{generate-topology (range lessThan } \cup \text{ range greaterThan)} \)
proof (rule ext, safe)
  fix \( S :: \text{real set} \)
  assume \( \text{open } S \)
  then obtain \( f \) where \( \forall x \in S. 0 < f x \land (\forall y. \text{dist } y \ x < f x \longrightarrow y \in S) \)
  unfolding \text{open-dist behoice-iff} ..
  then have \( \ast : (\bigcup x \in S. \{x - f x \ldots\} \cap \{-< x + f x\}) = S \) (is \( ?S = S \))
  by (fastforce simp: dist-real-def)
moreover have \text{generate-topology (range lessThan } \cup \text{ range greaterThan)} \ ?S
ultimately show generate-topology (range lessThan ∪ range greaterThan) S by simp

next

fix S :: real set
assume generate-topology (range lessThan ∪ range greaterThan) S
moreover have ∀a::real. open {..<a}
    unfolding open-dist dist-real-def
proof clarify
  fix x a :: real
  assume x < a
  then have 0 < a - x ∧ (∀y. |y - x| < a - x → y ∈ {..<a}) by auto
  then show ∃e>0. ∀y. |y - x| < e → y ∈ {..<a} ..
qed
moreover have ∀a::real. open {a <..}
    unfolding open-dist dist-real-def
proof clarify
  fix x a :: real
  assume a < x
  then have 0 < x - a ∧ (∀y. |y - x| < x - a → y ∈ {a<..}) by auto
  then show ∃e>0. ∀y. |y - x| < e → y ∈ {a<..} ..
qed
ultimately show open S
by induct auto
qed

instance real :: linear-continuum-topology ..

lemmas open-real-greaterThan = open-greaterThan[where 'a=real]
lemmas open-real-lessThan = open-lessThan[where 'a=real]
lemmas open-real-greaterThanLessThan = open-greaterThanLessThan[where 'a=real]
lemmas closed-real-atMost = closed-atMost[where 'a=real]
lemmas closed-real-atLeast = closed-atLeast[where 'a=real]
lemmas closed-real-atLeastAtMost = closed-atLeastAtMost[where 'a=real]

instance real :: ordered-real-vector
  by standard (auto intro: mult-left-mono mult-right-mono)

108.8 Extra type constraints

Only allow open in class topological-space.

setup (Sign.add-const-constraint (const-name 'open', SOME typ ('a::topological-space set ⇒ bool')));

Only allow uniformity in class uniform-space.

setup (Sign.add-const-constraint (const-name uniformity, SOME typ ('a::uniformity × 'a filter')));

Only allow dist in class metric-space.
setup (Sign.add-const-constraint
  (const-name (dist), SOME typ (\'a::metric-space ⇒ \'a ⇒ real))
)

Only allow norm in class real-normed-vector.

setup (Sign.add-const-constraint
  (const-name (norm), SOME typ (\'a::real-normed-vector ⇒ real))
)

108.9 Sign function

lemma norm-sgn: norm (sgn x) = (if x = 0 then 0 else 1)
  for x :: 'a::real-normed-vector
  by (simp add: sgn-div-norm)

lemma sgn-zero [simp]: sgn (0::'a::real-normed-vector) = 0
  by (simp add: sgn-div-norm)

lemma sgn-zero-iff: sgn x = 0 ↔ x = 0
  for x :: 'a::real-normed-vector
  by (simp add: sgn-div-norm)

lemma sgn-minus: sgn (- x) = - sgn x
  for x :: 'a::real-normed-vector
  by (simp add: sgn-div-norm)

lemma sgn-scaleR: sgn (scaleR r x) = scaleR (sgn r) (sgn x)
  for x :: 'a::real-normed-vector
  by (simp add: sgn-div-norm ac-simps)

lemma sgn-one [simp]: sgn (1::'a::real-normed-algebra-1) = 1
  by (simp add: sgn-div-norm)

lemma sgn-of-real: sgn (of-real r :: 'a::real-normed-algebra-1) = of-real (sgn r)
  unfolding of-real-def by (simp only: sgn-scaleR sgn-one)

lemma sgn-mult: sgn (x * y) = sgn x * sgn y
  for x y :: 'a::real-normed-div-algebra
  by (simp add: sgn-div-norm norm-mult)

hide-fact (open) sgn-mult

lemma real-sgn-eq: sgn x = x / |x|
  for x :: real
  by (simp add: sgn-div-norm divide-inverse)

lemma zero-le-sgn-iff [simp]: 0 ≤ sgn x ↔ 0 ≤ x
  for x :: real
  by (cases 0::real x rule: linorder-cases simp-all)

lemma sgn-le-0-iff [simp]: sgn x ≤ 0 ↔ x ≤ 0
THEORY “Real-Vector-Spaces”

for x :: real by (cases 0 :: real rule: linorder-cases) simp-all

lemma norm-conv-dist: norm x = dist x 0
  unfolding dist-norm by simp

declare norm-conv-dist [symmetric, simp]

lemma dist-0-norm [simp]: dist 0 x = norm x for x :: 'a :: real-normed_vector
  by (simp add: dist-norm)

lemma dist-diff [simp]: dist a (a - b) = norm b dist (a - b) a = norm b
  by (simp-all add: dist-norm)

lemma dist-of-int: dist (of-int m) (of-int n :: 'a :: real-normed-algebra-1) = of-int |m - n|
  proof -
    have dist (of-int m) (of-int n :: 'a) = dist (of-int m :: 'a) (of-int m - (of-int (m - n)))
      by simp
    also have ... = of-int |m - n| by (subst dist-diff, subst norm-of-int) simp
    finally show ?thesis .
  qed

lemma dist-of-nat: dist (of-nat m) (of-nat n :: 'a :: real-normed-algebra-1) = of-int |int m - int n|
  by (subst (1 2) of-int-of-nat-eq [symmetric]) (rule dist-of-int)

108.10 Bounded Linear and Bilinear Operators

lemma linearI: linear f
  if \( \forall b_1 b_2. f (b_1 + b_2) = f b_1 + f b_2 \)
  \( \forall r. f (r * R b) = r * R f b \)
  using that
  by unfold-locales (auto simp: algebra-simps)

lemma linear-iff:
  linear f \( \iff \forall x y. f (x + y) = f x + f y \) \( \land \forall c. f (c * R x) = c * R f x \)
  (is linear f \( \iff \) ?rhs)
  proof
    assume linear f
    then interpret f : linear f .
    show ?rhs by (simp add: f.add f.scale)
  next
    assume ?rhs
    then show linear f by (intro linearI) auto
  qed
lemma linear-of-real [simp]: linear of-real
  by (simp add: linear-iff scaleR-conv-of-real)

lemmas linear-scaleR-left = linear-scale-left
lemmas linear-imp-scaleR = linear-imp-scale

corollary real-linearD:
fixes f :: real ⇒ real
assumes linear f obtains c where f = (∗) c
by (rule linear-imp-scaleR [OF assms]) (force simp: scaleR-conv-of-real)

lemma linear-times-of-real: linear (λx. a ∗ of-real x)
by (auto intro!: linearI simp: distrib-left)
  (metis mult-scaleR-right scaleR-conv-of-real)

locale bounded-linear = linear f for f :: 'a::real-normed-vector ⇒ 'b::real-normed-vector
+ assumes bounded: ∃ K. ∀ x. norm (f x) ≤ norm x ∗ K
begin

lemma pos-bounded: ∃ K>0. ∀ x. norm (f x) ≤ norm x ∗ K
proof –
obtain K where K: ∀x. norm (f x) ≤ norm x ∗ K
  using bounded by blast
show ?thesis
proof (intro ezI impI conjI allI)
  show 0 < max 1 K
    by (rule order-less-le-trans [OF zero-less-one max.cobounded1])
next
  fix x
  have norm (f x) ≤ norm x ∗ K using K .
  also have . . . ≤ norm x ∗ max 1 K
    by (rule mult-left-mono [OF max.cobounded2 norm-ge-zero])
  finally show norm (f x) ≤ norm x ∗ max 1 K .
qed

lemma nonneg-bounded: ∃ K≥0. ∀ x. norm (f x) ≤ norm x ∗ K
using pos-bounded by (auto intro: order-less-imp-le)

lemma linear: linear f
by (fact local.linear-axioms)

end

lemma bounded-linear-intro:
assumes ∃x y. f (x + y) = f x + f y
and ∃r x. f (scaleR r x) = scaleR r (f x)
and ∃x. norm (f x) ≤ norm x ∗ K
shows bounded-linear f
by standard (blast intro: assms)+

locale bounded-bilinear =
fixes prod :: 'a::real-normed-vector ⇒ 'b::real-normed-vector ⇒ 'c::real-normed-vector
(infixl *** 70)
assumes add-left: prod (a + a') b = prod a b + prod a' b
and add-right: prod a (b + b') = prod a b + prod a b'
and scaleR-left: prod (scaleR r a) b = scaleR r (prod a b)
and scaleR-right: prod a (scaleR r b) = scaleR r (prod a b)
and bounded: ∃ K. ∀ a b. norm (prod a b) ≤ norm a * norm b * K

begin

lemma pos-bounded: ∃ K>0. ∀ a b. norm (a *** b) ≤ norm a * norm b * K
proof –
  obtain K where ∃ a b. norm (a *** b) ≤ norm a * norm b * K
  using bounded by blast
  then have norm (a *** b) ≤ norm a * norm b * (max 1 K) for a b
  by (rule order.trans) (simp add: mult-left-mono)
  then show ?thesis
  by force
qed

lemma nonneg-bounded: ∃ K≥0. ∀ a b. norm (a *** b) ≤ norm a * norm b * K
using pos-bounded by (auto intro: order-less-imp-le)

lemma additive-right: additive (λb. prod a b)
by (rule additive.intro, rule add-right)

lemma additive-left: additive (λa. prod a b)
by (rule additive.intro, rule add-left)

lemma zero-left: prod 0 b = 0
by (rule additive.zero [OF additive-left!])

lemma zero-right: prod a 0 = 0
by (rule additive.zero [OF additive-right!])

lemma minus-left: prod (- a) b = - prod a b
by (rule additive.minus [OF additive-left!])

lemma minus-right: prod a (- b) = - prod a b
by (rule additive.minus [OF additive-right!])

lemma diff-left: prod (a - a') b = prod a b - prod a' b
by (rule additive.diff [OF additive-left!])

lemma diff-right: prod a (b - b') = prod a b - prod a b'
by (rule additive.diff [OF additive-right!])
lemma sum-left: \( \text{prod } (\text{sum } g \ S) \ x = \text{sum } ((\lambda i. \text{prod } (g \ i)) \ x) \ S \)
by (rule additive.sum[OF additive-left])

lemma sum-right: \( \text{prod } x (\text{sum } g \ S) = \text{sum } ((\lambda i. (\text{prod } x (g \ i))) \ S) \)
by (rule additive.sum[OF additive-right])

lemma bounded-linear-left: \( \text{bounded-linear } (\lambda a. \ a ** b) \)
proof
  obtain \( K \) where \( \forall a. b. \ \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K \)
  using pos-bounded by blast
  then show \(?\)thesis
  by (rule-tac \( K = \text{norm } b * K \) in bounded-linear-intro) (auto simp: algebra-simps)
qed

lemma bounded-linear-right: \( \text{bounded-linear } (\lambda b. \ a ** b) \)
proof
  obtain \( K \) where \( \forall a. b. \ \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K \)
  using pos-bounded by blast
  then show \(?\)thesis
  by (rule-tac \( K = \text{norm } a * K \) in bounded-linear-intro) (auto simp: algebra-simps)
qed

lemma prod-diff-prod: \( (x ** y - a ** b) = (x - a) ** (y - b) + (x - a) ** b + a ** (y - b) \)
by (simp add: diff-left diff-right)

lemma flip: \( \text{bounded-bilinear } (\lambda x y. y ** x) \)
proof
  show \( \exists K. \ \forall a. b. \ \text{norm } (b ** a) \leq \text{norm } a * \text{norm } b * K \)
  by (metis bounded mult.commute)
qed (simp-all add: add-right add-left scaleR-right scaleR-left)

lemma comp1:
  assumes \( \text{bounded-linear } g \)
  shows \( \text{bounded-bilinear } (\lambda x. (**) (g x)) \)
proof
  unfold-locales
  interpret \( g: \text{bounded-linear } g \) by fact
  show \( \forall a. a' b. g (a + a') ** b = g a ** b + g a' ** b \)
  by (auto simp: g.add add-left add-right g.scaleR scaleR-right scaleR-left)
  from g.nonneg bounded nonneg-bounded obtain \( K L \)
  where \( \forall m. 0 \leq K \ 0 \leq L \)
  and \( K: \forall x. \ \text{norm } (g \ x) \leq \text{norm } x * K \)
and \( L \): \( \forall a \ b. \ \text{norm} \ (a ** b) \leq \text{norm} \ a * \text{norm} \ b * L \)

by auto

have \( \text{norm} \ (g a ** b) \leq \text{norm} \ a * K * \text{norm} \ b * L \) for \( a \ b \)


then show \( \exists K. \ \forall a \ b. \ \text{norm} \ (g a ** b) \leq \text{norm} \ a * \text{norm} \ b * K \)

by (auto intro: exI[where \( x=K * L \)] simp: ac-simps)

qed

lemma \text{comp}: bounded-linear \( f \implies \) bounded-linear \( g \implies \) bounded-bilinear \( (\lambda x \ y. \ f x *^\bullet g y) \)

by (rule bounded-bilinear.flip[OF bounded-bilinear.comp1[OF bounded-bilinear.flip[OF comp1]]])

end

lemma bounded-linear-ident\[simp\]: bounded-linear \( \lambda x. \ x \)

by standard (auto intro: exI[of - 1])

lemma bounded-linear-zero\[simp\]: bounded-linear \( \lambda x. \ 0 \)

by standard (auto intro: exI[of - 1])

lemma bounded-linear-add:

assumes bounded-linear \( f \)

and bounded-linear \( g \)

shows bounded-linear \( \lambda x. \ f x + g x \)

proof –

interpret \( f \): bounded-linear \( f \) by fact

interpret \( g \): bounded-linear \( g \) by fact

show \( \exists \ ? \) thesis

proof

from \( f \).bounded obtain \( Kf \) where \( Kf: \ \text{norm} \ (f x) \leq \text{norm} \ x * Kf \) for \( x \)

by blast

from \( g \).bounded obtain \( Kg \) where \( Kg: \ \text{norm} \ (g x) \leq \text{norm} \ x * Kg \) for \( x \)

by blast

show \( \exists K. \ \forall x. \ \text{norm} \ (f x + g x) \leq \text{norm} \ x * K \)

using add-mono[OF \( Kf Kg \)]

by (intro exI[of - \( Kf + Kg \)]) (auto simp: field-simps intro: norm-triangle-ineq order-trans)

qed (simp-all add: f.add g.add f.scaleR g.scaleR scaleR-right-distrib)

qed

lemma bounded-linear-minus:

assumes bounded-linear \( f \)

shows bounded-linear \( \lambda x. \ - f x \)

proof –

interpret \( f \): bounded-linear \( f \) by fact

show \( \exists \ ? \) thesis

by unfold-locales (simp-all add: f.add f.scaleR f.bounded)

qed
lemma bounded-linear-sub: bounded-linear f \implies bounded-linear g \implies bounded-linear \( \lambda x. f x - g x \)

using bounded-linear-add[of f \lambda x. - g x] bounded-linear-minus[of g]

by (auto simp: algebra-simps)

lemma bounded-linear-sum:

fixes f :: 'i \Rightarrow 'a::real-normed-vector
shows \((\forall i. i \in I \implies bounded-linear (f i)) \implies bounded-linear (\lambda x. \sum_{i\in I} f i)\)

by (induct I rule: infinite-finite-induct) (auto intro!: bounded-linear-add)

lemma bounded-linear-compose:

assumes bounded-linear f and bounded-linear g

shows bounded-linear \( \lambda x. f (g x) \)

proof -

interpret f: bounded-linear f by fact

interpret g: bounded-linear g by fact

show ?thesis

proof (unfold-locales)

show f (g (x + y)) = f (g x) + f (g y) for x y

by (simp only: f.add g.add)

show f (g (scaleR r x)) = scaleR r (f (g x)) for r x

by (simp only: f.scaleR g.scaleR)

from f.pos-bounded obtain Kf where f: \( \forall x. \text{norm} (f x) \leq \text{norm} x \ast Kf \) and Kf: \( 0 < Kf \)

by blast

from g.pos-bounded obtain Kg where g: \( \forall x. \text{norm} (g x) \leq \text{norm} x \ast Kg \)

by blast

show \( \exists K. \forall x. \text{norm} (f (g x)) \leq \text{norm} x \ast K \)

proof (intro exI allI)

fix x

have \( \text{norm} (f (g x)) \leq \text{norm} (g x) \ast Kf \)

using f .

also have \( \ldots \leq (\text{norm} x \ast Kg) \ast Kf \)

using g Kf [THEN order-less-imp-le] by (rule mult-right-mono)

also have \( \text{norm} x \ast Kg \leq \text{norm} x \ast (Kg \ast Kf) \)

by (rule mult.assoc)

finally show \( \text{norm} (f (g x)) \leq \text{norm} x \ast (Kg \ast Kf) \).

qed

qed

lemma bounded-bilinear-mult: bounded-bilinear \((\ast) :: 'a \Rightarrow 'a \Rightarrow 'a::real-normed-algebra)\)

proof (rule bounded-bilinear.intro)

show \( \exists K. \forall a b::'a. \text{norm} (a \ast b) \leq \text{norm} a \ast \text{norm} b \ast K \)

by (rule_tac x=1 in exI) (simp add: norm-mult-ineq)

qed (auto simp: algebra-simps)
lemma bounded-linear-mult-left: bounded-linear (λx::'a::real-normed-algebra. x * y)
  using bounded-bilinear-mult
  by (rule bounded-bilinear.bounded-linear-left)
lemma bounded-linear-mult-right: bounded-linear (λy::'a::real-normed-algebra. x * y)
  using bounded-bilinear-mult
  by (rule bounded-bilinear.bounded-linear-right)
lemmas bounded-linear-mult-const =
  bounded-linear-mult-left THEN bounded-linear-compose
lemmas bounded-linear-mult-mut =
  bounded-linear-mult-right THEN bounded-linear-compose
lemma bounded-linear-divide: bounded-linear (λx. x / y)
  for y :: 'a::real-normed-field
  unfolding divide-inverse by (rule bounded-linear-mult-left)
lemma bounded-linear-scaleR: bounded-bilinear scaleR
  proof (rule bounded-bilinear.intro)
    show ∃ K. ∀ a b. norm (a *R b) ≤ norm a * norm b * K
      using less-eq-real-def by auto
  qed (auto simp: algebra-simps)
lemma bounded-linear-scaleR-left: bounded-linear (λr. scaleR r x)
  using bounded-bilinear-scaleR
  by (rule bounded-bilinear.bounded-linear-left)
lemma bounded-linear-scaleR-right: bounded-linear (λx. scaleR r x)
  using bounded-bilinear-scaleR
  by (rule bounded-bilinear.bounded-linear-right)
lemmas bounded-linear-scaleR-const =
  bounded-linear-scaleR-left[THEN bounded-linear-compose]
lemmas bounded-linear-scaleR-scaleR =
  bounded-linear-scaleR-right[THEN bounded-linear-compose]
lemma bounded-linear-of-real: bounded-linear (λr. of-real r)
  unfolding of-real-def by (rule bounded-linear-scaleR-left)
lemma real-bounded-linear: bounded-linear f ↔ (∃ c::real. f = (λx. x * c))
  for f :: real ⇒ real
  proof −
    { fix x
assume bounded-linear f
then interpret bounded-linear f.
from scaleR[of x 1] have f x = x * f 1
  by simp }
then show ?thesis
  by (auto intro: exI[of - f 1] bounded-linear-mult-left)
qed

instance real-normed-algebra-1 ⊆ perfect-space
proof
  fix x::'a
  have ∃e. 0 < e ⇒ ∃y. norm (y - x) < e ∧ y ≠ x
    by (rule_tac x = x + of_real (e/2) in exI) auto
  then show ¬open {x}
    by (clarsimp simp: open-dist dist-norm)
qed

108.11 Filters and Limits on Metric Space

lemma (in metric-space) nhds-metric: nhds x = (INF e∈{0 <..}. principal {y. dist y x < e})
unfolding nhds-def
proof (safe intro!: INF-eq)
  fix S
  assume open S x ∈ S
  then obtain e where {y. dist y x < e} ⊆ S 0 < e
    by (auto simp: open-dist subset-eq)
  then show ∃e∈{0<..}. principal {y. dist y x < e} ≤ principal S
    by auto
qed (auto intro!: exI[of - {y. dist x y < e} for e] open-ball simp: dist-commute)

lemma tendsto-iff-uniformity:
  — More general analogous of tendsto-iff below. Applies to all uniform spaces, not just metric ones.
  fixes l :: ('b :: uniform-space)
  shows ⟨(f ----→ l) F ⟷ (∀E. eventually E uniformity ⇒ (∀F x in F. E (f x, l)))⟩
proof (intro iffI allI impI)
  fix E :: ('b × 'b) ⇒ bool
  assume ⟨(f ----→ l) F ⟷ eventually E uniformity⟩
  from ⟨eventually E uniformity⟩
  have ⟨eventually (λ(x, y). E (y, x)) uniformity⟩
    by (simp add: uniformity-sym)
  then have ∀F y x in uniformity, y = l ⇒ E (x, y)
    using eventually-mono by fastforce
  with ⟨(f ----→ l) F ⟷ eventually (λx. E (x, l)) (filtermap f F)⟩
    by (simp add: filterlim-def le-filter-def eventually-nhds-uniformity)
then show $\forall_F \in F. E(f, l)$
  by (simp add: eventually-filtermap)
next
  assume assms: $\forall E. \text{eventually } E \text{ uniformity } \rightarrow (\forall_F \in F. E(f, l))$
  have (eventually $P$ (filtermap $f$ $F$)): If $\forall_F (x, y) \in \text{uniformity}. x = l \rightarrow P y$
    for $P$
  proof
    from that have $\forall_F (y, x) \in \text{uniformity}. x = l \rightarrow P y$
    using uniformity-sym [where $E = \lambda(x, y). x = l \rightarrow P y$] by auto
    have (eventually $P$ (filtermap $f$ $F$))
    if $\forall_F (y, x) \in \text{uniformity}. x = l \rightarrow P y$
  qed
  then show $\forall f \rightarrow l$ $F$
    by (simp add: filterlim-def le-filter-def eventually-nhds-uniformity)
qed

lemma (in metric-space) tendsto_iff: ($f \rightarrow l$) $F$ $\iff$ ($\forall e > 0. \text{eventually } (\lambda x. \text{dist}(f x) l < e) F$)
unfolding nhds_metric filterlim_INF filterlim_principal by auto

lemma (in metric-space) tendsto_dist_iff: ((($f \rightarrow l$) $F$) $\iff$ (((($\lambda x. \text{dist}(f x) l < e) F$) $\iff$ ($f \rightarrow l$) $F$)
unfolding tendsto_iff by simp

lemma (in metric-space) tendstoI [intro!]:
  ($\forall e > 0. \text{eventually } (\lambda x. \text{dist}(f x) l < e) F$) $\implies$ ($f \rightarrow l$) $F$
by (auto simp: tendsto_iff)

lemma (in metric-space) tendstoD: ($f \rightarrow l$) $F$ $\implies$ ($0 < e \implies \text{eventually } (\lambda x. \text{dist}(f x) l < e) F$
by (auto simp: tendsto_iff)

lemma (in metric-space) eventually_nhds_metric:
  eventually $P$ (nhds $a$) $\iff$ ($\exists d > 0. \forall x < d \rightarrow P x$)
unfolding nhds_metric by (subst eventually-INF-base)
  (auto simp: eventually_principal Bex-def subset_eq intro: exI[of - min $a$ $b$ for $a$ $b$])

lemma eventually_at: eventually $P$ (at $a$ within $S$) $\iff$ ($\exists d > 0. \forall x \in S. x \neq a$ \land \text{dist}(x, a) < d \rightarrow P x$)
  for $a :: 'a :: metric-space$
by (auto simp: eventually_at_filter eventually_nhds_metric)

lemma frequently_at: frequently $P$ (at $a$ within $S$) $\iff$ ($\forall d > 0. \exists x \in S. x \neq a$ \land \text{dist}(x, a) < d \land P x$)
  for $a :: 'a :: metric-space$
unfolding frequently-def eventually-at by auto

lemma eventually-at-le: \( \text{eventually } P \) (at a within S) \( \iff \) \( (\exists d > 0. \forall x \in S. x \neq a \land \text{dist } x \text{ a } \leq d \implies P x) \)

  for a :: 'a::metric-space
  unfolding eventually-at-filter eventually-nhds-metric
  apply safe
  apply (rule_tac x=d / 2 in exI, auto)
  done

lemma eventually-at-left-real: \( a > (b :: \text{real}) \implies \text{eventually } (\lambda x. x \in \{b <..< a\}) \)

  (at-left a)

  by (subst eventually-at, rule exI[of - a - b]) (force simp: dist-real-def)

lemma eventually-at-right-real: \( a < (b :: \text{real}) \implies \text{eventually } (\lambda x. x \in \{a <..< b\}) \)

  (at-right a)

  by (subst eventually-at, rule exI[of - b - a]) (force simp: dist-real-def)

lemma metric-tendsto-imp-tendsto:

  fixes a :: 'a :: metric-space and b :: 'b :: metric-space
  assumes f : (f ---+ a) F
  and le : \( \text{eventually } (\lambda x. \text{dist } (g x) b \leq \text{dist } (f x) a) \) F
  shows (g ---+ b) F

proof (rule tendstoI)

  fix e :: real
  assume 0 < e

  with f have \( \text{eventually } (\lambda x. \text{dist } (f x) a < e) \) F by (rule tendstoD)

  with le show \( \text{eventually } (\lambda x. \text{dist } (g x) b < e) \) F

    using le-less-trans by (rule eventually-elim2)

qed

lemma filterlim-real-sequentially: \( LIM x \text{ sequentially. real } x :> \text{at-top} \)

proof (clarisimp simp: filterlim-at-top)

  fix Z

  show \( \forall F x \text{ in sequentially. } Z \leq \text{real } x \)

    by (meson eventually-sequentiallyI nat-ceiling-le-eq)

qed

lemma filterlim-nat-sequentially: filterlim nat sequentially at-top

proof 

  have \( \forall F x \text{ in at-top. } Z \leq \text{nat } x \text{ for } Z \)

    by (auto intro!: eventually-at-top-linorderI[where c=int Z])

  then show \?thesis

    unfolding filterlim-at-top ..

qed

lemma filterlim-floor-sequentially: filterlim floor at-top at-top

proof 


have \( \forall x \text{ in at-top}, \ Z \leq |x| \) for \( Z \)

by (auto simp: le-floor-iff intro: eventually-at-top-linorderI[of \( c=\text{of-int} \ Z \)])

then show \(?thesis\)

unfolding filterlim-at-top ..

qed

lemma filterlim-sequentially-iff-filterlim-real:

\( \text{filterlim } f \text{ sequentially } F \iff \text{filterlim } (\lambda x. \text{ real } (f x)) \text{ at-top } F \) (is \( ?lhs = ?rhs \))

proof

assume \( ?lhs \) then show \( ?rhs \)

using filterlim-compose filterlim-real-sequentially by blast

next

assume \( R: \ ?rhs \)

show \( ?lhs \)

proof

−

have \( \text{filterlim } (\lambda x. \text{ nat } (\text{floor } (\text{real } (f x)))) \text{ sequentially } F \)

by (intro filterlim-compose[of filterlim-nat-sequentially]

filterlim-compose[of filterlim-floor-sequentially] \( R \))

then show \( ?thesis \) by simp

qed

qed

108.11.1 Limits of Sequences

lemma lim-sequentially: \( X \rightarrow L \leftrightarrow (\forall r>0. \ \exists \ no. \ \forall n\geq no. \ dist (X n) \ L < r) \)

for \( L :: 'a::\text{metric-space} \)

unfolding tendsto-iff eventually-sequentially ..

lemmas LIMSEQ-def = lim-sequentially

lemma LIMSEQ-iff-nz: \( X \rightarrow L \leftrightarrow (\forall r>0. \ \exists no>0. \ \forall n\geq no. \ dist (X n) \ L < r) \)

for \( L :: 'a::\text{metric-space} \)

unfolding lim-sequentially by (metis Suc-leD zero_less_Suc)

lemma metric-LIMSEQ-I: \( (\forall r. \ 0 < r \Rightarrow \exists no. \ \forall n\geq no. \ dist (X n) \ L < r) \Rightarrow \)

\( X \rightarrow L \)

for \( L :: 'a::\text{metric-space} \)

by (simp add: lim-sequentially)

lemma metric-LIMSEQ-D: \( X \rightarrow L \Rightarrow 0 < r \Rightarrow \exists no. \ \forall n\geq no. \ dist (X n) \ L < r \)

for \( L :: 'a::\text{metric-space} \)

by (simp add: lim-sequentially)

lemma LIMSEQ-norm-0:

assumes \( \forall n::\text{nat}. \ \text{norm } (f n) < 1 / \text{ real } (\text{Suc } n) \)

shows \( f \rightarrow 0 \)
proof (rule metric-LIMSEQ-I)
  fix ε :: real
  assume ε > 0
  then obtain N::nat where ε > inverse N N > 0
    by (metis neq0-conv real-arch-inverse)
  then have norm (f n) < ε if n ≥ N for n
  proof —
    have 1 / (Suc n) ≤ 1 / N
      using 0 < N; inverse-of-nat-le le-SucI that by blast
    also have ... < ε
      by (metis (no-types) inverse (real N) < ε)
    finally show ?thesis
  qed
  then show ∃ no. ∀ n≥no. dist (f n) 0 < ε
  by auto
  qed

108.11.2 Limits of Functions

lemma LIM-def: f − a → L ←→ (∀ r > 0. ∃ s > 0. ∀ x. x ≠ a ∧ dist x a < s —
dist (f x) L < r)
  for a :: 'a::metric-space and L :: 'b::metric-space
unfolding tendsto-iff eventually-at by simp

lemma metric-LIM-I:
  (∀r. 0 < r —> ∃ s>0. ∀ x. dist x a < s —> dist (f x) L < r) —> f
−a→ L
  for a :: 'a::metric-space and L :: 'b::metric-space
by (simp add: LIM-def)

lemma metric-LIM-D: f −a→ L —> 0 < r —> ∃s>0. ∀ x. x ≠ a ∧ dist x a < s —
dist (f x) L < r
  for a :: 'a::metric-space and L :: 'b::metric-space
by (simp add: LIM-def)

lemma metric-LIM-imp-LIM:
  fixes l :: 'a::metric-space
  and m :: 'b::metric-space
  assumes f: f −a→ l
  and le: ∃x. x ≠ a —> dist (g x) m ≤ dist (f x) l
  shows g −a→ m
  by (rule metric-tendsto-imp-tendsto [OF f]) (auto simp: eventually-at-topological le)

lemma metric-LIM-equal2:
  fixes a :: 'a::metric-space
  assumes g −a→ l 0 < R
  and ∃x. x ≠ a —> dist x a < R —> f x = g x
shows \( f - \mathbb{R} \rightarrow l \)

proof –
  have \( \bigwedge S [\text{open } S ; l \in S ; \forall x \in S] \Rightarrow \forall x \in S, f x \in S \)
  apply (simp add: eventually-at)
  by (metis assms(2) assms(3) dual-order.strict-trans linorder-neqE-linordered-idom)
  then show \( \exists \theta \)
    using assms by (simp add: tendsto-def)
qed

lemma metric-LIM-compose2:
  fixes a :: 'a::metric-space
  assumes f: \( f - \mathbb{R} \rightarrow b \)
  and g: \( g - b \rightarrow c \)
  and inj: \( \exists d>0. \forall x. x \neq a \land \text{dist } x a < d \rightarrow f x \neq b \)
  shows \( (\lambda x. g (f x)) - \mathbb{R} \rightarrow c \)
  using inj by (intro tendsto-compose-eventually[OF g f]) (auto simp: eventually-at)

lemma metric-isCont-LIM-compose2:
  fixes f :: 'a::metric-space \( \Rightarrow - \)
  assumes f [unfolded isCont-def]: isCont f a
  and g: \( g - f a \rightarrow l \)
  and inj: \( \exists d>0. \forall x. x \neq a \land \text{dist } x a < d \rightarrow f x \neq f a \)
  shows \( (\lambda x. g (f x)) - a \rightarrow l \)
  by (rule metric-LIM-compose2[OF f g inj])

108.12 Complete metric spaces

108.13 Cauchy sequences

lemma (in metric-space) Cauchy-def: Cauchy \( X = (\forall e>0. \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X m) (X n) < e) \)
proof –
  have \( \ast \): eventually P (INF M. principal \{ (X m, X n) | n m. m \geq M \land n \geq M \})
  \( \mapsto (\exists M. \forall m \geq M. \forall n \geq M. P (X m, X n)) \) for P
  apply (subst eventually-INF-base)
  subgoal by simp
  subgoal for a b
    by (intro bexI[of - max a b]) (auto simp: eventually-principal subset-eq)
  subgoal by (auto simp: eventually-principal, blast)
  done
  have Cauchy X \( \mapsto (INF M. \text{principal } \{ (X m, X n) | n m. m \geq M \land n \geq M \}) \)
  \( \leq \) uniformity
    unfolding Cauchy-uniform-iff le-filter-def * ..
  also have \( \ldots = (\forall e>0. \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X m) (X n) < e) \)
    unfolding uniformity-dist le-INF-iff by (auto simp: * le-principal)
  finally show \( \exists \theta \)
  qed
lemma (in metric-space) Cauchy-altdef: Cauchy $f \longleftrightarrow (\forall e>0. \exists M. \forall m \geq M. \forall n>m. \ dist (f m) (f n) < e)$

(is ?lhs $\longleftrightarrow$ ?rhs)

proof
  assume ?rhs
  show ?lhs unfolding Cauchy-def
  proof (intro allI impI)
    fix $e :: real$
    assume $e > 0$
    with ?rhs obtain $M$ where $M: m \geq M \Longrightarrow n > m \Longrightarrow dist (f m) (f n) < e$ for $m n$
    by blast
    have $dist (f m) (f n) < e$ if $m \geq M$ and $n > m$
    using $M[of m n]$ that by (cases m n rule: linorder-cases) (auto simp: dist-commute)
    then show $\exists M. \forall m \geq M. \forall n \geq M. \forall m \geq M. \forall n > m. \ dist (f m) (f n) < e$
    by blast
  qed
next
  assume ?lhs
  show ?rhs unfolding Cauchy-def by blast
  qed

lemma (in metric-space) Cauchy-altdef2: Cauchy $s \longleftrightarrow (\forall e>0. \exists N :: \text{nat}. \forall n \geq N. \ dist (s n) (s N) < e)$ (is ?lhs $= ?rhs$)

proof
  assume Cauchy $s$
  then show ?rhs by (force simp: Cauchy-def)
next
  assume ?rhs
  { fix $e :: real$
    assume $e > 0$
    with ?rhs obtain $N$ where $N: \forall n \geq N. \ dist (s n) (s N) < e/2$
    by (erule-tac $x = e/2$ in allE) auto
    { fix $n m$
      assume $nm: N \leq m \land N \leq n$
      then have $dist (s m) (s n) < e$ using $N$
      using dist-triangle-half[rule: [of $s m s N e s n$]]
    }
by blast

then have \( \exists N. \forall m n. N \leq m \land N \leq n \rightarrow \text{dist} (s m) (s n) < e \)
  by blast

then have ?lhs
  unfolding Cauchy-def by blast
then show ?lhs
  by blast
qed

lemma (in metric-space) metric-CauchyI:
  \((\forall e. \exists M. \forall m \geq M. \forall n \geq M. \text{dist} (X m) (X n) < e) \Longrightarrow \text{Cauchy} X\)
  by (simp add: Cauchy-def)

lemma (in metric-space) CauchyI:\
  \((\forall e. \exists M. \forall m \geq M. \forall n > m. \text{dist} (X m) (X n) < e) \Longrightarrow \text{Cauchy} X\)
  unfolding Cauchy-altdef by blast

lemma (in metric-space) metric-CauchyD:
  \[\text{Cauchy} X \Longrightarrow \exists M. \forall n \geq M. \forall m \geq M. \text{dist} (X m) (X n) < e\]
  by (simp add: Cauchy-def)

lemma (in metric-space) metric-Cauchy-iff2:
  \[\text{Cauchy} X \iff \forall j. (\exists M. \forall m \geq M. \forall n \geq M. |X m - X n| < inverse (real (Suc j)))\]
  apply (auto simp add: Cauchy-def)
  by (metis less-trans of-nat-Suc reals-Archimedean)

lemma Cauchy-iff2: \[\text{Cauchy} X \iff (\forall j. (\exists M. \forall m \geq M. \forall n \geq M. |X m - X n| < inverse (real (Suc j))))\]
  by (simp only: metric-Cauchy-iff2 dist-real-def)

lemma lim-1-over-n [tendsto-intros]: \((\lambda n. 1 / \text{of-nat} n) \longrightarrow (0 ::'a::real-normed-field))\)
sequentially
proof (subst lim-sequentially, intro allI impI allI)
  fix e::real and n
  assume e: e > 0
  have inverse e < of-nat (nat [inverse e + 1]) by linarith
  also assume n \geq nat [inverse e + 1]
  finally show \[\text{dist} (1 / \text{of-nat} n :: 'a) 0 < e\]
    using e by (simp add: field-split-simps norm-divide)
  qed

lemma (in metric-space) complete-def:
  shows complete S = \(\forall f. (\forall n. f n \in S) \land \text{Cauchy} f \longrightarrow (\exists l \in S. f \longrightarrow l)\)
  unfolding complete-uniform
proof safe
  fix f :: nat \Rightarrow 'a
assume \( f : \forall n. f\ n \in S \text{ Cauchy } f \)
and \( \ast : \forall F \leq \text{principal } S. F \neq \bot \longrightarrow \text{cauchy-filter } F \longrightarrow (\exists x \in S. F \leq \text{nhds } x) \)
then show \( \exists l \in S. f \longrightarrow l \)
unfolding \text{filterlim-def} using \( f \)
by (intro \( \ast \) \{rule-format\})
(auto simp: \text{filtermap-sequentially-ne-bot le-principal eventually-filtermap Cauchy-uniform})
next
fix \( F \) :: 'a filter
assume \( F \leq \text{principal } S \) \( F \neq \bot \text{cauchy-filter } F \)
assume \( \text{seq} : \forall f. (\forall n. f\ n \in S) \land \text{Cauchy } f \longrightarrow (\exists x \in S. f \longrightarrow x) \)
from \( \langle F \leq \text{principal } S \rangle \langle \text{cauchy-filter } F \rangle \) have \( \text{FF-le} : F \times F \leq \text{uniformity-on } S \)
by (simp add: \text{cauchy-filter-def principal-prod-principal} \{symmetric\} \text{prod-filter-mono})
let \( ?P = \lambda P e. \text{eventually } P F \land (\forall x. P x \longrightarrow x \in S) \land (\forall x y. P x \longrightarrow P y \longrightarrow \text{dist } x y < e) \)
have \( P : \exists P. ?P P \epsilon \) if \( 0 < \epsilon \) for \( \epsilon :: \text{real} \)
proof
from that have eventually \( (\lambda(x, y). x \in S \land y \in S \land \text{dist } x y < \epsilon) \) (uniformity-on \( S \))
by (auto simp: \text{eventually-inf-principal eventually-uniformity-metric})
from \text{filter-leD}[\text{OF FF-le this}] show \( \text{thesis} \)
by (auto simp: \text{eventually-prod-same})
qed
have \( \exists P. \forall n. ?P (P\ n) (1 / Suc\ n) \land P (Suc\ n) \leq P\ n \)
proof (rule \text{dependent-nat-choice})
show \( \exists P. ?P P (1 / Suc\ 0) \)
using \( P[\text{of } 1] \) by auto
next
fix \( n \) assume \( ?P P (1 / \text{Suc } n) \)
moreover obtain \( Q \) where \( ?P Q (1 / \text{Suc } (Suc\ n)) \)
using \( P[\text{of } 1 / \text{Suc } (Suc\ n)] \) by auto
ultimately show \( \exists Q. ?P Q (1 / \text{Suc } (Suc\ n)) \land Q \leq P \)
by (intro \text{exf[of - } \lambda x. P x \land Q x]) (auto simp: \text{eventually-conj-iff})
qed
then obtain \( P \) where \( \text{P: eventually } (P\ n) F P\ n\ x \longrightarrow x \in S \)
\( P\ n\ x \longrightarrow P\ n\ y \longrightarrow \text{dist } x y < 1 / \text{Suc } n P (Suc\ n) \leq P\ n \)
for \( n x y \)
by \text{metis}
have \( \text{antimono } P \)
using \( P(4) \) by (rule \text{decseq-SucI})
obtain \( X \) where \( X : P\ n (X\ n) \) for \( n \)
using \( P(1)[\text{THEN eventually-happens'[OF } \langle F \neq \bot \rangle]\] \) by \text{metis}
have \( \text{Cauchy } X \)
unfolding \( \text{metric-Cauchy-iff2 inverse-eq-divide} \)
proof (intro exI allI impI)
  fix j m n :: nat
  assume j ≤ m j ≤ n
  with ( antimono P) X have P j (X m) P j (X n)
    by (auto simp: antimono-def)
  then show dist (X m) (X n) < 1 / Suc j
    by (rule P)
qed

moreover have ∀n. X n ∈ S
  using P (2) X by auto
ultimately obtain x where X ----→ x x ∈ S
  using seq by blast

show ∃x∈S. F ≤ nhds x
proof (rule bexI)
  have eventually (λy. dist y x < e) F if 0 < e for e :: real
proof
  from that have (λn. 1 / Suc n :: real) ----→ 0 ∧ 0 < e / 2
    by (subst filterlim-sequentially-Suc) (auto intro: lim-1-over-n)
  then have ∀F n in sequentially. dist (X n) x < e / 2 ∧ 1 / Suc n < e / 2
    using (X ----→ x)
    unfolding tendssto iff order-tendsto iff[where 'a = real] eventually-conj iff
    by blast
  then obtain n where dist x (X n) < e / 2 ∧ 1 / Suc n < e / 2
    by (auto simp: eventually-sequentially dist-commute)
  show ?thesis
    using (eventually (P n) F)
    proof (eventually-elim)
      case (elim y)
      then have dist y (X n) < 1 / Suc n
        by (intro X P)
      also have . . . < e / 2 by fact
      finally show dist y x < e
        by (rule dist-triangle-half l) fact
    qed
  qed
  then show F ≤ nhds x
    unfolding nhds-metric le-INF iff le-principal by auto
  qed
qed

apparently unused

lemma (in metric-space) totally-bounded-metric:
totally-bounded S ←→ (∀e>0. ∃k. finite k ∧ S ⊆ (∪x∈k. {y. dist x y < e})))
unfolding totally-bounded-def eventually-uniformity metric imp-ex
apply (subst all-comm)
apply (intro arg-cong[where f=All] ext, safe)
subgoal for e
  apply (erule allE[af - λ(x, y). dist x y < e])
apply auto
done

subgoal for e P k
  apply (intro exI[of - k])
  apply (force simp: subset_eq)
done
done

setup (Sign.add_const_constraint (const_name dist, SOME typ ('a::dist ⇒ 'a ⇒ real))

lemma cauchy-filter-metric:
  fixes F :: 'a::{uniformity-dist,uniform-space} filter
  shows cauchy-filter F (∀ e. e > 0 → (∃ P. eventually P F ∧ (∀ x y. P x ∧ P y → dist x y < e)))
  proof (unfold cauchy-filter-def le-filter-def)
    assume asm: ∀ e > 0. ∃ P. eventually P (filtermap f F)
    then show (eventually P uniformity) ⟹ eventually P (F × F F)
      apply (auto simp: eventually-uniformity-metric)
      using eventually-prod-same by blast
  next
    fix e :: real
    assume asm: e > 0
    assume asm: ∀ P. eventually P uniformity ⟹ eventually P (F × F F)
    define P where P ≡ λ(x,y :: 'a). dist x y < e
    with asm (e > 0), have (eventually P (F × F F))
      by (metis case-prod-conv eventually-uniformity-metric)
    then show (∃ P. eventually P F ∧ (∀ x y. P x ∧ P y → dist x y < e))
      by (auto simp add: eventually-prod-same P-def)
  qed

lemma cauchy-filter-metric-filtermap:
  fixes f :: 'a ⇒ 'b::{uniformity-dist,uniform-space}
  shows cauchy-filter (filtermap f F) (∀ e. e > 0 → (∃ P. eventually P F ∧ (∀ x y. P x ∧ P y → dist (f x) (f y) < e)))
  proof (subst cauchy-filter-metric, intro iffI allI impl)
    assume asm: ∀ e > 0. ∃ P. eventually P (filtermap f F) ∧ (∀ x y. P x ∧ P y → dist (f x) (f y) < e)
    then show e > 0 ⟹ ∃ P. eventually P F ∧ (∀ x y. P x ∧ P y → dist (f x) (f y) < e)
      unfolding eventually-filtermap by blast
  next
    assume asm: ∀ e > 0. ∃ P. eventually P F ∧ (∀ x y. P x ∧ P y → dist (f x) (f y) < e)
    then show (eventually P uniformity) ⟹ eventually P (F × F F)
      apply (auto simp: eventually-uniformity-metric)
      using eventually-prod-same by blast
  next

y) < e):
  fix e:real assume (e > 0)
  then obtain P where (eventually P F) and PPe: (P x ∧ P y) → dist (f x) (f y) < e for x y
  using asm by blast

  show (∃ P. eventually P (filtermap f F) ∧ (∀ x y. P x ∧ P y) → dist x y < e)
  apply (rule exI[of - λ x. ∃ y. P y ∧ x = f y])
  using PPe (eventually P F) apply (auto simp: eventually-filtermap)
  by (smt (verit, ccfv-SIG eventually-elim2)

qed

setup (Sign.add-const-constraint (const-name ‹dist›, SOME typ ‹'a::metric-space ⇒ 'a ⇒ real›)

108.13.1 Cauchy Sequences are Convergent

class complete-space = metric-space +
  assumes Cauchy-convergent: Cauchy X ⇒ convergent X

lemma Cauchy-convergent-iff: Cauchy X ⇔ convergent X
  for X :: nat ⇒ 'a::complete-space
  by (blast intro: Cauchy-convergent convergent-Cauchy)

To prove that a Cauchy sequence converges, it suffices to show that a sub-
sequence converges.

lemma Cauchy-converges-subseq:
  fixes u::nat ⇒ 'a::metric-space
  assumes Cauchy u
  strict-mono r
  (u ∘ r) → l
  shows u → l

proof –
  have ∗: eventually (λ n. dist (u n) l) < e) sequentially if e > 0 for e
  proof –
    have e/2 > 0 using that by auto
    then obtain N1 where N1: ∀ m. m ≥ N1 ⇒ n ≥ N1 ⇒ dist (u m) (u n) < e/2
    using Cauchy w unfolding Cauchy-def by blast
    obtain N2 where N2: ∀ n. n ≥ N2 ⇒ dist ((u ∘ r) n) l < e / 2
    using order-tendsToD(2)[OF iffD1[OF tendsTo-dist-ifff (u ∘ r) → l] e/2 > 0]
    unfolding eventually-sequentially by auto
  have dist (u n) l < e if n ≥ max N1 N2 for n
  proof –
    have dist (u n) l ≤ dist (u n) ((u ∘ r) n) + dist ((u ∘ r) n) l
    by (rule dist-triangle)
    also have ... < e/2 + e/2
  proof (intro add-strict-mono)
show \( \text{dist}((u \circ r) n) < e / 2 \)
using \( N1[\text{of } n \ r n \ N2[\text{of } n] \) that unfolding comp-def
by (meson assms(2) le-trans max.bounded_iff strict_mono_imp_increasing)
show \( \text{dist}((u \circ r) n) l < e / 2 \)
using \( N2 \) that by auto
qed

finally show \(?thesis \) by simp
qed

then show \(?thesis unfolding eventually_seq by blast
qed

have \( \lambda n. \text{dist}((u n) l) \rightarrow 0 \)
by (simp add: less_trans * order_tendsto)
then show \(?thesis \)
using tendsto_dist_iff by auto
qed

108.14 The set of real numbers is a complete metric space

Proof that Cauchy sequences converge based on the one from http://pirate.
shu.edu/~wachsmut/ira/nunseq/proofs/cauconv.html

If sequence \( X \) is Cauchy, then its limit is the lub of \( \{ r. \exists N. \forall n \geq N. r < X n \} \)

lemma increasing-LIMSEQ:
fixes \( f :: \text{nat} \Rightarrow \text{real} \)
assumes inc: \( \forall n. f n \leq f (Suc n) \)
and bdd: \( \forall n. f n \leq l \)
and en: \( \forall e. 0 < e \Rightarrow \exists n. l \leq f n + e \)
shows \( f \rightarrow l \)
proof (rule increasing_tendsto)
fix \( x \)
assume \( x < l \)
with dense[of 0 l - x] obtain \( e \) where \( 0 < e \) \( e < l - x \)
by auto
from en[OF \( \langle 0 < e \rangle \) obtain \( n \) where \( l - e \leq f n \)
by (auto simp: field_simps)
with \( \langle e < l - x \rangle \) \( \langle 0 < e \rangle \) have \( x < f n \)
by simp
with incseq_SucI[of \( f \), OF inc] show eventually \( \langle \lambda n. x < f n \rangle \) sequentially
defined by (auto simp: eventually_sequentially incseq_def intro: less_trans)
qed

lemma real-Cauchy-convergent:
fixes \( X :: \text{nat} \Rightarrow \text{real} \)
assumes \( X :: \text{Cauchy X} \)
sows \( \text{convergent} X \)
proof

define \( S :: \text{real set where } S = \{ x. \exists N. \forall n \geq N. x < X n \} \)
then have mem-S: \( \forall N x. \forall n \geq N. x < X n \Rightarrow x \in S \)
by auto
have bound-isUb: \( y \leq x \) if \( N: \forall n \geq N. X n < x \) and \( y \in S \) for \( N \) and \( x y :: \) real
proof —
  from that have \( \exists M. \forall n \geq M. y < X n \)
    by (simp add: S-def)
then obtain \( M \) where \( \forall n \geq M. y < X n \).
then have \( y < X (\max M N) \) by simp
also have \( \ldots < x \) using \( N \) by simp
finally show \( ?thesis \) by (rule order-less-imp-le)
qed

obtain \( N \) where \( \forall m \geq N. \forall n \geq N. \dist (X m) (X n) < 1 \)
using \( X[THEN \text{metric-CauchyD}, \OF \text{zero-less-one}] \) by auto
then have \( N: \forall n \geq N. \dist (X n) (X N) < 1 \) by simp
have \([simp]: S \neq \{\}\)
proof (intro exI ex-in-conv \[THEN \text{iffD1}\])
  from \( N \) have \( \forall n \geq N. X N - 1 < X n \)
    by (simp add: abs-diff-less-iff dist-real-def)
then show \( X N - 1 \in S \) by (rule mem-S)
qed
have \([simp]: \text{bdd-above } S\)
proof
  from \( N \) have \( \forall n \geq N. X n < X N + 1 \)
    by (simp add: abs-diff-less-iff dist-real-def)
then show \( \\\land s. s \in S \Longrightarrow s \leq X N + 1 \)
    by (rule bound-isUb)
qed
have \( X \longrightarrow \Sup S \)
proof (rule metric-LIMSEQ-I)
  fix \( r :: \) real
assume \( 0 < r \)
then have \( r: 0 < r/2 \) by simp
obtain \( N \) where \( \forall n \geq N. \forall m \geq N. \dist (X n) (X m) < r/2 \)
  using \( X[THEN \text{metric-CauchyD}, \OF \text{X r}] \) by auto
then have \( \forall n \geq N. \dist (X n) (X N) < r/2 \) by simp
then have \( N: \forall n \geq N. X N - r/2 < X n \land X n < X N + r/2 \)
  by (simp only: dist-real-def abs-diff-less-iff)
from \( N \) have \( \forall n \geq N. X n - r/2 < X n \) by blast
then have \( X N - r/2 \in S \) by (rule mem-S)
then have \( 1: X N - r/2 \leq \Sup S \) by (simp add: cSup-upper)
from \( N \) have \( \forall n \geq N. X n < X N + r/2 \) by blast
from bound-isUb \([OF this]\)
have \( 2: \Sup S \leq X N + r/2 \)
  by (intro cSup-least) simp-all
show \( \exists N. \forall n \geq N. \dist (X n) (S) < r \)
proof (intro cxI allI impI)
fix \( n \)
assume \( n : N \leq n \)
from \( N n \) have \( X n < X N + r/2 \) and \( X N - r/2 < X n \)
by simp-all
then show \( \text{dist} (X n) (\text{Sup } S) < r \) using 1 2
by (simp add: abs-diff-less-iff dist-real-def)
qed
qed
then show \( ?\text{thesis} \) by (auto simp: convergent-def)
qed

instance real :: complete-space
by intro-classes (rule real-Cauchy-convergent)

class banach = real-normed-vector + complete-space

instance real :: banach ..

lemma tendsto-at-top1-sequentially:
fixes \( f : \text{real} \Rightarrow \text{'}b : \text{first-countable-topology} \)
assumes \( \star : \forall X. \text{filterlim } X \text{ at-top sequentially } \Rightarrow (\lambda n. f (X n)) \longrightarrow y \)
shows \( (f \longrightarrow y) \text{ at-top} \)
proof
obtain \( A \) where \( A \): decseq \( A \) open \( (A n) y \in A n \text{ nhds } y = (\text{INF } n. \text{principal } (A n)) \) for \( n \)
by (rule nhds-countable[of y]) (rule that)
have \( \forall m. \exists k. \forall x \geq k. f x \in A m \)
proof (rule ccontr)
assume \( \neg (\forall m. \exists k. \forall x \geq k. f x \in A m) \)
then obtain \( m \) where \( \exists k. \exists x \geq k. f x \notin A m \)
by auto
then have \( \exists X. \forall n. (f (X n) \notin A m) \land \max n (X n) + 1 \leq X (\text{Suc } n) \)
by (intro dependent-nat-choice) (auto simp del: max.bounded-iff)
then obtain \( X \) where \( X : \forall n. f (X n) \notin A m \land n. \max n (X n) + 1 \leq X (\text{Suc } n) \)
by auto
have \( I \leq n \Rightarrow \text{real } n \leq X n \) for \( n \)
using \( X[\text{of } n - 1] \) by auto
then have \( \text{filterlim } X \text{ at-top sequentially} \)
by (force intro: filterlim-at-top-mono[OF filterlim-real-sequentially]
simp: eventually-sequentially)
from topological-tendstoD[OF \( \star \)[OF this] A(2, 3), of m] X(1) show False
by auto
qed
then obtain \( k \) where \( k m \leq x \Rightarrow f x \in A m \) for \( m x \)
by metis
then show \( ?\text{thesis} \)
unfolding at-top-def \( A \) by (intro filterlim-base[where \( i=k \)]) auto
lemma tendsto-at-topI-sequentially-real:
fixes $f :: \mathbb{R} \Rightarrow \mathbb{R}$
assumes mono: mono $f$
and limseq: $(\lambda n. f (\text{real } n)) \longrightarrow y$
shows $(f \longrightarrow y)$ at-top
proof (rule tendstoI)
fix $e :: \mathbb{R}$
assume $0 < e$
with limseq obtain $N :: \mathbb{N}$ where $N \leq n \Rightarrow |f (\text{real } n) - y| < e$ for $n$
by (auto simp: lim-sequentially dist-real-def)
have le: $f x \leq y$ for $x :: \mathbb{R}$
proof
obtain $n$ where $x \leq \text{real-of-nat } n$
using real-arch-simple[of $x$]
note monoD[OF mono this]
also have $f (\text{real-of-nat } n) \leq y$
by (rule LIMSEQ-le-const[OF limseq]) (auto intro: exI[of - $n$] monoD[OF mono])
finally show \text{thesis}.
qed
have eventually $(\lambda x. \text{real } N \leq x)$ at-top
proof (rule eventually-ge-at-top)
then show eventually $(\lambda x. \text{dist } (f x) y < e)$ at-top
proof (eventually-elim)
assume (elim $x$)
with $N$[of $N$] le have $y - f (\text{real } N) < e$ by auto
moreover note monoD[OF mono elim]
ultimately show \text{dist } $(f x) y < e$
using le[of $x$] by (auto simp: dist-real-def field-simps)
qed
qed
end

109 Limits on Real Vector Spaces

theory Limits
imports Real-Vector-Spaces
begin

lemma range-mult [simp]:
fixes $a :: \mathbb{R}$ shows range $((\ast) \ a) = (\text{if } a=0 \text{ then } \{0\} \text{ else } \text{UNIV})$
by (simp add: surj-def) (meson dvdE dvd-field-iff)

109.1 Filter going to infinity norm

definition at-infinity :: 'a::real-normed-vector filter
where \( \text{at-infinity} = (\text{INF } r. \text{ principal } \{ x. \ r \leq \text{norm } x \}) \)

**lemma** eventually-at-infinity: eventually \( P \) at-infinity \( \iff (\exists b. \ \forall x. \ b \leq \text{norm } x \rightarrow P x) \)

**unfolding** at-infinity-def

**by** (auto simp: eventually-INF-base)

**corollary** eventually-at-infinity-pos: eventually \( p \) at-infinity \( \iff (\exists b. \ 0 < b \land (\forall x. \ \text{norm } x \geq b \rightarrow p x)) \)

**unfolding** eventually-at-infinity

**lemma** at-infinity-eq-at-top-bot: (at-infinity :: real filter) = sup at-top at-bot

**proof**

1. \( \forall n \geq u. \ A n; \forall n \leq v. \ A n \rightarrow \exists b. \ \forall x. \ b \leq |x| \rightarrow A x \) for \( A \) and \( u \) and \( v \) and \( A \) and \( u \) and \( v \) and \( A \)

2. \( \forall x. \ u \leq |x| \rightarrow A x \rightarrow \exists N. \ \forall n \geq N. \ A n \) for \( A \) and \( u \) and \( v \) and \( A \) and \( u \) and \( v \) and \( A \)

3. \( \forall x. \ u \leq |x| \rightarrow A x \rightarrow \exists N. \ \forall n \leq N. \ A n \) for \( A \) and \( u \) and \( v \) and \( A \) and \( u \) and \( v \) and \( A \)

**show** ?thesis

**by** (auto simp: filter-eq-iff eventually-sup eventually-at-infinity eventually-at-top-linorder eventually-at-bot-linorder intro: 1 2 3)

**qed**

**lemma** at-top-le-at-infinity: at-top \( \leq \) (at-infinity :: real filter)

**unfolding** at-infinity-eq-at-top-bot by simp

**lemma** at-bot-le-at-infinity: at-bot \( \leq \) (at-infinity :: real filter)

**unfolding** at-infinity-eq-at-top-bot by simp

**lemma** filterlim-at-top-imp-at-infinity: filterlim \( f \) at-top \( F \) \( \Rightarrow \) filterlim \( f \) at-infinity \( F \)

**for** \( f :: - \rightarrow real \)

**by** (rule filterlim-mono[OF - at-top-le-at-infinity order-refl])

**lemma** filterlim-real-at-infinity-sequentially: filterlim real at-infinity sequentially

**by** (simp add: filterlim-at-top-imp-at-infinity filterlim-real-sequentially)

**lemma** lim-infinity-imp-sequentially: \((f \rightarrow l) \at-inf \Rightarrow (\lambda n. \ f(n)) \rightarrow \ l) \) sequentially

**by** (simp add: filterlim-at-top-imp-at-infinity filterlim-compose filterlim-real-sequentially)

## 109.1.1 Boundedness

**definition** \( \text{Bfun} :: \ ('a \Rightarrow 'b::metric-space) \Rightarrow 'a \text{ filter} \Rightarrow \text{bool} \)

**where** \( \text{Bfun-metric-def} : \text{Bfun } f F = (\exists y. \exists K>0. \text{ eventually } (\lambda x. \ \text{dist } (f x \ y) \leq \text{norm } x) \rightarrow F \)
THEORY “Limits”

abbreviation Bseq :: (nat ⇒ 'a::metric-space) ⇒ bool
  where Bseq X ≡ Bfun X sequentially

lemma Bseq-conv-Bfun: Bseq X ⟷ Bfun X sequentially ..

lemma Bseq-ignore-initial-segment: Bseq X ⟷ Bseq (λn. X (n + k))
  unfolding Bfun-metric-def by (subst eventually-sequentially-seq)

lemma Bseq-offset: Bseq (λn. X (n + k)) ≡ Bseq X
  unfolding Bfun-metric-def by (subst (asm) eventually-sequentially-seq)

lemma Bfun-def: Bfun f F ≡ (∃ K > 0. eventually (λx. norm (f x) ≤ K) F)
  unfolding Bfun-metric-def norm-conv-dist
  proof safe
  fix y K
  assume K: 0 < K and *: eventually (λx. dist (f x) y ≤ K) F
  moreover have eventually (λx. dist (f x) 0 ≤ dist (f x) y + dist 0 y) F
    by (intro always-eventually) (metis dist-commute dist-triangle)
  with * have eventually (λx. dist (f x) 0 ≤ K + dist 0 y) F
    by eventually-elim auto
  with ⟨0 < K⟩ show ∃ K > 0. eventually (λx. dist (f x) 0 ≤ K) F
    by (intro exI[of - K + dist 0 y] add-pos-nonneg conjI zero-le-dist) auto
  qed (force simp del: norm-conv-dist [symmetric])

lemma BfunI:
  assumes K: eventually (λx. norm (f x) ≤ K) F
  shows Bfun f F
  unfolding Bfun-def
  proof (intro exI conjI allI)
    show 0 < max K 1 by simp
    show eventually (λx. norm (f x) ≤ max K 1) F
      using K by (rule eventually-mono) simp
  qed

lemma BfunE:
  assumes Bfun f F
  obtains B where 0 < B and eventually (λx. norm (f x) ≤ B) F
  using assms unfolding Bfun-def by blast

lemma Cauchy-Bseq:
  assumes Cauchy X shows Bseq X
  proof -
    have ∃ y K. 0 < K ∧ (∃ N. ∀ n≥N. dist (X n) y ≤ K)
      if ∀ m n. [m ≥ M; n ≥ M] ==> dist (X m) (X n) < 1 for M
    by (meson order.order-iff-strict that zero-less-one)
    with assms show ?thesis
      by (force simp: Cauchy-def Bfun-metric-def eventually-sequentially)
109.1.2 Bounded Sequences

**lemma** BseqI': \((\forall n. \text{norm} (X n) \leq K) \implies Bseq X\)

**by** (intro BfunI) (auto simp: eventually-sequentially)

**lemma** Bseq-def: \(Bseq X \iff (\exists K > 0. \forall n. \text{norm} (X n) \leq K)\)

**proof** safe

fix \(N K\)

assume \(0 < K \forall n \geq N. \text{norm} (X n) \leq K\)

then show \(\exists K > 0. \forall n. \text{norm} (X n) \leq K\)

by (intro exI [of - max (Max (\text{norm} ' X ' {..N}))) \(K\)] max.strict-coboundedI2)

(auto intro !: imageI not-less [where \(a = \text{nat}\), THEN iffD1] Max-ge simp: le-max-iff-disj)

qed auto

**lemma** BseqE: \(Bseq X \implies (\exists K. 0 < K \implies \forall n. \text{norm} (X n) \leq K \implies Q) \implies Q\)

**unfolding** Bseq-def **by** auto

**lemma** BseqD: \(Bseq X \implies \exists K. 0 < K \land (\forall n. \text{norm} (X n) \leq K)\)

**by** (simp add: Bseq-def)

**lemma** BseqI: \(0 < K \implies \forall n. \text{norm} (X n) \leq K \implies Bseq X\)

**by** (auto simp: Bseq-def)

**lemma** Bseq-bdd-above: \(Bseq X \implies \text{bdd-above} (\text{range} X)\)

for \(X :: \text{nat} \Rightarrow \text{real}\)

**proof** (elim BseqE, intro bdd-aboveI2)

fix \(K n\)

assume \(0 < K \forall n. \text{norm} (X n) \leq K\)

then show \(X n \leq K\)

by (auto elim!: allE[of - n])

qed

**lemma** Bseq-bdd-above': \(Bseq X \implies \text{bdd-above} (\lambda n. \text{norm} (X n))\)

for \(X :: \text{nat} \Rightarrow 'a :: \text{real-normed-vector}\)

**proof** (elim BseqE, intro bdd-aboveI2)

fix \(K n\)

assume \(0 < K \forall n. \text{norm} (X n) \leq K\)

then show \(\text{norm} (X n) \leq K\)

by (auto elim!: allE[of - n])

qed

**lemma** Bseq-bdd-below: \(Bseq X \implies \text{bdd-below} (\text{range} X)\)

for \(X :: \text{nat} \Rightarrow \text{real}\)

**proof** (elim BseqE, intro bdd-belowI2)
THEORY "Limits"

fix K n
assume 0 < K ∀ n. norm (X n) ≤ K
then show − K ≤ X n
by (auto elim!: allE[of - n])
qed

lemma Bseq-eventually-mono:
assumes eventually (λn. norm (f n) ≤ norm (g n)) sequentially Bseq g
shows Bseq f
proof
from assms(2) obtain K where 0 < K and eventually (λn. norm (g n) ≤ K)
sequentially unfolding Bfun-def by fast
with assms(1) have eventually (λn. norm (f n) ≤ K) sequentially
by (fast elim: eventually-elim2 order-trans)
with ‹0 < K› show Bseq f
unfolding Bfun-def by fast
qed

lemma lemma-NBseq-def:
(∃ K > 0. ∀ n. norm (X n) ≤ K) ←→ (∃ N. ∀ n. norm (X n) ≤ real(Suc N))
proof safe
fix K :: real
from reals-Archimedean2 obtain n :: nat where K < real n..
then have K ≤ real (Suc n) by auto
moreover assume ∀ m. norm (X m) ≤ K
ultimately have ∀ n. norm (X n) ≤ real (Suc n)
by (blast intro: order-trans)
then show ∃ N. ∀ n. norm (X n) ≤ real (Suc N)..
next
show ∀ N. ∀ n. norm (X n) ≤ real (Suc N) −→ ∃ K > 0. ∀ n. norm (X n) ≤ K
using of-nat-0-less-iff by blast
qed

Alternative definition for Bseq.
lemma Bseq-iff: Bseq X −→ (∃ N. ∀ n. norm (X n) ≤ real(Suc N))
by (simp add: Bseq-def) (simp add: lemma-NBseq-def)

lemma lemma-NBseq-def2: (∃ K > 0. ∀ n. norm (X n) ≤ K) = (∃ N. ∀ n. norm (X n) < real(Suc N))
proof −
have *: ∀ N. ∀ n. norm (X n) ≤ 1 + real N −→
∃ N. ∀ n. norm (X n) < 1 + real N
by (metis add.commute le-less-trans less-add-one of-nat-Suc)
then show *thesis
unfolding lemma-NBseq-def
by (metis less-le-not-le not-less-iff-gr-or-eq of-nat-Suc)
qed

Yet another definition for Bseq.
lemma Bseq-iff1a: Bseq X ←→ (∃N. ∀n. norm (X n) < real (Suc N))
by (simp add: Bseq-def lemma-NBseq-def2)

109.1.3 A Few More Equivalence Theorems for Boundedness

Alternative formulation for boundedness.
lemma Bseq-iff2: Bseq X ←→ (∃k > 0. ∃x. ∀n. norm (X n + x) ≤ k)
by (metis BseqE BseqI add.commute add-cancel-right-left add-uminus-conv-diff
norm-add-leD
norm-minus-cancel norm-minus-commute)

Alternative formulation for boundedness.
lemma Bseq-iff3: Bseq X ←→ (∃k > 0. ∃N. ∀n. norm (X n + X N) ≤ k)
(is ?P ←→ ?Q)
proof
assume ?P
then obtain K where *: 0 < K and **: ∀n. norm (X n) ≤ K
by (auto simp: Bseq-def)
from * have 0 < K + norm (X 0) by (rule order-less-le-trans) simp
from ** have ∀n. norm (X n - X 0) ≤ K + norm (X 0)
  by (auto intro: order-trans norm-triangle-ineq4)
then have ∀n. norm (X n - X 0) ≤ K + norm (X 0)
  by simp
with 0 < K + norm (X 0) show ?Q by blast
next
assume ?Q
then show ?P by (auto simp: Bseq-iff2)
qed

109.1.4 Upper Bounds and Lubs of Bounded Sequences

lemma Bseq-minus-iff: Bseq (λn. - (X n) :: 'a::real-normed-vector) ←→ Bseq X
by (simp add: Bseq-def)

lemma Bseq-add:
fixes f :: nat ⇒ 'a::real-normed-vector
assumes Bseq f
shows Bseq (λx. f x + c)
proof
from assms obtain K where K: ∀x. norm (f x) ≤ K
  unfolding Bseq-def by blast
{ fix x :: nat
  have norm (f x + c) ≤ norm (f x) + norm c by (rule norm-triangle-ineq)
  also have norm (f x) ≤ K by (rule K)
  finally have norm (f x + c) ≤ K + norm c by simp
}
then show ?thesis by (rule BseqI)
qed
lemma Bseq-add-iff: Bseq (λx. f x + c) ↔ Bseq f
for f :: nat ⇒ 'a::real-normed-vector
using Bseq-add[of f c] Bseq-add[of λx. f x + c - c] by auto

lemma Bseq-mult:
fixes f g :: nat ⇒ 'a::real-normed-field
assumes Bseq f and Bseq g
shows Bseq (λx. f x * g x)
proof
  from assms obtain K1 K2 where K: norm (f x) ≤ K1 K1 > 0 norm (g x) ≤ K2 K2 > 0
  for x
  unfolding Bseq-def by blast
  then have norm (f x * g x) ≤ K1 * K2 for x
  by (auto simp: norm-mult intro!: mult-mono)
  then show ?thesis by (rule BseqI')
qed

lemma Bfun-const [simp]: Bfun (λ-.. c) F
unfolding Bfun-metric-def by (auto intro!: exI[of - c] exI[of - 1::real])

lemma Bseq-cmult-iff:
fixes c :: 'a::real-normed-field
assumes c ≠ 0
shows Bseq (λx. c * f x) ↔ Bseq f
proof
  assume Bseq (λx. c * f x)
  then obtain K where K: ⋀x. norm (f x) :: 'a::real-normed-vector ≤ K1 K1 > 0 norm (g x) ≤ K2 K2 > 0
  for x
  unfolding Bseq-def by blast
  then have norm (c * f x) ≤ K1 * K2 for x
  by (auto simp: norm-mult intro!: mult-mono)
  then show ?thesis by (rule BseqI')
qed

lemma Bseq-subseq: Bseq f → Bseq (λx. f (g x))
for f :: nat ⇒ 'a::real-normed-vector
unfolding Bseq-def by auto

lemma Bseq-Suc-iff: Bseq (λn. f (Suc n)) ↔ Bseq f
for f :: nat ⇒ 'a::real-normed-vector
using Bseq-offset[of f 1] by (auto intro: Bseq-subseq)

lemma increasing-Bseq-subseq-iff:
assumes ⋀x y. x ≤ y → norm (f x :: 'a::real-normed-vector) ≤ norm (f y)
strict-mono g
shows Bseq (λx. f (g x)) ↔ Bseq f
proof
  assume Bseq (λx. f (g x))
  then obtain K where K: ⋀x. norm (f (g x)) ≤ K

unfolding Bseq-def by auto
{ fix x :: nat from filterlim-subseq[OF assms(2)] obtain y where g y ≥ x by (auto simp: filterlim-at-top eventually-at-top-linorder) then have norm (f x) ≤ norm (f (g y)) using assms(1) by blast also have norm (f (g y)) ≤ K by (rule K) finally have norm (f x) ≤ K . }
then show Bseq f by (rule BseqI') qed (use Bseq-subseq[of f g] in simp-all)

lemma nonneg-incseq-Bseq-subseq-iff:
fixes f :: nat ⇒ real and g :: nat ⇒ nat
assumes ∃ x. f x ≥ 0 incseq f strict-mono g
shows Bseq (λx. f (g x)) ←→ Bseq f
using assms by (intro increasing-Bseq-subseq-iff) (auto simp: incseq-def)

lemma Bseq-eq-bounded: range f ⊆ {a..b} =⇒ Bseq f
for a b :: real
proof (rule BseqI'[where K=max (norm a) (norm b)])
fix n assume range f ⊆ {a..b} then have f n ∈ {a..b} by blast
then show norm (f n) ≤ max (norm a) (norm b) by auto
qed

lemma incseq-bounded: incseq X =⇒ ∀i. X i ≤ B =⇒ Bseq X
for B :: real
by (intro Bseq-eq-bounded[of X X 0 B]) (auto simp: incseq-def)

lemma decseq-bounded: decseq X =⇒ ∀i. B ≤ X i =⇒ Bseq X
for B :: real
by (intro Bseq-eq-bounded[of X B X 0]) (auto simp: decseq-def)

109.1.5 Polynomial function extremal theorem, from HOL Light

lemma polyfun-extremal-lemma:
fixes c :: nat ⇒ 'a::real-normed-div-algebra
assumes 0 < e
shows ∃ M. ∀ z. M ≤ norm(z) =⇒ norm (∑ i≤n. c(i) * z^i) ≤ e * norm(z)
(Suc n)
proof (induct n)
case 0 with assms
show ?case
apply (rule-tac x=norm (e 0) / e in extI)
apply (auto simp: field-simps)
done
next
case (Suc n)
  obtain M where M: \( \forall z. M \leq \text{norm } z \implies \text{norm } (\sum_{i \leq n} c_i \cdot z^i) \leq e \cdot \text{norm } z \sim \text{Suc } n \)
  using Suc assms by blast
  show ?case
  proof (rule exI)
    where x = max M (1 + norm (c (Suc n)) / e)
  
    assume z1: \( M \leq \text{norm } z \) and \( 1 + \text{norm } (c (\text{Suc } n)) / e \leq \text{norm } z \)
    using assms by (simp add: field-simps)
    have \( \text{norm } (\sum_{i \leq n} c_i \cdot z^i) \leq e \cdot \text{norm } z \sim \text{Suc } n \)
    using M [OF z1] by simp
    then have \( \text{norm } (\sum_{i \leq n} c_i \cdot z^i) + \text{norm } (c (\text{Suc } n) \cdot z \sim \text{Suc } n) \leq e \cdot \text{norm } z \sim \text{Suc } n + \text{norm } (c (\text{Suc } n) \cdot z \sim \text{Suc } n) \)
    by simp
    then have \( \text{norm } ((\sum_{i \leq n} c_i \cdot z^i) + c (\text{Suc } n) \cdot z \sim \text{Suc } n) \leq e \cdot \text{norm } z \sim \text{Suc } n \)
    by (blast intro: norm-triangle-le elim: )
    also have \( \ldots \leq (e + \text{norm } (c (\text{Suc } n))) \cdot \text{norm } z \sim \text{Suc } n \)
    by (simp add: norm-power norm-mult algebra-simps)
    also have \( \ldots \leq (e \cdot \text{norm } z) \cdot \text{norm } z \sim \text{Suc } n \)
    by (metis z2_mult.commute mult-left-mono norm-ge-zero norm-power)
    finally show \( \text{norm } ((\sum_{i \leq n} c_i \cdot z^i) + c (\text{Suc } n) \cdot z \sim \text{Suc } n) \leq e \cdot \text{norm } z \sim \text{Suc } n \)
    by simp
  qed
qed

lemma polyfun-extremal:
  fixes c :: nat \Rightarrow 'a::real-normed-div-algebra
  assumes k: c k \neq 0
  \( 1 \leq k \) and kn: k \leq n
  shows eventually (\( \lambda z. \text{norm } (\sum_{i \leq n} c(i) \cdot z^i) \geq B \)) at-infinity
using kn
proof (induction n)
case 0
  then show ?case
  using k by simp
next
case (Suc m)
  show ?case
  proof (cases c (Suc m) = 0)
    case True
    then show ?thesis using Suc k
    by auto (metis antisym-conv less-eq-Suc-le not-le)
  next
case False
then obtain M where M:
  \( \forall z. \: M \leq \text{norm } z \implies \text{norm } (\sum i \leq m. \: c \: i \ast z^i) \leq \text{norm } (c \: \text{Suc } m) / 2 \)
* \text{norm } z \leq \text{Suc } m
  using polyfun-extremal-lemma [of norm(c (Suc m)) / 2 \: c \: m] \: Suc
by auto
have \( \exists b. \: \forall z. \: b \leq \text{norm } z \longrightarrow B \leq \text{norm } (\sum i \leq \text{Suc } m. \: c \: i \ast z^i) \)
proof (rule exI [where \( x=\max M \: (\max 1 \: (\text{Suc } m) / 2) \)])
claarsimp simp del:
\( \text{THEORY } \text{"Limits" } 2177 \)
(apply (apply (apply (rule exI [where \( x=\max M \: (\max 1 \: (\text{Suc } m) / 2) \)])
claarsimp simp del:
\( \text{by } (\text{simp add: field-simps}) \)
have \( nz: \: \text{norm } z \leq \text{norm } z \sim \text{Suc } m \)
by (metis \( \langle 1 \leq \text{norm } z \rangle \: \text{One-nat-def less-eq-Suc-le power-increasing power-one-right zero-less-Suc} \)
have \( \ast: \: \forall y \: x. \: \text{norm } (c \: \text{Suc } m) / 2 \leq \text{norm } y - \text{norm } x \implies B \leq \text{norm } (x + y) \)
by (metis abs-le-iff add.commute norm-diff-ineq order-trans z2)
have \( \text{norm } z \ast \text{norm } (c \: \text{Suc } m) / 2 \ast \text{norm } (\sum i \leq m. \: c \: i \ast z^i) \leq \text{norm } (c \: \text{Suc } m) / \text{norm } z \ast \text{norm } (c \: \text{Suc } m) / \text{norm } z \sim \text{Suc } m \)
using \( M \: \text{Suc } z1 \) \by auto
also have \( \ast: \: \leq 2 \ast \text{norm } (c \: \text{Suc } m) / \text{norm } z \sim \text{Suc } m \)
using \( nz \) \by (simp add: mult-mono del: \text{power-Suc})
finally show \( B \leq \text{norm } (\sum i \leq m. \: c \: i \ast z^i) + c \: \text{Suc } m \ast z \sim \text{Suc } m \)
using Suc.IH
apply (auto simp: eventually-at-infinity)
apply (rule *)
apply (simp add: field-simps norm-mult norm-power)
done
qed
then show \( \?thesis \)
by (simp add: eventually-at-infinity)
qed

109.2 Convergence to Zero

definition \( Zfun: \: (\forall a \rightarrow b::\text{real-normed-vector}) \rightarrow (\forall \text{ a filter } \rightarrow \text{ bool}) \)
where \( Zfun \: f \: F = (\forall r>0. \: \text{eventually } (\lambda x. \: \text{norm } (f \: x) < r) \: F) \)
lemma \( Zfun1: \: (\forall r. \: 0 < r \implies \text{eventually } (\lambda x. \: \text{norm } (f \: x) < r) \: F) \implies Zfun \: f \: F \)
by (simp add: \text{Zfun-def})

lemma \( ZfunD: \: Zfun \: f \: F \implies 0 < r \implies \text{eventually } (\lambda x. \: \text{norm } (f \: x) < r) \: F \)
by (simp add: \text{Zfun-def})

lemma \( Zfun-ssubb: \: \text{eventually } (\lambda x. \: f \: x = g \: x) \: F \implies Zfun \: g \: F \implies Zfun \: f \: F \)
unfolding \( \text{Zfun-def} \) by (auto elim!: eventually-rev-mp)

lemma Zfun-zero: \( \text{Zfun} (\lambda x. 0) F \)

unfolding \( \text{Zfun-def} \) by simp

lemma Zfun-norm-iff: \( \text{Zfun} (\lambda x. \text{norm}(f x)) F = \text{Zfun} (\lambda x. f x) F \)

unfolding \( \text{Zfun-def} \) by simp

lemma Zfun-imp-Zfun:
assumes \( f: \text{Zfun} f F \)
and \( g: \text{eventually} (\lambda x. \text{norm}(g x) \leq \text{norm}(f x) * K) F \)
shows \( \text{Zfun} (\lambda x. g x) F \)

proof (cases \( 0 < K \))
case \( K: True \)
show \( ?\text{thesis} \)
proof (rule ZfunI)
  fix \( r :: \text{real} \)
  assume \( 0 < r \)
  then have \( 0 < r / K \) using \( K \) by simp
  then have \( \lambda x. \text{norm}(f x) < r / K \) \( F \)
  using \( \text{ZfunD [OF f]} \) by blast
  with \( g \) show \( \lambda x. \text{norm}(g x) < r \) \( F \)
  proof eventually-elim
    case (elim \( x \))
    then have \( \text{norm}(f x) * K < r \)
    using \( K \) \( \text{norm-ge-zero} \) by (rule mult-left-mono)
    finally show \( ?\text{case} \)
      by (simp add: order-le-less-trans [OF elim(1)])
  qed
  qed
next
case \( False \)
then have \( K: K \leq 0 \) by (simp only: not-less)
show \( ?\text{thesis} \)
proof (rule ZfunI)
  fix \( r :: \text{real} \)
  assume \( 0 < r \)
  from \( g \) show \( \lambda x. \text{norm}(g x) < r \) \( F \)
  proof eventually-elim
    case (elim \( x \))
    also have \( \text{norm}(f x) * K \leq \text{norm}(f x) * 0 \)
    using \( K \) \( \text{norm-ge-zero} \) by (rule mult-left-mono)
    finally show \( ?\text{case} \)
      using \( \langle 0 < r \rangle \) by simp
  qed
  qed
  qed

lemma Zfun-le: \( \text{Zfun} g F \implies \forall x. \text{norm}(f x) \leq \text{norm}(g x) \implies \text{Zfun} f F \)
by (erule Zfun-imp-Zfun [where $K = 1$]) simp

lemma Zfun-add:
  assumes $f$: Zfun $f$ $F$
  and $g$: Zfun $g$ $F$
  shows Zfun $(\lambda x. f x + g x) F$
proof (rule ZfunI)
  fix $r$ :: real
  assume $0 < r$
  then have $r < r / 2$ by simp
  have eventually $(\lambda x. \text{norm}(f x) < r / 2) F$
    using $f$ $r$
  moreover
  have eventually $(\lambda x. \text{norm}(g x) < r / 2) F$
    using $g$ $r$
  ultimately
  show eventually $(\lambda x. \text{norm}(f x + g x) < r) F$
proof eventually-elim
  case (elim $x$)
  have $\text{norm}(f x + g x) \leq \text{norm}(f x) + \text{norm}(g x)$
    by (rule norm-triangle-ineq)
  also have $\ldots < r / 2 + r / 2$
    using $\text{elim}$ by (rule add-strict-mono)
  finally show $\text{?case}$
    by simp
  qed
qed

lemma Zfun-minus:
  Zfun $f$ $F$ $\implies$ Zfun $(\lambda x. - f x) F$
proof
  unfolding Zfun-def by simp

lemma Zfun-diff:
  Zfun $f$ $F$ $\implies$ Zfun $g$ $F$ $\implies$ Zfun $(\lambda x. f x - g x) F$
using Zfun-add [of $f$ $F$ $(\lambda x. - g x)$] by (simp add: Zfun-minus)

lemma (in bounded-linear) Zfun:
  assumes $g$: Zfun $g$ $F$
  shows Zfun $(\lambda x. f (g x)) F$
proof
  obtain $K$ where $\text{norm}(f x) \leq \text{norm} x * K$ for $x$
    using bounded by blast
  then have eventually $(\lambda x. \text{norm}(f (g x)) \leq \text{norm}(g x) * K) F$
    by simp
  with $g$ show $\text{?thesis}$
    by (rule Zfun-imp-Zfun)
  qed

lemma (in bounded-bilinear) Zfun:
  assumes $f$: Zfun $f$ $F$
  and $g$: Zfun $g$ $F$
shows $\text{Zfun } (\lambda x. f x \, ** \, g x) \, F$
proof (rule $\text{ZfunI}$)
fix $r :: \text{real}$
assume $r : 0 < r$
obtain $K$ where $K : 0 < K$
and norm-le: $\text{norm } (x ** y) \leq \text{norm } x \cdot \text{norm } y \cdot K$ for $x \, y$
using pos-bounded by blast
from $K$ have $K' : 0 < \text{inverse } K$
by (rule positive-imp-inverse-positive)
have eventually $(\lambda x. \text{norm } (f x) < r) \, F$
using $f \, r$ by (rule $\text{ZfunD}$)
moreover have eventually $(\lambda x. \text{norm } (g x) < \text{inverse } K) \, F$
using $g \, K'$ by (rule $\text{ZfunD}$)
ultimately show eventually $(\lambda x. \text{norm } (f x ** g x) < r) \, F$
proof eventually-elim
 case (elim $x$)
 have $\text{norm } (f x ** g x) \leq \text{norm } (f x) \cdot \text{norm } (g x) \cdot K$
 by (rule norm-le)
 also have $\text{norm } (f x) \cdot \text{norm } (g x) \cdot K < r \cdot \text{inverse } K \cdot K$
 by (intro mult-strict-right-mono mult-strict-mono' norm-ge-zero elim $K$)
 also from $K$ have $r \cdot \text{inverse } K \cdot K = r$
 by simp
 finally show $?case$.
qed
qed

lemma (in bounded-bilinear) $\text{Zfun-left}$: $\text{Zfun } f \, F \Longrightarrow \text{Zfun } (\lambda x. f x \, ** \, a) \, F$
by (rule bounded-linear-left [THEN bounded-linear.$\text{Zfun}$])

lemma (in bounded-bilinear) $\text{Zfun-right}$: $\text{Zfun } f \, F \Longrightarrow \text{Zfun } (\lambda x. a \, ** \, f x) \, F$
by (rule bounded-linear-right [THEN bounded-linear.$\text{Zfun}$])

lemmas $\text{Zfun-mult} = \text{bounded-bilinear}.$Zfun [OF bounded-bilinear-mult]
lemmas $\text{Zfun-mult-right} = \text{bounded-bilinear}.$Zfun-right [OF bounded-bilinear-mult]
lemmas $\text{Zfun-mult-left} = \text{bounded-bilinear}.$Zfun-left [OF bounded-bilinear-mult]

lemma $\text{tendsto-Zfun-iff}$: $(f \longrightarrow a) \, F = \text{Zfun } (\lambda x. f x - a) \, F$
by (simp only: tendsto-iff Zfun-def dist-norm)

lemma $\text{tendsto-0-le}$:
$(f \longrightarrow 0) \, F \Longrightarrow \text{eventually } ((\lambda x. \text{norm } (g x) \leq \text{norm } (f x) \cdot K) \, F \Longrightarrow (g \longrightarrow 0) \, F)$
by (simp add: Zfun-imp-Zfun tendsto-Zfun-iff)

109.2.1 Distance and norms

lemma $\text{tendsto-dist}$ [tendsto-intros]:
fixes \( l m :: \text{metric-space} \)
assumes \( f :: (\lambda a. f a) \to \text{metric-space} \)
and \( g :: (\lambda a. g a) \to \text{metric-space} \)
shows \( ((\lambda x. \text{dist} (f x) (g x)) \to \text{dist l m}) \)
proof (rule tendstoI)
  fix \( e :: \text{real} \)
  assume \( 0 < e \)
  then have \( e2 :: 0 < e/2 \) by simp
  from tendstoD [OF f e2] tendstoD [OF g e2]
  show eventually \((\lambda x. \text{dist} (\text{dist} (f x) (g x)) (\text{dist l m})) < e) \to \text{F} \)
proof (eventually-elim)
  case (elim x)
  then show \( \text{dist} (\text{dist} (f x) (g x)) (\text{dist l m}) < e \)
  unfolding dist-real-def
  using dist-triangle2 [of f x g x l]
  and dist-triangle2 [of g x l m]
  and dist-triangle3 [of l m f x]
  and dist-triangle3 [of l m g x]
  by arith
qed

lemma continuous-dist[continuous-intros]:
  fixes \( f g :: \to \text{metric-space} \)
  shows \( \text{continuous F f} \to \text{continuous F g} \to \text{continuous F} (\lambda x. \text{dist} (f x) (g x)) \)
  unfolding continuous-def by (rule tendsto-dist)

lemma continuous-on-dist[continuous-intros]:
  fixes \( f g :: \to \text{metric-space} \)
  shows \( \text{continuous-on s f} \to \text{continuous-on s g} \to \text{continuous-on s} (\lambda x. \text{dist} (f x) (g x)) \)
  unfolding continuous-on-def by (auto intro: tendsto-dist)

lemma continuous-at-dist: isCont (dist a) b
  using continuous-on-dist [OF continuous-on-const continuous-on-id] continuous-on-eq-continuous-within by blast

lemma tendsto-norm [tendsto-intros]: \( (\lambda x. \text{norm} (f x)) \to \text{norm a}) \to \text{F} \)
  unfolding norm-conv-dist by (intro tendsto-intros)

lemma continuous-norm [continuous-intros]: continuous \( f \to \text{continuous} \)
  unfolding continuous-def by (rule tendsto-norm)

lemma continuous-on-norm [continuous-intros]:
  continuous-on s \( f \to \text{continuous-on s} \)
  unfolding continuous-on-def by (auto intro: tendsto-norm)
lemma continuous-on-norm-id [continuous-intros]: continuous-on S norm
by (intro continuous-on-id continuous-on-norm)

lemma tendsto-norm-zero: \( f \rightarrow 0 \) \( F \) \( \Rightarrow \) \((\lambda x. \text{norm} (f x)) \rightarrow 0\) \( F \)
by (drule tendsto-norm) simp

lemma tendsto-norm-zero-cancel: \((\lambda x. \text{norm} (f x)) \rightarrow 0\) \( F \) \( \Rightarrow \) \( (f \rightarrow 0) \) \( F \)
unfolding tendsto_iff dist-norm by simp

lemma tendsto-norm-zero-iff: \((\lambda x. \text{norm} (f x)) \rightarrow 0\) \( F \) \( \leftrightarrow \) \((f \rightarrow 0) \) \( F \)
unfolding tendsto_iff dist-norm by simp

lemma tendsto-rabs [tendsto-intros]: \( f \rightarrow l \) \( F \) \( \Rightarrow \) \((\lambda x. |f x|) \rightarrow |l|\) \( F \)
for \( l :: \text{real} \)
by (fold real-norm-def) (rule tendsto-norm)

lemma continuous-rabs [continuous-intros]:
continuous \( F f \) \( \Rightarrow \) continuous \( F \) \((\lambda x. |f x| :: \text{real}|)\)
unfolding real-norm-def[symmetric] by (rule continuous-norm)

lemma continuous-on-rabs [continuous-intros]:
continuous-on \( s f \) \( \Rightarrow \) continuous-on \( s \) \((\lambda x. |f x| :: \text{real}|)\)
unfolding real-norm-def[symmetric] by (rule continuous-on-norm)

lemma tendsto-rabs-zero: \( f \rightarrow (0 :: \text{real}) \) \( F \) \( \Rightarrow \) \((\lambda x. |f x|) \rightarrow 0\) \( F \)
by (fold real-norm-def) (rule tendsto-norm-zero)

lemma tendsto-rabs-zero-cancel: \((\lambda x. |f x|) \rightarrow (0 :: \text{real})\) \( F \) \( \Rightarrow \) \( (f \rightarrow 0) \) \( F \)
by (fold real-norm-def) (rule tendsto-norm-zero-cancel)

lemma tendsto-rabs-zero-iff: \((\lambda x. |f x|) \rightarrow (0 :: \text{real})\) \( F \) \( \leftrightarrow \) \((f \rightarrow 0) \) \( F \)
by (fold real-norm-def) (rule tendsto-norm-zero-iff)

109.3 Topological Monoid

class topological-monoid-add = topological-space + monoid-add +
assumes tendsto-add-Pair: LIM x (nhds a ×F nhds b). \( \text{fst} x + \text{snd} x \Rightarrow \text{nhds} (a + b) \)

class topological-comm-monoid-add = topological-monoid-add + comm-monoid-add

lemma tendsto-add [tendsto-intros]:
fixes a b :: 'a::topological-monoid-add
shows \( f \rightarrow a \) \( F \) \( \Rightarrow \) \( g \rightarrow b \) \( F \) \( \Rightarrow \) \((\lambda x. f x + g x) \rightarrow a + b) \) \( F \)
using filterlim-compose[OF tendsto-add-Pair, of \( \lambda x. (f x, g x) a b F \)]
by (simp add: nhds-prod[symmetric] tendsto-Pair)
lemma continuous-add [continuous-intros]:
  fixes f g :: - ⇒ 'b::topological-monoid-add
  shows continuous F f ⇒ continuous F g ⇒ continuous F (λx. f x + g x)
  unfolding continuous-def by (rule tendsto-add)

lemma continuous-on-add [continuous-intros]:
  fixes f g :: - ⇒ 'b::topological-monoid-add
  shows continuous-on s f ⇒ continuous-on s g ⇒ continuous-on s (λx. f x + g x)
  unfolding continuous-on-def by (auto intro: tendsto-add)

lemma tendsto-add-zero:
  fixes f g :: - ⇒ 'b::topological-monoid-add
  shows (f −→ 0) F ⇒ (g −→ 0) F ⇒ ((λx. f x + g x) −→ 0) F
  by (drule (1) tendsto-add) simp

lemma tendsto-sum [tendsto-intros]:
  fixes f :: 'a ⇒ 'b ⇒ 'c::topological-comm-monoid-add
  assumes (∀i. i ∈ I ⇒ (f i −→ a i) F) ⇒ ((λx. ∑i∈I. f i x) −→ (∑i∈I. a i)) F
  by (induct I rule: infinite-finite-induct) (simp-all add: tendsto-add)

lemma tendsto-null-sum:
  fixes f :: 'a ⇒ 'b ⇒ 'c::topological-comm-monoid-add
  assumes (∀i. i ∈ I ⇒ continuous F (f i)) ⇒ continuity F (λx. ∑i∈I. f i x)
  using tendsto-sum [of I λx y. f y x λx. 0] assms by simp

lemma continuous-sum [continuous-intros]:
  fixes f :: 'a ⇒ 'b::t2-space ⇒ 'c::topological-comm-monoid-add
  shows (∀i. i ∈ I ⇒ continuous F (f i)) ⇒ continuous F (λx. ∑i∈I. f i x)
  unfolding continuous-def by (rule tendsto-sum)

lemma continuous-on-sum [continuous-intros]:
  fixes f :: 'a ⇒ 'b::topological-space ⇒ 'c::topological-comm-monoid-add
  shows (∀i. i ∈ I ⇒ continuous-on S (f i)) ⇒ continuous-on S (λx. ∑i∈I. f i x)
  unfolding continuous-on-def by (auto intro: tendsto-sum)

instance nat :: topological-comm-monoid-add
  by standard
  (simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal)

instance int :: topological-comm-monoid-add
  by standard
  (simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal)
109.3.1 Topological group

class topological-group-add = topological-monoid-add + group-add +
  assumes tendsto-uminus-nhds: (uminus ----> a) (nhds a)
begin

lemma tendsto-minus [tendsto-intros]: (f ----> a) F ==> ((λx. - f x) ----> - a) F
  by (rule filterlim-compose[OF tendsto-uminus-nhds])

end

class topological-ab-group-add = topological-group-add + ab-group-add

instance topological-ab-group-add < topological-comm-monoid-add ..

lemma continuous-minus [continuous-intros]: continuous F f ==> continuous F (λx. - f x)
  for f :: 'a::t2-space => 'b::topological-group-add
unfolding continuous-def by (rule tendsto-minus)

lemma continuous-on-minus [continuous-intros]: continuous-on s f ==> continuous-on s (λx. - f x)
  for f :: 'a::topological-group-add
unfolding continuous-on-def by (auto intro: tendsto-minus)

lemma tendsto-minus-cancel: ((λx. - f x) ----> - a) F ==> (f ----> a) F
  for a :: 'a::topological-group-add
by (drule tendsto-minus) simp

lemma tendsto-minus-cancel-left: (f ----> - (y::::topological-group-add)) F <= ((λx. - f x) ----> y) F

lemma tendsto-diff [tendsto-intros]:
  fixes a b :: 'a::topological-group-add
  shows (f ----> a) F ==> (g ----> b) F ==> ((λx. f x - g x) ----> a - b) F
using tendsto-add[of f a F λx. - g x - b] by (simp add: tendsto-minus)

lemma continuous-diff [continuous-intros]:
  fixes f g :: 'a::t2-space => 'b::topological-group-add
  shows continuous F f ==> continuous F g ==> continuous F (λx. f x - g x)
unfolding continuous-def by (rule tendsto-diff)

lemma continuous-on-diff [continuous-intros]:
  fixes f g :: 'a::topological-group-add
  shows continuous-on s f ==> continuous-on s g ==> continuous-on s (λx. f x - g x)
unfolding continuous-on-def by (auto intro: tendsto-diff)
lemma continuous-on-op-minus: continuous-on (s::'a::topological-group-add set) 
(\((-\) x)
by (rule continuous-intros | simp)+

instance real-normed-vector < topological-ab-group-add 
proof
  fix a b :: 'a
  show ((\x. fst x + snd x) \longrightarrow a + b) (nhds a \times_F nhds b)
    unfolding tendsto-Zfun-iff add-diff-add
      using tendsto-fst[OF filterlim-ident, of (a,b)] tendsto-snd[OF filterlim-ident, of (a,b)]
        by (intro Zfun-add)
  simp add: tendsto-Zfun-iff
qed

lemmas real-tendsto-sandwich = tendsto-sandwich[where 'a=real]

109.3.2 Linear operators and multiplication

lemma linear-times [simp]: linear (\(\lambda x. c * x\))
  for c :: 'a::real-algebra
  by (auto simp: linearI distrib-left)

lemma (in bounded-linear) tendsto: (g \longrightarrow a) F \Longrightarrow ((\(\lambda x. f (g x)\)) \longrightarrow f a) F
  by (simp only: tendsto-Zfun-iff diff [symmetric] Zfun)

lemma (in bounded-linear) continuous: continuous F g \Longrightarrow continuous F (\(\lambda x. f (g x)\))
  using tendsto[of g - F] by (auto simp: continuous-def)

lemma (in bounded-linear) continuous-on: continuous-on s g \Longrightarrow continuous-on s (\(\lambda x. f (g x)\))
  using tendsto[of g] by (auto simp: continuous-on-def)

lemma (in bounded-linear) tendsto-zero: (g \longrightarrow 0) F \Longrightarrow ((\(\lambda x. f (g x)\)) \longrightarrow 0) F
  by (drule tendsto) (simp only: zero)

lemma (in bounded-bilinear) tendsto:
(f \longrightarrow a) F \Longrightarrow (g \longrightarrow b) F \Longrightarrow ((\(\lambda x. f x ** g x\)) \longrightarrow a ** b) F
  by (simp only: tendsto-Zfun-iff prod-diff-prod Zfun Zfun-left Zfun-right)

lemma (in bounded-bilinear) continuous:
continuous \( F \ f \implies \) continuous \( F \ g \implies \) continuous \( F \ (\lambda x. f x \ ** g x) \)
using \( \text{tendsto}[of f - F g] \) by (auto simp: continuous-def)

lemma (in bounded-bilinear) continuous-on:
continuous-on \( s \ f \implies \) continuous-on \( s \ g \implies \) continuous-on \( s \ (\lambda x. f x \ ** g x) \)
using \( \text{tendsto}[of f - g] \) by (auto simp: continuous-on-def)

lemma (in bounded-bilinear) tendsto-zero:
assumes \( f \to 0 \)
and \( g \to 0 \)
shows \( ((\lambda x. f x \ ** g x) \to 0) \)
using \( \text{tendsto}[OF f g] \) by (simp add: zero-left)

lemma (in bounded-bilinear) tendsto-left-zero:
\( f \to 0 \)
shows \( ((\lambda x. c \ ** f x) \to 0) \)
by (rule bounded-linear.tendsto-zero [OF bounded-linear-left])

lemma (in bounded-bilinear) tendsto-right-zero:
\( f \to 0 \)
shows \( ((\lambda x. c \ ** f x) \to 0) \)
by (rule bounded-linear.tendsto-zero [OF bounded-linear-right])

lemmas tendsto-of-real [tendsto-intros] =
bounded-linear.tendsto [OF bounded-linear-of-real]

lemmas tendsto-scaleR [tendsto-intros] =
bounded-bilinear.tendsto [OF bounded-bilinear-scaleR]

Analogous type class for multiplication

class topological-semigroup-mult = topological-space + semigroup-mult +
  assumes tendsto-mult-Pair: \( \text{LIM} \ x \ (\text{nhds} \ a \times F \ \text{nhds} \ b) \). \( \text{fst} \ x \ ** \text{snd} \ x :: \text{nhds} \ (a \ast b) \)

instance real-normed-algebra < topological-semigroup-mult
proof
  fix \( a \ b :: 'a \)
  show \( ((\lambda x. \text{fst} \ x \ ** \text{snd} \ x) \to a \ast b) \ (\text{nhds} \ a \times F \ \text{nhds} \ b) \)
    unfolding \( \text{nhds-prod}[\text{symmetric}] \)
    using \( \text{tendsto-fst}[\text{OF} \ \text{filterlim-ident}, \ \text{of} (a,b)] \) \( \text{tendsto-snd}[\text{OF} \ \text{filterlim-ident}, \ \text{of} (a,b)] \)
    by (simp add: bounded-bilinear.tendsto [OF bounded-bilinear-mult])
qed

lemma tendsto-mult [tendsto-intros]:
  fixes \( a \ b :: 'a::\text{topological-semigroup-mult} \)
  shows \( (f \to a) \ F \implies (g \to b) \ F \implies ((\lambda x. f x \ ** g x) \to a \ast b) \ F \)
  using \( \text{filterlim-compose}[\text{OF} \ \text{tendsto-mult-Pair}, \ \text{of} \ (a,b)] \) \( \text{tendsto-Pair} \)
  by (simp add: \( \text{nhds-prod}[\text{symmetric}] \) tendsto-Pair)

lemma tendsto-mult-left: \( (f \to l) \ F \implies ((\lambda x. c \ ** (f x)) \to c \ast l) \ F \)
for $c :: 'a::topological-semigroup-mult$
by (rule tendsto-mult [OF tendsto-const])

lemma tendsto-mult-right: $(f \to l) \Rightarrow ((\lambda x. (f x) * c) \to l * c) F$
for $c :: 'a::topological-semigroup-mult$
by (rule tendsto-mult [OF - tendsto-const])

lemma tendsto-mult-left-iff [simp]:
$c \neq 0 \Rightarrow \text{tendsto} (\lambda x. c * f x) (c * l) F \iff \text{tendsto} f l F$
for $c :: 'a::\{topological-semigroup-mult, field\}$
by (auto simp: tendsto-mult-left dest: tendsto-mult-left [where $c = 1/c$])

lemma tendsto-mult-right-iff [simp]:
$c \neq 0 \Rightarrow \text{tendsto} (\lambda x. f x * c) (l * c) F \iff \text{tendsto} f l F$
for $c :: 'a::\{topological-semigroup-mult, field\}$
by (auto simp: tendsto-mult-right dest: tendsto-mult-left [where $c = 1/c$])

lemma tendsto-zero-mult-left-iff [simp]:
fixes $c :: 'a::\{topological-semigroup-mult, field\}$ assumes $c \neq 0$
shows $(\lambda n. c * a n) \to 0 \iff a \to 0$
using assms tendsto-mult-left tendsto-mult-left-iff by fastforce

lemma tendsto-zero-mult-right-iff [simp]:
fixes $c :: 'a::\{topological-semigroup-mult, field\}$ assumes $c \neq 0$
shows $(\lambda n. a n * c) \to 0 \iff a \to 0$
using assms tendsto-mult-right tendsto-mult-right-iff by fastforce

lemma tendsto-zero-divide-iff [simp]:
fixes $c :: 'a::\{topological-semigroup-mult, field\}$ assumes $c \neq 0$
shows $(\lambda n. a n / c) \to 0 \iff a \to 0$
using tendsto-zero-mult-right-iff [of 1/c a] assms by (simp add: field-simps)

lemma lim-const-over-n [tendsto-intros]:
fixes $a :: 'a::real-normed-field$
shows $(\lambda n. a / of-nat n) \to 0$
using tendsto-const [OF tendsto-const [of a]] lim-1-over-n by simp

lemmas continuous-of-real [continuous-intros] =
bounded-linear.continuous [OF bounded-linear-of-real]
lemmas continuous-scaleR [continuous-intros] =
bounded-bilinear.continuous [OF bounded-bilinear-scaleR]
lemmas continuous-mult [continuous-intros] =
bounded-bilinear.continuous [OF bounded-bilinear-mult]
lemmas continuous-on-of-real [continuous-intros] =
bounded-linear.continuous-on [OF bounded-linear-of-real]
lemmas continuous-on-scaleR [continuous-intros] =
bounded-bilinear.continuous-on [OF bounded-bilinear-scaleR]
lemmas continuous-on-mult: \[ \text{fixes } c :: 'a::real-normed-algebra \]
shows \( \text{continuous } F f \implies \text{continuous } (\lambda x. c \ast f x) \)
by (rule continuous-mult [OF continuous-const])

lemma continuous-mult-right:
fixes \( c :: 'a::real-normed-algebra \)
shows \( \text{continuous } F f \implies \text{continuous } (\lambda x. f x \ast c) \)
by (rule continuous-mult [OF - continuous-const])

lemma continuous-on-mult-left:
fixes \( c :: 'a::real-normed-algebra \)
shows \( \text{continuous-on } s f \implies \text{continuous-on } s (\lambda x. c \ast f x) \)
by (rule continuous-on-mult [OF continuous-on-const])

lemma continuous-on-mult-right:
fixes \( c :: 'a::real-normed-algebra \)
shows \( \text{continuous-on } s f \implies \text{continuous-on } s (\lambda x. f x \ast c) \)
by (rule continuous-on-mult [OF - continuous-on-const])

lemma continuous-on-mult-const [simp]:
fixes \( c :: 'a::real-normed-algebra \)
shows \( \text{continuous-on } s (\ast c) \)
by (intro continuous-on-mult-left continuous-on-id)

lemma tendsto-divide-zero:
fixes \( c :: 'a::real-normed-field \)
shows \( (f \longrightarrow 0) F \implies ((\lambda x. f x / c) \longrightarrow 0) F \)
by (cases \( c=0 \)) (simp-all add: divide-inverse tendsto-mult-left-zero)

lemma tendsto-power [tendsto-intros]: \( (f \longrightarrow a) F \implies ((\lambda x. f x ^ n) \longrightarrow a ^ n) F \)
for \( f :: 'a \Rightarrow 'b::\{\text{power,real-normed-algebra}\} \)
by (induct \( n \)) (simp-all add: tendsto-mult)
lemma tendsto-null-power: \[ ([f \rightarrow 0]) F; 0 < n ] \implies ((\lambda x. f x \cdot n) \rightarrow 0) F \]
for \( f :: 'a \Rightarrow 'b::{\text{power,real-normed-algebra-1}} \)
using tendsto-power [of \( F \) \( n \)] by (simp add: power-0-left)

lemma continuous-power [continuous-intros]: continuous \( F f \implies \) continuous \( F \)
\( (\lambda x. (f x)^n) \)
for \( f :: 'a::\text{t2-space} \Rightarrow 'b::\{}\text{power,real-normed-algebra}\}\)
unfolding continuous-def by (rule tendsto-power)

lemma continuous-on-power [continuous-intros]:
fixes \( f :: - \Rightarrow 'b::\{}\text{power,real-normed-algebra}\}\)
shows continuous-on \( s f \implies \) continuous-on \( s (\lambda x. (f x)^n) \)
unfolding continuous-on-def by (auto intro: tendsto-power)

lemma tendsto-prod [tendsto-intros]:
fixes \( f :: 'a \Rightarrow 'b::\{}\text{real-normed-algebra,comm-ring-1}\}\)
shows \((\wedge i. i \in S \Rightarrow (f i \rightarrow L i)) \Rightarrow ((\lambda x. \prod i \in S. f i x) \rightarrow (\prod i \in S. L i)) \)
by (induct \( S \) rule: infinite-finite-induct) (simp-all add: tendsto-mult)

lemma continuous-prod [continuous-intros]:
fixes \( f :: - \Rightarrow 'b::\{}\text{real-normed-algebra,comm-ring-1}\}\)
shows \((\wedge i. i \in S \Rightarrow \text{continuous} f (f i)) \Rightarrow \text{continuous} F (\lambda x. \prod i \in S. f i x) \)
unfolding continuous-def by (rule tendsto-prod)

lemma continuous-on-prod [continuous-intros]:
fixes \( f :: 'a \Rightarrow - \Rightarrow 'c::\{}\text{real-normed-algebra,comm-ring-1}\}\)
shows \((\wedge i. i \in S \Rightarrow \text{continuous-on} s f (f i)) \Rightarrow \text{continuous-on} s (\lambda x. \prod i \in S. f i x) \)
unfolding continuous-on-def by (auto intro: tendsto-prod)

lemma tendsto-of-real-iff:
\((\lambda x. \text{of-real} (f x) :: 'a::\text{real-normed-div-algebra}) \rightarrow \text{of-real} c) \ F \leftrightarrow (f \rightarrow c) \ F \)
unfolding tendsto-iff by simp

lemma tendsto-add-const-iff:
\((\lambda x. c + f x :: 'a::\text{topological-group-add}) \rightarrow c + d) \ F \leftrightarrow (f \rightarrow d) \ F \)
using tendsto-add[OF tendsto-const[of \( c \), of \( f \) \( d \)]
and tendsto-add[OF tendsto-const[of \( -c \), of \( \lambda x. c + f x + d \)] by auto

class topological-monoid-mult = topological-semigroup-mult + monoid-mult
class topological-comm-monoid-mult = topological-monoid-mult + comm-monoid-mult

lemma tendsto-power-strong [tendsto-intros]:
fixes \( f :: - \Rightarrow 'b :: \text{topological-monoid-mult} \)
assumes \((f \rightarrow a) \ F (g \rightarrow b) \ F \)
shows \((\lambda x. f x ^\cdot g x) \rightarrow a ^\cdot b) \ F \)
proof –
  have \((\lambda x. f x \sim b) \longrightarrow a \sim b) F\)
    by (induction b) (auto intro: tendsto-intros assms)
  also from assms(2) have eventually \((\lambda x. g x = b) F\)
    by (simp add: nhds-discrete filterlim-principal)
  hence eventually \((\lambda x. f x \sim b = f x \sim g x) F\)
    by eventually-elim simp
  hence \((\lambda x. f x \sim b = f x \sim g x) F\)
    by (intro filterlim-cong refl)
  finally show \(?thesis\).

qed

lemma \(\text{continuous-mult}'\) [continuous-intros]:
  fixes f g :: \(-\Rightarrow \)\(\cdot\)
  shows \(\text{continuous } F f \implies \text{continuous } F g \implies \text{continuous } F (\lambda x. f x \ast g x)\)
    unfolding \(\text{continuous-def}\) by (rule tendsto-mult)

lemma \(\text{continuous-power}'\) [continuous-intros]:
  fixes f :: \(-\Rightarrow \)\(\cdot\)
  shows \(\text{continuous } F f \implies \text{continuous } F g \implies \text{continuous } F (\lambda x. f x ^ g x)\)
    unfolding \(\text{continuous-def}\) by (rule tendsto-power-strong)

lemma \(\text{continuous-on-mult}'\) [continuous-intros]:
  fixes f g :: \(-\Rightarrow \)\(\cdot\)
  shows \(\text{continuous-on } A f \implies \text{continuous-on } A g \implies \text{continuous-on } A (\lambda x. f x \ast g x)\)
    unfolding \(\text{continuous-on-def}\) by (auto intro: tendsto-mult)

lemma \(\text{continuous-on-power}'\) [continuous-intros]:
  fixes f :: \(-\Rightarrow \)\(\cdot\)
  shows \(\text{continuous-on } A f \implies \text{continuous-on } A g \implies \text{continuous-on } A (\lambda x. f x ^ g x)\)
    unfolding \(\text{continuous-on-def}\) by (auto intro: tendsto-power-strong)

lemma tendsto-mult-one:
  fixes f g :: \(-\Rightarrow \)\(\cdot\)
  shows \((f \longrightarrow 1) F \implies (g \longrightarrow 1) F \implies ((\lambda x. f x \ast g x) \longrightarrow 1) F\)
    by (drule (1) tendsto-mult) simp

lemma tendsto-prod' [tendsto-intros]:
  fixes f :: \('a \Rightarrow \cdot\)\('b\)\('c\)
  shows \((\Lambda i. i \in I \implies (f \longrightarrow a i) F) \implies ((\lambda x. \prod i \in I. f i x) \longrightarrow (\prod i \in I. a i)) F\)
    by (induct I rule: infinite-finite-induct) (simp-all add: tendsto-mult)

lemma tendsto-one-prod':
  fixes f :: \('a \Rightarrow \cdot\)\('b\)\('c\)
  assumes \((\lambda i. i \in I \implies ((\lambda x. f x i) \longrightarrow 1) F)\)
  shows \((\lambda i. \prod (f i I) \longrightarrow 1) F\)
using tendsto-prod' [of I λx y. f y x λx. 1] assms by simp

lemma LIMSEQ-prod-0:
  fixes f :: nat ⇒ 'a::{semidom,topological-space}
  assumes f i = 0
  shows (λn. prod f {..<n}) −−−−→ 0
proof (subst tendsto-cong)
  show ∀F n in sequentially. prod f {..<n} = 0
    using assms eventually-at-top-linorder by auto
qed auto

lemma LIMSEQ-prod-nonneg:
  fixes f :: nat ⇒ 'a::{linordered-semidom,linorder-topology}
  assumes 0: ∀n. 0 ≤ f n and a: (λn. prod f {..<n}) −−−−→ a
  shows a ≥ 0
  by (simp add: 0 prod-nonneg LIMSEQ-le-const [OF a])

lemma continuous-prod' [continuous-intros]:
  fixes 'a => 'b::t2-space => 'c::topological-comm-monoid-mult
  shows (∀i. i ∈ I ⇒ continuous F (f i)) ⇒ continuous F (λx. ∏i∈I. f i x)
unfolding continuous-def by (rule tendsto-prod')

lemma continuous-on-prod' [continuous-intros]:
  fixes 'a => 'b::topological-space => 'c::topological-comm-monoid-mult
  shows (∀i. i ∈ I ⇒ continuous-on S (f i)) ⇒ continuous-on S (λx. ∏i∈I. f i x)
unfolding continuous-on-def by (auto intro: tendsto-prod')

instance nat :: topological-comm-monoid-mult
  by standard
  (simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal)

instance int :: topological-comm-monoid-mult
  by standard
  (simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal)

class comm-real-normed-algebra-1 = real-normed-algebra-1 + comm-monoid-mult

class comm-real-normed-algebra-1

instance real-normed-field
begin

subclass comm-real-normed-algebra-1
proof
  from norm-mult[of 1 :: 'a 1] show norm 1 = 1 by simp
qed (simp-all add: norm-mult)

end
109.3.3 Inverse and division

**Lemma (in bounded-bilinear) Zfun-prod-Bfun:**
assumes \( f : \text{Zfun } f \) F
and \( g : \text{Bfun } g \) F
shows \( \text{Zfun } (\lambda x. f \times g) \) F

**Proof** –
- obtain \( K \) where \( K \): \( 0 \leq K \)
  - and \( \text{norm-le: } \forall x y. \text{norm } (x \times y) \leq \text{norm } x \times \text{norm } y \times K \)
  - using \( \text{nonneg-bounded by blast} \)
- obtain \( B \) where \( B : 0 < B \)
  - and \( \text{norm-g: } \text{eventually } (\lambda x. \text{norm } (f \times g) \leq \text{norm } (f) \times (B \times K)) \) F
  - using \( g \) by (rule BfunE)
  - have \( \text{eventually } (\lambda x. \text{norm } (f \times g) \leq \text{norm } (f) \times (B \times K)) \) F
  - using \( \text{norm-g proof eventually-elim} \)
    - case (elim \( x \))
      - have \( \text{norm } (f \times g) \leq \text{norm } (f) \times \text{norm } (g) \times K \)
      - by (rule \( \text{norm-le} \))
      - also have \( \ldots \leq \text{norm } (f) \times (B \times K) \)
      - by (intro \( \text{mult-mono’ order-refl norm-g norm-ge-zero mult-nonneg-nonneg K elim} \))
      - also have \( \ldots = \text{norm } (f) \times (B \times K) \)
      - by (rule \( \text{mult.assoc} \))
      - finally show \( \text{norm } (f \times g) \leq \text{norm } (f) \times (B \times K) \).
  - qed
- with \( f \) show \(? \text{thesis} \)
  - by (rule \( \text{Zfun-imp-Zfun} \))
- qed

**Lemma (in bounded-bilinear) Bfun-prod-Zfun:**
assumes \( f : \text{Bfun } f \) F
and \( g : \text{Zfun } g \) F
shows \( \text{Zfun } (\lambda x. f \times g) \) F
using \( \text{flip } g f \) by (rule bounded-bilinear.Zfun-prod-Bfun)

**Lemma Bfun-inverse:**
fixes \( a :: \text{‘a::real-normed-div-algebra} \)
assumes \( f: (f \longrightarrow a) \) F
assumes \( a: a \neq 0 \)
shows \( \text{Bfun } (\lambda x. \text{inverse } (f x)) \) F

**Proof** –
- from \( a \) have \( 0 < \text{norm } a \) by simp
- then have \( \exists r > 0. r < \text{norm } a \) by (rule dense)
- then obtain \( r \) where \( r1: 0 < r \) and \( r2: r < \text{norm } a \)
  - by blast
  - have \( \text{eventually } (\lambda x. \text{dist } (f \times a < r)) \) F
    - using \( \text{tendstoD } [OF f r1] \) by blast
  - then have \( \text{eventually } (\lambda x. \text{norm } \text{inverse } (f x) \leq \text{inverse } (\text{norm } a - r)) \) F
  - proof eventually-elim
    - case (elim \( x \))
then have 1: \( \| f(x) - a \| < r \)
  by (simp add: dist-norm)
then have 2: \( f(x) \neq 0 \) using \( r^2 \) by auto
then have \( \| \text{inverse} (f(x)) \| = \text{inverse} (\| f(x) \|) \)
  by (rule nonzero-norm-inverse)
also have \( \ldots \leq \text{inverse} (\| a - r \|) \)
proof (rule le_imp_inverse_le)
  show \( 0 < \| a - r \| \)
    using \( r^2 \) by simp
  have \( \| f(x) \| - \| a \| \leq \| f(x) - a \| \)
    by (rule norm_triangle_ineq2)
  also have \( \ldots < r \) using 1.
finally show \( \| a - r \| \leq \| f(x) \| \)
  by simp
qed
finally show \( \text{thesis} \)
  by (rule BfunI)
qed

lemma tendsto_inverse [tendsto-intros]:
  fixes \( a :: 'a::real-normed-div-algebra \)
  assumes \( f : (f ---\to a) F \)
  and \( a: a \neq 0 \)
  shows \( ((\lambda x. \text{inverse} (f x)) ---\to \text{inverse} a) F \)
proof
  from \( a \)
  have \( 0 < \| a \| \)
    by simp
with \( f \)
  have eventually \( (\lambda x. \text{dist} (f x) \leq \| a \|) F \)
    by (rule tendstoD)
then have eventually \( (\lambda x. f x \neq 0) F \)
  unfolding dist-norm by (auto elim!: eventually_mono)
with \( a \)
  have eventually \( (\lambda x. \text{inverse} (f x) - \text{inverse} a = \)
    \(- (\text{inverse} (f x) * (f x - a) * \text{inverse} a)) F \)
  by (auto elim!: eventually_mono simp: inverse_diff_inverse)
moreover have \( \text{Zfun} (\lambda x. \text{inverse} (f x) * (f x - a) * \text{inverse} a) F \)
  by (intro \text{Zfun_minus} \text{Zfun_mult_left}
    \text{bounded-bilinear} \text{Bfun_prod_Zfun} \text{OF} \text{bounded-bilinear_mult}
    \text{Bfun_inverse} \text{OF} \text{f} \text{f} \text{[unfolded tendsto-Zfun_iff]}])
ultimately show \( \text{thesis} \)
  unfolding tendsto_Zfun_iff by (rule \text{Zfun_ssubst})
qed

lemma continuous_inverse:
  fixes \( f :: 'a::t2-space \Rightarrow 'b::real-normed-div-algebra \)
  assumes continuous \( F f \)
  and \( f (\text{Lim} F (\lambda x. x)) \neq 0 \)
  shows continuous \( F (\lambda x. \text{inverse} (f x)) \)
using assms unfolding continuous-def by (rule tendsto-inverse)

lemma continuous-at-within-inverse[continuous-intros]:
  fixes f :: 'a::t2-space ⇒ 'b::real-normed-div-algebra
  assumes continuous (at a within s) f
       and f a ≠ 0
  shows continuous (at a within s) (λx. inverse (f x))
  using assms unfolding continuous-within by (rule tendsto-inverse)

lemma continuous-on-inverse[continuous-intros]:
  fixes f :: 'a::topological-space ⇒ 'b::real-normed-div-algebra
  assumes continuous-on s f
       and ∀x ∈ s. f x ≠ 0
  shows continuous-on s (λx. inverse (f x))
  using assms unfolding continuous-on-def by (blast intro: tendsto-inverse)

lemma tendsto-divide [tendsto-intros]:
  fixes a b :: 'a::real-normed-field
  shows (f −→ a) F ⇒ (g −→ b) F ⇒ b ≠ 0 ⇒ ((λx. f x / g x) −→ a / b) F
  by (simp add: tendsto-mult tendsto-inverse divide-inverse)

lemma continuous-divide:
  fixes f g :: 'a::t2-space ⇒ 'b::real-normed-field
  assumes continuous F f
       and continuous F g
       and g (Lim F (λx. x)) ≠ 0
  shows continuous F (λx. (f x) / (g x))
  using assms unfolding continuous-def by (rule tendsto-divide)

lemma continuous-at-within-divide[continuous-intros]:
  fixes f g :: 'a::t2-space ⇒ 'b::real-normed-field
  assumes continuous (at a within s) f continuous (at a within s) g
       and a ≠ 0
  shows continuous (at a within s) (λx. (f x) / (g x))
  using assms unfolding continuous-within by (rule tendsto-divide)

lemma isCont-divide[continuous-intros, simp]:
  fixes f g :: 'a::t2-space ⇒ 'b::real-normed-field
  assumes isCont f a isCont g a g a ≠ 0
  shows isCont (λx. (f x) / (g x)) a
  using assms unfolding continuous-at by (rule tendsto-divide)

lemma continuous-on-divide[continuous-intros]:
  fixes f :: 'a::topological-space ⇒ 'b::real-normed-field
  assumes continuous-on s f continuous-on s g
       and ∀x ∈ s. g x ≠ 0
  shows continuous-on s (λx. (f x) / (g x))
  using assms unfolding continuous-on-def by (blast intro: tendsto-divide)
lemma tendsto-power-int [tendsto-intros]:
  fixes a :: 'a::real-normed-div-algebra
  assumes f: (f --- a) F  
  and a: a ≠ 0
  shows ((λx. power-int (f x) n) --- power-int a n) F
  using assms by (cases n rule: int-cases4) (auto intro!: tendsto-intros simp: power-int-minus)

lemma continuous-power-int:
  fixes f :: 'a::t2-space ⇒ 'b::real-normed-div-algebra
  assumes continuous F f  
  and f (Lim F (λx. x)) ≠ 0
  shows continuous F (λx. power-int (f x) n)
  using assms unfolding continuous-def by (rule tendsto-power-int)

lemma continuous-at-within-power-int [continuous-intros]:
  fixes f :: 'a::t2-space ⇒ 'b::real-normed-div-algebra
  assumes continuous (at a within s) f  
  and f a ≠ 0
  shows continuous (at a within s) (λx. power-int (f x) n)
  using assms unfolding continuous-within by (rule tendsto-power-int)

lemma continuous-on-power-int [continuous-intros]:
  fixes f :: 'a::topological-space ⇒ 'b::real-normed-div-algebra
  assumes continuous-on s f and ∀x∈s. f x ≠ 0
  shows continuous-on s (λx. power-int (f x) n)
  using assms unfolding continuous-on-def by (blast intro: tendsto-power-int)

lemma tendsto-power-int' [tendsto-intros]:
  fixes a :: 'a::real-normed-div-algebra
  assumes f: (f --- a) F  
  and a: a ≠ 0 ∨ n ≥ 0
  shows ((λx. power-int (f x) n) --- power-int a n) F
  using assms by (cases n rule: int-cases4) (auto intro!: tendsto-intros simp: power-int-minus)

lemma tendsto-sgn [tendsto-intros]: (f --- l) F --- l ≠ 0 --- ((λx. sgn (f x)) --- sgn l) F
  for l :: 'a::real-normed-vector
  unfolding sgn-div-norm by (simp add: tendsto-intros)

lemma continuous-sgn:
  fixes f :: 'a::t2-space ⇒ 'b::real-normed-vector
  assumes continuous F f  
  and f (Lim F (λx. x)) ≠ 0
  shows continuous F (λx. sgn (f x))
  using assms unfolding continuous-def by (rule tendsto-sgn)
lemma continuous-at-within-sgn[continuous-intros]:
  fixes f :: 'a::t2-space ⇒ 'b::real-normed-vector
  assumes continuous (at a within s) f
      and f a ≠ 0
  shows continuous (at a within s) (λx. sgn (f x))
  using assms unfolding continuous-within by (rule tendsto-sgn)

lemma isCont-sgn[continuous-intros]:
  fixes f :: 'a::t2-space ⇒ 'b::real-normed-vector
  assumes isCont f a
      and f a ≠ 0
  shows isCont (λx. sgn (f x)) a
  using assms unfolding continuous-at by (rule tendsto-sgn)

lemma continuous-on-sgn[continuous-intros]:
  fixes f :: 'a::topological-space ⇒ 'b::real-normed-vector
  assumes continuous-on s f
      and ∀x∈s. f x ≠ 0
  shows continuous-on s (λx. sgn (f x))
  using assms unfolding continuous-on-def by blast intro: tendsto-sgn

lemma filterlim-at-infinity:
  fixes f :: 'a::real-normed-vector
  assumes 0 ≤ c
  shows (LIM x F. f x :> at-infinity) ←→ (∀r>c. eventually (λx. r ≤ norm (f x)) F)
  unfolding filterlim-iff eventually-at-infinity
proof safe
  fix P :: 'a ⇒ bool
  fix b
  assume *: ∀r>c. eventually (λx. r ≤ norm (f x)) F
  assume P: ∀x. b ≤ norm x → P x
  have max b (c + 1) > c by auto
  with * have eventually (λx. max b (c + 1) ≤ norm (f x)) F
    by auto
  then show eventually (λx. P (f x)) F
proof eventually-elim
    case (elim x)
    with P show P (f x) by auto
qed

lemma filterlim-at-infinity-imp-norm-at-top:
  fixes F
  assumes filterlim f at-infinity F
  shows filterlim (λx. norm (f x)) at-top F
proof −
{  
  fix r :: real
have \( \forall F \ x \in F. \ r \leq \text{norm} (f \ x) \) using \text{filterlim-at-infinity}[\text{of } 0 f F] \ assms

by (cases \( r > 0 \))

(auto simp: not-less intro: always-eventually order.trans[OF - norm-ge-zero])

\}

thus \text{?thesis} by (auto simp: \text{filterlim-at-top})

qed

lemma \text{filterlim-norm-at-top-imp-at-infinity}:

fixes \( F \)

assumes \text{filterlim}\ (\lambda x. \text{norm} (f \ x)) \text{ at-top } F

shows \text{filterlim } f \text{ at-infinity } F

using \text{filterlim-at-infinity}[\text{of } 0 f F] \ assms by (auto simp: \text{filterlim-at-top})

lemma \text{filterlim-norm-at-top}: \text{filterlim norm at-top at-infinity}

by (rule \text{filterlim-at-infinity-imp-norm-at-top}) (rule \text{filterlim-ident})

lemma \text{filterlim-at-infinity-conv-norm-at-top}:

\text{filterlim } f \text{ at-infinity } G \leftrightarrow \text{filterlim}\ (\lambda x. \text{norm} (f \ x)) \text{ at-top } G

by (auto simp: \text{filterlim-at-infinity}[\text{of } 0 f F] \text{滤器lim-at-top-gt}[\text{of } - - 0])

lemma \text{eventually-not-equal-at-infinity}:

\text{eventually}\ (\lambda x. x \neq (a :: 'a :: \{real-normed-vector\})) \text{ at-infinity}

proof –

from \text{filterlim-norm-at-top}[where 'a = 'a]

have \( \forall F \ x \in \text{at-infinity}. \ \text{norm} \ a < \text{norm} (x::'a) \) by (auto simp: \text{filterlim-at-top-dense})

thus \text{?thesis} by \text{eventually-elim auto}

qed

lemma \text{filterlim-int-of-nat-at-topD}:

fixes \( F \)

assumes \text{filterlim}\ (\lambda x. f (\text{int} x)) \text{ F at-top}

shows \text{filterlim } f \text{ F at-top}

proof –

have \text{filterlim}\ (\lambda x. f (\text{int} (\text{nat} x))) \text{ F at-top}

by (rule \text{filterlim-compose}[\text{OF assms filterlim-nat-sequentially}])

also have \text{?this} \leftrightarrow \text{filterlim } f \text{ F at-top}

by (intro \text{filterlim-cong refl eventually-mono} [\text{OF eventually-ge-at-top}[\text{of } - : \text{int}]])

auto

finally show \text{?thesis}.

qed

lemma \text{filterlim-int-sequentially} [\text{tendsto-intros}]:

\text{filterlim int at-top sequentially}

unfolding \text{filterlim-at-top}

proof

fix \( C :: \text{int} \)

show \text{eventually}\ (\lambda n. \text{int} \ n \geq C) \text{ at-top}

using \text{eventually-ge-at-top}[\text{of } \text{nat} \ [C]] \text{ by eventually-elim linarith}

qed
lemma filterlim-real-of-int-at-top [tendsto-intros]:
filterlim real-of-int at-top at-top
unfolding filterlim-at-top
proof
  fix C :: real
  show eventually (\lambda n. real-of-int n ≥ C) at-top
  using eventually-ge-at-top[of ⌈C⌉] by eventually-elim linarith
qed

lemma filterlim-abs-real: filterlim (abs :: real ⇒ real) at-top at-top
proof (subst filterlim-cong[of refl refl])
  from eventually-ge-at-top[of 0 :: real]
  show eventually (\lambda x::real. |x| = x) at-top
  by eventually-elim simp
qed (simp-all add: filterlim-ident)

lemma filterlim-of-real-at-infinity [tendsto-intros]:
filterlim (of-real :: real ⇒ 'a :: real-normed-algebra-1) at-infinity at-top
by (intro filterlim-norm-at-top-imp-at-infinity) (auto simp: filterlim-abs-real)

lemma not-tendsto-and-filterlim-at-infinity:
  fixes c :: 'a::real-normed-vector
  assumes F ≠ bot
  and (f −−−→ c) F
  and filterlim f at-infinity F
  shows False
proof
  from tendstoD[of assms(2), of 1/2]
  have eventually (\lambda x. dist (f x) c < 1/2) F
  by simp
moreover
  from filterlim-at-infinity[of norm c f F] assms(3)
  have eventually (\lambda x. norm (f x) ≥ norm c + 1) F by simp
ultimately have eventually (\lambda x. False) F
proof eventually-elim
  fix x
  assume A: dist (f x) c < 1/2
  assume norm (f x) ≥ norm c + 1
  also have norm (f x) = dist (f x) 0 by simp
  also have ... ≤ dist (f x) c + dist c 0 by (rule dist-triangle)
  finally show False using A by simp
qed
with assms show False by simp
qed

lemma filterlim-at-infinity-imp-not-convergent:
  assumes filterlim f at-infinity sequentially
  shows ¬ convergent f
by (rule notI, rule not-tendsto-and-filterlim-at-infinity[of - assms])
lemma filterlim-at-infinity-imp-eventually-ne:  
asumes filterlim f at-infinity F  
shows eventually (λz. f z ≠ c) F  
proof  
  have norm c + 1 > 0  
    by (intro add-nonneg-pos) simp-all  
  with filterlim-at-infinity[of order.refl, of f F]  
  have eventually (λz. norm (f z) ≥ norm c + 1) F  
    by blast  
  then show ?thesis  
    by eventually-elim auto  
qed  

lemma tendsto-of-nat:  
  filterlim (of-nat :: nat ⇒ 'a::real-normed-algebra-1) at-infinity sequentially  
proof (subst filterlim-at-infinity[of order.refl], intro allI impI)  
  fix r :: real  
  assume r: r > 0  
  define n where n = nat ⌈r⌉  
  from r have n: ∀m≥n. of-nat m ≥ r  
  unfolding n-def by linarith  
  from eventually-ge-at-top[of n] show eventually (λm. norm (of-nat m :: 'a) ≥ r) sequentially  
    by eventually-elim (use n in simp-all)  
qed  

109.4 Relate at, at-left and at-right  

This lemmas are useful for conversion between at x to at-left x and at-right x and also at-right (0::'a).  

lemmas filterlim-split-at-real = filterlim-split-at[where 'a=real]  

lemma filtermap-nhds-shift: filtermap (λx. x - d) (nhds a) = nhds (a - d)  
  for a d :: 'a::real-normed-vector  
  by (rule filtermap-fun-inverse[where g=λx. x + d])  
    (auto intro!: tendsto-eq-intros filterlim-ident)  

lemma filtermap-nhds-minus: filtermap (λx. - x) (nhds a) = nhds (- a)  
  for a :: 'a::real-normed-vector  
  by (rule filtermap-fun-inverse[where g=uminus])  
    (auto intro!: tendsto-eq-intros filterlim-ident)  

lemma filtermap-at-shift: filtermap (λx. x - d) (at a) = at (a - d)  
  for a d :: 'a::real-normed-vector  
  by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-shift[symmetric])  

lemma filtermap-at-right-shift: filtermap (λx. x - d) (at-right a) = at-right (a -
lemma filterlim-shift:
  fixes d :: 'a::real-normed-vector
  assumes filterlim f F (at a)
  shows filterlim (f ◦ (+) d) F (at (a − d))
  unfolding filterlim-iff
proof (intro strip)
  fix P
  assume eventually P F
  then have ∀ F x in filtermap (λ y. y − d) (at a). P (f (d + x))
    using assms by (force simp add: filterlim-iff eventually-filtermap)
  then show ∀ F x in at (a − d). P ((f ◦ (+) d) x)
    by (force simp add: filtermap-at-shift)
qed

lemma filterlim-shift-iff:
  fixes d :: 'a::real-normed-vector
  shows filterlim (f ◦ (+) d) F (at (a − d)) = filterlim f F (at a) (is ?lhs = ?rhs)
proof
  assume L: ?lhs show ?rhs
    using filterlim-shift [OF L, of −d] by (simp add: filterlim-iff)
qed (metis filterlim-shift)

lemma at-right-to-0: at-right a = filtermap (λx. x + a) (at-right 0)
  for a :: real
  using filtermap-at-right-shift[of a 0] by simp

lemma filterlim-at-right-to-0:
  filterlim f F (at-right a) ↔ filterlim (λx. f (x + a)) F (at-right 0)
  for a :: real
  unfolding filterlim-def filtermap-filtermap at-right-to-0[of a] ..

lemma eventually-at-right-to-0:
  eventually P (at-right a) ↔ eventually (λx. P (x + a)) (at-right 0)
  for a :: real
  unfolding at-right-to-0[of a] by (simp add: eventually-filtermap)

lemma at-to-0: at a = filtermap (λx. x + a) (at 0)
  for a :: 'a::real-normed-vector
  using filtermap-at-shift[of −a 0] by simp

lemma filterlim-at-to-0:
  filterlim f F (at a) ↔ filterlim (λx. f (x + a)) F (at 0)
  for a :: 'a::real-normed-vector
  unfolding filterlim-def filtermap-filtermap at-to-0[of a] ..
lemma eventually-at-to-0:
  eventually P (at a) ←→ eventually (λx. P (x + a)) (at 0)
  for a :: 'a::real-normed-vector
  unfolding at-to-0[of a] by (simp add: eventually-filtermap)

lemma filtermap-at-minus: filtermap (λx. - x) (at a) = at (- a)
  for a :: 'a::real-normed-vector
  by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus[symmetric])

lemma at-left-minus: at-left a = filtermap (λx. - x) (at-right (- a))
  for a :: real
  by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus[symmetric])

lemma at-right-minus: at-right a = filtermap (λx. - x) (at-left (- a))
  for a :: real
  by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus[symmetric])

lemma filterlim-at-left-to-right:
  filterlim f F (at-left a) ←→ filterlim (λx. f (- x)) F (at-right (- a))
  for a :: real
  unfolding filterlim-def filtermap-filtermap at-left-minus[of a] ..

lemma eventually-at-left-to-right:
  eventually P (at-left a) ←→ eventually (λx. P (- x)) (at-right (- a))
  for a :: real
  unfolding at-left-minus[of a] by (simp add: eventually-filtermap)

lemma filterlim-uminus-at-top-at-bot: LIM x at-bot. - x :: real => at-top
  unfolding filterlim-at-bot eventually-at-bot-dense
  by (metis leI minus-less-iff order-less-asym)

lemma filterlim-uminus-at-bot-at-top: LIM x at-top. - x :: real => at-bot
  unfolding filterlim-at-top eventually-at-top-dense
  by (metis leI less-minus-iff order-less-asym)

lemma at-bot-mirror :
  shows (at-bot::('a::{ordered-ab-group-add,linorder} filter)) = filtermap uminus at-top
  proof (rule filtermap-fun-inverse[symmetric])
    show filterlim uminus at-top (at-bot::'a filter)
      using eventually-at-bot-linorder filterlim-at-top le-minus-iff by force
    show filterlim uminus (at-bot::'a filter) at-top
      by (simp add: filterlim-at-bot minus-le-iff)
  qed auto

lemma at-top-mirror :
  shows (at-top::('a::{ordered-ab-group-add,linorder} filter)) = filtermap uminus at-bot
  proof (rule filtermap-fun-inverse[symmetric])
    show filterlim uminus at-bot (at-top::'a filter)
      using eventually-at-top-linorder filterlim-at-bot le-minus-iff by force
    show filterlim uminus (at-top::'a filter) at-bot
      by (simp add: filterlim-at-top minus-le-iff)
  qed auto
theorem filterlim-at-top-mirror: \( \lim_{x \to \top} f x \colon F \iff \lim_{x \to \bot} f (-x) \colon F \) 
unfolding filterlim-def at-top-mirror filtermap-filtermap ..

lemma filterlim-at-bot-mirror: \( \lim_{x \to \bot} f x \colon F \iff \lim_{x \to \top} f (-x) \colon F \) 
unfolding filterlim-def at-bot-mirror filtermap-filtermap ..

lemma filterlim-uminus-at-top: \( \lim_{x \to F} f x \colon \top \iff \lim_{x \to F} (-f x) \colon \bot \) 
using filterlim-compose [OF filterlim-uminus-at-bot-at-top, of f F] and filterlim-compose [OF filterlim-uminus-at-top-at-bot, of \( \lambda x. -f x \)]
by auto

lemma tendsto-at-botI-sequentially: 
fixes \( f : \mathbb{R} \to 'b : \text{first-countable-topology} \)
assumes \( \forall X. \lim X \text{ at-bot sequentially} \implies (\lambda n. f (X n)) \dasharrow y \)
shows \( f \dasharrow y \text{ at-bot} \)
unfolding filterlim-at-bot-mirror
proof (rule tendsto-at-topI-sequentially)
fix \( X : \mathbb{N} \to \mathbb{R} \) assume filterlim X at-top sequentially 
thus \( (\lambda n. f (-X n)) \dasharrow y \) by (intro *) (auto simp: filterlim-uminus-at-top)
qed

lemma filterlim-at-infinity-imp-filterlim-at-top: 
assumes \( \lim_{x \to \infty} (f :: 'a \Rightarrow \mathbb{R}) \) 
assumes eventually \( (\lambda x. f x > 0) \) \( F \)
shows \( \lim f \) \( at-top \) \( F \)
proof 
from assms(2) have \( * : \text{eventually} (\lambda x. \text{norm} (f x) = f x) \) \( F \) by eventually-elim simp
from assms(1) show \(?thesis unfolding filterlim-at-infinity-conv-norm-at-top 
by (subst (asm) filterlim-cong[OF refl refl *])
qed

lemma filterlim-at-infinity-imp-filterlim-at-bot: 
assumes \( \lim_{x \to \infty} (f :: 'a \Rightarrow \mathbb{R}) \) 
assumes eventually \( (\lambda x. f x < 0) \) \( F \)
shows \( \lim f \) \( at-bot \) \( F \)
proof 
from assms(2) have \( * : \text{eventually} (\lambda x. \text{norm} (f x) = -f x) \) \( F \) by eventually-elim simp
from assms(1) have \( \lim (\lambda x. -f x) \) \( at-top \) \( F \)
unfolding filterlim-at-infinity-conv-norm-at-top 
by (subst (asm) filterlim-cong[OF refl refl *])
thus \( \text{thesis by (simp add: filterlim-uminus-at-top)} \)
qed

lemma filterlim-uminus-at-bot: \( (\text{LIM } x \text{ F. } f x :> \text{at-bot}) \iff (\text{LIM } x \text{ F. } - (f x) :: \text{real} :> \text{at-top}) \)

unfolding filterlim-uminus-at-top by simp

lemma filterlim-inverse-at-top-right: \( \text{LIM } x \text{ at-right } (0::\text{real}). \text{inverse } x :> \text{at-top} \)

unfolding filterlim-at-top-gt[where \( c=0 \)] eventually-at-filter

proof safe
fix \( Z :: \text{real} \)
assume \( \text{arith}: 0 < Z \)

then have eventually \( (\lambda x. x < \text{inverse } Z) (\text{nhds } 0) \)
by (auto simp: eventually-nhds-metric dist-real-def intro: !: exI [of - | inverse Z |])

then show eventually \( (\lambda x. x \neq 0 \rightarrow x \in \{0<..\} \rightarrow Z \leq \text{inverse } x) (\text{nhds } 0) \)
by (auto elim!: eventually-mono simp: inverse-eq-divide field-simps)

qed

lemma tendsto-inverse-0:
fixes \( x :: -'a::\text{real-normed-algebra} \)
shows \( (\text{inverse } \rightarrow (0::'a)) \text{ at-infinity} \)

unfolding tendsto-Zfun-iff diff-0-right Zfun-def eventually-at-infinity

proof safe
fix \( r :: \text{real} \)
assume \( 0 < r \)

show \( \exists b. \forall x. b \leq \text{norm } x \rightarrow \text{norm } x :: 'a < r \)
proof (intro exI[of - inverse (r / 2)] allI impl)
fix \( x :: 'a \)
from \( 0 < r \) have \( 0 < \text{inverse } (r / 2) \) by simp
also assume \( *: \text{inverse } (r / 2) \leq \text{norm } x \)
finally show \( \text{norm } (\text{inverse } x) < r \)
using \( *: 0 < r \)
by (subst nonzero-norm-inverse) (simp-all add: inverse-eq-divide field-simps)

qed

lemma tendsto-add-filterlim-at-infinity:
fixes \( c :: 'b::\text{real-normed-vector} \)
and \( F :: 'a \text{ filter} \)
assumes \( (f \rightarrow c) F \)
and filterlim g at-infinity F
shows \( \text{filterlim } (\lambda x. f x + g x) \text{ at-infinity } F \)

proof (subst filterlim-at-infinity[OF order-refl], safe)
fix \( r :: \text{real} \)
assume \( r: r > 0 \)
from assms(1) have \( ((\lambda x. \text{norm } (f x)) \rightarrow \text{norm } c) F \)
by (rule tendsto-norm)
then have eventually \( (\lambda x. \text{norm } (f x) < \text{norm } c + 1) F \)
by (rule order-tendstoD) simp-all
moreover from $r$ have $r + \text{norm } c + 1 > 0$
  by (intro add-pos-nonneg) simp-all
with assms(2) have eventually $(\lambda x. \text{norm } (g x) \geq r + \text{norm } c + 1) \ F$
  unfolding filterlim-at-infinity[OF order-refl]
  by (elim allE[of $\cdot$ $r + \text{norm } c + 1$]) simp-all
ultimately show eventually $(\lambda x. \text{norm } (f x + g x) \geq r) \ F$
proof eventually-elim
fix $x :: 'a$
assume $A$: $\text{norm } (f x) < \text{norm } c + 1$ and $B$: $r + \text{norm } c + 1 \leq \text{norm } (g x)$
from $A \ B$ have $r \leq \text{norm } (g x) - \text{norm } (f x)$
  by simp
also have $\text{norm } (g x) - \text{norm } (f x) \leq \text{norm } (g x + f x)$
  by (rule norm-diff-ineq)
finally show $r \leq \text{norm } (f x + g x)$
  by (simp add: add-ac)
qed

lemma tendsto-add-filterlim-at-infinity':
  fixes $c :: 'b::real-normed-vector$
    and $F :: 'a filter$
  assumes filterlim $f$ at-infinity $F$
    and $(g \longrightarrow c) \ F$
  shows filterlim $(\lambda x. f x + g x)$ at-infinity $F$
by (subst add.commute) (rule tendsto-add-filterlim-at-infinity assms)+

lemma filterlim-inverse-at-right-top:
  $\lim x$ at-top. inverse $x$ := at-right $(0 :: real)$
unfolding filterlim-at
by (auto simp: eventually-at-top-dense)
  (metis tendsto-inverse-0 filterlim-mono at-top-le-at-infinity order-refl)

lemma filterlim-inverse-at-top:
  $(f \longrightarrow (0 :: real)) \ F \implies$ eventually $(\lambda x. 0 < f x) \ F \implies \lim x \ F$. inverse $(f x) :=$ at-top
by (intro filterlim-compose[OF filterlim-inverse-at-top-right])
  (simp add: filterlim-def eventually-filtermap eventually-mono at-within-def le-principal)

lemma filterlim-inverse-at-bot-neg:
  $\lim x$ (at-left $(0 :: real)$). inverse $x$ := at-bot
by (simp add: filterlim-inverse-at-top-right filterlim-uminus-at-bot filterlim-at-left-to-right)

lemma filterlim-inverse-at-bot:
  $(f \longrightarrow (0 :: real)) \ F \implies$ eventually $(\lambda x. f x < 0) \ F \implies \lim x \ F$. inverse $(f x) :=$ at-bot
unfolding filterlim-uminus-at-bot inverse-minus-eq[ symmetric]
by (rule filterlim-inverse-at-top) (simp add: tendsto-minus-cancel-left[ symmetric])

lemma at-right-to-top: (at-right $(0 :: real))$ = filtermap inverse at-top
by (intro filtermap-fun-inverse[symmetric, where g=inverse])
(auto intro: filterlim-inverse-at-top-right filterlim-inverse-at-right-top)

lemma eventually-at-right-to-top:
eventually P (at-right (0::real)) <-> eventually (λx. P (inverse x)) at-top
unfolding at-right-to-top eventually-filtermap ..

lemma filterlim-at-right-to-top:
filterlim f F (at-right (0::real)) <-> (LIM x at-top. f (inverse x) :- F)
unfolding filterlim-def at-right-to-top filtermap-filtermap ..

lemma at-top-to-right: at-top = filtermap inverse (at-right (0::real))
unfolding at-right-to-top filtermap-filtermap inverse-inverse-eq filtermap-ident ..

lemma eventually-at-top-to-right:
eventually P at-top <-> eventually (λx. P (inverse x)) (at-right (0::real))
unfolding at-top-to-right eventually-filtermap ..

lemma filterlim-at-top-to-right:
filterlim f F at-top <-> (LIM x (at-right (0::real)). f (inverse x) :- F)
unfolding filterlim-def at-top-to-right filtermap-filtermap ..

lemma filterlim-inverse-at-infinity:
fixes x :: 'a::real-normed-div-algebra, division-ring
shows filterlim inverse at-infinity (at (0::'a))
unfolding filterlim-at-infinity[OF order-refl]
proof safe
fix r :: real
assume 0 < r
then show eventually (λx::'a. r ≤ norm (inverse x)) (at 0)
  unfolding eventually-at norm-inverse
  by (intro exI[of - inverse r])
  (auto simp: norm-conv-dist[symmetric] field-simps inverse-eq-divide)
qed

lemma filterlim-inverse-at-iff:
fixes g :: 'a => 'b::real-normed-div-algebra, division-ring
shows (LIM x F. inverse (g x) :- at 0) <-> (LIM x F. g x :- at-infinity)
unfolding filterlim-def filtermap-filtermap[symmetric]
proof
assume filtermap g F ≤ at-infinity
then have filtermap inverse (filtermap g F) ≤ filtermap inverse at-infinity
  by (rule filtermap-mono)
also have ... ≤ at 0
  using tendssto-inverse-0[where 'a='b]
  by (auto intro: exI[of - 1]
    simp: le-principal eventually-filtermap filterlim-def at-within-def eventually-at-infinity)
finally show filtermap inverse (filtermap g F) ≤ at 0 .
next

assume filtermap inverse (filtermap g F) ≤ at 0
then have filtermap inverse (filtermap inverse (filtermap g F)) ≤ filtermap inverse (at 0)
  by (rule filtermap-mono)
with filterlim-inverse-at-infinity show filtermap g F ≤ at-infinity
  by (auto intro: order-trans simp: filterlim-def filtermap-filtermap)
qed

lemma tendsto-mult-filterlim-at-infinity:
fixes c :: 'a::real-normed-field
assumes (f ----> c) F c ≠ 0
assumes filterlim g at-infinity F
shows filterlim (λx. f x * g x) at-infinity F
proof
  have ((λx. inverse (f x) * inverse (g x)) ----> inverse c * 0) F
    by (intro tendsto-mult tendsto-inverse assms filterlim-compose[OF tendsto-inverse-0])
  then have filterlim (λx. inverse (f x) * inverse (g x)) (at (inverse c * 0)) F
    unfolding filterlim-at using assms by (auto intro: filterlim-at-infinity-imp-eventually-ne tendsto-imp-eventually-ne eventually-conj)
  then show ?thesis
    using ⟨n < 0⟩ by (subst filterlim-inverse-at-iff; simp-all)
qed

lemma filterlim-power-int-neg-at-infinity:
fixes f :: 'a::{real-normed-div-algebra, division-ring}
assumes n < 0 and lim: (f ----> 0) F and ev: eventually (λx. f x ≠ 0) F
shows filterlim (λx. f x powi n) at-infinity F
proof
  have lim': ((λx. f x powi n) ----> 0) F
    by (rule tendsto-eq-intros lim)+ (use ⟨n < 0⟩ in auto)
  have ev': eventually (λx. f x powi n ≠ 0) F
    using ev by eventually-elim (use ⟨n < 0⟩ in auto)
  have filterlim (λx. inverse (f x powi n)) at-infinity F
    by (intro filterlim-compose[OF filterlim-inverse-at-infinity])
      (use lim' ev' in (auto simp: filterlim-at))
  thus ?thesis
    using ⟨n < 0⟩ by (simp add: power-int-def power-inverse)
qed

lemma tendsto-inverse-0-at-top: LIM x F. f x :: real ----> 0 = ((λx. inverse (f x) :: real) ----> 0) F
  by (metis filterlim-at filterlim-mono[OF at-top-le-at-infinity order-refl] filterlim-inverse-at-iff)

lemma filterlim-inverse-at-top-iff:
eventually (λx. 0 < f x) F = (LIM x F. inverse (f x) :: real ----> 0)
lemma filterlim-at-top-iff-inverse-0:
  eventually \((\lambda x. 0 < f x) \mapsto (LIM x F. f x :> at-top) \iff ((\text{inverse } \circ f) \mapsto 0)\)
  if \((0 :: \text{real})\) by (auto dest: tendsto-inverse-0-at-top filterlim-inverse-at-top)

lemma real-tendsto-divide-at-top:
  fixes \(c :: \text{real}\)
  assumes \((f \mapsto c) \mapsto F\)
  assumes filterlim \(g \atop F\)
  shows \((\lambda x. f x / g x) \mapsto 0) \mapsto F\)
  by (auto simp: divide-inverse-commute intro!: tendsto-mult [THEN tendsto-eq-rhs] tendsto-inverse-0-at-top assms)

lemma mult-nat-left-at-top:
  \(c > 0 \Rightarrow \text{filterlim} \ (\lambda x. c \ast x) \atop F\)
  for \(c :: \text{nat}\)
  by (rule filterlim-subseq) (auto simp: strict-mono-def)

lemma mult-nat-right-at-top:
  \(c > 0 \Rightarrow \text{filterlim} \ (\lambda x. x \ast c) \atop F\)
  for \(c :: \text{nat}\)
  by (rule filterlim-subseq) (auto simp: strict-mono-def)

lemma filterlim-times-pos:
  \(LIM x F1. c \ast f x :> \text{at-right } l\)
  if filterlim \(f \atop (\text{at-right } p)\)
  \(F1 \ 0 < c \ast l = c \ast p\)
  for \(c :: 'a::\{\text{linordered-field, linorder-topology}\}\)

  unfolding filterlim-iff

  proof safe
    fix \(P\)
    assume \(\forall F \ x \in \text{at-right } l. P \ x\)
    then obtain \(d \ \text{where} \ c \ast p < d \ \forall y. y > c \ast p \Rightarrow y < d \Rightarrow P \ y\)
      unfolding \(\langle l = - \rangle \ \text{eventually-at-right-field}\)
    by auto
    then have \(\forall F \ a \in \text{at-right } p. P \ (c \ast a)\)
      by (auto simp: eventually-at-right-field \(\langle 0 < c \rangle \ \text{field-simps intro!: exI[where x=d/c]\}
    from that(1)[unfolded filterlim-iff, rule-format, OF this]
    show \(\forall F \ x \in F1. P \ (c \ast f x)\).
  qed

lemma filtermap-nhds-times:
  \(c \neq 0 \Rightarrow \text{filtermap} \ (\text{times } c) \ (\text{nhds } a) = \text{nhds} \ (c \ast a)\)
  for \(a \ c :: 'a::\text{real-normed-field}\)
  by (rule filtermap-fun-inverse[where \(g=\lambda x. \text{inverse } c \ast x\)])
    (auto intro!: tendsto-eq-intros filterlim-ident)

lemma filtermap-times-pos-at-right:
fixes \( c ':\alpha::\{\text{linordered-field}, \text{linorder-topology}\}\)
assumes \( c > 0\)
shows \( \text{filtermap} (\times c) (\text{at-right} \ p) = \text{at-right} (c \times p)\)
using assms
by (intro filtermap-open-inverse[where \( g = \lambda x. \text{inverse} \ c \times x\)])
(auto intro: filterlim-ident filterlim-times-pos)

lemma at-to-infinity: \((0::\alpha::\{\text{real-normed-field, field}\}) = \text{filtermap} \text{inverse} \text{at-infinity}\)
proof (rule antisym)
  have \((\text{inverse} \longrightarrow (0::\alpha)) \text{ at-infinity}\)
  by (fact tendsto-inverse-0)
  then show \(\text{filtermap} \text{inverse} \text{at-infinity} \leq \text{at} (0::\alpha)\)
  using filterlim-def filterlim-ident filterlim-inverse-at-iff by fastforce
next
  have \(\text{filtermap} \text{inverse} (\text{filtermap} \text{inverse} (\text{at} (0::\alpha))) \leq \text{filtermap} \text{inverse} \text{at-infinity}\)
  using filterlim-inverse-at-infinity unfolding filterlim-def
  by (rule filtermap-mono)
  then show \((0::\alpha) \leq \text{filtermap} \text{inverse} \text{at-infinity}\)
  by (simp add: filtermap-ident filtermap-filtermap)
qed

lemma lim-at-infinity-0:
fixes \( l :: \alpha::\{\text{real-normed-field, field}\}\)
shows \((f \\longrightarrow l) \text{ at-infinity} \longleftrightarrow ((f \circ \text{inverse} ) \\longrightarrow l) (\text{at} \ (0::\alpha))\)
by (simp add: tendsto-compose-filtermap at-to-infinity filtermap-filtermap)

lemma lim-zero-infinity:
fixes \( l :: \alpha::\{\text{real-normed-field, field}\}\)
shows \((\lambda x. f(1/x)) \\longrightarrow l) (\text{at} \ (0::\alpha)) \text{ implies} (f \\longrightarrow l) \text{ at-infinity}\)
by (simp add: inverse-eq-divide lim-at-infinity-0 comp-def)

We only show rules for multiplication and addition when the functions are
either against a real value or against infinity. Further rules are easy to derive
by using \(\text{filterlim} ?f \text{ at-top} ?F = (\text{LIM} x ?F. \quad ?f x :> \text{at-bot}).\)

lemma filterlim-tendsto-pos-mult-at-top:
assumes \( f: (f \\longrightarrow c) \quad F\)
  and \( c: 0 < c\)
  and \( g: \text{LIM} x F. \quad g x :> \text{at-top}\)
shows \(\text{LIM} x F. \quad (f \times g x :: \text{real}) :> \text{at-top}\)
unfolding filterlim-at-top-gt[where \( c = 0\)]
proof safe
  fix \( Z :: \text{real}\)
  assume \( 0 < Z\)
  from \( f < 0 < c\) have \(\text{eventually} (\lambda x. c / 2 < f x) \quad F\)
  by (auto dest!: tendstoD[where \( c = c / 2\) elim!: eventually-mono
      simp: dist-real-def abs-real-def split: if-split_asm])
  moreover from \( g\) have \(\text{eventually} (\lambda x. Z / c \times 2 \leq g x) \quad F\)
  unfolding filterlim-at-top by auto
  ultimately show \(\text{eventually} (\lambda x. Z \times f x < g x) \quad F\)
proof eventually-elim
  case (elim x)
  with \(0 < Z\), \(0 < c\) have \(c / 2 \cdot (Z / c \cdot 2) \leq f \cdot g\)
  by (intro mult-mono) (auto simp: zero-le-divide-iff)
  with \(0 < c\) show \(Z \leq f \cdot g\)
  by simp
qed

lemma filterlim-at-top-mult-at-top:
  assumes f: LIM x F. f x :> at-top
  and g: LIM x F. g x :> at-top
  shows LIM x F. (f x :: real) :> at-top
  unfolding filterlim-at-top-gt[where c=0]
proof safe
  fix Z :: real
  assume 0 < Z
  from f have eventually (\(\lambda x\). 1 \leq f x) F
  unfolding filterlim-at-top by auto
  moreover from g have eventually (\(\lambda x\). Z \leq g x) F
  unfolding filterlim-at-top by auto
  ultimately show eventually (\(\lambda x\). Z \leq f \cdot g) F
proof eventually-elim
  case (elim x)
  with \(0 < Z\), \(1 \cdot Z \leq f \cdot g\)
  by (intro mult-mono) (auto simp: zero-le-divide-iff)
  then show \(Z \leq f \cdot g\)
  by simp
qed

lemma filterlim-at-top-mult-tendsto-pos:
  assumes f: (f ----> c) F
  and c: 0 < c
  and g: LIM x F. g x :> at-top
  shows LIM x F. (g x \cdot f x :: real) :> at-top
  by (auto simp: mult.commute intro: filterlim-tendsto-pos-mult-at-top f c g)

lemma filterlim-tendsto-pos-mult-at-bot:
  fixes c :: real
  assumes (f ----> c) F 0 < c filterlim g at-bot F
  shows LIM x F. f x \cdot g x :> at-bot
  using filterlim-tendsto-pos-mult-at-top[OF assms(1,2), of \(\lambda x\) \cdot g x] assms(3)
  unfolding filterlim-uminus-at-bot by simp

lemma filterlim-tendsto-neg-mult-at-bot:
  fixes c :: real
  assumes c: (f ----> c) F c < 0 and g: filterlim g at-top F
  shows LIM x F. f x \cdot g x :> at-bot
using $c$ filterlim-tendsto-pos-mult-at-top[of $\lambda x.\ - f x - c F$, OF = $g$]
unfolding filterlim-uminus-at-bot tendsto-minus-cancel-left by simp

lemma filterlim-cmult-at-bot-at-top:
  assumes filterlim $(h :: \rightarrow real)$ at-top $F$ $c \neq 0$ $G = (\text{if } c > 0 \text{ then at-top else at-bot})$
  shows $(\lambda x.\ c * h x) G F$
  using assms filterlim-tendsto-pos-mult-at-top[OF tendsto-const[of $c$, of $h F$]]
           filterlim-tendsto-neg-mult-at-bot[OF tendsto-const[of $c$, of $h F$]] by simp

lemma filterlim-pow-at-top:
  fixes $f :: 'a \Rightarrow real$
  assumes $0 < n$
  and $f$: LIM $x F$. $f x : >$ at-top
  shows LIM $x F$. $(f x) ^ n :: real : >$ at-top
  using $(0 < n)$
  proof (induct $n$)
    case 0
    then show ?case by simp
  next
    case (Suc $n$) with $f$ show ?case
    by (cases $n = 0$) (auto intro!: filterlim-at-top-mult-at-top)
  qed

lemma filterlim-pow-at-bot-even:
  fixes $f :: real \Rightarrow real$
  shows $0 < n \Longrightarrow LIM x F. f x : > at-bot \Longrightarrow even n \Longrightarrow LIM x F. (f x) ^ n : > at-top$
  using filterlim-pow-at-top[of $n \lambda x.\ - f x F$] by (simp add: filterlim-uminus-at-top)

lemma filterlim-pow-at-bot-odd:
  fixes $f :: real \Rightarrow real$
  shows $0 < n \Longrightarrow LIM x F. f x : > at-bot \Longrightarrow odd n \Longrightarrow LIM x F. (f x) ^ n : > at-bot$
  using filterlim-pow-at-top[of $n \lambda x.\ - f x F$] by (simp add: filterlim-uminus-at-top)

lemma filterlim-power-at-infinity [tendsto-intros]:
  fixes $F$ and $f :: 'a \Rightarrow 'b :: real-normed-div-algebra$
  assumes filterlim $f$ at-infinity $F$ $n > 0$
  shows filterlim $(\lambda x.\ f x ^ n)$ at-infinity $F$
  by (rule filterlim-norm-at-top-imp-at-infinity)
     (auto simp: norm-power intro!: filterlim-pow-at-top assms intro: filterlim-at-infinity-imp-norm-at-top)

lemma filterlim-tendsto-add-at-top:
  assumes $f$: $(f \rightarrow c) F$
  and $g$: LIM $x F$. $g x : > at-top$
  shows LIM $x F$. $(f x + g x :: real) : > at-top$
unfolding filterlim-at-top-gt[where $c=0$]

proof safe
fix $Z :: \text{real}$
assume $0 < Z$
from $f$ have eventually $(\lambda x. c - 1 < f x) \; F$
by (auto dest: tendstoD[where $e=1$] elim!: eventually-mono simp: dist-real-def)
moreover from $g$ have eventually $(\lambda x. Z - (c - 1) \leq g x) \; F$
unfolding filterlim-at-top by auto
ultimately show eventually $(\lambda x. Z \leq f x + g x) \; F$
by eventually-elim simp

qed

lemma LIM-at-top-divide:
fixes $f \; \cdot \; 'a \Rightarrow \text{real}$
assumes $f$: $(f \longrightarrow a) \; F \; 0 < a$
and $g$: $(g \longrightarrow 0) \; F$ eventually $(\lambda x. 0 < g x) \; F$
shows LIM $x \; F$. $(f x / g x) \; : > \; \text{at-top}$
unfolding divide-inverse
by (rule filterlim-tendsto-pos-mul-at-top[OF $f$]) (rule filterlim-inverse-at-top[OF $g$])

lemma filterlim-at-top-add-at-top:
assumes $f$: LIM $x \; F$. $(f x) \; : > \; \text{at-top}$
and $g$: LIM $x \; F$. $(g x) \; : > \; \text{at-top}$
shows LIM $x \; F$. $(f x + g x) \; : > \; \text{at-top}$
unfolding filterlim-at-top-gt[where $c=0$]

proof safe
fix $Z :: \text{real}$
assume $0 < Z$
from $f$ have eventually $(\lambda x. 0 < f x) \; F$
unfolding filterlim-at-top by auto
moreover from $g$ have eventually $(\lambda x. Z \leq g x) \; F$
unfolding filterlim-at-top by auto
ultimately show eventually $(\lambda x. Z \leq f x + g x) \; F$
by eventually-elim simp

qed

lemma tendsto-divide-0:
fixes $f :: \Rightarrow 'a::{\text{real-normed-div-algebra},\text{division-ring}}$
assumes $f$: $(f \longrightarrow c) \; F$
and $g$: LIM $x \; F$. $(g x) \; : > \; \text{at-infinity}$
shows $(\lambda x. f x / g x) \longrightarrow 0) \; F$
using tendsto-mult[OF $f$ filterlim-compose[OF tendsto-inverse-0 $g$]]
by (simp add: divide-inverse)

lemma linear-plus-1-le-power:
fixes $x :: \text{real}$
assumes $x$: $0 \leq x$
shows real $n * x + 1 \leq (x + 1) ^ n$
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from x have real (Suc n) * x + 1 ≤ (x + 1) * (real n * x + 1)
    by (simp add: field-simps)
  also have ... ≤ (x + 1) ^ Suc n
    using Suc x by (simp add: mult-left-mono)
  finally show ?case.
qed

lemma filterlim-realpow-sequentially-gt1:
  fixes x :: 'a :: real_normed_div_algebra
  assumes x [arith]: 1 < norm x
  shows LIM n sequentially. x ^ n > at_infinity
proof (intro filterlim_at_infinity[THEN iffD2] allI impI)
  fix y :: real
  assume 0 < y
  obtain N :: nat where y < real N * (norm x - 1)
    by (meson diff_gt_0_iff_gt reals_Archimedean3 x)
  also have ... ≤ real N * (norm x - 1 + 1)
    by simp
  also have ... ≤ (norm x - 1 + 1) ^ N
    by (rule linear_plus_1_le_power simp)
  also have ... = norm x ^ N
    by simp
  finally have ∀ n≥N. y ≤ norm x ^ n
    by (metis order_less_le_trans power_increasing order_less_imp_le x)
  then show eventually (λn. y ≤ norm (x ^ n)) sequentially
    unfolding eventually_sequentially
    by (auto simp: norm_power)
qed simp

lemma filterlim-divide-at-infinity:
  fixes f g :: 'a ⇒ 'a :: real_normed_field
  assumes filterlim f (nhds c) F filterlim g (at 0) F c ≠ 0
  shows filterlim (λx. f x / g x) at_infinity F
proof
  have filterlim (λx. f x / inverse (g x)) at_infinity F
    by (intro tendsto_mult_filterlim_at_infinity[OF assms(1,3)])
  thus ?thesis by (simp add: field_simps)
qed

109.5 Floor and Ceiling

lemma eventually-floor-less:

fixes $f :: 'a \Rightarrow 'b::\{order-topology,floor-ceiling\}$
assumes $f: (f \rightarrow\rightarrow l) \ F$
and $l: l \notin \mathbb{Z}$
shows $\forall x \in F. \ of-int (floor l) < f x$
by (intro order-tendstoD[OF $f$]) (metis Ints-of-int antisym_conv2 floor_correct $l$)

lemma eventually-less-ceiling:
fixes $f :: 'a \Rightarrow 'b::\{order-topology,floor-ceiling\}$
assumes $f: (f \rightarrow\rightarrow l) \ F$
and $l: l \notin \mathbb{Z}$
shows $\forall x \in F. \ f x < of-int \ (ceiling l)$
by (intro order-tendstoD[OF $f$]) (metis Ints-of-int le_of_int_ceiling less_le)

lemma eventually-floor-eq:
fixes $f :: 'a \Rightarrow 'b::\{order-topology,floor-ceiling\}$
assumes $f: (f \rightarrow\rightarrow l) \ F$
and $l: l \notin \mathbb{Z}$
shows $\forall x \in F. \ floor (f x) = floor l$
using eventually_floor_less[OF assms] eventually_less_ceiling[OF assms]
by eventually_elim (meson floor_less_iff less_ceiling_iff not_less_iff_gr_or_eq)

lemma eventually-ceiling-eq:
fixes $f :: 'a \Rightarrow 'b::\{order-topology,floor-ceiling\}$
assumes $f: (f \rightarrow\rightarrow l) \ F$
and $l: l \notin \mathbb{Z}$
shows $\forall x \in F. \ ceiling (f x) = ceiling l$
using eventually_floor_less[OF assms] eventually_less_ceiling[OF assms]
by eventually_elim (meson floor_less_iff less_ceiling_iff not_less_iff_gr_or_eq)

lemma tendsto_of_int_floor:
fixes $f :: 'a \Rightarrow 'b::\{order-topology,floor-ceiling\}$
assumes $(f \rightarrow\rightarrow l) \ F$
and $l: l \notin \mathbb{Z}$
shows $((\lambda x. \ of-int \ (floor (f x))) :: 'c::\{ring-1,topological-space\}) \rightarrow\rightarrow \ of-int \ (floor l) \ F$
using eventually_floor_eq[OF assms]
by (simp add: eventually_mono topological_tendstoI)

lemma tendsto_of_int_ceiling:
fixes $f :: 'a \Rightarrow 'b::\{order-topology,floor-ceiling\}$
assumes $(f \rightarrow\rightarrow l) \ F$
and $l: l \notin \mathbb{Z}$
shows $((\lambda x. \ of-int \ (ceiling (f x))) :: 'c::\{ring-1,topological-space\}) \rightarrow\rightarrow \ of-int \ (ceiling l) \ F$
using eventually_ceiling_eq[OF assms]
by (simp add: eventually_mono topological_tendstoI)

lemma continuous_on_of_int_floor:
continuous-on $(UNIV - \mathbb{Z} :: 'a::\{order-topology,floor-ceiling\} \ set)$
THEORY “Limits”

(\lambda x. \textit{of-int} (\textit{floor} x) :: 'b::{ring_1, topological-space})

unfolding continuous-on-def
by (auto intro: tendsto-of-int-floor)

lemma continuous-on-of-int-ceiling:
continuous-on (UNIV - \mathbb{Z} :: 'a::{order-topology, floor-ceiling} set)
(\lambda x. \textit{of-int} (\textit{ceiling} x) :: 'b::{ring_1, topological-space})

unfolding continuous-on-def
by (auto intro: tendsto-of-int-ceiling)

109.6 Limits of Sequences

lemma [trans]: \(X = Y \implies Y \longrightarrow z \implies X \longrightarrow z\)
by simp

lemma LIMSEQ-iff:
fixes \(L :: 'a::\textit{real-normed-vector}\)
shows (\(X \longrightarrow L\) = (\(\forall r > 0. \exists \text{n. \forall n \geq \text{n. norm} (X n - L) < r}\))

unfolding lim-sequentially dist-norm ..

lemma LIMSEQ-I: (\(\forall r < r \implies \exists \text{n. \forall n \geq \text{n. norm} (X n - L) < r}\)) \(\implies X \longrightarrow L\)
for \(L :: 'a::\textit{real-normed-vector}\)
by (simp add: LIMSEQ-iff)

lemma LIMSEQ-D: \(X \longrightarrow L\) \(\implies 0 < r \implies \exists \text{n. \forall n \geq \text{n. norm} (X n - L) < r}\)
< r
for \(L :: 'a::\textit{real-normed-vector}\)
by (simp add: LIMSEQ-iff)

lemma LIMSEQ-linear: \(X \longrightarrow x \implies l > 0 \implies (\lambda n. X (n * l)) \longrightarrow x\)

unfolding tendsto-def eventually-sequentially
by (metis div-le-dividend div-mult_self1_is_m le_trans mult.commute)

Transformation of limit.

lemma Lim-transform: (\(g \longrightarrow a\)) \(F \implies ((\lambda x. f x - g x) \longrightarrow 0) \implies (f \longrightarrow a) F\)
for \(a b :: 'a::\textit{real-normed-vector}\)
using tendsto-add [of g a F \lambda x. f x - g x 0] by simp

lemma Lim-transform2: (\(f \longrightarrow a\)) \(F \implies ((\lambda x. f x - g x) \longrightarrow 0) \implies (g \longrightarrow a) F\)
for \(a b :: 'a::\textit{real-normed-vector}\)
by (erule Lim-transform) (simp add: tendsto-minus-cancel)

proposition Lim-transform-eq: (\((\lambda x. f x - g x) \longrightarrow 0) \implies (f \longrightarrow a) F\)
\longleftrightarrow (\(g \longrightarrow a\) F)
for \(a :: 'a::\textit{real-normed-vector}\)
using Lim-transform Lim-transform2 by blast
lemma **Lim-transform-eventually**: 
\[ ([f \rightarrow l] F; \text{eventually} (\lambda x. f x = g x) F) \implies (g \rightarrow l) F \]
using **eventually-elim2** by (fastforce simp add: tendsto_def)

lemma **Lim-transform-within**: 
assumes \( (f \rightarrow l) \) \((at x within S)\)
and \(0 < d\)
and \(\forall x'. x' \in S \implies 0 < \text{dist } x' x \implies \text{dist } x' x < d \implies f x' = g x'\)
shows \((g \rightarrow l) \) \((at x within S)\)
proof (rule **Lim-transform-eventually**) 
show \(\text{eventually } (\lambda x. f x = g x) \) \((at x within S)\)
using assms by (auto simp: eventually-at)
show \( (f \rightarrow l) \) \((at x within S)\)
by fact
qed

lemma **filterlim-transform-within**: 
assumes \(\text{filterlim } g G \) \((at x within S)\)
assumes \(G \leq F\) \(0 < d\) \((\forall x'. x' \in S \implies 0 < \text{dist } x' x \implies \text{dist } x' x < d \implies f x' = g x'\)
shows \(\text{filterlim } f F \) \((at x within S)\)
proof (rule **Lim-transform-eventually**) 
show \(\text{eventually } (\lambda x. f x = g x) \) \((at x within S)\)
using assms apply (elim filterlim-mono-eventually)
unfolding eventually-at by auto

Common case assuming being away from some crucial point like 0.

lemma **Lim-transform-away-within**: 
fixes \(a \ b:: ',a::t1-space\)
assumes \(a \neq b\)
and \(\forall x \in S. x \neq a \land x \neq b \implies f x = g x\)
and \((f \rightarrow l) \) \((at a within S)\)
shows \((g \rightarrow l) \) \((at a within S)\)
proof (rule **Lim-transform-eventually**) 
show \(\text{eventually } (\lambda x. f x = g x) \) \((at a within S)\)
unfolding eventually-at-topological
by (rule exI [where \(x=-\{b\}\)] simp add: open-Compl assms)
qed

lemma **Lim-transform-away-at**: 
fixes \(a \ b:: ',a::t1-space\)
assumes \(ab: a \neq b\)
and \(fg: \forall x. x \neq a \land x \neq b \implies f x = g x\)
and \(fl: (f \rightarrow l) \) \((at a)\)
shows \((g \rightarrow l) \) \((at a)\)
using **Lim-transform-away-within**[OF \(ab, of UNIV f g l\) \(fg fl\) by simp]

Alternatively, within an open set.
lemma \textbf{Lim-transform-within-open}:
assumes \((f \longrightarrow l) \text{ (at } a \text{ within } T)\)
and open \(s\) and \(a \in s\)
and \(\forall x. x \in s \implies x \neq a \implies f x = g x\)
shows \((g \longrightarrow l) \text{ (at } a \text{ within } T)\)
proof (rule \textit{Lim-transform-eventually})
show eventually \((\lambda x. f x = g x) \text{ (at } a \text{ within } T)\)
unfolding eventually-at-topological
using \textit{assms} by auto
show \((f \longrightarrow l) \text{ (at } a \text{ within } T)\) by fact
qed

A congruence rule allowing us to transform limits assuming not at point.

lemma \textbf{Lim-cong-within}:
assumes \(a = b\)
and \(x = y\)
and \(S = T\)
and \(\forall x. x \neq b \implies x \in T \implies f x = g x\)
shows \((f \longrightarrow x) \text{ (at } a \text{ within } S) \iff (g \longrightarrow y) \text{ (at } b \text{ within } T)\)
unfolding tendsto-def eventually-at-topological
using \textit{assms} by simp

An unbounded sequence’s inverse tends to 0.

lemma \textbf{LIMSEQ-inverse-zero}:
assumes \(\forall r::\text{real}. \exists N. \forall n \geq N. r < X n\)
shows \((\lambda n. \text{inverse (}X n)) \longrightarrow 0\)
apply (rule filterlim-compose[of tendsto-inverse-0])
by (metis \textit{assms} eventually-at-top-linorderI filterlim-at-top-dense filterlim-at-top-imp-at-infinity)

The sequence \((1::'a) / n\) tends to 0 as \(n\) tends to infinity.

lemma \textbf{LIMSEQ-inverse-real-of-nat}:
\((\lambda n. \text{inverse (}real (\text{Suc } n))) \longrightarrow 0\)
by (metis filterlim-compose tendsto-inverse-0 filterlim-mono order-refl filterlim-Suc filterlim-compose[of filterlim-real-sequentially] at-top-le-at-infinity)

The sequence \(r + (1::'a) / n\) tends to \(r\) as \(n\) tends to infinity is now easily proved.

lemma \textbf{LIMSEQ-inverse-real-of-nat-add}:
\((\lambda n. r + \text{inverse (}real (\text{Suc } n))) \longrightarrow r\)
using tendsto-add [OF tendsto-const \textbf{LIMSEQ-inverse-real-of-nat}] by auto

lemma \textbf{LIMSEQ-inverse-real-of-nat-add-minus}:
\((\lambda n. r + -\text{inverse (}real (\text{Suc } n))) \longrightarrow r\)
using tendsto-add [OF tendsto-const tendsto-minus [OF \textbf{LIMSEQ-inverse-real-of-nat}]] by auto

lemma \textbf{LIMSEQ-inverse-real-of-nat-add-minus-mult}:
\((\lambda n. r * (1 + -\text{inverse (}real (\text{Suc } n)))) \longrightarrow r\)
using tendsto-mult [OF tendsto-const \textbf{LIMSEQ-inverse-real-of-nat-add-minus} [of 1]]
by auto

lemma lim-inverse-n: ((\n. inverse(of-nat n)) ----> (0::'a::real-normed-field)) sequentially
  using lim-1-over-n by (simp add: inverse-eq-divide)

lemma lim-inverse-n': ((\n / n ----> 0) sequentially
  using lim-inverse-n by (simp add: inverse-eq-divide)

lemma LIMSEQ-Suc-n-over-n: (\n. of-nat (Suc n) / of-nat n :: 'a :: real-normed-field)
  ----> 1
proof (rule Lim-transform-eventually)
  show eventually (\n. 1 + inverse (of-nat n :: 'a) = of-nat (Suc n) / of-nat n) sequentially
    using eventually-gt-at-top[of 0::nat]
    by eventually-elim (simp add: field-simps)
  have (\n / inverse (of-nat n :: 'a) ----> 1 + 0
    by (intro tendsto-add tendsto-const lim-inverse-n)
  then show (\n / inverse (of-nat n :: 'a) ----> 1
    by simp
qed

lemma LIMSEQ-n-over-Suc-n: (\n. of-nat n / of-nat (Suc n) :: 'a :: real-normed-field)
  ----> 1
proof (rule Lim-transform-eventually)
  show eventually (\n. inverse (of-nat (Suc n) / of-nat n :: 'a) = of-nat n / of-nat (Suc n)) sequentially
    using eventually-gt-at-top[of 0::nat]
    by eventually-elim (simp add: field-simps del: of-nat-Suc)
  have (\n / inverse (of-nat (Suc n) / of-nat n :: 'a)) ----> inverse 1
    by (intro tendsto-inverse LIMSEQ-Suc-n-over-n simp-all)
  then show (\n / inverse (of-nat (Suc n) / of-nat n :: 'a)) ----> 1
    by simp
qed

109.7 Convergence on sequences

lemma convergent-cong:
  assumes eventually (\x. f x = g x) sequentially
  shows convergent f ----> convergent g
unfolding convergent-def
by (subst filterlim-cong[OF refl refl assms]) (rule refl)

lemma convergent-Suc-iff: convergent (\n. f (Suc n)) ----> convergent f
by (auto simp: convergent-def filterlim-sequentially-Suc)

lemma convergent-ignore-initial-segment: convergent (\n. f (n + m)) = convergent f
THEORY “Limits”

proof (induct m arbitrary: f)
case 0
  then show ?case by simp
next
case (Suc m)
  have convergent (λn. f (n + Suc m)) ←→ convergent (λn. f (Suc n + m))
    by simp
  also have . . . ←→ convergent (λn. f (n + m))
    by (rule convergent-Suc-iff)
  also have . . . ←→ convergent f
    by (rule Suc)
  finally show ?case.
qed

lemma convergent-add:
  fixes X Y :: nat ⇒ 'a::topological-monoid-add
  assumes convergent (λn. X n)
    and convergent (λn. Y n)
  shows convergent (λn. X n + Y n)
  using assms unfolding convergent-def by (blast intro: tendsto-add)

lemma convergent-sum:
  fixes X :: 'a ⇒ nat ⇒ 'b::topological-comm-monoid-add
  shows (∀i. i ∈ A ⇒ convergent (λn. X i n)) ⇒ convergent (λn. ∑i∈A. X i n)
  by (induct A rule: infinite-finite-induct) (simp-all add: convergent-const convergent-add)

lemma (in bounded-linear) convergent:
  assumes convergent (λn. X n)
  shows convergent (λn. f (X n))
  using assms unfolding convergent-def by (blast intro: tendsto)

lemma (in bounded-bilinear) convergent:
  assumes convergent (λn. X n)
    and convergent (λn. Y n)
  shows convergent (λn. X n ** Y n)
  using assms unfolding convergent-def by (blast intro: tendsto)

lemma convergent-minus-iff:
  fixes X :: nat ⇒ 'a::topological-group-add
  shows convergent X ←→ convergent (λn. − X n)
  unfolding convergent-def by (force dest: tendsto-minus)

lemma convergent-diff:
  fixes X Y :: nat ⇒ 'a::topological-group-add
  assumes convergent (λn. X n)
  assumes convergent (λn. Y n)
  shows convergent (λn. X n − Y n)
using assms unfolding convergent-def by (blast intro: tendsto-diff)

lemma convergent-norm:
  assumes convergent f
  shows convergent (λn. norm (f n))
proof –
  from assms have f −−−−→ lim f
    by (simp add: convergent-LIMSEQ-iff)
  then have (λn. norm (f n)) −−−−→ norm (lim f)
    by (rule tendsto-norm)
  then show ?thesis
    by (auto simp: convergent-def)
qed

lemma convergent-of-real:
  convergent f =⇒ convergent (λn. of-real (f n) :: 'a::real-normed-algebra-1)
unfolding convergent-def by (blast intro: tendsto-of-real)

lemma convergent-add-const-iff:
  convergent (λn. c + f n :: 'a::topological-ab-group-add) ←→ convergent f
proof
  assume convergent (λn. c + f n)
  from convergent-diff[OF this convergent-const[of c]] show convergent f
    by simp
next
  assume convergent f
  from convergent-add[OF convergent-const[of c] this] show convergent (λn. c + f n)
    by simp
qed

lemma convergent-add-const-right-iff:
  convergent (λn. f n + c :: 'a::topological-ab-group-add) ←→ convergent f
using convergent-add-const-right-iff[of f c] by (simp add: add-ac)

lemma convergent-diff-const-right-iff:
  convergent (λn. f n − c :: 'a::topological-ab-group-add) ←→ convergent f
using convergent-add-const-right-iff[of f − c] by (simp add: add-ac)

lemma convergent-mult:
  fixes X Y :: nat ⇒ 'a::topological-semigroup-mult
  assumes convergent (λn. X n)
  and convergent (λn. Y n)
  shows convergent (λn. X n * Y n)
using assms unfolding convergent-def by (blast intro: tendsto-mult)

lemma convergent-mult-const-iff:
  assumes c ≠ 0
  shows convergent (λn. c * f n :: 'a::{field,topological-semigroup-mult}) ←→ con-
vergent f 
proof
  assume convergent (λn. c * f n)
  from assms convergent-mult[OF this convergent-const[of inverse c]]
  show convergent f by (simp add: field-simps)
next
  assume convergent f
  from convergent-mult[OF convergent-const[of c] this] show convergent (λn. c * f n)
    by simp
qed

lemma convergent-mult-const-right-iff:
  fixes c :: 'a::{field,topological-semigroup-mult}
  assumes c ≠ 0
  shows convergent (λn. f n * c) ←→ convergent f
  using convergent-mult-const-iff[OF assms, of f] by (simp add: mult-ac)

lemma convergent-imp-Bseq: convergent f ⇒ Bseq f
  by (simp add: Cauchy-Bseq convergent-Cauchy)

A monotone sequence converges to its least upper bound.

lemma LIMSEQ-incseq-SUP:
  fixes X :: nat ⇒ 'a::{conditionally-complete-linorder,linorder-topology}
  assumes u: bdd-above (range X)
    and X: incseq X
  shows X −→ (SUP i. X i)
  by (rule order-tendstoI)
    (auto simp: eventually-sequentially u less-cSUP-iff
     intro: X[THEN incseqD] less-le-trans cSUP-lessD[OF u])

lemma LIMSEQ-decseq-INF:
  fixes X :: nat ⇒ 'a::{conditionally-complete-linorder, linorder-topology}
  assumes u: bdd-below (range X)
    and X: decseq X
  shows X −→ (INF i. X i)
  by (rule order-tendstoI)
    (auto simp: eventually-sequentially u cINF-less-iff
     intro: X[THEN decseqD] le-less-trans less-cINF-D[OF u])

Main monotonicity theorem.

lemma Bseq-mono-convergent: Bseq X ⇒ monoseq X ⇒ convergent X
  for X :: nat ⇒ real
  by (auto simp: monoseq-iff convergent-def intro: LIMSEQ-decseq-INF LIMSEQ-incseq-SUP
    dest: Bseq-bdd-above Bseq-bdd-below)

lemma Bseq-mono-convergent: Bseq X ⇒ (∀ m n. m ≤ n → X m ≤ X n) ⇒
  convergent X
  for X :: nat ⇒ real
THEORY “Limits”

by (auto intro!: Bseq-monoseq-convergent incseq-imp-monoseq simp: incseq-def)

lemma monoseq-imp-convergent iff-Bseq: monoseq f \iff convergent f \iff Bseq f
for f :: nat \Rightarrow real
using Bseq-monoseq-convergent[of f] convergent-imp-Bseq[of f] by blast

lemma Bseq-monoseq-convergent’-inc:
fixes f :: nat \Rightarrow real
shows Bseq (λn. f (n + M)) \implies (∀m n. M ≤ m \implies m ≤ n \implies f m ≤ f n)
\implies convergent f
by (subst convergent-ignore-initial-segment [symmetric, of - M])
(auto intro!: Bseq-monoseq-convergent simp: monoseq-def)

lemma Bseq-monoseq-convergent’-dec:
fixes f :: nat \Rightarrow real
shows Bseq (λn. f (n + M)) \implies (∀m n. M ≤ m \implies m ≤ n \implies f m ≥ f n)
\implies convergent f
(auto intro!: Bseq-monoseq-convergent simp: monoseq-def)

lemma Cauchy-iff: Cauchy X \iff (∀ε>0. ∃M. ∀m≥M. ∀n≥M. norm (X m - X n) < ε)
for X :: nat \Rightarrow 'a::real-normed-vector
unfolding Cauchy-def dist-norm ..

lemma CauchyI: (∀ε. 0 < ε \implies ∃M. ∀m≥M. ∀n≥M. norm (X m - X n) < ε) \implies Cauchy X
for X :: nat \Rightarrow 'a::real-normed-vector
by (simp add: Cauchy-iff)

lemma CauchyD: Cauchy X \implies 0 < ε \implies ∃M. ∀m≥M. ∀n≥M. norm (X m - X n) < ε
for X :: nat \Rightarrow 'a::real-normed-vector
by (simp add: Cauchy-iff)

lemma incseq-convergent:
fixes X :: nat \Rightarrow real
assumes incseq X
and ∀i. X i ≤ B
obtains L where X \longrightarrow L \forall i. X i ≤ L
proof atomize-elim
from incseq-bounded[OF assms] (incseq X) Bseq-monoseq-convergent[of X]
obtain L where X \longrightarrow L
by (auto simp: convergent-def monoseq-def incseq-def)
with (incseq X) show \exists L. X \longrightarrow L \land (∀i. X i ≤ L)
by (auto intro!: exI[of - L] incseq-le)
qed

lemma decseq-convergent:
fixes $X : \mathbb{N} \rightarrow \mathbb{R}$
assumes $\text{decseq } X$
and $\forall i. B \leq X_i$
obtains $L$ where $X \xrightarrow{\quad} L \land \forall i. L \leq X_i$
proof
atomize-elim
from $\text{decseq-bounded}[\text{OF assms}] \langle \text{decseq } X \rangle$ $\text{Bseq-monoseq-convergent}[\text{OF } X]$
obtain $L$ where $X \xrightarrow{\quad} L$
by (auto simp: convergent_def monoseq_def decseq_def)
with $\langle \text{decseq } X \rangle$
show $\exists L. X \xrightarrow{\quad} L \land (\forall i. L \leq X_i)$
by (auto intro!: exI $\langle \text{OF - } L \rangle$ decseq_ge)
qed

lemma monoseq-convergent:
fixes $X : \mathbb{N} \rightarrow \mathbb{R}$
assumes $X : \text{monoseq } X$ and $B : \forall i. |X_i| \leq B$
obtains $L$ where $X \xrightarrow{\quad} L$
using $X$ unfolding monoseq_iff
proof
assume incseq X
show thesis
using $\text{abs-le-D1} \ [\text{OF } B] \ \text{incseq-convergent} \ [\text{OF } \langle \text{incseq } X \rangle]$ that by meson
next
assume decseq X
show thesis
using decseq-convergent $\langle \text{OF } \langle \text{decseq } X \rangle \rangle$ that
by (metis B abs-le_iff add.inverse_inverse neg-le_iff_le)
qed

109.8 More about filterlim (thanks to Wenda Li)

lemma filterlim-at-infinity-times:
fixes $f : 'a \Rightarrow 'b :: \text{real-normed-field}$
assumes $\text{filterlim } f \text{ at-infinity } F \ \text{filterlim } g \text{ at-infinity } F$
shows $\text{filterlim } (\lambda x. f x \ast g x) \text{ at-infinity } F$
proof
have $((\lambda x. \text{inverse } (f x) \ast \text{inverse } (g x)) \xrightarrow{\quad} 0 \ast 0) \ F$
by (intro tendsto_mult tendsto_inverse assms filterlim-compose $\langle \text{OF tendsto-inverse-0} \rangle$)
then have $\text{filterlim } (\lambda x. \text{inverse } (f x) \ast \text{inverse } (g x)) \ (\text{at } 0) \ F$
unfolding filterlim-at using assms
by (auto intro: filterlim-at-infinity-imp-eventually-ne tendsto-imp-eventually-ne eventually-conj)
then show $\top$thesis
by (subst filterlim_inverse-at_iff[symmetric]) simp_all
qed

lemma filterlim-at-top-at-bot[elim]:
fixes $f : 'a \Rightarrow 'b :: \text{unbounded-dense-linorder}$ and $F : 'a \text{ filter}$
assumes $\text{top} \ \text{filterlim } f \text{ at-top } F \ \text{and} \ \text{bot} : \text{filterlim } f \text{ at-bot } F \ \text{and} \ \ F \neq \text{bot}$
shows $\top$
proof
proof
  obtain $c :: 'b$ where True by auto
  have $\forall F \in F. c < f$ using top unfolding filterlim-at-top-dense by auto
  moreover have $\forall F \in F. f < c$
    using bot unfolding filterlim-at-bot-dense by auto
  ultimately have $\forall F \in F. f < f \land f < c$
    using eventually-conj by auto
  then have $\forall F \in F. False$ by (auto elim: eventually-mono)
  then show False using $\langle F \neq bot \rangle$ by auto
qed

lemma filterlim-at-top-nhds[elim] :
  fixes $f :: 'a \Rightarrow 'b :: \{ unbounded-dense-linorder, order-topology \}$ and $F :: 'a filter$
  assumes top: filterlim $f$ at-top $F$ and tendsto: $(f \longrightarrow c)$ $F$ and $F \neq bot$
  shows False
proof
  obtain $c :: 'b$ where $c' < c$ using gt-ex by blast
  have $\forall F \in F. c' < f$ using top unfolding filterlim-at-top-dense by auto
  moreover have $\forall F \in F. f < c'$
    using order-tendsto[OF tendsto, of $c' < c$] by auto
  ultimately have $\forall F \in F. f < f \land f < c'$
    using eventually-conj by auto
  then have $\forall F \in F. False$ by (auto elim: eventually-mono)
  then show False using $\langle F \neq bot \rangle$ by auto
qed

lemma filterlim-at-bot-nhds[elim] :
  fixes $f :: 'a \Rightarrow 'b :: \{ unbounded-dense-linorder, order-topology \}$ and $F :: 'a filter$
  assumes top: filterlim $f$ at-bot $F$ and tendsto: $(f \longrightarrow c)$ $F$ and $F \neq bot$
  shows False
proof
  obtain $c :: 'b$ where $c' < c$ using lt-ex by blast
  have $\forall F \in F. c' > f$ using top unfolding filterlim-at-bot-dense by auto
  moreover have $\forall F \in F. f > c'$
    using order-tendsto[OF tendsto, of $c' < c$] by auto
  ultimately have $\forall F \in F. c' < f \land f < c'$
    using eventually-conj by auto
  then have $\forall F \in F. False$ by (auto elim: eventually-mono)
  then show False using $\langle F \neq bot \rangle$ by auto
qed

lemma eventually-times-inverse-1 :
  fixes $f :: 'a \Rightarrow 'b :: \{ field, t2-space \}$
  assumes $(f \longrightarrow c)$ $F \neq 0$
  shows $\forall F \in F. inverse (f) * f = 1$
by (smt (verit assms eventually-mono mult.commute right-inverse tendsto-imp-eventually-ne)
lemma filterlim-at-infinity-divide-iff:
  fixes f::'a ⇒ 'b::real-normed-field
  assumes (f ----> c) F c≠0
  shows (LIM x F. f x / g x :> at-infinity) ↔ (LIM x F. g x :> at 0)
proof
  assume LIM x F. f x / g x :> at-infinity
  then have LIM x F. inverse (f x) * (f x / g x) :> at-infinity 
    using assms tendsto-inverse tendsto-mult-filterlim-at-infinity by fastforce
  then have LIM x F. inverse (g x) :> at-infinity 
    apply (elim filterlim-mono-eventually)
    using eventually-times-inverse-1[OF assms]
    by (auto elim: eventually-mono simp add: field-simps)
  then show filterlim g (at 0) F using filterlim-inverse-at-iff[symmetric] by force

next
  assume filterlim g (at 0) F
  then have filterlim (λx. inverse (g x)) at-infinity F 
    using filterlim-compose filterlim-inverse-at-infinity by blast
  then have LIM x F. f x * inverse (g x) :> at-infinity 
    using tendsto-mult-filterlim-at-infinity[OF assms, of λx. inverse(g x)]
    by simp
  then show LIM x F. f x / g x :> at-infinity by (simp add: divide-inverse)
qed

lemma filterlim-tendsto-pos-mult-at-top-iff:
  fixes f::'a ⇒ real
  assumes (f ----> c) F and 0 < c
  shows (LIM x F. (f x * g x) :> at-top) ↔ (LIM x F. g x :> at-top)
proof
  assume filterlim g at-top F
  then have LIM x F. f x * g x :> at-top 
    using filterlim-tendsto-pos-mult-at-top[OF assms] by auto
next
  assume asm:LIM x F. f x * g x :> at-top 
  have ((λx. inverse (f x))) ----> inverse c F 
    using tendsto-inverse[OF assms(1)] 〈0<c〉 by auto
  moreover have inverse c >0 using assms(2) by auto
  ultimately have LIM x F. inverse (f x) * (f x * g x) :> at-top 
    using filterlim-tendsto-pos-mult-at-top[OF - - asm, of λx. inverse (f x) inverse c]
    by auto
  then show LIM x F. g x :> at-top 
    apply (elim filterlim-monoc-eventually)
    apply simp-all[2]
    using eventually-times-inverse-1[OF assms(1)] 〈c>0〉 eventually-mono by fastforce
qed

lemma filterlim-tendsto-pos-mult-at-bot-iff:

THEORY “Limits”

fixes $c :: \mathbb{R}$
assumes $(f \longrightarrow c) \cdot F \cdot 0 < c$
shows $(\lim x. F \cdot f \cdot x \cdot g \cdot x) \iff \lim g \cdot \text{at-bot} \cdot F$
using $\limlim-\text{tendsto-\text{pos-mult-\text{at-top-iff}}[OF \ \text{assms}(1,2), \ of \ \lambda x. - g \cdot x]}$
unfolding $\limlim-\text{uminus-\text{at-bot}} \ \text{by simp}$

lemma $\limlim-\text{tendsto-\text{neg-mult-\text{at-top-iff}}}$:
fixes $f :: 'a \Rightarrow \mathbb{R}$
assumes $(f \longrightarrow c) \cdot F \cdot c < 0$
shows $(\lim x. F \cdot f \cdot x \cdot g \cdot x) \iff (\lim x. F \cdot g \cdot x) \cdot \text{at-top}$
proof –
have $(\lim x. F \cdot f \cdot x \cdot g \cdot x) = (\lim x. F \cdot - g \cdot x)$
apply (rule $\limlim-\text{tendsto-\text{pos-mult-\text{at-top-iff}}[OF \ \text{assms}(1,2), \ of \ \lambda x. - f \cdot x \cdot c \cdot F \cdot \lambda x. - g \cdot x, \ \text{simplified}]}$
using $\text{assms \ by \ (auto \ intro: \ \text{tendsto-intros})}$
also have \ldots $= (\lim x. F \cdot g \cdot x) \cdot \text{at-bot}$
using $\limlim-\text{uminus-\text{at-top}}[\text{symmetric}] \ \text{by \ auto}$
finally show $?\text{thesis}$.
qed

lemma $\limlim-\text{tendsto-\text{neg-mult-\text{at-bot-iff}}}$:
fixes $c :: \mathbb{R}$
assumes $(f \longrightarrow c) \cdot F \cdot 0 > c$
shows $(\lim x. F \cdot f \cdot x \cdot g \cdot x) \iff \limlim g \cdot \text{at-top} \cdot F$
using $\limlim-\text{tendsto-\text{neg-mult-\text{at-top-iff}}[OF \ \text{assms}(1,2), \ of \ \lambda x. - g \cdot x]}$
unfolding $\limlim-\text{uminus-\text{at-top}} \ \text{by simp}$

109.9 Power Sequences

lemma $\Bseq-\text{realpow}$: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow \Bseq (\lambda n. x \cdot ^n n)$
for $x :: \mathbb{R}$
by (metis decseq-bounded decseq-def power-decreasing zero_le_power)

lemma $\text{monoseq-\text{realpow}}$: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow \text{monoseq} (\lambda n. x \cdot ^n n)$
for $x :: \mathbb{R}$
using $\text{monoseq-def power-decreasing \ by \ blast}$

lemma $\text{convergent-\text{realpow}}$: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow \text{convergent} (\lambda n. x \cdot ^n n)$
for $x :: \mathbb{R}$
by (blast intro:: $\Bseq$-monoseq-convergent $\Bseq$-realpow monoseq-realpow)

lemma $\text{LIMSEQ-\text{inverse-\text{realpow-zero}}}$: $1 < x \Longrightarrow (\lambda n. \text{inverse} (x \cdot ^n n)) \longrightarrow 0$
for $x :: \mathbb{R}$
by (rule $\limlim-\text{compose}[OF \ \text{tendsto-\text{inverse-0}}, \ \text{filterlim-\text{realpow-\text{sequentially-\text{gt1}}}]}$)

simp

lemma $\text{LIMSEQ-\text{realpow-zero}}$:
fixes $x :: \mathbb{R}$
assumes $0 \leq x < 1$
shows \((\lambda n. x \sim n) \longrightarrow 0\)

proof (cases \(x = 0\))
  case False
  with \(0 \leq x\) have \(1 < \text{inverse } x\)
    using \(x < 1\) by (simp add: one-less-inverse)
  then have \((\lambda n. \text{inverse } (\text{inverse } x \sim n)) \longrightarrow 0\)
    by (rule LIMSEQ-inverse-realpow-zero)
  then show \(\text{thesis}\) by (simp add: power-inverse)
next
  case True
  show \(\text{thesis}\)
    by (rule LIMSEQ-imp-Suc) (simp add: True)
qed

lemma LIMSEQ-power-zero [tendsto-intros]: \(\text{norm } x < 1 \implies (\lambda n. x \sim n) \longrightarrow 0\)
  for \(x :: 'a::real-normed-algebra-1\)
  apply (drule LIMSEQ-realpow-zero [OF norm-ge-zero])
  by (simp add: Zfun-le norm-power-ineq tendsto-Zfun-iff)

lemma LIMSEQ-divide-realpow-zero: \(1 < x \implies (\lambda n. a / (x \sim n) :: real) \longrightarrow 0\)
  by (rule tendsto-divide-0 [OF tendsto-const filterlim-realpow-sequentially-gt1]) simp

lemma tendsto-power-zero:
  fixes \(x::'a::real-normed-algebra-1\)
  assumes \(\text{norm } x < 1\)
  shows \((\lambda y. x ^ (f y)) \longrightarrow 0)\ F\)
proof (rule tendstoI)
  fix \(e::\text{real}\)
  assume \(0 < e\)
  from tendstoD [OF LIMSEQ-power-zero [OF \(\text{norm } x < 1\)] [OF \(0 < e\)]
  have \(\forall F \cdot x \sim n < e\)
    by simp
  then obtain \(N\) where \(N: \text{norm } (x \sim n) < e\) if \(n \geq N\) for \(n\)
    by (auto simp: eventually-sequentially)
  have \(\forall F \cdot i \in F. f i \geq N\)
    using \(\text{filterlim } f \text{ sequentially } F\)
    by (simp add: filterlim-at-top)
  then show \(\forall F \cdot i \in F. \text{dist } (x \sim f i) 0 < e\)
    by eventually-elim (auto simp: N)
qed

Limit of \(c^n\) for \(|c| < (1::'a)\).

lemma LIMSEQ-abs-realpow-zero: \(|c| < 1 \implies (\lambda n. |c| \sim n :: real) \longrightarrow 0\)
  by (rule LIMSEQ-realpow-zero [OF abs-ge-zero])

lemma LIMSEQ-abs-realpow-zero2: \(|c| < 1 \implies (\lambda n. c \sim n :: real) \longrightarrow 0\)
by (rule LIMSEQ-power-zero) simp

109.10 Limits of Functions

lemma LIM-eq: f −→ L = (∀ r > 0. ∃ s > 0. ∀ x. x ≠ a ∧ norm (x - a) < s →→ norm (f x - L) < r)
  for a :: 'a::real-normed-vector and L :: 'b::real-normed-vector
  by (simp add: LIM-def dist-norm)

lemma LIM-I: (∀r. 0 < r ⇒ ∃ s > 0. ∀ x. x ≠ a ∧ norm (x - a) < s →→ norm (f x - L) < r)
  for a :: 'a::real-normed-vector and L :: 'b::real-normed-vector
  by (simp add: LIM-eq)

lemma LIM-D: f −→ L ⇒ 0 < r ⇒ ∃ s > 0. ∀ x. x ≠ a ∧ norm (x - a) < s →→ norm (f x - L) < r
  for a :: 'a::real-normed-vector and L :: 'b::real-normed-vector
  by (simp add: LIM-eq)

lemma LIM-offset: f −→ L ⇒ (λx. f (x + k)) −→ (a - k) −→ L
  for a :: 'a::real-normed-vector
  by (simp add: filtermap-at-shift[symmetric, of a k] filterlim-def filtermap-filtermap)

lemma LIM-offset-zero: (λh. f (a + h)) −→ 0 −→ L
  for a :: 'a::real-normed-vector
  by (drule LIM-offset [where k = a]) (simp add: add.commute)

lemma LIM-offset-zero-cancel: (λh. f (a + h)) −→ 0 −→ L "⇒" f −→ L
  for a :: 'a::real-normed-vector
  by (drule LIM-offset [where k = - a]) simp

lemma LIM-offset-zero-iff: NO-MATCH 0 a "⇒" f −→ L ⇐≤ (λh. f (a + h)) −→ 0 −→ L
  for f :: 'a :: real-normed-vector ⇒ -
  using LIM-offset-zero-cancel[of f a L] LIM-offset-zero[of f L a] by auto

lemma tendsto-offset-zero-iff:
  fixes f :: 'a :: real-normed-vector ⇒ -
  assumes NO-MATCH 0 a a ∈ S open S
  shows (f −→ L) (at a within S) "⇐≤" ((λh. f (a + h)) −→ L) (at 0)
  using assms by (simp add: tendsto-within-open NO-MATCH LIM-offset-zero-iff)

lemma LIM-zero: (f −→ l) F "⇒" (λx. f x - l) −→ 0 F
  for f :: 'a ⇒ 'b::real-normed-vector
  unfolding tendsto-iff dist-norm by simp

lemma LIM-zero-cancel:
  fixes f :: 'a ⇒ 'b::real-normed-vector
shows \((\lambda x. f x - l) \longrightarrow 0\) \(F \Rightarrow (f \longrightarrow l) F\)

unfolding tendsto_iff dist-norm by simp

lemma LIM-zero_iff: \((\lambda x. f x - l) \longrightarrow 0\) \(F = (f \longrightarrow l) F\)

for \(f :: 'a \Rightarrow 'b::real-normed-vector\)

unfolding tendsto_iff dist-norm by simp

lemma LIM-imp-LIM:
  fixes \(f :: 'a::topological-space \Rightarrow 'b::real-normed-vector\)
  assumes \(f : f - a \rightarrow l\)
  and le: \(\forall x. x \neq a \Longrightarrow \text{norm} (g x - m) \leq \text{norm} (f x - l)\)
  shows \(g - a \rightarrow m\)
  by (rule metric-LIM-imp-LIM [OF f]) (simp add: dist-norm le)

lemma LIM-equal2:
  fixes \(f g :: 'a::real-normed-vector \Rightarrow 'b::topological-space\)
  assumes \(0 < R\)
  and \(\forall x. x \neq a \Longrightarrow \text{norm} (x - a) < R \Longrightarrow f x = g x\)
  shows \(g - a \rightarrow l = f - a \rightarrow l\)
  by (rule metric-LIM-equal2 [OF - assms]) (simp-all add: dist-norm)

lemma LIM-compose2:
  fixes \(a :: 'a::real-normed-vector\)
  assumes \(f : f - a \rightarrow b\)
  and \(g : g - b \rightarrow c\)
  and inj: \(\exists d > 0. \forall x. x \neq a \wedge \text{norm} (x - a) < d \rightarrow f x \neq b\)
  shows \((\lambda x. g (f x)) - a \rightarrow c\)
  by (rule metric-LIM-compose2 [OF f g inj [folded dist-norm]])

lemma real-LIM-sandwich-zero:
  fixes \(f g :: 'a::topological-space \Rightarrow \text{real}\)
  assumes \(f : f - a \rightarrow 0\)
  and \(1: \forall x. x \neq a \Longrightarrow \text{norm} (x - a) \leq g x\)
  and \(2: \forall x. x \neq a \Longrightarrow g x \leq f x\)
  shows \(g - a \rightarrow 0\)
  proof (rule LIM-imp-LIM [OF f])
    fix \(x\)
    assume \(x : x \neq a\)
    with \(1\) have \(\text{norm} (g x - 0) = g x\) by simp
    also have \(g x \leq f x\) by (rule 2 [OF \(x\)])
    also have \(f x \leq |f x|\) by (rule abs-ge-self)
    also have \(|f x| = \text{norm} (f x - 0)\) by simp
    finally show \(\text{norm} (g x - 0) \leq \text{norm} (f x - 0)\).
  qed

109.11 Continuity

lemma LIM-isCont_iff: \((f -a f a) = ((\lambda h. f (a + h)) -0 \rightarrow f a)\)
for $f :: 'a::real-normed-vector \Rightarrow 'b::topological-space$
by (rule iffI [OF LIM-offset-zero LIM-offset-zero-cancel])

lemma isCont-iff: isCont $f x = (\lambda h. f (x + h)) -0 \rightarrow f x$
for $f :: 'a::real-normed-vector \Rightarrow 'b::topological-space$
by (simp add: isCont-def LIM-isCont-iff)

lemma isCont-LIM-compose2:
fixes $a :: 'a::real-normed-vector$
assumes $f$ [unfolded isCont-def]: isCont $f a$
and $g$: $g -f a \rightarrow l$
and inj: $\exists d>0. \forall x. x \neq a \land \text{norm} (x - a) < d \rightarrow f x \neq f a$
shows $(\lambda x. g (f x)) -a \rightarrow l$
by (rule LIM-compose2 [OF $f \ g \ \text{inj}$])

lemma isCont-norm [simp]: isCont $f a \Rightarrow isCont (\lambda x. \text{norm} (f x)) a$
for $f :: 'a::t2-space \Rightarrow 'b::real-normed-vector$
by (fact continuous-norm)

lemma isCont-rabs [simp]: isCont $f a \Rightarrow isCont (|f x|) a$
for $f :: 'a::t2-space \Rightarrow 'b::real-normed-vector$
by (fact continuous-rabs)

lemma isCont-add [simp]: isCont $f a \Rightarrow isCont (g a \Rightarrow isCont (\lambda x. f x + g x)) a$
for $f :: 'a::t2-space \Rightarrow 'b::topological-monoid-add$
by (fact continuous-add)

lemma isCont-minus [simp]: isCont $f a \Rightarrow isCont (\lambda x. - f x) a$
for $f :: 'a::t2-space \Rightarrow 'b::real-normed-vector$
by (fact continuous-minus)

lemma isCont-diff [simp]: isCont $f a \Rightarrow isCont (g a \Rightarrow isCont (\lambda x. f x - g x)) a$
for $f :: 'a::t2-space \Rightarrow 'b::real-normed-vector$
by (fact continuous-diff)

lemma isCont-mult [simp]: isCont $f a \Rightarrow isCont (g a \Rightarrow isCont (\lambda x. f x * g x)) a$
for $f g :: 'a::t2-space \Rightarrow 'b::real-normed-algebra$
by (fact continuous-mult)

lemma (in bounded-linear) isCont: isCont $g a \Rightarrow isCont (\lambda x. f (g x)) a$
by (fact continuous)

lemma (in bounded-bilinear) isCont: isCont $f a \Rightarrow isCont g a \Rightarrow isCont (\lambda x. f x ** g x) a$
by (fact continuous)

lemmas isCont-scaleR [simp] =
bounded-bilinear.isCont [OF bounded-bilinear-scaleR]
lemmas isCont-of-real [simp] = 
bounded-linear.isCont [OF bounded-linear-of-real]

lemma isCont-power [simp]: isCont f a \implies isCont (λx. f x ^ n) a 
for f :: 'a::t2-space ⇒ 'b::(power,real-normed-algebra) 
by (fact continuous-power)

lemma isCont-sum [simp]: ∀ i ∈ A. isCont (f i) a \implies isCont (λx. ∑ i ∈ A. f i x) a 
for f :: 'a ⇒ 'b::t2-space ⇒ 'c::topological-comm-monoid-add 
by (auto intro: continuous-sum)

109.12 Uniform Continuity

lemma uniformly-continuous-on-def:
  fixes f :: 'a::metric-space ⇒ 'b::metric-space 
  shows uniformly-continuous-on s f ↔
  (∀ e > 0. ∃ d > 0. ∀ x ∈ s. ∀ x' ∈ s. dist x x' < d → dist (f x) (f x') < e)
  unfolding uniformly-continuous-on-uniformity 
  uniformity-dist filterlim-INF filterlim-principal eventually-inf-principal 
  by (force simp: uniformity-dist [symmetric] eventually-uniformity-metric)

abbreviation isUCont :: 'a::metric-space ⇒ 'b::metric-space ⇒ bool 
  where isUCont f ≡ uniformly-continuous-on UNIV f

lemma isUCont-def: isUCont f ↔ (∀ r > 0. ∃ s > 0. ∀ x y. dist x y < s → dist (f x) (f y) < r)
  by (auto simp: uniformly-continuous-on-def dist-commute)

lemma isUCont-isCont: isUCont f \implies isCont f x 
  by (drule uniformly-continuous-imp-continuous) (simp add: continuous-on-eq-continuous-at)

lemma uniformly-continuous-on-Cauchy:
  fixes f :: 'a::metric-space ⇒ 'b::metric-space 
  assumes uniformly-continuous-on S f Cauchy X \\ A n. X n ∈ S 
  shows Cauchy (λn. f (X n)) 
  using assms 
  unfolding uniformly-continuous-on-def by (meson Cauchy-def)

lemma isUCont-Cauchy: isUCont f \implies Cauchy X \implies Cauchy (λn. f (X n)) 
  by (rule uniformly-continuous-on-Cauchy [where S=UNIV and f=f]) simp-all

lemma (in bounded-linear) isUCont: isUCont f 
  unfolding isUCont-def dist-norm 
proof (intro allI impI)
  fix r :: real 
  assume r: 0 < r 
  obtain K where K: 0 < K and norm-le: norm (f x) ≤ norm x * K for x 
  using pos-bounded by blast 
  show ∃ s > 0. ∀ x y. norm (x − y) < s → norm (f x − f y) < r
proof (rule exI, safe)
from r K show \( 0 < r / K \) by simp
next
  fix x y :: 'a
  assume xy: norm \((x - y)\) < r / K
  have norm \((f x - f y)\) = norm \((f (x - y))\) by (simp only: diff)
  also have \(\ldots \leq norm (x - y) * K \) by (rule norm-le)
  also from K xy have \(\ldots < r \) by (simp only: pos-less-divide-eq)
  finally show norm \((f x - f y)\) < r .
qed

lemma (in bounded-linear) Cauchy: Cauchy X \(\equiv\) Cauchy (\(\lambda n. f \(X n)\))
by (rule isUCont [THEN isUCont-Cauchy])

lemma LIM-less-bound:
  fixes f :: real \(\Rightarrow\) real
  assumes ev: \(b < x \forall x' \in \{ b <..< x \}. \) \(0 \leq f x' \) and isCont f x
  shows \(0 \leq f x\)
proof (rule tendsto-lowerbound)
  show \((f \longmapsto f x) \) (at-left x)
    using \(\{isCont f x\}\) by (simp add: filterlim-at-split isCont-def)
  show eventually \((\lambda x. \) \(0 \leq f x) \) (at-left x)
    using ev by (auto simp: eventually-at dist-real-def intro: \!)
qed simp

109.13 Nested Intervals and Bisection – Needed for Compactness

lemma nested-sequence-unique:
  assumes \(\forall n. f n \leq f (Suc n) \forall n. g (Suc n) \leq g n \forall n. f n \leq g n \) \(\lambda n. f n - g n\) \(\longmapsto 0 \)
  shows \(\exists l :: real. ((\forall n. f n \leq l) \land f \longmapsto l) \land ((\forall n. l \leq g n) \land g \longmapsto l)\)
proof –
  have incseq f unfolding incseq-Suc-iff by fact
  have decseq g unfolding decseq-Suc-iff by fact
  have \(f n \leq g 0 \) for \(n\)
  proof –
    from \(\{decseq g\}\) have \(g n \leq g 0 \)
      by (rule decseqD) simp
    with \(\forall n. f n \leq g n\)[THEN spec, of \(n\)] show \(?thesis\)
      by auto
  qed
  then obtain \(u\) where \(f \longmapsto u \forall i. f i \leq u\)
    using incseq-convergent[OF \(\{incseq f\}\)] by auto
  moreover have \(f 0 \leq g n \) for \(n\)
  proof –
    from \(\{incseq f\}\) have \(f 0 \leq f n\) by (rule incseqD) simp
    with \(\forall n. f n \leq g n\)[THEN spec, of \(n\)] show \(?thesis\)
by simp
qed
then obtain \( l \) where \( g \longrightarrow l \forall i. \ l \leq g \ i \)
using decseq-convergent[OF `decseq g`] by auto
moreover note LIMSEQ-unique[OF assms(4) tendsto-diff[OF `f \longrightarrow u \ g` \longrightarrow l.]]
ultimately show \(?thesis by auto\)
qed

lemma Bolzano[consumes 1, case-names trans local]:
fixes \( P :: \ real \Rightarrow \ real \Rightarrow \ bool \)
assumes [arith]: \( a \leq b \)
and trans: \( \forall a \ b \ c. \ P \ a \ b \Longrightarrow P \ b \ c \Longrightarrow a \leq b \Longrightarrow b \leq c \Longrightarrow P \ a \ c \)
and local: \( \forall x. \ a \leq x \Longrightarrow x \leq b \Longrightarrow \exists d>0. \forall a \ b. \ a \leq x \land x \leq b \land b - a < d \Longrightarrow P \ a \ b \)
shows \( P \ a \ b \)
proof -
define bisect where \( \text{bisect} \equiv \lambda (x,y). \) if \( P \ x \ ((x+y)/2) \) then \((x+y)/2, y)\) else \((x, (x+y)/2)\)
define \( l\ u \) where \( l \ n \equiv \text{fst} \ ((\text{bisect}^\sim n)(a,b)) \) and \( u \ n \equiv \text{snd} \ ((\text{bisect}^\sim n)(a,b)) \)
for \( n \)
have \( l[simp]; \ l \ 0 = a \land l \ (Suc \ n) = (if \ P \ (l \ n) \ ((l \ n + u \ n) / 2) \) then \((l \ n + u \ n) / 2 \) else \( l \ n)\)
and \( u[simp]; \ u \ 0 = b \land u \ (Suc \ n) = (if \ P \ (l \ n) \ ((l \ n + u \ n) / 2) \) then \( u \ n \) else \((l \ n + u \ n) / 2)\)
by (simp-all add: l-def u-def bisect-def split: prod.split)
have \( \exists x. \ (\forall n. \ l \ n \leq x) \land l \longrightarrow x \) \land \((\forall n. \ x \leq u \ n) \land u \longrightarrow x)\)
proof (safe intro; nested-sequence-unique)
  show \( l \ n \leq l \ (Suc \ n) \ u \ (Suc \ n) \leq u \ n \) for \( n \)
  by (induct \( n \) auto)
next
have \( l \ n - u \ n = (a - b) / 2^n \) for \( n \)
  by (induct \( n \) (auto simp: field-simps)
then show \( (\lambda n. \ l \ n - u \ n) \longrightarrow 0 \)
  by (simp add: LIMSEQ-divide-realpow-zero)
qed fact
then obtain \( x \) where \( x: \ (\forall n. \ l \ n \leq x \land l \longrightarrow x \ \land \ (\forall n. \ x \leq u \ n) \land l \longrightarrow x) \)
by auto
obtain \( d \) where \( \theta < d \) and \( d: a \leq x \Longrightarrow x \leq b \Longrightarrow b - a < d \Longrightarrow P \ a \ b \) for \( a\ b \)
  using \( (l \ n \leq x) \ (x \leq u \ 0) \) local[of \( x \)] by auto
show \( P \ a \ b \)
proof (rule ccontr)
  assume \( \neg \ P \ a \ b \)
  have \( \neg \ P \ (l \ n) \ (u \ n) \) for \( n \)
proof (induct n)
  case 0
  then show ?case
    by (simp add: ¬ P a b)
next
  case (Suc n)
  with trans[of l n (l n + u n) / 2 u n] show ?case
    by auto
qed
moreover
  { have eventually (λn. x - d / 2 < l n) sequentially
      using (intro order-tendstoD[of - x]) auto
  moreover have eventually (λn. u n < x + d / 2) sequentially
      using (intro order-tendstoD[of - x]) auto
  ultimately have eventually (λn. P (l n) (u n)) sequentially
    proof eventually-elim
      case (elim n)
      from add-strict-mono[OF this] have u n - l n < d by simp
      with x show P (l n) (u n) by (rule d)
    qed
  } ultimately show False by simp
qed

lemma compact-Icc[simp, intro]: compact {a .. b::real}
proof (cases a ≤ b, rule compactI)
  fix C
  assume C: a ≤ b ∀ t ∈ C. open t {a .. b} ⊆ ∪ C
  define T where T = {a .. b}
  from C(1,3) show ∃ C' ⊆ C. finite C' ∧ {a .. b} ⊆ ∪ C'
    proof (induct rule: Bolzano)
      case (trans a b c)
      then have *: {a .. c} = {a .. b} ∪ {b .. c}
        by auto
      with trans obtain C1 C2
        where C1 ⊆ C finite C1 {a .. b} ⊆ ∪ C1 C2 ⊆ C finite C2 {b .. c} ⊆ ∪ C2
        by auto
      with unfolding ?case
        unfolding * by (intro exI[of - C1 ∪ C2]) auto
    next
      case (local x)
      with C have x ∈ ∪ C by auto
      with C(2) obtain c where x ∈ c open c c ∈ C
        by auto
      then obtain e where 0 < e {x - e <..< x + e} ⊆ c
        by (auto simp: open-dist dist-real-def subset-eq Ball-def abs-less-iff)
      with c ∈ C show ?case
by (safe intro: exI[of - e/2] exI[of - {c}]) auto
qed
qed simp

lemma continuous-image-closed-interval:
  fixes a b and f :: real ⇒ real
  defines S ≡ {a..b}
  assumes a ≤ b and f: continuous-on S f
  shows ∃c d. f'S = {c..d} ∧ c ≤ d
proof –
  have S: compact S S ≠ {} using ‹a ≤ b› by (auto simp: S-def)
  obtain c where c ∈ S ∀d ∈ S. f d ≤ f c using continuous-attains-sup[OF S f] by auto
  moreover obtain d where d ∈ S ∀c ∈ S. f d ≤ f c using continuous-attains-inf[OF S f] by auto
  moreover have connected (f'S)
  using connected-continuous-image[OF f] connected-Icc by (auto simp: S-def)
  ultimately have f ' S = {f d .. f c} ∧ f d ≤ f c by (auto simp: connected-iff-interval)
  then show ?thesis by auto
qed

lemma open-Collect-positive:
  fixes f :: 'a::topological-space ⇒ real
  assumes f: continuous-on s f
  shows ∃A. open A ∧ A ∩ s = {x ∈ s. 0 < f x}
  using continuous-on-open-invariant[THEN iffD1, OF f, rule-format, of {0 <..}] by (auto simp: Int-def field-simps)

lemma open-Collect-less-Int:
  fixes f g :: 'a::topological-space ⇒ real
  assumes f: continuous-on s f
  and g: continuous-on s g
  shows ∃A. open A ∧ A ∩ s = {x ∈ s. f x < g x}
  using open-Collect-positive[OF continuous-on-diff[OF g f]] by (simp add: field-simps)

109.14 Boundedness of continuous functions

By bisection, function continuous on closed interval is bounded above

lemma isCont-eq-Ub:
  fixes f :: real ⇒ 'a::linorder-topology
  shows a ≤ b ⇒ ∀x::real. a ≤ x ∧ x ≤ b → isCont f x →
  ∃M. (∀x. a ≤ x ∧ x ≤ b → f x ≤ M) ∧ (∃x. a ≤ x ∧ x ≤ b ∧ f x = M)
  using continuous-attains-sup[of {a..b} f]
  by (auto simp: continuous-at-imp-continuous-on Ball-def Bex-def)
lemma isCont-eq-Lb:
  fixes f :: real ⇒ 'a::linorder_topology
  shows a ≤ b ⇒ ∀ x. a ≤ x ∧ x ≤ b → isCont f x ⇔
    ∃ M. (∀ x. a ≤ x ∧ x ≤ b → f x ≤ M) ∧ (∃ x. a ≤ x ∧ x ≤ b ∧ f x = M)
  using continuous-attains-inf[of {a..b} f]
  by (auto simp: continuous-at-imp-continuous-on Ball_def Bex_def)

lemma isCont-bounded:
  fixes f :: real ⇒ 'a::linorder_topology
  shows a ≤ b ⇒ ∀ x. a ≤ x ∧ x ≤ b → isCont f x ⇒ ∃ M. ∀ x. a ≤ x ∧ x ≤ b → f x ≤ M
  using isCont-eq-Ub[of a b f] by auto

lemma isCont-has-Ub:
  fixes f :: real ⇒ 'a::linorder_topology
  shows a ≤ b ⇒ ∀ x. a ≤ x ∧ x ≤ b → isCont f x ⇒
    ∃ M. (∀ x. a ≤ x ∧ x ≤ b → f x ≤ M) ∧ (∃ N. N < M → (∃ x. a ≤ x ∧ x ≤ b ∧ f x = M) ∧ (∀ x. N ≤ x → (∀ x. a ≤ x ∧ x ≤ b → f x ≤ N)))
  using isCont-eq-Ub[of a b f] by auto

lemma isCont-Lb-Ub:
  fixes f :: real ⇒ real
  assumes a ≤ b ∀ x. a ≤ x ∧ x ≤ b → isCont f x
  shows ∃ L M. (∀ x. a ≤ x ∧ x ≤ b → L ≤ f x ∧ f x ≤ M) ∧
    (∀ y. L ≤ y ∧ y ≤ M → (∃ x. a ≤ x ∧ x ≤ b ∧ (f x = y)))
  proof -
    obtain M where M: a ≤ M M ≤ b ∀ x. a ≤ x ∧ x ≤ b → f x ≤ f M
      using isCont-eq-Ub[OF assms] by auto
    obtain L where L: a ≤ L L ≤ b ∀ x. a ≤ x ∧ x ≤ b → f L ≤ f x
      using isCont-eq-Lb[OF assms] by auto
    have (∀ x. a ≤ x ∧ x ≤ b → f L ≤ f x ∧ f x ≤ f M)
      using M L by simp
  moreover
  have (∀ y. f L ≤ y ∧ y ≤ f M → (∃ x≥a. x ≤ b ∧ f x = y))
  proof (cases L ≤ M)
    case True then show ?thesis
      using IVT[of f L - M] M L assms by (metis order.trans)
  next
    case False then show ?thesis
      using IVT2[of f L - M]
      by (metis L(2) M(1) assms(2) le_cases order.trans)
  qed
  ultimately show ?thesis
    by blast
  qed

Continuity of inverse function.

lemma isCont-inverse-function:
  fixes f g :: real ⇒ real
THEORY "Limits"

assumes $d: 0 < d$

and inj: $\forall z. |z-x| \leq d \Rightarrow g(fz) = z$

and cont: $\forall z. |z-x| \leq d \Rightarrow \text{isCont } fz$

shows isCont $g(fx)$

proof -

let $?A = f(x - d)$

let $?B = f(x + d)$

let $?D = \{x - d..x + d\}$

have $f: \text{continuous-on } ?D f$

using cont by (intro continuous-at-imp-continuous-on ballI) auto

then have $g: \text{continuous-on } (f)?D g$

using inj by (intro continuous-on-inv) auto

from $df$ have $\{\min ?A ?B <..< \max ?A ?B\} \subseteq f'?D$

by (intro connected-contains-luo connected-continuous-image) (auto split: split-min split-max)

with $g$ have $\text{continuous-on } \{\min ?A ?B <..< \max ?A ?B\} g$

by (rule continuous-on-subset)

moreover have $(?A < f x \land f x < ?B) \lor (?B < f x \land f x < ?A)$

using $d$ inj by (intro continuous-inj-imp-mono[of - - f inj-on-imageI2[of g, OF inj-onI]) auto

then have $fx \in \{\min ?A ?B <..< \max ?A ?B\}$

by auto

ultimately show $\text{thesis}$

by (simp add: continuous-on-eq-continuous-at)

qed

lemma isCont-inverse-function2:

fixes $f g :: \text{real} \Rightarrow \text{real}$

shows $[a < x; x < b;$

$\forall z. [a \leq z; z \leq b] \Rightarrow g(fz) = z;$

$\forall z. [a \leq z; z \leq b] \Rightarrow \text{isCont } fz] \Rightarrow \text{isCont } g(fx)$

apply (rule isCont-inverse-function [where $f=\text{f}$ and $d=\text{min } (x - a) (b - x)$])

apply (simp-all add: abs-le-iff)

done

Bartle/Sherbert: Introduction to Real Analysis, Theorem 4.2.9, p. 110.

lemma LIM-fun-gt-zero: $f -c\rightarrow l \Rightarrow 0 < l \Rightarrow \exists r. \theta < r \land (\forall x. x \neq c \land |c - x| < r \rightarrow 0 < f x)$

for $f :: \text{real} \Rightarrow \text{real}$

by (force simp: dest: LIM-D)

lemma LIM-fun-less-zero: $f -c\rightarrow l \Rightarrow l < 0 \Rightarrow \exists r. \theta < r \land (\forall x. x \neq c \land |c - x| < r \rightarrow f x < 0)$

for $f :: \text{real} \Rightarrow \text{real}$

by (drule LIM-D [where $r=-l$]) force+
lemma \textbf{LIM-fun-not-zero}: \( f - c \rightarrow l \not= 0 \implies \exists r. \ 0 < r \land (\forall x. x \neq c \land |c - x| < r \implies f x \not= 0) \)

for \( f :: \text{real} \rightarrow \text{real} \)

using \textbf{LIM-fun-gt-zero[of f l c]} \textbf{LIM-fun-less-zero[of f l c]} by (auto simp: neq_iff)

lemma \textbf{Lim-topological}:
\( (f \longrightarrow l) \net \Longleftrightarrow \text{trivial-limit net} \lor (\forall S. \text{open } S \implies l \in S \implies \text{eventually } (\lambda x. f x \in S) \net) \)

unfolding \text{tendsto-def} trivial-limit-eq by auto

lemma \textbf{eventually-within-Un}:
\( \text{eventually } P (\text{at } x \text{ within } (s \cup t)) \Longleftrightarrow \text{eventually } P (\text{at } x \text{ within } s) \land \text{eventually } P (\text{at } x \text{ within } t) \)

unfolding \text{eventually-at-filter} by (auto elim!: eventually-rev_mp)

lemma \textbf{Lim-within-Un}:
\( (f \longrightarrow l) (\text{at } x \text{ within } (s \cup t)) \longleftrightarrow (f \longrightarrow l) (\text{at } x \text{ within } s) \land (f \longrightarrow l) (\text{at } x \text{ within } t) \)

unfolding \text{tendsto-def} by (auto simp: eventually-within-Un)

end

deciduation
theory \textit{Inequalities}

\textbf{imports} \textit{Real-Vector-Spaces}

begin

lemma \textbf{Chebyshev-sum-upper}:
\( \text{fixes } a b :: \text{nat} \Rightarrow 'a::linordered-idom} \)

assumes \( \land i j. i \leq j \implies j < n \implies a i \leq a j \)

assumes \( \land i j. i \leq j \implies j < n \implies b i \geq b j \)

shows of-nat n * (\( \sum k=0..<n. a k \ast b k) \leq (\sum k=0..<n. a k) \ast (\sum k=0..<n. b k) \)

proof 
\let \( ?S = (\sum j=0..<n. (\sum k=0..<n. (a j - a k) \ast (b j - b k))) \)

\( \text{have } 2 \ast (\text{of-nat } n \ast (\sum j=0..<n. (a j \ast b j))) - (\sum j=0..<n. b j) \ast (\sum k=0..<n. a k)) = ?S \)

\( \text{by (simp only: one-add-one[symmetric] algebra-simps)} \)

\( \text{(simp add: algebra-simps sum-subtract sum.distrib sum.swap[of } \lambda i j. a i \ast b j \text{ sum-distrib-left)} \)

\( \text{also} \)
\{ \text{fix } i j :: \text{nat} \text{ assume } i < n j < n \)

\( \text{hence } a i - a j \leq 0 \land b i - b j \geq 0 \lor a i - a j \geq 0 \land b i - b j \leq 0 \)

\( \text{using } \text{assms by (cases } i \leq j \text{)} (\text{auto simp: algebra-simps)} \)

\} \text{ then have } ?S \leq 0 \)

\( \text{by (auto intro: sum-nonpos simp: mult-le-0-iff)} \)
finally show thesis by (simp add: algebra-simps)
qed

lemma Chebyshev-sum-upper-nat:
  fixes a b :: nat ⇒ nat
  shows (∀ i j. [ i≤j; j<n ] ⇒ a i ≤ a j) ⇒
         (∀ i j. [ i≤j; j<n ] ⇒ b i ≥ b j) ⇒
         n * (∑ i=0..<n. a i) * (∑ i=0..<n. b i)
  using Chebyshev-sum-upper[where 'a=real, of n a b]

end

110 Infinite Series

theory Series
imports Limits Inequalities
begin

110.1 Definition of infinite summability

definition sums :: (nat ⇒ 'a::{topological-space, comm-monoid-add}) ⇒ 'a ⇒ bool
  (infixr sums 80)
where f sums s ←→ (∀n. ∑ i<n. f i) −→ s

definition summable :: (nat ⇒ 'a::{topological-space, comm-monoid-add}) ⇒ bool
where summable f ←→ (∃ s. f sums s)

definition suminf :: (nat ⇒ 'a::{topological-space, comm-monoid-add}) ⇒ 'a
  (binder ∑)
where suminf f = (THE s. f sums s)

Variants of the definition

lemma sums-def': f sums s ←→ (∀n. ∑ i=0..n. f i) −→ s
  unfolding sums-def
  apply (subst filterlim-sequentially-Suc [symmetric])
  apply (simp only: lessThan-Suc-atMost atLeast0AtMost)
  done

lemma sums-def-lc: f sums s ←→ (∀n. ∑ i≤n. f i) −→ s
  by (simp add: sums-def' atMost-atLeast0)

lemma bounded-imp-summable:
  fixes a :: nat ⇒ 'a::{conditionally-complete-linorder,linorder-topology,linordered-comm-semiring-strict}
  assumes 0: ∀ n. a n ≥ 0 and bounded: ∀ n. (∑ k≤n. a k) ≤ B
  shows summable a
proof
  have bdd-above (range(∀n. ∑ k≤n. a k))
    by (meson bdd-aboveI2 bounded)
moreover have incseq \((\lambda n. \sum_{k \leq n} a k)\)
by \((\text{simp add: mono-def 0 sum-mono2})\)
ultimately obtain \(s\) where \((\lambda n. \sum_{k \leq n} a k) \longrightarrow s\)
using LIMSEQ-incseq-SUP by blast
then show \(\text{thesis}\)
by \((\text{auto simp: sums-def-le summable-def})\)
qed

110.2 Infinite summability on topological monoids

lemma sums-subst\([\text{trans}]: f = g \implies g \text{ sums } z \implies f \text{ sums } z\)
by simp

lemma sums-cong: \((\forall n. f n = g n) \implies f \text{ sums } c \iff g \text{ sums } c\)
by presburger

lemma sums-summable: \(f \text{ sums } l \implies \text{summable } f\)
by \((\text{simp add: sums-def summable-def, blast})\)

lemma summable-iff-convergent: \(\text{summable } f \iff \text{convergent } (\lambda n. \sum_{i < n} f i)\)
by \((\text{simp add: summable-def sums-def convergent-def})\)

lemma summable-iff-convergent': \(\text{summable } f \iff \text{convergent } (\lambda n. \text{sum } f \{..n\})\)
by \((\text{simp add: convergent-def summable-def sums-def-le})\)

lemma suminf-eq-lim: \(\text{suminf } f = \lim (\lambda n. \sum_{i < n} f i)\)
by \((\text{simp add: suminf-def sums-def lim-def})\)

lemma sums-zero\([\text{simp, intro}]: (\lambda n. 0) \text{ sums } 0\)
unfolding sums-def by simp

lemma summable-zero\([\text{simp, intro}]: \text{summable } (\lambda n. 0)\)
by \((\text{rule sums-zero [THEN sums-summable]})\)

lemma sums-group: \(f \text{ sums } s \implies 0 < k \implies (\lambda n. \text{sum } f \{n * k ..< n * k + k\}) \text{ sums } s\)
apply \((\text{simp only: sums-def sum.nat-group tendsto-def eventually-sequentially})\)
apply \((\text{erule all-forward imp-forward exE assumption})\)+
by \((\text{metis le-square mult.commute mult.left-neutral mult.le-cancel2 mult.le-mono})\)

lemma suminf-cong: \((\forall n. f n = g n) \implies \text{suminf } f = \text{suminf } g\)
by presburger

lemma summable-cong:
  fixes \(f, g:: \text{\em real-normed-vector}\)
  assumes eventually \((\lambda x. f x = g x)\) sequentially
  shows \(\text{summable } f = \text{summable } g\)
proof
  from assms obtain \(N\) where \(\forall n \geq N. f n = g n\)
by (auto simp: eventually-at-top-linorder)

define C where \( C = \sum_{k < N} f \cdot k - g \cdot k \)

from eventually-ge-at-top[of N]

have eventually \((\lambda n. \text{sum } f \{..<n\} = C + \text{sum } g \{..<n\})\) sequentially

proof eventually-elim
  case (elim n)
  then have \{..<n\} = \{..<N\} \cup \{N..<n\}
    by auto
  also have \(\text{sum } f \{..<n\} = \text{sum } f \{..<N\} + \text{sum } f \{N..<n\}\)
    by (intro sum.union-disjoint simp)
  also from N have \(\text{sum } f \{N..<n\} = \text{sum } g \{N..<n\}\)
    by (intro sum.cong simp)
  also have \(\text{sum } f \{..<N\} + \text{sum } g \{N..<n\} = C + (\text{sum } g \{..<N\} + \text{sum } g \{N..<n\})\)
    unfolding C-def by (simp add: algebra-simps sum-subtractf)
  also have \(\text{sum } g \{N..<n\} = \text{sum } g \{N..<n\}\)
    by (intro sum.union-disjoint [symmetric]) auto
  also from elim have \{..<N\} \cup \{N..<n\} = \{..<n\}
    by auto
  finally show \(\text{sum } f \{..<n\} = C + \text{sum } g \{..<n\}\).
  qed

from convergent-cong[OF this] show \(\text{thesis}\)
  by (simp add: summable-iff-convergent convergent-add-const-iff)
  qed

lemma sums-finite:
  assumes \(\text{simp}: \text{finite } N\)
  and \(f: \forall n. n \notin N \Rightarrow f \cdot n = 0\)
  shows \(\text{f sums } (\sum_{n \in N} f \cdot n)\)

proof
  have eq: \(\text{sum } f \{..<n + \text{Suc } (\text{Max } N)\} = \text{sum } f \cdot N \text{ for } n\)
    by (rule sum.mono-neutral-right) (auto simp: add-increasing less Suc eq-le f)
  show \(\text{thesis}\)
    unfolding sums-def
    by (rule LIMSEQ_offset[of - Suc (Max N)])
    (simp add: eq atLeast0LessThan del: add-Suc-right)
  qed

corollary sums-0: \((\forall n. f \cdot n = 0) \Rightarrow (f \text{ sums } 0)\)
  by (metis (no_types) finite.emptyI sum.empty sums-finite)

lemma summable-finite: \(\text{finite } N \Rightarrow (\forall n. n \notin N \Rightarrow f \cdot n = 0) \Rightarrow \text{summable } f\)
  by (rule sums-summable) (rule sums-finite)

lemma sums-If-finite-set: \(\text{finite } A \Rightarrow (\lambda r. \text{if } r \in A \text{ then } f \cdot r \text{ else } 0) \text{ sums } (\sum_{r \in A} f \cdot r)\)
  using sums-finite[of A (\lambda r. \text{if } r \in A \text{ then } f \cdot r \text{ else } 0)] by simp

lemma summable-If-finite-set[simp, intro]: \(\text{finite } A \Rightarrow \text{summable } (\lambda r. \text{if } r \in A \text{ then } f \cdot r \text{ else } 0)\)
THEORY "Series"

then f r else 0)

by (rule sums-summable) (rule sums-If-finite-set)

lemma sums-If-finite: finite {r. P r} \implies (\lambda r. if P r then f r else 0) sums (∑ r | P r, f r)

using sums-If-finite-set[of {r. P r}] by simp

lemma summable-If-finite[simp, intro]: finite {r. P r} \implies summable (\lambda r. if P r then f r else 0)

by (rule sums-summable) (rule sums-If-finite)

lemma sums-single: (λr. if r = i then f r else 0) sums f i

using sums-If-finite[of λr. r = i] by simp

lemma summable-single[simp, intro]: summable (λr. if r = i then f r else 0)

by (rule sums-summable) (rule sums-single)

context

fixes f :: nat ⇒ 'a::{t2-space,comm-monoid-add}

begin

lemma summable-sums[intro]: summable f \implies f sums (suminf f)

by (simp add: summable-def sums-def suminf-def)

(metis convergent-LIMSEQ-iff convergent-def lim-def)

lemma summable-LIMSEQ: summable f \implies (λn. ∑ i<n. f i) −−−−→ suminf f

by (rule summable-sums [unfolded sums-def])

lemma summable-LIMSEQ': summable f \implies (λn. ∑ i≤n. f i) −−−−→ suminf f

using sums-def-le by blast

lemma sums-unique: f sums s \implies s = suminf f

by (metis limI suminf-eq-lim sums-def)

lemma sums-iff: f sums x \iff summable f ∧ suminf f = x

by (metis summable-sums sums-summable sums-unique)

lemma summable-sums-iff: summable f \iff f sums suminf f

by (auto simp: sums-iff summable-sums)

lemma sums-unique2: f sums a \implies f sums b \implies a = b

for a b :: 'a

by (simp add: sums-iff)

lemma sums-Uniq: ∃ a. f sums a

for a b :: 'a

by (simp add: sums-unique2 Uniq-def)

lemma suminf-finite:
assumes $N$: finite $N$
  and $f$: $\forall n. n \notin N \implies f n = 0$
shows $\suminf f = (\sum_{n \in N} f n)$
using sums-finite[OF assms, THEN sums-unique] by simp

end

lemma suminf-zero(simp): $\suminf (\lambda n. 0 :: 'a::{t2-space, comm-monoid-add}) = 0$
  by (rule sums-zero [THEN sums-unique, symmetric])

110.3 Infinite summability on ordered, topological monoids

lemma sums-le: $\forall n. f n \leq g n \implies f \sums s \implies g \sums t \implies s \leq t$
  for $f, g :: \text{nat} \Rightarrow 'a::{ordered-comm-monoid-add, linorder-topology}$
  by (rule LIMSEQ-le) (auto intro: sum-mono simp: sums-def)

context
  fixes $f :: \text{nat} \Rightarrow 'a::{ordered-comm-monoid-add, linorder-topology}$
begin

lemma suminf-le: $\forall n. f n \leq g n \implies \sum_{n \in I} f \implies \sum_{n \in I} g \implies \suminf f \leq \suminf g$
  using sums-le by blast

lemma sum-inf-le-suminf:
  shows $\forall f. \text{summable } f \implies \text{finite } I \implies (\forall n. n \notin I \implies 0 \leq f n) \implies \sum f I \leq \suminf f$
  by (rule sums-le[OF - sums-If-finite-set summable-sums]) auto

lemma suminf-nonneg: $\forall f. \text{summable } f \implies (\forall n. 0 \leq f n) \implies 0 \leq \suminf f$
  using sum-le-suminf by force

lemma suminf-le-const: $\forall f. \text{summable } f \implies (\forall n. \sum f \{..< \ n\} \leq x) \implies \suminf f \leq x$
  by (metis LIMSEQ-le-const2 summable-LIMSEQ)

lemma suminf-eq-zero-iff:
  assumes $\forall n. 0 \leq f n$
  shows $\suminf f = 0 \iff (\forall n. f n = 0)$
proof
  assume $L$: $\suminf f = 0$
  then have $f$: $(\forall n. \sum_{i<n} f i) \longrightarrow 0$
    using summable-LIMSEQ[of $f$] assms by simp
  then have $\forall i. (\sum_{n \in \{i\}} f n) \leq 0$
    by (metis $L$ summable $\forall \text{ order-refl pos sum.infinite sum.le-suminf}$)
  with $\text{pos}$ show $\forall n. f n = 0$
    by (simp add: order.antisym)
qed (metis suminf-zero fun-eq-iff)
lemma `suminf-pos-iff`:
summable f \(\Rightarrow (\forall n. 0 \leq f n \Rightarrow 0 < \text{suminf} f \iff (\exists i. 0 < f i))\)
using `sum-le-suminf[of \{\}]` `suminf-eq-zero-iff` by (simp add: less-le)

lemma `suminf-pos2`:
assumes `summable f` \(\forall n. 0 \leq f n \Rightarrow 0 < f i\)
sows `0 < \text{suminf} f`
proof
\(\begin{align*}
\text{have } 0 < (\sum n < Suc. f n) & \quad \text{using `assms` by (intro `sum-pos2[where i=i]`) auto} \\
\text{also have } \ldots < \text{suminf} f & \quad \text{using `assms` by (intro `sum-le-suminf`) auto} \\
\text{finally show } \text{?thesis}.
\end{align*}\)
qed

lemma `suminf-pos`:
summable f \(\Rightarrow (\forall n. 0 < f n \Rightarrow 0 < \text{suminf} f)\)
by (intro `suminf-pos2[where i=0]`) (auto intro: less-imp-le)

end

context`fixes f :: nat \Rightarrow 'a::{ordered-cancel-comm-monoid-add,linorder-topology}
begin

lemma `sum-less-suminf2`:
summable f \(\Rightarrow (\forall m. m \geq n \Rightarrow 0 \leq f m \Rightarrow n < i \Rightarrow 0 < f i \Rightarrow \text{sum} f \{.. < i\} < \text{suminf} f)\)
using `sum-le-suminf[of f \{.. < Suc i\}]`
also have `\ldots < \text{suminf} f` by (auto simp: less-imp-le ac-simps)

lemma `sum-less-suminf`:
summable f \(\Rightarrow (\forall m. m \geq n \Rightarrow 0 < f m \Rightarrow \text{sum} f \{.. < n\} < \text{suminf} f)\)
using `sum-less-suminf2[of n n]` by (simp add: less-imp-le)

end

lemma `summableI-nonneg-bounded`:
fixes f :: nat \Rightarrow 'a::{ordered-comm-monoid-add,linorder-topology,conditionally-complete-linorder}
assumes `pos\[simp]\:: \\forall n. 0 \leq f n`
also have `\\forall i. (\sum i < n. f i) \leq x`
shows summable f
unfolding `summable-def sums-def [abs-def]`
proof (rule exI LIMSEQ-incseq-SUP)+
\(\begin{align*}
\text{show } \text{bdd-above} \quad (\text{range } (\lambda n. \text{sum} f \{.. < n\})) & \quad \text{using `le` by (auto simp: bdd-above-def)} \\
\text{show } \text{incseq} \quad (\lambda n. \text{sum} f \{.. < n\}) & \quad \text{by (auto simp: mono-def intro!: `sum-mono2`)}
\end{align*}\)
lemma summableI[intro, simp]: summable f
  for f :: nat ⇒ 'a::{canonically-ordered-monoid-add,linorder-topology,complete-linorder}
  by (intro summableI-nonneg-bounded[where x=top] zero-le top-greatest)

lemma suminf-eq-SUP-real:
  assumes X: summable X ⋀ i. 0 ≤ X i shows suminf X = (SUP i. ∑ n<i. X i)
  by (intro LIMSEQ-unique[OF summable-LIMSEQ X LIMSEQ-incseq-SUP]
      (auto intro!: bdd-aboveI2[where M=∑ i. X i]
       sum-le-suminf X monoI sum-mono2))

110.4 Infinite summability on topological monoids

begin
  fixes f g :: nat ⇒ 'a::{t2-space,topological-comm-monoid-add}
  begin

  lemma sums-Suc:
    assumes (λn. f (Suc n)) sums l
    shows f sums (l + f 0)
    proof
      have (λn. (∑ i<n. f (Suc i)) + f 0) ----→ l + f 0
        using assms by (auto intro: tendsto-add simp: sums-def)
      moreover have (∑ i<n. f (Suc i)) + f 0 = (∑ i< Suc n. f i) for n
        unfolding lessThan-Suc-eq-insert-0
        by (simp add: ac-simps sum.atLeast1-atMost-eq image-Suc-lessThan)
      ultimately show ?thesis
        by (auto simp: sums-def simp del: sum.lessThan-Suc intro: filterlim-sequentially-Suc[THEN iffD1])
    qed

  lemma sums-add: f sums a ⇒ g sums b ⇒ (λn. f n + g n) sums (a + b)
    unfolding sums-def by (simp add: sum.distrib tendsto-add)

  lemma summable-add: summable f ⇒ summable g ⇒ summable (λn. f n + g n)
    unfolding summable-def by (auto intro: sums-add)

  lemma suminf-add: summable f ⇒ summable g ⇒ suminf f + suminf g = (∑ n. f n + g n)
    by (intro sums-unique sums-add summable-sums)
  end

context
  fixes f :: 'i ⇒ nat ⇒ 'a::{t2-space,topological-comm-monoid-add}
  and I :: 'i set
begin
lemma sums-sum: \((\forall i \in I \Rightarrow (f i) \text{ sums } (x i)) \Rightarrow (\lambda n. \sum_{i \in I} f i \ n) \text{ sums } (\sum_{i \in I} x i)\) 
by (induct I rule: infinite-finite-induct) (auto intro!: sums-add)

lemma suminf-sum: \((\forall i \in I \Rightarrow \text{summable } (f i)) \Rightarrow (\sum n. \sum_{i \in I} f i \ n) = (\sum_{i \in I} \sum n. f i \ n)\) 
using sums-unique[OF sums-sum, OF summable-sums] by simp

lemma summable-sum: \((\forall i \in I \Rightarrow \text{summable } (f i)) \Rightarrow \text{summable } (\lambda n. \sum_{i \in I} f i \ n)\) 
using sums-summable[OF sums-sum, OF summable-sums].

end

lemma sums-If-finite-set':
fixes f g :: nat ⇒ 'a::{t2-space, topological-ab-group-add}
assumes g sums S and finite A and S' = S + (\sum n \in A. f n - g n)
shows \((\lambda n. \text{if } n \in A \text{ then } f n \text{ else } g n) \text{ sums } S'\)
proof\[
\begin{array}{l}
  \text{have } (\lambda n. g n + (\text{if } n \in A \text{ then } f n - g n \text{ else } 0)) \text{ sums } (S + (\sum n \in A. f n - g n)) \\
  \text{by (intro sums-add assms sums-If-finite-set)}
\end{array}
\]
also have \((\lambda n. g n + (\text{if } n \in A \text{ then } f n - g n \text{ else } 0)) = (\lambda n. \text{if } n \in A \text{ then } f n \text{ else } g n)\)
by (simp add: fun-eq-iff)
finally show \(?thesis using assms by simp\)
qed

110.5 Infinite summability on real normed vector spaces

context
fixes f :: nat ⇒ 'a::real-normed-vector
begin

lemma sums-Suc-iff: \((\lambda n. f (\text{Suc } n)) \text{ sums } s \iff f \text{ sums } (s + f 0)\)
proof\[
\begin{array}{l}
  \text{have } f \text{ sums } (s + f 0) \iff (\lambda i. \sum_{j<i} f j) \text{ sums } s + f 0 \\
  \text{by (subst filterlim-sequentially-Suc) (simp add: sums-def)}
\end{array}
\]
also have \(\ldots \iff (\lambda i. (\sum_{j<i} f (\text{Suc } j)) + f 0) \text{ sums } s + f 0\)
by (simp add: ac-simps lessThan-Suc-insert-0 image-Suc-lessThan sum.atLeast1-atMost-eq)
also have \(\ldots \iff (\lambda n. f (\text{Suc } n)) \text{ sums } s\)
proof\[
\begin{array}{l}
  \text{assume } (\lambda i. (\sum_{j<i} f (\text{Suc } j)) + f 0) \text{ sums } s + f 0 \\
  \text{with tendsto-add[OF this tendsto-const, of } - f 0\text{] show } (\lambda i. f (\text{Suc } i)) \text{ sums } s \\
  \text{by (simp add: sums-def)}
\end{array}
\]
qed (auto intro: tendsto-add simp: sums-def)
finally show \(?thesis ..\)
qed
lemma summable-Suc-iff: summable (λn. f (Suc n)) = summable f
proof
  assume summable f
  then have f sums suminf f
    by (rule summable-sums)
  then have (λn. f (Suc n)) sums (suminf f - f 0)
    by (simp add: sums-Suc-iff)
  then show summable (λn. f (Suc n))
    unfolding summable-def by blast
qed (auto simp: sums-Suc-iff summable-def)

lemma sums-Suc-imp: f 0 = 0 ⇒ (λn. f (Suc n)) sums s ⇒ (λn. f n) sums s
  using sums-Suc-iff by simp
end

context
  fixes f :: nat ⇒ 'a::real-normed-vector
begin

lemma sums-diff: f sums a ⇒ g sums b ⇒ (λn. f n - g n) sums (a - b)
  unfolding sums-def by (simp add: sum-subtractf tendsto-diff)

lemma summable-diff: summable f ⇒ summable g ⇒ summable (λn. f n - g n)
  unfolding summable-def by (auto intro: sums-diff)

lemma suminf-diff: summable f ⇒ summable g ⇒ suminf f - suminf g = (∑n. f n - g n)
  by (intro sums-unique sums-diff summable-sums)

lemma sums-minus: f sums a ⇒ (λn. - f n) sums (- a)
  unfolding sums-def by (simp add: sum-negf tendsto-minus)

lemma summable-minus: summable f ⇒ summable (λn. - f n)
  unfolding summable-def by (auto intro: sums-minus)

lemma suminf-minus: summable f ⇒ (∑n. - f n) = - (∑n. f n)
  by (intro sums-unique [symmetric] sums-minus summable-sums)

lemma sums-iff-shift: (λi. f (i + n)) sums s ⇔ f sums (s + (∑i<n. f i))
proof (induct n arbitrary: s)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then have (λi. f (Suc i + n)) sums s ⇔ (λi. f (i + n)) sums (s + f n)
    by (subst sums-Suc-iff) simp
with Suc show \(\text{?case}\)
  by (simp add: ac-simps)
qed

corollary sums-iff-shift': \((\lambda i. f (i + n)) \text{ sums } (s - (\sum_{i<n} f i)) \iff f \text{ sums } s\)
  by (simp add: sums-iff-shift)

lemma sums-zero-iff-shift:
  assumes \(\forall i. i < n \implies f i = 0\)
  shows \((\lambda i. f (i+n)) \text{ sums } s \iff (\lambda i. f i) \text{ sums } s\)
  by (simp add: assms sums-iff-shift)

lemma summable-iff-shift [simp]: \(\text{summable } (\lambda n. f (n + k)) \iff \text{summable } f\)
  by (metis diff-add-cancel summable-def sums-iff-shift [abs-def])

lemma sums-split-initial-segment: \(f \text{ sums } s \implies (\lambda i. f (i + n)) \text{ sums } (s - (\sum_{i<n} f i))\)
  by (simp add: sums-iff-shift)

lemma summable-ignore-initial-segment: \(\text{summable } f \implies \text{summable } (\lambda n. f (n + k))\)
  by (simp add: summable-iff-shift)

lemma suminf-minus-initial-segment: \(\text{summable } f \implies \left(\sum_{n} f(n + k)\right) = (\sum_{n} f(n + k)) - (\sum_{i<k} f i)\)
  by (rule sums-unique[symmetric]) (auto simp: sums-iff-shift)

lemma suminf-split-initial-segment: \(\text{summable } f \implies \text{suminf } f = (\sum_{n} f(n + k)) + (\sum_{i<k} f i)\)
  by (auto simp add: suminf-minus-initial-segment)

lemma suminf-split-head: \(\text{summable } f \implies (\sum_{n} f (Suc n)) = \text{suminf } f - f 0\)
  using suminf-split-initial-segment[of 1] by simp

lemma suminf-exist-split:
  fixes \(r :: \text{real}\)
  assumes \(0 < r \land \text{summable } f\)
  shows \(\exists N. \forall n \geq N. \text{norm} (\sum_{i} f (i + n)) < r\)
  proof
    -- from LIMSEQ-D[OF \text{summable-LIMSEQ}[OF \text{summable } f]; \(0 < r\)]
    obtain N :: nat where \(\forall n \geq N. \text{norm} (\text{sum } f \{..<n\} - \text{suminf } f) < r\)
      by auto
    then show \(\text{thesis}\)
      by (auto simp: norm-minus-commute suminf-minus-initial-segment[OF \text{summable } f])
    qed

lemma summable-LIMSEQ-zero:
  assumes \(\text{summable } f\)
  shows \(f \longrightarrow 0\)
proof
  have Cauchy (λn. sum f {..<n})
    using LIMSEQ-imp-Cauchy assms summable-LIMSEQ by blast
  then show ?thesis
    unfolding Cauchy-iff LIMSEQ-iff
    by (metis add.commute add-diff-cancel-right' diff-zero le-SucI sum.lessThan-Suc)
qed

lemma summable-imp-convergent: summable f ⇒ convergent f
  by (force dest: summable-LIMSEQ-zero simp: convergent-def)

lemma summable-imp-Bseq: summable f ⇒ Bseq f
  by (simp add: convergent-imp-Bseq summable-imp-convergent)

end

lemma (in bounded-linear) sums: (λn. X n) sums a ⇒ (λn. f (X n)) sums (f a)
  unfolding sums-def by (drule tendsto) (simp only: sum)

lemma (in bounded-linear) summable: summable (λn. X n) ⇒ summable (λn. f (X n))
  unfolding summable-def by (auto intro: sums)

lemma (in bounded-linear) suminf: summable (λn. X n) ⇒ f (∑n. X n) = (∑ n. f (X n))
  by (intro sums-unique sums summable-sums)

lemmas sums-of-real = bounded-linear.sums [OF bounded-linear-of-real]
lemmas summable-of-real = bounded-linear.summable [OF bounded-linear-of-real]
lemmas suminf-of-real = bounded-linear.suminf [OF bounded-linear-of-real]

lemmas sums-scaleR-left = bounded-linear.sums[OF bounded-linear-scaleR-left]
lemmas summable-scaleR-left = bounded-linear.summable[OF bounded-linear-scaleR-left]
lemmas suminf-scaleR-left = bounded-linear.suminf[OF bounded-linear-scaleR-left]

lemmas sums-scaleR-right = bounded-linear.sums[OF bounded-linear-scaleR-right]
lemmas summable-scaleR-right = bounded-linear.summable[OF bounded-linear-scaleR-right]
lemmas suminf-scaleR-right = bounded-linear.suminf[OF bounded-linear-scaleR-right]

lemma summable-const-iff: summable (λ-. c) ≡ c = 0
  for c :: 'a::real-normed-vector
proof
  have ¬ summable (λ-. c) if c ≠ 0
  proof
    from that have filterlim (λn. of-nat n * norm c) at-top sequentially
by (subst mult.commute)
  (auto intro: filterlim-tendsto-pos-mult-at-top filterlim-real-sequentially)
then have ¬ convergent (λn. norm (∑ k<n. c))
  by (intro filterlim-at-infinity-imp-not-convergent filterlim-at-top-imp-at-infinity)
  (simp all add: sum-constant-scaleR)
then show ?thesis
  unfolding summable-iff-convergent using convergent-norm by blast
qed
then show ?thesis by auto
qed

110.6 Infinite summability on real normed algebras

context
  fixes f :: nat ⇒ ′a::real_normed_algebra
begin

lemma sums-mult: f sums a ⇒ (λn. c * f n) sums (c * a)
  by (rule bounded-linear.sums [OF bounded-linear-mult-right])

lemma summable-mult: summable f ⇒ summable (λn. c * f n)
  by (rule bounded-linear.summable [OF bounded-linear-mult-right])

lemma suminf-mult: summable f ⇒ suminf (λn. c * f n) = c * suminf f
  by (rule bounded-linear.suminf [OF bounded-linear-mult-right, symmetric])

lemma sums-mult2: f sums a ⇒ (λn. f n * c) sums (a * c)
  by (rule bounded-linear.sums [OF bounded-linear-mult-left])

lemma summable-mult2: summable f ⇒ summable (λn. f n * c)
  by (rule bounded-linear.summable [OF bounded-linear-mult-left])

lemma suminf-mult2: summable f ⇒ suminf f * c = (∑ n. f n * c)
  by (rule bounded-linear.suminf [OF bounded-linear-mult-left])

end

lemma sums-mult-iff:
  fixes f :: nat ⇒ ′a::real_normed_algebra
  assumes c ≠ 0
  shows (λn. c * f n) sums (c * d) ←→ f sums d
  using sums-mult[of f d c] sums-mult[of λn. c * f n c * d inverse c]
  by (force simp: field_simps assms)

lemma sums-mult2-iff:
  fixes f :: nat ⇒ ′a::real_normed_algebra
  assumes c ≠ 0
  shows (λn. f n * c) sums (d * c) ←→ f sums d
  using sums-mult-iff[of f d c] by (simp add: mult.commute)
THEORY “Series” 2250

lemma sums-of-real-iff:
  \((\lambda n. \text{of-real } (f n)) :: 'a::real-normed-div-algebra) \text{ sums of-real } c \iff f \text{ sums } c\)

110.7 Infinite summability on real normed fields

context fixes c :: 'a::real-normed-field
begin

lemma sums-divide: \(f \text{ sums } a \iff (\lambda n. f n / c) \text{ sums } (a / c)\)
by (rule bounded-linear.sums [OF bounded-linear-divide])

lemma summable-divide: \(\text{summable } f \iff \text{summable } (\lambda n. f n / c)\)
by (rule bounded-linearsummable [OF bounded-linear-divide])

lemma suminf-divide: \(\text{summable } f \iff \text{suminf } (\lambda n. f n / c) = \text{suminf } f / c\)
by (rule bounded-linear.suminf [OF bounded-linear-divide, symmetric])

lemma summable-inverse-divide: \(\text{summable } (f \circ \text{inverse}) \iff \text{summable } (\lambda n. c / f n)\)
by (auto dest: summable-mult [of - c] simp: field-simps)

lemma sums-mult-D: \(\text{ sums } (c \ast f n) \iff c \neq 0 \iff f \text{ sums } (a/c)\)
using sums-mult-iff by fastforce

lemma summable-mult-D: \(\text{summable } (\lambda n. c \ast f n) \iff c \neq 0 \iff \text{summable } f\)
by (auto dest: summable-divide)

Sum of a geometric progression.

lemma geometric-sums:
  assumes norm c < 1
  shows \((\lambda n. c \cdot n) \text{ sums } (1 / (1 - c))\)
proof
  have neq-0: \(c - 1 \neq 0\)
    using assms by auto
  then have \((\lambda n. c \cdot n / (c - 1) - 1 / (c - 1)) \longrightarrow 0 / (c - 1) - 1 / (c - 1)\)
    by (intro tendsto-intros assms)
  then have \((\lambda n. (c \cdot n - 1) / (c - 1)) \longrightarrow 1 / (1 - c)\)
    by (simp add: nonzero-minus-divide-right [OF neq-0] diff-divide-distrib)
  with neq-0 show \((\lambda n. c \cdot n) \text{ sums } (1 / (1 - c))\)
    by (simp add: sums-def geometric-sum)
qed

lemma summable-geometric: \(\text{norm } c < 1 \iff \text{summable } (\lambda n. c \cdot n)\)
by (rule geometric-sums [THEN sums-summable]);
lemma suminf-geometric: norm c < 1 \implies \suminf (\lambda n. c^n) = 1 / (1 - c)
by (rule sums-unique[ symmetric]) (rule geometric-sums)

lemma summable-geometric-iff [simp]: summable (\lambda n. c^n) \iff norm c < 1
proof
assume summable (\lambda n. c^n :: 'a :: real_normed_field)
then have (\lambda n. norm (c^n)) ----> 0
by (simp add: norm-power [symmetric] tendsto-norm-zero-iff summable-LIMSEQ-zero)
from order-tendstoD [OF this zero-less-one] obtain n where norm (c^n) < 1
by (auto simp: eventually_at_top_linorder)
then show norm c < 1 using one_le_power [of norm c n]
by (cases norm c \geq 1) (linarith, simp)
qed (rule summable-geometric)

end

Biconditional versions for constant factors

context
fixes c :: 'a::real_normed_field
begin

lemma summable-cmult-iff [simp]: summable (\lambda n. c * f n) \iff c = 0 \lor summable f
proof
have [summable (\lambda n. c * f n); c \neq 0] \implies summable f
using summable-mult-D by blast
then show \thesis
by (auto simp: summable-mult)
qed

lemma summable-divide-iff [simp]: summable (\lambda n. f n / c) \iff c = 0 \lor summable f
proof
have [summable (\lambda n. f n / c); c \neq 0] \implies summable f
by (auto dest: summable-divide [where c = 1/c])
then show \thesis
by (auto simp: summable-divide)
qed

end

lemma power-half-series: (\lambda n. (1/2::real)^Suc n) sums 1
proof
have 2: (\lambda n. (1/2::real)^n) sums 2
using geometric-sums [of 1/2::real] by auto
have (\lambda n. (1/2::real)^Suc n) = (\lambda n. (1/2)^n / 2)
by (simp add: mult.commute)
then show \thesis
using sums-divide [OF 2, of 2] by simp
110.8 Telescoping

**Lemma telescope-sums**:  
fixes \( c :: 'a::real-normed-vector \)  
assumes \( f \longrightarrow c \)  
shows \((\lambda n. f (Suc n) - f n) \) sums \( (c - f 0) \)  
unfolding sums-def  
**Proof**  
 subst filterlim-sequentially-Suc \([\text{symmetric}]\)  
have \((\lambda n. \sum_{k< Suc n} f (Suc k) - f k) = (\lambda n. f (Suc n) - f 0) \)  
by \((\text{simp add: lessThan-Suc-atMost atLeast0AtMost [symmetric] sum-Suc-diff})\)  
also have \( \ldots \longrightarrow c - f 0 \)  
by \((\text{intro tendsto-diff LIMSEQ-Suc[OF assms]} \) tendsto-const\)  
finally show \((\lambda n. \sum_{n<Suc n} f (Suc n) - f n) \longrightarrow c - f 0 \) .  
qed

**Lemma telescope-sums'**:  
fixes \( c :: 'a::real-normed-vector \)  
assumes \( f \longrightarrow c \)  
shows \((\lambda n. f n - f (Suc n)) \) sums \( (f 0 - c) \)  
using sums-minus\([\text{OF telescope-sums[OF assms]}\) by \((\text{simp add: algebra-simps})\)

**Lemma telescope-summable**:  
fixes \( c :: 'a::real-normed-vector \)  
assumes \( f \longrightarrow c \)  
shows summable \((\lambda n. f (Suc n) - f n) \)  
using telescope-sums\([\text{OF assms]}\) by \((\text{simp add: sums-iff})\)

**Lemma telescope-summable'**:  
fixes \( c :: 'a::real-normed-vector \)  
assumes \( f \longrightarrow c \)  
shows summable \((\lambda n. f n - f (Suc n)) \)  
using summable-minus\([\text{OF telescope-summable[OF assms]}\) by \((\text{simp add: algebra-simps})\)

110.9 Infinite summability on Banach spaces

Cauchy-type criterion for convergence of series (c.f. Harrison).

**Lemma summable-Cauchy**:  
summable \( f \iff (\forall e>0. \exists N. \forall m\geq N. \forall n. \text{norm} \ (\sum f \{m..<n\}) < e) \) (is - = ?rhs)  
for \( f :: \text{nat} \Rightarrow 'a::banach \)  
**Proof**  
assume \( f \) : summable \( f \)  
show ?rhs  
**Proof** clarify  
fix \( e :: real \)  
assume \( 0 < e \)
then obtain $M$ where $M: \forall m \ n. [m \geq M; n \geq M] \implies \text{norm} (\sum f \{m..<n\}) < e$

- sum $f \{m..<n\} < e$

using $f$ by (force simp add: summable-iff-convergent Cauchy-convergent-iff

[symmetric] Cauchy-iff)

have $\text{norm} (\sum f \{m..<n\}) < e$ if $m \geq M$ for $m \ n$

proof (cases $m \ n$ rule: linorder-class.le-cases)

assume $m \leq n$

then show $\exists M. \forall m \geq M. \forall n. \text{norm} (\sum f \{m..<n\}) < e$

by (metis (mono-tags, opaque-lifting) $M$ atLeast0LessThan order-trans sum-diff-nat-ivl that zero-le)

next

assume $n \leq m$

then show $\exists M. \forall m \geq M. \forall n. \text{norm} (\sum f \{m..<n\}) < e$

by blast

qed

next

assume $r: ?\text{rhs}$

then show summable $f$

unfolding summable-iff-convergent Cauchy-convergent-iff [symmetric] Cauchy-iff

proof clarify

fix $e :: \text{real}$

assume $\theta < e$

with $r$ obtain $N$ where $N: \forall m \ n. m \geq N \implies \text{norm} (\sum f \{m..<n\}) < e$

by blast

have $\text{norm} (\sum f \{m..<n\} - \sum f \{..<n\}) < e$ if $m \geq N \ n \geq N$ for $m \ n$

proof (cases $m \ n$ rule: linorder-class.le-cases)

assume $m \leq n$

then show $\exists M. \forall m \geq M. \forall n. \text{norm} (\sum f \{m..<n\} - \sum f \{..<n\}) < e$

by (metis $N$ finite-lessThan lessThan-minus-lessThan lessThan-subset-iff norm-minus-commute sum-diff $\{m\geq N\}$)

next

assume $n \leq m$

then show $\exists M. \forall m \geq M. \forall n. \text{norm} (\sum f \{m..<n\} - \sum f \{..<n\}) < e$

by blast

qed

qed

lemma summable-Cauchy':

fixes $f :: \text{nat \Rightarrow \text{a :: banach}}$

assumes ev: eventually $(\lambda m. \forall n \geq m. \text{norm} (\sum f \{m..<n\}) \leq g \ m)$ sequentially

assumes $g0: g \longrightarrow 0$

shows summable $f$
proof (subst summable-Cauchy, intro allI impI, goal-cases)
case (1 e)
then have \( \forall x \in \text{sequentially}. \ g x < e \)
  using \( \text{g0 order-tendstoD(2) by blast} \)
with \( \text{ev} \) have \( \text{eventually (\lambda m. \forall n. norm (sum f \{ m..<n \}) < e)} \) at-top
proof eventually-elim
  case (elim m)
  show ?case
  proof
    fix n
    from elim show \( \text{norm (sum f \{ m..<n \}) < e} \)
    by (cases \( n \geq m \)) auto
  qed
  qed
thus ?case by (auto simp: eventually-at-top-linorder)
qed

context fixes \( f :: \text{nat} \Rightarrow \text{'}a::banach \)
begin

Absolute convergence implies normal convergence.

lemma summable-norm-cancel: \( \text{summable (\lambda n. \text{norm (f n)})} \Rightarrow \text{summable f} \)
unfolding summable-Cauchy
apply (erule all-forward imp-forward ex-forward | assumption)+
apply (fastforce simp add: order-le-less-trans [OF norm-sum] order-le-less-trans [OF abs-ge-self])
done

lemma summable-norm: \( \text{summable (\lambda n. \text{norm (f n)})} \Rightarrow \text{norm (suminf f) \leq (\sum n. norm (f n))} \)
by (auto intro: LIMSEQ-le tendsto-norm summable-norm-cancel summable-LIMSEQ norm-sum)

Comparison tests.

lemma summable-comparison-test:
  assumes \( \exists N. \forall n\geq N. \text{norm (f n)} \leq g n \) and \( g : \text{summable g} \)
shows \( \text{summable f} \)
proof
  obtain \( N \) where \( N : \forall n\geq N \Rightarrow \text{norm (f n)} \leq g n \)
  using assms by blast
  show ?thesis
  proof (clarsimp simp add: summable-Cauchy)
    fix \( e :: \text{real} \)
    assume \( 0 < e \)
    then obtain \( N g \) where \( N g : \forall m n \geq N g \Rightarrow \text{norm (sum g \{ m..<n \}) < e} \)
    using \( g \) by (fastforce simp: summable-Cauchy)
    with \( N \) have \( \text{norm (sum f \{ m..<n \}) < e} \) if \( m \geq \text{max N} N g \) for \( m n \)
    proof
have norm (sum f {m..<n}) ≤ sum g {m..<n}
    using N that by (force intro: sum-norm-le)
also have ... ≤ norm (sum g {m..<n})
    by simp
also have ... < e
    using Ng that by auto
finally show thesis.
  qed
then show ∃N. ∀m≥N. ∀n. norm (sum f {m..<n}) < e
by blast
qed
qed

lemma summable-comparison-test-ev:
  eventually (λn. norm (f n) ≤ g n) sequentially ⇒ summable g ⇒ summable f
by (rule summable-comparison-test) (auto simp: eventually-at-top-linorder)

A better argument order.

lemma summable-comparison-test': summable g ⇒ (∀n. n ≥ N ⇒ norm (f n) ≤ g n) ⇒ summable f
by (rule summable-comparison-test) auto

110.10 The Ratio Test

lemma summable-ratio-test:
  assumes c < 1 ∧ n ≥ N ⇒ norm (f (Suc n)) ≤ c * norm (f n)
  shows summable f
proof (cases 0 < c)
  case True
  show summable f
proof (rule summable-comparison-test)
    show ∃N’. ∀n≥N’. norm (f n) ≤ (norm (f N) / (c ^ N)) * c ^ n
proof (intro exI allI impI)
      fix n
      assume N ≤ n
      then show norm (f n) ≤ (norm (f N) / (c ^ n)) * c ^ n
proof (induct rule: inc-induct)
      case base
      with True show ?case by simp
next
case (step m)
    have norm (f (Suc m)) / c ^ Suc m * c ^ n ≤ norm (f m) / c ^ m * c ^ n
      using ⟨0 < c, c < 1⟩ assms(2)[OF ⟨N ≤ m⟩] by (simp add: field-simps)
    with step show ?case by simp
  qed
  qed
  show summable (λn. norm (f N) / c ^ N * c ^ n)
    using ⟨0 < c, c < 1⟩ by (intro summable-mult summable-geometric) simp
q}d
next
case False
have f (Suc n) = 0 if n ≥ N for n
proof -
from that have norm (f (Suc n)) ≤ c * norm (f n)
by (rule assms(2))
also have ... ≤ 0
using False by (simp add: not-less mult-nonpos-nonneg)
finally show ?thesis
by auto
qed
then show summable f
by (intro sums-summable[OF sums-finite, of {.. Suc N}]) (auto simp: not-le Suc-less-eq2)
qed

end

Application to convergence of the log function

lemma norm-summable-ln-series:
fixes z :: 'a :: {real_normed_field, banach}
assumes norm z < 1
shows summable (λn. norm (z ^ n / of_nat n))
proof (rule summable-comparison-test)
show summable (λn. norm (z ^ n))
using assms unfolding norm-power by (intro summable-geometric) auto
have norm z ^ n / real n ≤ norm z ^ n for n
proof (cases n = 0)
case False
hence norm z ^ n * 1 ≤ norm z ^ n * real n
by (intro mult-left-mono) auto
thus ?thesis
using False by (simp add: field-simps)
qed auto
thus ∃N. ∀n≥N. norm (norm (z ^ n / of-nat n)) ≤ norm (z ^ n)
by (intro exI[of _ 0]) (auto simp: norm-power norm-divide)
qed

Relations among convergence and absolute convergence for power series.

lemma Abel-lemma:
fixes a :: nat ⇒ 'a::real_normed_vector
assumes r: 0 ≤ r
and r0: r < r0
and M: ∀n. norm (a n) * r0^−n ≤ M
shows summable (λn. norm (a n) * r^−n)
proof (rule summable-comparison-test')
show summable (λn. M * (r / r0) ^ −n)
using assms by (auto simp add: summable-mult summable-geometric)
show norm (norm (a n) * r ^ −n) ≤ M * (r / r0) ^ −n for n
THEORY "Series" 2257

using r r0 M [of n] dual-order.order-iff-strict
by (fastforce simp add: abs-mult field-simps)
qed

Summability of geometric series for real algebras.

lemma complete-algebra-summable-geometric:
  fixes x :: 'a::{real-normed-algebra-1,banach}
  assumes norm x < 1
  shows summable (λn. x ^ n)
proof (rule summable-comparison-test)
  show ∃ N. ∀ n≥N. norm (x ^ n) ≤ norm x ^ n
    by (simp add: norm-power-ineq)
  from assms show summable (λn. norm x ^ n)
    by (simp add: summable-geometric)
qed

110.11 Cauchy Product Formula

Proof based on Analysis WebNotes: Chapter 07, Class 41 http://www.math.unl.edu/~webnotes/classes/class41/prp77.htm

lemma Cauchy-product-sums:
  fixes a b :: nat ⇒ 'a::{real-normed-algebra,banach}
  assumes a: summable (λk. norm (a k))
    and b: summable (λk. norm (b k))
  shows (λk. ∑ i≤k. a i * b (k - i)) sums ((∑ k. a k) * (∑ k. b k))
proof –
  let ?S1 = λn::nat. {..<n} × {..<n}
  let ?S2 = λn::nat. {(i,j). i + j < n}
  have S1 mono: ∀ m n. m ≤ n ⇒ ?S1 m ⊆ ?S1 n by auto
  have S2 le S1: ∀ n. ?S2 n ⊆ ?S1 n by auto
  have S1 le S2: ∀ n. ?S1 (n div 2) ⊆ ?S2 n by auto
  have finite S1: ∀ n. finite (?S1 n) by simp
  with S2 le S1 have finite S2: ∀ n. finite (?S2 n) by (rule finite-subset)

  let ?g = λ(i,j). a i * b j
  let ?f = λ(i,j). norm (a i) * norm (b j)
  have f nonneg: ∀ x. 0 ≤ ?f x by auto
  then have norm sum f: ∀ A. norm (sum ?f A) = sum ?f A
    unfolding real norm def
      by (simp only: abs of nonneg sum nonneg [rule format])

  have (λn. (∑ k<n. a k) * (∑ k<n. b k)) ----> (∑ k<n. a k) * (∑ k<n. b k)
    by (intro tendsto_mult summable LIMSEQ summable norm cancel [OF a] summable norm cancel
        [OF b])
  then have 1: (λn. sum ?g (?S1 n)) ----> (∑ k<n. a k) * (∑ k<n. b k)
    by (simp only: sum product sum Sigma [rule format] finite lessThan)

  have (λn. (∑ k<n. norm (a k)) * (∑ k<n. norm (b k))) ----> (∑ k<n. norm (a k)) * (∑ k<n. norm (b k))
using a b by (intro tendsto-mult summable-LIMSEQ)
then have (λ n. sum ?f (?S1 n)) ----> (∑ k. norm (a k)) * (∑ k. norm (b k))
  by (simp only: sum-product sum.Sigma [rule-format] finite-lessThan)
then have convergent (λ n. sum ?f (?S1 n))
  by (rule convergentI)
then have Cauchy: Cauchy (λ n. sum ?f (?S1 n))
  by (rule convergent-Cauchy)
have Zfun (λ n. sum ?f (?S1 n - ?S2 n)) sequentially
proof (rule ZfunI, simp only: eventually-sequentially norm-sum-f)
  fix r :: real
  assume r: 0 < r
from CauchyD [OF Cauchy r] obtain N
  where ∀ m≥N. ∀ n≥N. norm (sum ?f (?S1 m) - sum ?f (?S1 n)) < r ..
then have ∃ m n. N ≤ n ⇒ n ≤ m ⇒ norm (sum ?f (?S1 m - ?S1 n)) < r
  by (simp only: sum-diff finite-S1 S1-mono)
then have N: ∀ m n. N ≤ n ⇒ n ≤ m ⇒ sum ?f (?S1 m - ?S1 n) < r
  by (simp only: norm-sum-f)
proof (intro cexI allI impI)
  fix n
  assume 2 * N ≤ n
then have n: N ≤ n div 2 by simp
have sum ?f (?S1 n - ?S2 n) ≤ sum ?f (?S1 n - ?S1 (n div 2))
  by (intro sum-mono2 finite-Diff finite-S1 f-nonneg Diff-mono subset-refl)
also have .. . < r
  using n div-le-dividend by (rule N)
finally show sum ?f (?S1 n - ?S2 n) < r .
qed
qed
then have Zfun (λ n. sum ?g (?S1 n - ?S2 n)) sequentially
apply (rule Zfun-le [rule-format])
apply (simp only: norm-sum-f)
apply (rule order-trans [OF norm-sum sum-mono])
apply (auto simp add: norm-mult-ineq)
done
then have 2: (λ n. sum ?g (?S1 n) - sum ?g (?S2 n)) ----> 0
  unfolding tendsto-Zfun-iff diff-0-right
by (simp only: sum-diff finite-S1 S2-le-S1)
with 1 have (λ n. sum ?g (?S2 n)) ----> (∑ k. a k) * (∑ k. b k)
  by (rule Lim-transform2)
then show ?thesis
  by (simp only: sums-def sum.triangle-reindex)
qed

lemma Cauchy-product:
  fixes a b :: nat ⇒ 'a::{real-normed-algebra,banach}
  assumes summable (λk. norm (a k))
THEORY “Series”  

and summable (∑ k. a k) \* (∑ k. b k) = (∑ k. ∑ i≤k. a i \* b (k - i))

using assms by (rule Cauchy-product-sums [THEN sums-unique])

lemma summable-Cauchy-product:
fixes a b :: nat ⇒ 'a::{real-normed-algebra,banach}
 assumes summable (∑ k. norm (a k))
and summable (∑ k. norm (b k))
shows summable (∑ i≤k. a i \* b (k - i))

110.12 Series on reals

lemma summable-norm-comparison-test:
\∃ N. \forall n≥N. norm (f n) ≤ g n ⇒ summable g ⇒ summable (λn. norm (f n))
by (rule summable-comparison-test) auto

lemma summable-rabs-comparison-test: \∃ N. \forall n≥N. |f n| ≤ g n ⇒ summable g
⇒ summable (λn. |f n|)
for f :: nat ⇒ real
by (rule summable-comparison-test) auto

lemma summable-rabs-cancel: summable (λn. |f n|) ⇒ summable f
for f :: nat ⇒ real
by (rule summable-norm-cancel) simp

lemma norm-suminf-le:
assumes (∑ n. norm (f n :: 'a :: banach)) ≤ g n summable g
shows norm (suminf f) ≤ suminf g
proof
  have *: summable (λn. norm (f n))
  using assms summable-norm-comparison-test by blast
  hence norm (suminf f) ≤ (∑ n. norm (f n)) by (intro summable-norm) auto
  also have ... ≤ suminf g by (intro suminf-le * assms allI)
  finally show ?thesis .

qed

lemma summable-zero-power [simp]: summable (λn. 0 ^ n :: 'a::{comm-ring-1,topological-space})
proof
  have (λn. 0 ^ n :: 'a) = (λn. if n = 0 then 0 else 0)
  by (intro ext) (simp add: zero-power)
  moreover have summable ... by simp
  ultimately show ?thesis by simp

qed
lemma summable-zero-power [simp]: summable (λ n. f n * 0 ^ n :: 'a::{ring-1, topological-space})
proof
  have (λ n. f n * 0 ^ n :: 'a) = (λ n. if n = 0 then f 0 * 0 ^ 0 else 0)
    by (intro ext) (simp add: zero-power)
moreover have summable ... by simp
ultimately show thesis by simp
qed

lemma summable-power-series:
  fixes z :: real
  assumes le-1: (∀ i. f i ≤ 1
and nonneg: (∀ i. 0 ≤ f i
and z: 0 ≤ z < 1
shows summable (λ i. f i * z ^ i)
proof (rule summable-comparison-test [OF - summable-geometric])
  show norm z < 1
    using z by (auto simp: less_imp_le
  show (∀ n. ∃ N. ∀ m ≥ N. norm (f m * z ^ m) ≤ z ^ m)
    using z
      by (auto intro: ext [of - 0] mult-left-le-one-le simp: abs_mult nonneg power-abs
less_imp_le le-1)
qed

lemma summable-0-powser: summable (λ n. f n * 0 ^ n :: 'a::real-normed-div-algebra)
  by simp

lemma summable-powser-split-head:
  summable (λ n. f (Suc n) * z ^ n :: 'a::real-normed-div-algebra) = summable (λ n. f n * z ^ n)
proof
  have summable (λ n. f (Suc n) * z ^ n) ⇔ summable (λ n. f (Suc n) * z ^ Suc n)
    (is ?lhs ⇔ ?rhs)
proof
  show ?rhs if ?lhs
    using summable-mult2 [OF that, of z]
    by (simp add: power-commutes algebra-simps)
  show ?lhs if ?rhs
    using summable-mult2 [OF that, of inverse z]
    by (cases z ≠ 0, subst (asm) power-Suc2) (simp-all add: algebra-simps)
qed
also have ... ⇔ summable (λ n. f n * z ^ n) by (rule summable-Suc-iff)
finally show thesis .
qed

lemma summable-powser-ignore-initial-segment:
  fixes f :: nat ⇒ 'a :: real-normed-div-algebra
  shows summable (λ n. f (n + m) * z ^ n) ⇔ summable (λ n. f n * z ^ n)
proof (induction m)
case (Suc m)
  have summable (λn. f (n + Suc m) * z ^ n) = summable (λn. f (Suc n + m) * z ^ n)
    by simp
  also have ... = summable (λn. f (n + m) * z ^ n)
    by (rule summable-powser-split-head)
  also have ... = summable (λn. f n * z ^ n)
    by (rule Suc.IH)
  finally show ?case.
qed simp-all

lemma powser-split-head:
  fixes f :: nat ⇒ 'a::{real-normed-div-algebra,banach}
  assumes summable (λn. f n * z ^ n)
  shows suminf (λn. f n * z ^ n) = f 0 + suminf (λn. f (Suc n) * z ^ n) * z
    and suminf (λn. f (Suc n) * z ^ n) * z = suminf (λn. f n * z ^ n) - f 0
    and summable (λn. f (Suc n) * z ^ n)
proof –
  from assms show summable (λn. f (Suc n) * z ^ n)
    by (subt summable-powser-split-head)
  from suminf-mult2[OF this, of z]
  have (∑n. f (Suc n) * z ^ n) * z = (∑n. f (Suc n) * z ^ Suc n)
    by (simp add: power-commutes algebra-simps)
  also from assms have ... = suminf (λn. f n * z ^ n) - f 0
    by (subt suminf-split-head) simp-all
  finally show suminf (λn. f n * z ^ n) = f 0 + suminf (λn. f (Suc n) * z ^ n) * z
    by simp
  then show suminf (λn. f (Suc n) * z ^ n) * z = suminf (λn. f n * z ^ n) - f 0
    by simp
qed

lemma summable-partial-sum-bound:
  fixes f :: nat ⇒ 'a :: banach
  and e :: real
  assumes summable: summable f
  and e: e > 0
  obtains N where \( m \in \mathbb{N} \land m \geq N \implies \text{norm} (\sum_{k=m..n} f k) < e \)
proof –
  from summable have Cauchy (λn. ∑k<n. f k)
    by (simp add: Cauchy-convergent-iff summable-iff-convergent)
  from CauchyD [OF this e] obtain N
    where N: \( m \in \mathbb{N} \land m \geq N \implies n \geq N \implies \text{norm} ((\sum_{k<m} f k) - (\sum_{k<n} f k)) < e \)
      by blast
  have norm (∑k=m..n. f k) < e if m: m ≥ N for m n
proof (cases n ≥ m)
  case True
    with m have norm ((∑k<Suc n. f k) - (∑k<m. f k)) < e
by (intro N) simp-all
also from True have \((\sum k<Suc \ n. f \ k) - (\sum k<m. f \ k) = (\sum k=m..n. f \ k)\)
by (subst sum-diff [symmetric]) (simp-all add: sum.last-plus)
finally show \(?thesis\).
next
case False
with e show \(?thesis\) by simp-all
qed
then show \(?thesis\) by (rule that)
qed

lemma \(\text{powser-sums-if}\):
\((\lambda n. (if n = m then \(1 :: 'a::\{\text{ring-1},\text{topological-space}\}\) else 0) \ast z^n)\) sums \(z^m\)
proof
have \((\lambda n. (if n = m then 1 else 0) \ast z^n) = (\lambda n. if n = m then z^n else 0)\)
by (intro ext) auto
then show \(?thesis\)
by (simp add: sums-single)
qed

lemma \(\text{fixes f :: nat} \Rightarrow \text{real}\)
assumes summable f
and inj g
and pos: \(\\forall x. 0 \leq f x\)
shows summable-reindex: summable \((f \circ g)\)
and suminf-reindex-mono: suminf \((f \circ g) \leq \text{suminf } f\)
and suminf-reindex: \(\text{suminf } (\\lambda x. x \notin \text{range } g \Rightarrow f x = 0) = \text{suminf } (f \circ g) = \text{suminf } f\)
proof
from \(\text{inj g}\) have \([\text{simp}]: \\Lambda A. \text{inj-on } g A\)
by (rule subset-inj-on) simp
have smaller: \(\\forall n. (\sum i<n. (f \circ g) i) \leq \text{suminf } f\)
proof
fix n
have \(\forall n' \in (g' \{..<n\}). n' < \text{Suc } (\text{Max } (g' \{..<n\}))\)
by (metis Max-ge finite-imageI finite-lessThan not-le not-less-eq)
then obtain m where \(n: \text{\forall } n'. n' < n \Rightarrow g n' < m\)
by blast
have \((\sum i<n. f (g \ i)) = \text{sum } f (g' \{..<n\})\)
by (simp add: sum.reindex)
also have \(\ldots \leq (\sum i<m. f \ i)\)
by (rule sum-mono2) (auto simp add: pos n[rule-format])
also have \(\ldots \leq \text{suminf } f\)
using \(\text{summable } f\)
by (rule sum-le-suminf) (simp-all add: pos)
finally show \((\sum i<n. (f \circ g) \ i) \leq \text{suminf } f\)
THEORY "Series"

by simp
qed

have incseq (λn. ∑ i<n. (f o g) i)
  by (rule incseq-SucI) (auto simp add: pos)
then obtain L where L: (λn. ∑ i<n. (f o g) i) → L
  using smaller by (rule incseq-convergent)
then have (f o g) sums L
  by (simp add: sums-def)
then show summable (f o g)
  by (auto simp add: sums-iff)
then have (λn. ∑ i<n. (f o g) i) → suminf (f o g)
  by (rule summable-LIMSEQ)
then show le: suminf (f o g) ≤ suminf f
  by (rule LIMSEQ-le-const2) (blast intro: smaller [rule-format])

assume f: ∀x. x /∈ range g ⇒ f x = 0
from ‹summable f› have suminf f ≤ suminf (f o g)
proof (rule suminf-le-const)
  fix n
  have ∃ n' ∈ (g - ‹..<n›). n' < Suc (Max (g - ‹..<n›))
    by (auto intro: Max-ge simp add: finite-vimageI less-Suc-eq-le)
  then obtain m where n: ∀n'. g n' < n ⇒ n' < m
    by blast
  have (∑ i<n. f i) = (∑ i∈{..<n} ∩ range g. f i)
    using f by (auto intro: sum.mono-neutral-cong-right)
  also have ... = (∑ i∈g - ‹..<n›. (f o g) i)
    by (rule sum.reindex-cong [where l = g])(auto)
  also have ... ≤ (∑ i<m. (f o g) i)
    by (rule sum.mono2)(auto simp add: pos n)
  also have ... ≤ suminf (f o g)
    using (summable (f o g)) by (rule sum-le-suminf) (simp-all add: pos)
  finally show sum f {..<n} ≤ suminf (f o g).
qed
with le show suminf (f o g) = suminf f
  by (rule antisym)
qed

lemma sums-mono-reindex:
assumes subseq: strict-mono g
  and zero: ∀n. n /∈ range g ⇒ f n = 0
shows (λn. f (g n)) sums c ←→ f sums c
unfolding sums-def
proof
  assume lim: (λn. ∑ k<n. f k) → c
  have (λn. ∑ k<n. f (g k)) = (λn. ∑ k<g n. f k)
  proof
THEORY "Series"

fix n :: nat
from subseq have \((\sum k< n. f (g k)) = (\sum k< g' ..< n. f k)\)
by (subst sum.reindex) (intro intro: strict-mono-imp-inj-on)
also from subseq have \((\sum k< g n. f k)\)
by (intro sum.mono-neutral-left ballI zero)
(auto simp: strict-mono-less strict-mono-less-eq)
finally show \((\sum k< n. f (g k)) = (\sum k< g n. f k)\).
qed
also from LIMSEQ-subseq-LIMSEQ[OF lim subseq] have \(\cdots \longrightarrow c\)
by (simp only: o-def)
finally show \((\lambda n. \sum k< n. f (g k)) \longrightarrow c\).
next
assume lim: \((\lambda n. \sum k< n. f (g k)) \longrightarrow c\)
define g-inv where \(g-inv n = \{\text{LEAST } m | g m \geq n\}\) for n
from filterlim-subseq[OF subseq] have g-inv-ex: \(\exists m. g m \geq n\) for n
by (auto simp: filterlim-at-top eventually-at-top-linorder)
then have g-inv: \(g (g-inv n) \geq n\) for n
unfolding g-inv-def by (rule LeastI-ex)
have g-inv-least: \(m \geq g-inv n\) if \(g m \geq n\) for \(m n\)
using that unfolding g-inv-def by (rule Least-le)
have g-inv-least': \(g m < n\) if \(m < g-inv n\) for \(m n\)
using that g-inv-least[of n m] by linarith
have \((\lambda n. \sum k< n. f k) = (\lambda n. \sum k< g-inv n. f (g k))\)
proof
fix n :: nat
{
fix k
assume k: \(k \in \{..< n\} - g' \{..< g-inv n\}\)
have k \notin range g
proof (rule notI, elim imageE)
fix l
assume l: \(k = g l\)
have g l < g (g-inv n)
by (rule less-le-trans[OF - g-inv]) (use k l in simp-all)
with subseq have l < g-inv n
by (simp add: strict-mono-less)
with k l show False
by simp
qed
then have f k = 0
by (rule zero)
}
with g-inv-least' g-inv have \((\sum k< n. f k) = (\sum k< g' \{..< g-inv n\}. f k)\)
by (intro sum.mono-neutral-right) auto
also from subseq have \((\sum k< g-inv n. f (g k))\)
using strict-mono-imp-inj-on by (subst sum.reindex) simp-all
finally show \((\sum k< n. f k) = (\sum k< g-inv n. f (g k))\).
qed
also {
fix $K$ $n :: nat$
assume $g K \leq n$
also have $n \leq g \ (g\text{-inv} \ n)$
by (rule $g\text{-inv}$)
finally have $K \leq g\text{-inv} \ n$
using subseq by (simp add: strict-mono-less-eq)
}
then have filterlim $g\text{-inv}$ at-top sequentially
by (auto simp add: filterlim-at-top eventually-at-top-linorder)
with $\lim$ have $(\lambda n. \sum k < g\text{-inv} \ n \ f \ (g \ k)) \longrightarrow c$
by (rule filterlim-compose)
finally show $(\lambda n. \sum k < n \ f \ k) \longrightarrow c$.
qed

lemma summable-mono-reindex:
assumes subseq: strict-mono $g$
and zero: $\forall n. n \notin \text{range} \ g \Longrightarrow f \ n = 0$
shows summable $(\lambda n. f \ (g \ n)) \Longleftrightarrow \text{summable} \ f$
using sums-mono-reindex[of $g \ f$, OF assms] by (simp add: summable-def)

lemma suminf-mono-reindex:
fixes $f :: nat \Rightarrow 'a :: \{t2\text{-space}, comm\text{-monoid\_add}\}$
assumes strict-mono $g \ \forall n. n \notin \text{range} \ g \Longrightarrow f \ n = 0$
shows suminf $(\lambda n. f \ (g \ n)) = \text{suminf} \ f$
proof (cases summable $f$)
case True
with sums-mono-reindex [of $g \ f$, OF assms]
and summable-mono-reindex [of $g \ f$, OF assms]
show thesis
by (simp add: sums-iff)
next
case False
then have $\neg(\exists c. f \sums c)$
unfolding summable-def by blast
then have suminf $f = \text{The} \ (\lambda -. \text{False})$
by (simp add: suminf-def)
moreover from False have $\neg \text{summable} \ (\lambda n. f \ (g \ n))$
using summable-mono-reindex[of $g \ f$, OF assms] by simp
then have $\neg(\exists c. \ (\lambda n. f \ (g \ n)) \sums c)$
unfolding summable-def by blast
then have suminf $(\lambda n. f \ (g \ n)) = \text{The} \ (\lambda -. \text{False})$
by (simp add: suminf-def)
ultimately show thesis by simp
qed

lemma summable-bounded-partials:
fixes $f :: nat \Rightarrow 'a :: \{\text{real\text{-}normed\_vector}, \text{complete\_space}\}$
assumes bound: eventually $(\lambda x0. \forall a \geq x0. \forall b > a. \text{norm} \ f \ \{a <.. b\}) \leq g \ a)$
sequentially
assumes $g$: $g \rightarrow 0$

shows summable $\text{f unfolding}$ summable-iff-convergent'

proof (intro Cauchy-convergent CauchyI', goal-cases)
  case (1 $\varepsilon$)
  with $g$ have eventually $(\lambda x. |g x| < \varepsilon)$ sequentially
    by (auto simp: tendsto_iff)
  from eventually-conj[OF this bound] obtain $x0$ where $x0$:
    $\forall x. x \geq x0 \implies |g x| < \varepsilon$ $\land a. x0 \leq a \implies a < b \implies \text{norm} \ (\text{sum f }\{a<..b\})$
  $\leq g a$
  unfolding eventually-at-top-linorder by auto
  show ?case
  proof (intro exI[of - $x0$] allI impI)
    fix $m$ $n$
    assume $mn$:
      $x0 \leq m$ $m < n$
    have $\text{dist} \ (\text{sum f }\{..m\}) \ (\text{sum f }\{..n\}) = \text{norm} \ (\text{sum f }\{..n\} - \text{sum f }\{..m\})$
      by (simp add: dist-norm norm-minus-commute)
    also have $\text{sum f }\{..n\} - \text{sum f }\{..m\} = \text{sum f }\{..n\} - \{..m\}$
      using $mn$ by (intro Groups-Big.sum-diff [symmetric]) auto
    also have $\{..n\} - \{..m\} = \{m<..n\}$ using $mn$ by auto
    also have $\ldots \leq |g m|$ by simp
    also have $\ldots < \varepsilon$ using $mn$ by (intro $x0$) auto
    finally show $\text{dist} \ (\text{sum f }\{..m\}) \ (\text{sum f }\{..n\}) < \varepsilon$ .
  qed
  qed
end

111 Differentiation

type Deriv
  imports Limits
begin

111.1 Frechet derivative

definition has-derivative :: "('a::real-normed-vector => 'b::real-normed-vector) =>
  ('a => 'b) => 'a filter => bool" (infix "has-derivative" 50)
where \(f \text{ has-derivative } f'\) $F \leftrightarrow$
  bounded-linear $f' \land$
  \((\lambda y. ((f y - f \ (\text{Lim F} \ (\lambda x. x)))) - f' \ (y - \text{Lim F} \ (\lambda x. x))) / \text{norm} \ (y - \text{Lim F} \ (\lambda x. x))) \rightarrow 0 \) $F$

Usually the filter $F$ is at $x$ within $s$. $(f \text{ has-derivative } D)$ (at $x$ within $s$)
means: $D$ is the derivative of function $f$ at point $x$ within the set $s$. Where $s$ is used to express left or right sided derivatives. In most cases $s$ is either
a variable or $\text{UNIV}$.

These are the only cases we’ll care about, probably.

lemma has-derivative-within: $(f \text{ has-derivative } f')$ (at $x$ within $s$) $\leftrightarrow$
bounded-linear \( f' \land ((\lambda y. (1 / \text{norm}(y - x)) \ast_R (f y - (f x + f'(y - x)))) \longrightarrow 0) \ (at \ x \ within \ s) \)

unfolding has-derivative-def tendsto-iff
by (subst eventually-Lim-ident-at) (auto simp add: field-simps)

lemma has-derivative-eq-rhs: \( (f \text{ has-derivative } f') \ F \Longrightarrow f' = g' \Longrightarrow (f \text{ has-derivative } g') \ F \)
by simp

definition has-field-derivative :: \( ('a::real-normed-field \Rightarrow 'a') \Rightarrow 'a \Rightarrow 'a \ filter \Rightarrow bool \)
(infix (has'-field'-derivative) 50)
where \( (f \text{ has-field-derivative } D) \ F \leftarrowto (f \text{ has-derivative } (\ast) \ D) \ F \)

lemma DERIV-cong: \( (f \text{ has-field-derivative } X) \ F \Longrightarrow X = Y \Longrightarrow (f \text{ has-field-derivative } Y) \ F \)
by simp

definition has-vector-derivative :: \( (\text{real} \Rightarrow 'b::real-normed-vector) \Rightarrow 'b \Rightarrow \text{real filter} \Rightarrow bool \)
(infix (has'-vector'-derivative) 50)
where \( (f \text{ has-vector-derivative } f') \ net \leftarrowto (f \text{ has-derivative } (\lambda x. x \ast_R f')) \ net \)

lemma has-vector-derivative-eq-rhs:
(\( f \text{ has-vector-derivative } X \) \ F \Longrightarrow X = Y \Longrightarrow (f \text{ has-vector-derivative } Y) \ F \)
by simp

named-theorems derivative-intros structural introduction rules for derivatives
setup \lang
let
val eq-thms = @{thms has-derivative-eq-rhs DERIV-cong has-vector-derivative-eq-rhs}
fun eq-rule thm = get-first (try (fn eq-thm => eq-thm OF [thm])) eq-thms
in
Global-Theory.add-thms-dynamic
(binding : derivative-eq-intros,
fn context =>
Named-Theorems.get (Context.proof-of-context) named-theorems derivative-intros
|> map-filter eq-rule)
end\nlang

The following syntax is only used as a legacy syntax.

abbreviation (input)
FDERIV :: \( ('a::real-normed-vector \Rightarrow 'b::real-normed-vector) \Rightarrow 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool \)
((FDERIV (-)/ (-)/ :: (-)) [1000, 1000, 60] 60)
where FDERIV \( f \) \( x \) :: \( f' \equiv (f \text{ has-derivative } f') \ (at \ x) \)

lemma has-derivative-bounded-linear: \( (f \text{ has-derivative } f') \ F \Longrightarrow \text{bounded-linear } f' \)
by (simp add: has-derivative-def)

lemma has-derivative-linear: \((f \text{ has-derivative } f') \Rightarrow \text{linear } f'\)
using bounded-linear.linear[\(\text{OF has-derivative-bounded-linear}\)] .

lemma has-derivative-ident[derivative-intros, simp]: \((\lambda x. x) \text{ has-derivative } (\lambda x. x)\) \(F\)
by (simp add: has-derivative-def)

lemma has-derivative-id [derivative-intros, simp]: \((\text{id} \text{ has-derivative } \text{id})\) \(F\)
by (metis eq-id-iff has-derivative-ident)

lemma shift-has-derivative-id: \((+ \ d) \text{ has-derivative } (\lambda x. x)\) \(F\)
using has-derivative-def by fastforce

lemma has-derivative-const[derivative-intros, simp]: \((\lambda x. c) \text{ has-derivative } (\lambda x. 0)\) \(F\)
by (simp add: has-derivative-def)

lemma (in bounded-linear) bounded-linear: bounded-linear \(f\) ..

lemma (in bounded-linear) has-derivative:
\((g \text{ has-derivative } g') \Rightarrow ((\lambda x. f (g x)) \text{ has-derivative } (\lambda x. f (g' x)))\) \(F\)
unfolding has-derivative-def
by (auto simp add: bounded-linear-compose [\(\text{OF bounded-linear}\) scaleR diff dest: tendsto)

lemma has-derivative-bot [intro]: bounded-linear \(f' \Rightarrow (f \text{ has-derivative } f')\) bot
by (auto simp: has-derivative-def)

lemma has-field-derivative-bot [simp, intro]: \((f \text{ has-field-derivative } f')\) bot
by (auto simp: has-field-derivative-def intro \(!:\text{has-derivative-bot bounded-linear-mult-right}\)

lemmas has-derivative-scaleR-right [derivative-intros] = bounded-linear.has-derivative [\(\text{OF bounded-linear-scaleR-right}\)]

lemmas has-derivative-scaleR-left [derivative-intros] = bounded-linear.has-derivative [\(\text{OF bounded-linear-scaleR-left}\)]

lemmas has-derivative-mult-right [derivative-intros] = bounded-linear.has-derivative [\(\text{OF bounded-linear-mult-right}\)]

lemmas has-derivative-mult-left [derivative-intros] = bounded-linear.has-derivative [\(\text{OF bounded-linear-mult-left}\)]

lemmas has-derivative-of-real[derivative-intros, simp] = bounded-linear.has-derivative[\(\text{OF bounded-linear-of-real}\)]

lemma has-derivative-add[simp, derivative-intros]:
assumes $f$: ($f$ has-derivative $f'$) $F$
and $g$: ($g$ has-derivative $g'$) $F$
shows (($\lambda x. f + g$) has-derivative $(\lambda x. f' + g'$)) $F$
unfolding has-derivative-def

proof

safe

let $\exists x = \operatorname{Lim} F (\lambda x. x)$

let $\exists D = \lambda f' f. (f' y - f' x)/(y - x)$ $\longrightarrow (0 + 0)$ $F$

using $f g$ by (intro tendsto-add) (auto simp: has-derivative-def)
then show $(\exists D (\lambda x. f + g x) (\lambda x. f' x + g' x) \longrightarrow 0) F$

by (simp add: field-simps scaleR-add-right scaleR-diff-right)

qed (blast intro: bounded-linear-add $f g$ has-derivative-bounded-linear)

lemma has-derivative-sum[simp, derivative-intros]:

$(\forall i. i \in I \Longrightarrow (f i \text{ has-derivative } f' i) F) \Longrightarrow$

($(\lambda x. \sum_{i \in I} f i x)$ has-derivative $(\lambda x. \sum_{i \in I} f' i x)) F$

by (induct I rule: infinite-finite-induct) simp-all

lemma has-derivative-minus[simp, derivative-intros]:

$(f \text{ has-derivative } f') F \Longrightarrow ((\lambda x. f x) \text{ has-derivative } (\lambda x. f' x)) F$

using has-derivative-scaleR-right[of $f f'$ $F$ $-1$] by simp

lemma has-derivative-diff[simp, derivative-intros]:

$(f \text{ has-derivative } f') F \Longrightarrow (g \text{ has-derivative } g') F \Longrightarrow$

($(\lambda x. f x - g x)$ has-derivative $(\lambda x. f' x - g' x)) F$

by (simp only: diff-conv-add-uminus has-derivative-add has-derivative-minus)

lemma has-derivative-at-within:

$(f \text{ has-derivative } f') (\text{at x within s}) \Longleftrightarrow$

$(\text{bounded-linear } f' \land (\lambda y. ((f y - f x) - f' (y - x)) / R \text{ norm } (y - x)) \longrightarrow 0) (\text{at x within s})$

proof (cases at x within s = $\text{bot}$)

case True

then show $?\text{thesis}$

by (metis (no-types, lifting) has-derivative-within tendsto-bot)

next

case False

then show $?\text{thesis}$

by (simp add: Lim-ident-at has-derivative-def)

qed

lemma has-derivative-iff-norm:

$(f \text{ has-derivative } f') (\text{at x within s}) \Longleftrightarrow$

$\text{bounded-linear } f' \land ((\lambda y. \text{norm } ((f y - f x) - f' (y - x)) / \text{norm } (y - x)) \longrightarrow 0) (\text{at x within s})$

using tendsto-norm-zero-iff[of - at x within s, where 'b='b, symmetric]

by (simp add: has-derivative-at-within divide-inverse ac-simps)

lemma has-derivative-at:
(f has-derivative D) (at x) ──→
(bounded-linear D ∧ (∃ h. norm (f (x + h) − f x − D h) / norm h) −0→ 0)
by (simp add: has-derivative-iff-norm LIM-offset-zero-iff)

lemma field-has-derivative-at:
fixes x :: 'a::real-normed-field
shows (f has-derivative (∗) D) (at x) ──→ (∃ h. (f (x + h) − f x) / h) −0→ D
(is ?lhs = ?rhs)
proof
have ?lhs = (∃ h. norm (f (x + h) − f x − D * h) / norm h) −0→ 0
  by (simp add: bounded-linear-mult-right has-derivative-at)
also have ... = (∃ y. norm ((f (x + y) − f x − D * y) / y)) −0→ 0
  by (simp cong: LIM-cong flip: nonzero-norm-divide)
also have ... = (∃ y. norm ((f (x + y) − f x) / y − D / y * y)) −0→ 0
  by (simp only: diff-divide-distrib times-divide_eq_left [symmetric])
also have ... = ?rhs
  by (simp add: tendsto-norm-zero-iff LIM-zero-iff cong: LIM-cong)
finally show ?thesis .

qed

lemma has-derivative-iff-Ex:
(f has-derivative f') (at x) ──→
bounded-linear f' ∧ (∃ e. (∀ h. f (x+h) = f x + f' h + e h) ∧ (∃ h. norm (e h) / norm h) −0→ 0) (at 0))
unfolding has-derivative-at by force

lemma has-derivative-at-within-iff-Ex:
assumes x ∈ S open S
shows (f has-derivative f') (at x within S) ──→
bounded-linear f' ∧ (∃ e. (∀ h. x + h ∈ S → f (x+h) = f x + f' h + e h) ∧ (∃ h. norm (e h) / norm h) −0→ 0) (at 0))
(is ?lhs = ?rhs)
proof safe
  show bounded-linear f'
    if (f has-derivative f') (at x within S)
    using has-derivative-bounded-linear that by blast
  show ∃ e. (∀ h. x + h ∈ S → f (x+h) = f x + f' h + e h) ∧ (∃ h. norm (e h))/ norm h) −0→ 0
    if (f has-derivative f') (at x within S)
    by (metis (full-types) assms that has-derivative-iff-Ex at-within-open)
  show (f has-derivative f') (at x within S)
    if bounded-linear f'
      and eq [rule-format]: ∀ h. x + h ∈ S → f (x + h) = f x + f' h + e h
      and 0: (λh. norm (e (h::'a::'b))/ norm h) −0→ 0
      for e
    proof
      have 1: f y − f x = f' (y − x) + e (y−x) if y ∈ S for y
        using eq [of y−x] that by simp
      have 2: ((λy. norm (e (y−x)) / norm (y − x)) −0→ 0) (at x within S)
THEORY "Deriv"

have \((\lambda y. \text{norm} (f y - f x - f' (y - x)) / \text{norm} (y - x)) \longrightarrow 0)\) (at \(x\) within \(S\))
  by (simp add: Lim-cong-within 1 2)
then show "thesis"
  by (simp add: has-derivative-iff-norm "bounded-linear f'")
qed

lemma has-derivativeI-
  "bounded-linear f' \implies ((\lambda y. \text{norm} (f y - f x - f' (y - x)) / \text{norm} (y - x)) \longrightarrow 0)\) (at \(x\) within \(s\))
  by (simp add: has-derivative-at-within)

lemma has-derivativeI-sandwich:
assumes e: "0 < e"
  and bounded: "bounded-linear f'"
  and sandwich: "(\lambda y. y \in s \implies y \neq x \implies \text{dist} y x < e \implies \text{norm} (f y - f x - f' (y - x)) / \text{norm} (y - x) \leq H y)"
  and (H \longrightarrow 0)\) (at \(x\) within \(s\))
show (f has-derivative f') (at \(x\) within \(s\))
unfolding has-derivative-iff-norm
proof safe
  show ((\lambda y. \text{norm} (f y - f x - f' (y - x)) / \text{norm} (y - x)) \longrightarrow 0)\) (at \(x\) within \(s\))
    by (rule tendsto-sandwich[where f=\(\lambda x. 0\)])
  show (H \longrightarrow 0)\) (at \(x\) within \(s\))
    proof (auto simp: le-divide-eq)
      qed
    qed
qed

lemma has-derivative-subset:
  "(f has-derivative f') (at \(x\) within \(s\)) \implies t \subseteq s \implies (f has-derivative f') (at \(x\) within \(t\))"
  by (auto simp add: has-derivative-iff-norm intro: tendsto-within-subset)

lemma has-derivative-within-singleton-iff:
  "(f has-derivative g) (at \(x\) within \(\{x\}\)) \iff \text{bounded-linear} g"
  by (auto intro!: has-derivativeI-sandwich[where e=1] has-derivative-bounded-linear)

111.1.1 Limit transformation for derivatives

lemma has-derivative-transform-within:
  assumes (f has-derivative f') (at \(x\) within \(s\))
  and 0 < d
  and \(x\) \in \(s\)
  by
and $\forall x'. [x' \in s; \text{dist } x' x < d] \Rightarrow f x' = g x'$

shows ($g$ has-derivative $f'$) (at $x$ within $s$)

using assms

unfolding has-derivative-within

by (force simp add: intro: Lim-transform-within)

lemma has-derivative-transform-within-open:

assumes ($f$ has-derivative $f'$) (at $x$ within $t$)

and open $s$

and $x \in s$

and $\forall x. x \in s \Rightarrow f x = g x$

shows ($g$ has-derivative $f'$) (at $x$ within $t$)

using assms unfolding has-derivative-within

by (force simp add: intro: Lim-transform-within-open)

lemma has-derivative-transform:

assumes $x \in s \land x \in s \Rightarrow g x = f x$

assumes ($f$ has-derivative $f'$) (at $x$ within $s$)

shows ($g$ has-derivative $f'$) (at $x$ within $s$)

using assms

by (intro has-derivative-transform-within[OF - zero-less-one, where $g=g$]) auto

lemma has-derivative-transform-eventually:

assumes ($f$ has-derivative $f'$) (at $x$ within $s$)

$(\forall F. x' \in \text{at } x \text{ within } s. f x' = g x')$

assumes $f x = g x x \in s$

shows ($g$ has-derivative $f'$) (at $x$ within $s$)

using assms

proof -

from assms(2,3) obtain $d$ where $d > 0 \land x'. x' \in s \Rightarrow \text{dist } x' x < d \Rightarrow f x' = g x'$

by (force simp: eventually-at)

from has-derivative-transform-within[OF assms(1) this(1) assms(4) this(2)]

show ?thesis.

qed

lemma has-field-derivative-transform-within:

assumes ($f$ has-field-derivative $f'$) (at $a$ within $S$)

and $0 < d$

and $a \in S$

and $\forall x. [x \in S; \text{dist } x a < d] \Rightarrow f x = g x$

shows ($g$ has-field-derivative $f'$) (at $a$ within $S$)

using assms unfolding has-field-derivative-def

by (metis has-derivative-transform-within)

lemma has-field-derivative-transform-within-open:

assumes ($f$ has-field-derivative $f'$) (at $a$)

and open $S a \in S$

and $\forall x. x \in S \Rightarrow f x = g x$
shows \((g \text{ has-field-derivative } f')\) \((at \ a)\)
using asms unfolding has-field-derivative-def
by (metis has-derivative-transform-within-open)

111.2 Continuity

lemma has-derivative-continuous:
assumes \(f: (f \text{ has-derivative } f')\) \((at \ x \ within \ s)\)
shows \(\text{continuous} \ (at \ x \ within \ s)\)
proof −
from \(f\) interpret \(F\): bounded-linear \(f'\)
by (rule has-derivative-bounded-linear)
note \(F.\)tendsto [tendsto-intros]
let \(?L = \lambda f. (f \longrightarrow 0)\) \((at \ x \ within \ s)\)
have \(?L (\lambda y. \text{norm} ((f y - f x) - f' (y - x)) / \text{norm} (y - x))\)
using \(f\) unfolding has-derivative-iff-norm by blast
then have \(?L (\lambda y. \text{norm} ((f y - f x) - f' (y - x)) / \text{norm} (y - x) * \text{norm} (y - x))\) \((is \ ?m)\)
by (rule tendsto-mult-zero) \(\text{auto intro: tendsto-eq-intros}\)
also have \(?m \leftarrow\rightarrow \ ?L (\lambda y. (f y - f x) - f' (y - x))\)
by (intro filterlim-cong) \(\text{simp-all add: eventually-at-filter}\)
finally have \(?L (\lambda y. (f y - f x))\)
by (rule tendsto-norm-zero-cancel)
then have \(?L (\lambda y. (f y - f x)) + f' (y - x)\)
by simp
from tendsto-add[OF this tendsto-const, of \(f x\)] show \(?\text{thesis}\)
by (simp add: continuous-within)
qed

111.3 Composition

lemma tendsto-at-iff-tendsto-nhds-within:
\(f x = y \implies (f \longrightarrow y)\) \((at \ x \ within \ s)\) \(\longleftrightarrow\) \((f \longrightarrow y)\) \((\text{inf} (\text{nhds} x) \text{ (principal \ s)})\)
unfolding tendsto-def eventually-inf-principal eventually-at-filter
by (intro ext all-conq imp-conq) \(\text{auto elim!: eventually-mono}\)

lemma has-derivative-in-compose:
assumes \(f: (f \text{ has-derivative } f')\) \((at \ x \ within \ s)\)
and \(g: (g \text{ has-derivative } g')\) \((at \ f x \ within \ (f's))\)
shows \((\lambda x. g (f x)) \text{ has-derivative} (\lambda x. g' (f' x))\) \((at \ x \ within \ s)\)
proof −
from \(f\) interpret \(F\): bounded-linear \(f'\)
by (rule has-derivative-bounded-linear)
from \(g\) interpret \(G\): bounded-linear \(g'\)
by (rule has-derivative-bounded-linear)
from \(F.bounded\) obtain \(kF\) where \(kF\): \(\lambda x. \text{norm} (f' x) \leq \text{norm} x * kF\)
by fast
from G.bounded obtain kG where kG: \( \forall x. \text{norm} \ (g' \ x) \leq \text{norm} \ x \ast kG \)

by fast

note G.tendsto[tendsto-intros]

let \(?L = \lambda f. (f \longrightarrow 0) \ (\text{at} \ x \ \text{within} \ s)\)

let \(?D = \lambda f \ x \ y. (f \ y - f \ x) \ast f' (y - x)\)

let \(?N = \lambda f \ x \ y. \text{norm} \ ((D \ f \ f' \ x \ y) / \text{norm} \ (y - x)\)

let \(?gf = \lambda x. g \ (f \ x)\)

and \(?gf' = \lambda x. g' \ (f' \ x)\)

define \(Nf\) where \(Nf = ?Ng f\)

define \(Ng\) where \[\text{abs-def}: Ng \ y = \ ?Ng \ g \ g' \ (f \ x) \ (f \ y) \text{ for } y\]

show \(\neg \text{thesis}\)

proof (rule has-derivativeI-sandwich[of 1])

show bounded-linear \((\lambda x. g' \ (f' \ x))\)

using \(fg\) by (blast intro: bounded-linear-compose has-derivative-bounded-linear)

next

fix \(y::'a\)

assume \(\neg g: g \neq x\)

have \(?N \ ?gf \ ?gf' \ x \ y = \text{norm} \ (g' \ ((D \ f \ f' \ x \ y) + D \ g \ g' \ (f \ x) \ (f \ y)) / \text{norm} \ (y - x)\)

by (simp add: G.diff G.add field-simps)

also have \(\ldots \leq \text{norm} \ ((D \ f \ f' \ x \ y) + \text{norm} \ (y - x) + Ng \ y \ast \text{norm} \ (f \ y - f \ x) / \text{norm} \ (y - x)\)

by (simp add: add-divide-distrib[symmetric] divide-right-mono norm-triangle-ineq)

G.zero \(Ng\-def\)

also have \(\ldots \leq Nf \ y \ast kG + Ng \ y \ast (Nf \ y + kF)\)

proof (intro add-mono multi-left-mono)

have \(\text{norm} \ (f \ y - f \ x) = \text{norm} \ ((D \ f \ f' \ x \ y + f' (y - x))\)

by simp

also have \(\ldots \leq \text{norm} \ ((D \ f \ f' \ x \ y) \ast \text{norm} \ (f' (y - x))\)

by (rule norm-triangle-ineq)

also have \(\ldots \leq \text{norm} \ ((D \ f \ f' \ x \ y) + \text{norm} \ (y - x) \ast kF\)

using \(kF\) by (intro add-mono) simp

finally show \(\text{norm} \ (f \ y - f \ x) / \text{norm} \ (y - x) \leq Nf \ y + kF\)

by (simp add: neq Nf-def field-simps)

qed (use kG in (simp-all add: Ng-def Nf-def neq zero-le-divide-iff field-simps))

finally show \(?N \ ?gf \ ?gf' \ x \ y \leq Nf \ y \ast kG + Ng \ y \ast (Nf \ y + kF)\).

next

have \([\text{tendsto-intros}]: ?L \ Nf\)

using \(f\) unfolding has-derivative-iff-norm Nf-def ..

from \(f\) have \((f \longrightarrow f \ x) \ (\text{at} \ x \ \text{within} \ s)\)

by (blast intro: has-derivative-continuous continuous-within[THEN iffD1])

then have \(f': \text{lim} \ x \ x \ \text{within} \ s. f \ x :> \inf \ (\text{nhds} \ (f \ x)) \ (\text{principal} \ (f\ s))\)

unfolding filterlim-def

by (simp add: eventually-filtermap eventually-at-filter le-principal)

have \(((?N \ y \ g \ g' \ (f \ x)) \longrightarrow 0) \ (\text{at} \ f) \ \text{within} \ f\ s\)

using \(g\) unfolding has-derivative-iff-norm ..

then have \(g': (((?N \ y \ g \ g' \ (f \ x)) \longrightarrow 0) \ (\text{inf} \ (\text{nhds} \ (f \ x)) \ (\text{principal} \ (f\ s)))\)


by (rule tendsto-at-iff-tendsto-nhds-within [THEN iffD1, rotated]) simp

have [tendsto-intros]: ?L Ng
  unfolding Ng-def by (rule filterlim-compose [OF g' f'])
  show ((λy. Nf y * kG + Ng y * (Nf y + kF)) −−→ 0) (at x within s)
    by (intro tendsto-eq-intros) auto
qed simp

lemma has-derivative-compose:
  (f has-derivative f') (at x within s) ⇒ (g has-derivative g') (at f x) ⇒
  ((λx. g (f x)) has-derivative (λy. g' (f' x))) (at x within s)
by (blast intro: has-derivative-in-compose has-derivative-subset)

lemma has-derivative-in-compose2:
  assumes ⋀ x. x ∈ t =⇒ (g has-derivative g' x) (at x within t)
  assumes f' s ⊆ t x ∈ s
  assumes (f has-derivative f') (at x within s)
  shows ((λx. g (f x)) has-derivative (λy. g' (f' x)) (f' y))) (at x within s)
using assms
by (auto intro: has-derivative-subset intro !: has-derivative-in-compose [of f f' x s])

lemma (in bounded-bilinear) FDERIV:
  assumes f: (f has-derivative f') (at x within s) and g: (g has-derivative g') (at x within s)
  shows ((λx. f x ** g x) has-derivative (λh. f x ** g' h + f' h ** g x)) (at x within s)
by (fast)

proof –
  from bounded-linear bounded [OF has-derivative-bounded-linear [OF f]]
  obtain KF where norm-F: ∃ x. norm (f' x) ≤ norm x * KF by fast

from pos-bounded obtain K
  where K: 0 < K and norm-prod: ∀ a b. norm (a ** b) ≤ norm a * norm b * K
  by fast
let ?D = λf' y. f y - f x - f' (y - x)
let ?N = λf' y. norm (?D f' y) / norm (y - x)
define Ng where Ng = ?N g g'
define Nf where Nf = ?N f f'

let ?fun1 = λy. norm (f y ** g y - f x ** g x - (f x ** g' (y - x) + f' (y - x) ** g x)) / norm (y - x)
let ?fun2 = λy. norm (f x) * Ng y * K + Nf y * normalize norm (g y) * K + K * normalize norm (g y - g x) * K
let ?F = at x within s

show ?thesis
proof (rule has-derivativeI-sandwich[of 1])
show bounded-linear \((\lambda h. f x ** g' h + f' h ** g x)\)
by (intro bounded-linear-add
bounded-linear-compose [OF bounded-linear-right] bounded-linear-compose
[OF bounded-linear-left]
has-derivative-bounded-linear [OF g] has-derivative-bounded-linear [OF f])
next
from \(g\) have \((g \longrightarrow g x)\)
by (intro continuous-within[THEN iffD1] has-derivative-continuous)
moreover from \(f\ g\) have \((Nf \longrightarrow 0)\)
by (simp-all add: has-derivative-iff-norm Ng-def Nf-def)
ultimately have \((?\text{fun2} \longrightarrow \text{norm} (f x) * 0 * K + 0 * \text{norm} (g x) * K + KF * \text{norm} (0::\text{b}) * K)\)
by (intro tendsto-intros) (simp-all add: LIM-zero-iff)
then show \((?\text{fun2} \longrightarrow 0)\)
by simp
next
fix \(y::\text{d}\)
assume \(y \neq x\)

have \(?\text{fun1} y = \text{norm} (f x ** (?D g g' y + ?D f f' y ** g y + f' (y - x) ** (g y - g x))) / \text{norm} (y - x)\)
by (simp add: diff-left diff-right add-left add-right field-simps)
also have \(\ldots \leq (\text{norm} (f x) * \text{norm} (?D g g' y) * K + \text{norm} (?D f f' y) * \text{norm} (g y) * K + KF * \text{norm} (g y - g x) * K) / \text{norm} (y - x)\)
by (intro divide-right-mono mult-mono
order-trans [OF norm-triangle-ineq add-mono]
order-trans [OF norm-prod mult-right-mono]
mult-nonneg-nonneg order-refl norm-ge-zero norm-F
K [THEN order-less-imp-le])
also have \(\ldots = (?\text{fun2} y)\)
by (simp add: add-divide-distrib Ng-def Nf-def)
finally show \(?\text{fun1} y \leq ?\text{fun2} y\).
qed simp

lemmas has-derivative-mult[simp, derivative-intros] = bounded-bilinear.FDERIV[OF bounded-bilinear-mult]
lemmas has-derivative-scaleR[simp, derivative-intros] = bounded-bilinear.FDERIV[OF bounded-bilinear-scaleR]

lemma has-derivative-prod[simp, derivative-intros]:
fixes \(f :: 'i \Rightarrow 'a::real-normed-vector \Rightarrow 'b::real-normed-field\)
shows \((\lambda g. \bigwedge i. i \in I \Longrightarrow (f i \text{ has-derivative } f' i) \text{ (at } x \text{ within } S)) \Longrightarrow
((\lambda g. \bigwedge i. i \in I. f' i y * (\prod j \in I - \{i\}. f j x))\)
\text{has-derivative} (\lambda g. \sum i \in I. f' i y * (\prod j \in I - \{i\}. f j x))\)
\text{ (at } x \text{ within } S)\)
proof (induct I rule: infinite-finite-induct)
case infinite
then show \(?\text{case} by simp\)
next
case empty
then show ?case by simp
next
case (insert $i$ $I$
let $?P = \lambda y. f\ i\ x * (\sum i\in I. f'\ i\ y * (\prod j\in I - \{i\}. f\ j\ x)) + (f'\ i\ y) * (\prod i\in I. f\ i\ x)$

have $((\lambda x. f\ i\ x * (\prod i\in I. f\ i\ x))$ has-derivative $?P)$ (at $x$ within $S$
using insert by (intro has-derivative-mult) auto
also have $?P = (\lambda y. \sum i'\in insert\ i\ I. f'\ i'\ y * (\prod j\in insert\ i\ I - \{i'\}. f\ j\ x))$
using insert(1,2)
by (auto simp add: sum-distrib-left insert-Diff-if intro! ext sum.cong)
finally show ?case
using insert by simp
qed

lemma has-derivative-power[simp, derivative-intros]:
fixes $f :: 'a :: real-normed-vector ⇒ 'b :: real-normed-field$
assumes $f$: ($f$ has-derivative $f'$) (at $x$ within $S$
shows $((\lambda x. f\ x^n)$ has-derivative $(\lambda y. of-nat\ n * f'\ y * f\ x^{(n - 1)}))$ (at $x$ within $S$
using has-derivative-prod[OF $f$, of {..< n}] by (simp add: prod-constant ac-simps)

lemma has-derivative-inverse':
fixes $x :: 'a::real-normed-dv-algebra$
assumes $x$: $x \neq 0$
shows (inverse has-derivative $(\lambda h. - (inverse\ x * h * inverse\ x)))$ (at $x$ within $S$
(is (- has-derivative $?f$) -)
proof (rule has-derivativeI-sandwich)
show bounded-linear $(\lambda h. - (inverse\ x * h * inverse\ x))$
by (simp add: bounded-linear-minus bounded-linear-mult-const bounded-linear-mult-right)
show $0 < norm\ x$ using $x$ by simp
have (inverse $\longrightarrow$ inverse $x$) (at $x$ within $S$
using tendsto-inverse tendsto-ident-at $x$ by auto
then show $((\lambda y. norm\ (inverse\ y - inverse\ x) * norm\ (inverse\ x))$ $\longrightarrow 0)$ (at $x$ within $S$
by (simp add: LIM-zero-iff tendsto-mul-left-zero tendsto-norm-zero)
next
fix $y :: 'a$
assume $h$: $y \neq x$ dist $y\ x < norm\ x$
then have $y \neq 0$ by auto
have norm $(inverse\ y - inverse\ x - ?f\ (y - x)) / norm\ (y - x) = norm\ ((inverse\ y * (y - x) * inverse\ x - inverse\ x * (y - x) * inverse\ x)) / norm\ (y - x)$
by (simp add: $\langle y \neq 0 \rangle$ inverse-diff-inverse)
also have $... = norm\ ((inverse\ y - inverse\ x) * (y - x) * inverse\ x) / norm\ (y - x)$
by (simp add: left-diff-distrib norm-minus-commute)
also have \( \ldots \leq \text{norm} (\text{inverse} \ y - \text{inverse} \ x) \ast \text{norm} (y - x) \ast \text{norm} (\text{inverse} \ x) / \text{norm} (y - x) \)
by (simp add: norm-mult)
also have \( \ldots = \text{norm} (\text{inverse} \ y - \text{inverse} \ x) \ast \text{norm} (\text{inverse} \ x) \)
by simp
finally show \( \text{norm} (\text{inverse} \ y - \text{inverse} \ x - f (y - x)) / \text{norm} (y - x) \leq \text{norm} (\text{inverse} \ y - \text{inverse} \ x) \ast \text{norm} (\text{inverse} \ x) \).
qed
THEORY "Deriv"

fixes $f : \mathbb{R} \rightarrow \mathbb{R}$
assumes $x : f \neq 0$
and $f : (f \text{ has-derivative } f') \ (at \ x \ within \ S)$
shows $((\lambda x. \ \text{power-int} \ (f \ x) \ n) \ \text{has-derivative} \ (\lambda h. \ f' \ h * (\text{of-int} \ n * \ \text{power-int} \ (f \ x) \ (n - 1))))$ 
(at $x$ within $S$)
using has-derivative-compose[OF $f$ has-derivative-power-int', OF $x$] .

Conventional form requires mult-AC laws. Types real and complex only.

lemma has-derivative-divide'[derivative-intros]:
fixes $f : \mathbb{R} \rightarrow \mathbb{R}$
assumes $f : (f \text{ has-derivative } f') \ (at \ x \ within \ S)$
and $g : (g \text{ has-derivative } g') \ (at \ x \ within \ S)$
and $x : g \ x \neq 0$
shows $((\lambda x. \ f \ x / g \ x) \ \text{has-derivative} \ (\lambda h. \ (f' \ h * g \ x - f \ x * g' \ h) / (g \ x * g \ x))) \ (at \ x \ within \ S)$
proof
- have $f' \ h / g \ x - f \ x * (\text{inverse} \ (g \ x) * g' \ h * \text{inverse} \ (g \ x)) = (f' \ h * g \ x - f \ x * g' \ h) / (g \ x * g \ x)$ for $h$
by (simp add: field-simps $x$)
then show ?thesis
using has-derivative-divide [OF $f$ $g$] $x$
by simp
qed

11.4 Uniqueness

This can not generally shown for (has-derivative), as we need to approach the point from all directions. There is a proof in Analysis for euclidean-space.

lemma has-derivative-at2: $(f \text{ has-derivative } f') \ (at \ x) \leftrightarrow$
bounded-linear $f' \land ((\lambda y. \ (1 / (\text{norm} \ (y - x)))) *_R \ (f \ y - (f \ x + f' \ (y - x))))$ 
$\rightarrow 0$ \ (at $x$)
using has-derivative-within [of $f$ $f'$ $x$ UNIV]
by simp

lemma has-derivative-zero-unique:
assumes $(\lambda x. \ 0) \ \text{has-derivative} \ F \ (at \ x)$
shows $F = (\lambda h. \ 0)$
proof
- interpret $F : \text{bounded-linear $F$} \ (at \ x)$
using assms by (rule has-derivative-bounded-linear)
let $?r = \lambda h. \ \text{norm} \ (F \ h) / \ \text{norm} \ h$
have $*: \ ?r - 0 \rightarrow 0$
using assms unfolding has-derivative-at by simp
show $F = (\lambda h. \ 0)$
proof
show $F \ h = 0$ for $h$
proof (rule ccontr)
assume **: ¬thesis
then have h: h ≠ 0
  by (auto simp add: F.zero)
with ** have 0 < r h
  by simp
from LIM-D [OF * this] obtain S
  where S: 0 < S and r: \(\forall x. x ≠ 0 \Rightarrow \text{norm } x < S \Rightarrow r x < r h\)
  by auto
from dense [OF S] obtain t where t: 0 < t ∧ t < S
let x = scaleR (t / norm h) h
have x ≠ 0 and norm x < S
  using t h by simp-all
then have r x < r h
  by (rule r)
then show False
  using t h by (simp add: F.scaleR)
qed

lemma has-derivative-unique:
  assumes (f has-derivative F) (at x)
  and (f has-derivative F') (at x)
  shows F = F'
proof
  have ((λx. 0) has-derivative (λh. F h - F' h)) (at x)
    using has-derivative-diff [OF assms] by simp
  then have (λh. F h - F' h) = (λh. 0)
    by (rule has-derivative-zero-unique)
  then show F = F'
    unfolding fun-eq_iff right-minus-eq .
qed

lemma has-derivative-Uniq: ∃≤1 F. (f has-derivative F) (at x)
  by (simp add: Uniq-def has-derivative-unique)

111.5 Differentiability predicate

definition differentiable :: ('a::real-normed-vector ⇒ 'b::real-normed-vector) ⇒ 'a filter ⇒ bool
  (infix differentiable 50)
where f differentiable F =⇒ (∃D. (f has-derivative D) F)

lemma differentiable-subset:
  f differentiable (at x within s) =⇒ t ⊆ s =⇒ f differentiable (at x within t)
  unfolding differentiable-def by (blast intro: has-derivative-subset)

lemmas differentiable-withinsubset = differentiable-subset
lemma differentiable-ident [simp, derivative-intros]: \((\lambda x. x)\) differentiable 
unfolding differentiable-def by (blast intro: has-derivative-ident)

lemma differentiable-const [simp, derivative-intros]: \((\lambda a. a)\) differentiable 
unfolding differentiable-def by (blast intro: has-derivative-const)

lemma differentiable-in-compose: 
\[ f \text{ differentiable (at } (g x) \text{ within } (g's) \) \implies g \text{ differentiable (at } x \text{ within } s \) \implies \]
\[(\lambda x. f (g x)) \text{ differentiable (at } x \text{ within } s \) \]
unfolding differentiable-def by (blast intro: has-derivative-in-compose)

lemma differentiable-compose: 
\[ f \text{ differentiable (at } (g x) \) \implies g \text{ differentiable (at } x \text{ within } s \) \implies \]
\[(\lambda x. f (g x)) \text{ differentiable (at } x \text{ within } s \) \]
by (blast intro: differentiable-in-compose differentiable-subset)

lemma differentiable-add [simp, derivative-intros]: 
\[ f \text{ differentiable } \implies g \text{ differentiable } \implies \]
\[(\lambda x. f x + g x) \text{ differentiable } \]
unfolding differentiable-def by (blast intro: has-derivative-add)

lemma differentiable-sum[simp, derivative-intros]: 
assumes finite s \(\forall a \in s. (f a)\) differentiable 
shows \((\lambda x. \sum (\lambda a. f a x) s)\) differentiable 
proof –
from bchoice[OF assms(2)[unfolded differentiable-def]]
show ?thesis 
  by (auto intro!: has-derivative-sum simp: differentiable-def)
qed

lemma differentiable-minus [simp, derivative-intros]: 
\[ f \text{ differentiable } \implies (\lambda x. - f x) \text{ differentiable } \]
unfolding differentiable-def by (blast intro: has-derivative-minus)

lemma differentiable-diff [simp, derivative-intros]: 
\[ f \text{ differentiable } \implies g \text{ differentiable } \implies (\lambda x. f x - g x) \text{ differentiable } \]
unfolding differentiable-def by (blast intro: has-derivative-diff)

lemma differentiable-mult [simp, derivative-intros]: 
fixes f g :: 'a::real-normed-vector \Rightarrow 'b::real-normed-algebra 
shows \(f \text{ differentiable (at } x \text{ within } s) \implies g \text{ differentiable (at } x \text{ within } s) \implies \]
\[(\lambda x. f x * g x) \text{ differentiable (at } x \text{ within } s) \]
unfolding differentiable-def by (blast intro: has-derivative-mult)

lemma differentiable-cmult-left-iff [simp]: 
fixes c::'a::real-normed-field 
shows \((\lambda t. c * q t)\) differentiable at t \iff c = 0 \lor (\lambda t. q t) differentiable at t 
(is \(?lhs = \?rhs\))
proof 
assume L: \(?lhs\)

\{\textbf{assume} \(c \neq 0\) \\
\textbf{then have} \(q\) differentiable at \(t\) \\
\textbf{using} differentiable-mult \([OF \text{ differentiable-const} \ L, \text{ of concl: } 1/c]\) \textbf{by} auto \\
\}\textbf{then show} \(?\text{rhs}\) \\
\textbf{by} auto \\
\textbf{qed} auto

\textbf{lemma} differentiable-cmult-right-iff \[\text{simp}\]:
\textbf{fixes} \(c : \cdot a : \text{real-normed-field}\)
\textbf{shows} \((\lambda t. q \ t \ast c)\) differentiable at \(t\) \iff \(c = 0 \vee (\lambda t. q \ t)\) differentiable at \(t\)
\textbf{(is \(?\text{lhs} = \ ?\text{rhs}\))} \\
\textbf{by} \((\text{simp add: mult.commute flip: differentiable-cmult-left-iff})\)

\textbf{lemma} differentiable-inverse \[\text{simp, derivative-intros}\]:
\textbf{fixes} \(f : \cdot a : \text{real-normed-vector} \Rightarrow \cdot b : \text{real-normed-field}\)
\textbf{shows} \(f\) differentiable \((at \ x \ within \ s) = \Rightarrow f \ x \neq 0 = \Rightarrow \)
\((\lambda x. \text{inverse} \ (f \ x))\) differentiable \((at \ x \ within \ s)\)
\textbf{unfolding} differentiable-def \textbf{by} \((\text{blast intro: has-derivative-inverse})\)

\textbf{lemma} differentiable-divide \[\text{simp, derivative-intros}\]:
\textbf{fixes} \(f, g : \cdot a : \text{real-normed-vector} \Rightarrow \cdot b : \text{real-normed-field}\)
\textbf{shows} \(f\) differentiable \((at \ x \ within \ s) = \Rightarrow g\) differentiable \((at \ x \ within \ s) = \Rightarrow \)
\(g \ x \neq 0 = \Rightarrow (\lambda x. \text{inverse} \ (f \ x / g \ x))\) differentiable \((at \ x \ within \ s)\)
\textbf{unfolding} divide-inverse \textbf{by} simp

\textbf{lemma} differentiable-power \[\text{simp, derivative-intros}\]:
\textbf{fixes} \(f, g : \cdot a : \text{real-normed-vector} \Rightarrow \cdot b : \text{real-normed-field}\)
\textbf{shows} \(f\) differentiable \((at \ x \ within \ s) = \Rightarrow (\lambda x. f \ x ^ n)\) differentiable \((at \ x \ within \ s)\)
\textbf{unfolding} differentiable-def \textbf{by} \((\text{blast intro: has-derivative-power})\)

\textbf{lemma} differentiable-power-int \[\text{simp, derivative-intros}\]:
\textbf{fixes} \(f : \cdot a : \text{real-normed-vector} \Rightarrow \cdot b : \text{real-normed-field}\)
\textbf{shows} \(f\) differentiable \((at \ x \ within \ s) = \Rightarrow f \ x \neq 0 = \Rightarrow \)
\((\lambda x. \text{power-int} \ (f \ x) n)\) differentiable \((at \ x \ within \ s)\)
\textbf{unfolding} differentiable-def \textbf{by} \((\text{blast intro: has-derivative-power-int})\)

\textbf{lemma} differentiable-scaleR \[\text{simp, derivative-intros}\]:
\textbf{fixes} \(f : \cdot a : \text{real-normed-vector} \Rightarrow \cdot b : \text{real-normed-field}\)
\textbf{shows} \(f\) differentiable \((at \ x \ within \ s) = \Rightarrow g\) differentiable \((at \ x \ within \ s) = \Rightarrow \)
\((\lambda x. f \ x \ast \cdot R \ g \ x)\) differentiable \((at \ x \ within \ s)\)
\textbf{unfolding} differentiable-def \textbf{by} \((\text{blast intro: has-derivative-scaleR})\)

\textbf{lemma} has-derivative-imp-has-field-derivative:
\((f \ has-derivative D) \ F = \Rightarrow (\lambda x. x \ast D' = D \ x) = \Rightarrow (f \ has-field-derivative D') \ F\)
\textbf{unfolding} has-field-derivative-def \textbf{by} \((\text{rule has-derivative-eq-rhs[of f D]})) \textbf{simp-all add: fan-iff mult.commute}\n
\textbf{lemma} has-field-derivative-imp-has-derivative:
\((f \ has-field-derivative D) \ F = \Rightarrow (f \ has-derivative (\ast) D) \ F\)
by (simp add: has-field-derivative-def)

lemma DERIV-subset:
(f has-field-derivative f' ) (at x within s) \Rightarrow t \subseteq s \Rightarrow
(f has-field-derivative f' ) (at x within t)
by (simp add: has-field-derivative-def has-derivative-subset)

lemma has-field-derivative-at-within:
(f has-field-derivative f' ) (at x) \Rightarrow (f has-field-derivative f' ) (at x within s)
using DERIV-subset by blast

abbreviation (input)
DERIV :: (real-normed-field \Rightarrow real-normed-field \Rightarrow bool)
((DERIV (-) (-)) :: (\Rightarrow)) [1000, 1000, 60] 60
where DERIV f x := (f has-field-derivative D) (at x)

abbreviation has-real-derivative :: (real \Rightarrow real \Rightarrow real filter \Rightarrow bool)
(infix ('has-real-derivative) 50)
where (f has-real-derivative D) F := (f has-field-derivative D) F

lemma real-differentiable-def:
f differentiable at x within s \iff \exists D. (f has-real-derivative D) (at x within s)
proof safe
assume f differentiable at x within s
then obtain f' where \ast: (f has-derivative f' ) (at x within s)
  unfolding differentiable-def by auto
then obtain c where f' = (\ast c)
by (metis real-bounded-linear has-derivative-bounded-linear mult.commute fun-eq-iff)
with \ast show \exists D. (f has-real-derivative D) (at x within s)
  unfolding has-field-derivative-def by auto
qed (auto simp: differentiable-def has-field-derivative-def)

lemma real-differentiableE [elim?]:
assumes f: f differentiable (at x within s)
obtains df' where (f has-real-derivative df' ) (at x within s)
using assms by (auto simp: real-differentiable-def)

lemma has-field-derivative-iff:
(f has-field-derivative D) (at x within S) \iff
((\lambda y. (f y - f x) / (y - x)) \longrightarrow D) (at x within S)
proof
  have \((\lambda y. norm (f y - f x - D * (y - x)) / norm (y - x)) \longrightarrow 0) (at x within S)
  = ((\lambda y. (f y - f x) / (y - x) - D) \longrightarrow 0) (at x within S)
  by (smt (verit, best) Lim-cong-within divide-diff-eq-iff norm-divide right-minus-eq
  tendsto-norm-zero-iff)
  then show \?thesis
  by (simp add: has-field-derivative-def has-derivative-iff-norm bounded-linear-mult-right
  LIM-zero-iff)

qed

lemma DERIV-def: DERIV f x := D ←→ (λh. (f (x + h) − f x) / h) −→ 0 −→ D
unfolding field-has-derivative-at has-field-derivative-def has-field-derivative-iff ..

lemma has-field-derivative-unique:
assumes (f has-field-derivative f'1) (at x within A)
assumes (f has-field-derivative f'2) (at x within A)
assumes at x within A ≠ bot
shows f'1 = f'2
using assms unfolding has-field-derivative-iff using tendsto-unique by blast

due to Christian Pardillo Laursen, replacing a proper epsilon-delta horror

lemma field-derivative-lim-unique:
assumes f : (f has-field-derivative df) (at z)
and s: s −−−−→ 0 ∧ n. s n ≠ 0
and a: (λn. (f (z + s n) − f z) / s n) −−−−→ a
shows df = a
proof −
have ((λk. (f (z + k) − f z) / k) −−−−→ df) (at 0)
  using f by (simp add: DERIV-def)
with s have ((λn. (f (z + s n) − f z) / s n) −−−−→ df)
  by (simp flip: LIMSEQ-SEQ-conv)
then show thesis
  using a by (rule LIMSEQ-unique)
qed

lemma mult-commute-abs: (λx. x * c) = (∗) c
for c :: 'a::ab-semigroup-mult
by (simp add: fun-iff mult.commute)

lemma DERIV-compose-FDERIV:
fixes f::real⇒real
assumes DERIV f (g x) := f'
assumes (g has-derivative g') (at x within s)
shows ((λx. f (g x)) has-derivative (λx. g' x ∗ f')) (at x within s)
using assms has-derivative-compose[of g g' x s f (∗) f']
by (auto simp: has-field-derivative-def ac-simps)

111.6 Vector derivative

It’s for real derivatives only, and not obviously generalisable to field derivatives

lemma has-real-derivative-iff-has-vector-derivative:
(f has-real-derivative y) F ←→ (f has-vector-derivative y) F
unfolding has-vector-derivative-def has-field-derivative-def real-scaleR-def mult-commute-efs
..
lemma has-field-derivative-subset:
(f has-field-derivative y) (at x within s) \implies t \subseteq s \implies
(f has-field-derivative y) (at x within t)  
by (fact DERIV-subset)

lemma has-vector-derivative-const[simp, derivative-intros]:
(\lambda x. c) has-vector-derivative 0  
by (auto simp: has-vector-derivative-def)

lemma has-vector-derivative-id[simp, derivative-intros]:
(\lambda x. x) has-vector-derivative 1  
by (auto simp: has-vector-derivative-def)

lemma has-vector-derivative-minus[derivative-intros]:
(f has-vector-derivative f') net = \implies
(\lambda x. -f x) has-vector-derivative (-f') net  
by (auto simp: has-vector-derivative-def)

lemma has-vector-derivative-add[derivative-intros]:
(f has-vector-derivative f') net = \implies
(g has-vector-derivative g') net = \implies
(\lambda x. f x + g x) has-vector-derivative (f' + g') net  
by (auto simp: has-vector-derivative-def scaleR-right-distrib)

lemma has-vector-derivative-sum[derivative-intros]:
(\forall i. i \in I \implies f i has-vector-derivative f'_i) net = \implies
(\lambda x. \sum_{i \in I} f i x) has-vector-derivative (\sum_{i \in I} f'_i i) net  
by (auto simp: has-vector-derivative-def fun-eq-iff scaleR-sum-right intro: derivative-eq-intros)

lemma has-vector-derivative-diff[derivative-intros]:
(f has-vector-derivative f') net = \implies
(g has-vector-derivative g') net = \implies
(\lambda x. f x - g x) has-vector-derivative (f' - g') net  
by (auto simp: has-vector-derivative-def scaleR-diff-right)

lemma has-vector-derivative-add-const:
((\lambda t. g t + z) has-vector-derivative f') net = ((\lambda t. g t) has-vector-derivative f') net  
apply (intro iff) 
apply (force dest: has-vector-derivative-diff [where g = \lambda t. z, OF - has-vector-derivative-const]) 
apply (force dest: has-vector-derivative-add [OF - has-vector-derivative-const]) 
done

lemma has-vector-derivative-diff-const:
((\lambda t. g t - z) has-vector-derivative f') net = ((\lambda t. g t) has-vector-derivative f') net  
using has-vector-derivative-add-const [where z = -z] 
by simp

lemma (in bounded-linear) has-vector-derivative:
assumes (g has-vector-derivative g') F
shows \((\lambda x. f (g x)) \text{ has-vector-derivative } f') F\)
using \text{has-derivative}[OF assms[unfolded has-vector-derivative-def]]
by (simp add: has-vector-derivative-def scaleR)

lemma (in bounded-bilinear) \text{has-vector-derivative}:
assumes \((f \text{ has-vector-derivative } f') (at x \text{ within } s)\)
and \((g \text{ has-vector-derivative } g') (at x \text{ within } s)\)
shows \((\lambda x. f x ** g x) \text{ has-vector-derivative } (f x ** g' + f' ** g x) (at x \text{ within } s)\)
using \text{FDERIV}[OF assms[1-2][unfolded has-vector-derivative-def]]
by (simp add: has-vector-derivative-def scaleR-right scaleR-left scaleR-right-distrib)

lemma \text{has-vector-derivative-scaleR}[derivative-intros]:
\((f \text{ has-field-derivative } f') (at x \text{ within } s) \Rightarrow \text{ has-vector-derivative } (g \text{ has-vector-derivative } g') (at x \text{ within } s) \Rightarrow \)
\((\lambda x. f x * R g x) \text{ has-vector-derivative } (f x * R g' + f' * R g x) (at x \text{ within } s)\)
unfolding \text{has-real-derivative-iff-has-vector-derivative}
by (rule bounded-bilinear.has-vector-derivative[OF bounded-bilinear-scaleR])

lemma \text{has-vector-derivative-mult}[derivative-intros]:
\((f \text{ has-vector-derivative } f') (at x \text{ within } s) \Rightarrow \text{ has-vector-derivative } (g \text{ has-vector-derivative } g') (at x \text{ within } s) \Rightarrow \)
\((\lambda x. f x * g x) \text{ has-vector-derivative } (f x * g' + f' * g x) (at x \text{ within } s)\)
for \(f g :: \text{real} \Rightarrow \text{a::real-normed-algebra}\)
by (rule bounded-bilinear.has-vector-derivative[OF bounded-bilinear-mult])

lemma \text{has-vector-derivative-of-real}[derivative-intros]:
\((f \text{ has-field-derivative } D) F \Rightarrow \((\lambda x. \text{of-real } (f x)) \text{ has-vector-derivative } (\text{of-real } D)\) F\)
by (rule bounded-linear.has-vector-derivative[OF bounded-linear-of-real])
(simp add: has-real-derivative-iff-has-vector-derivative)

lemma \text{has-vector-derivative-real-field}:
\((f \text{ has-field-derivative } f') (at (\text{of-real } a)) \Rightarrow ((\lambda x. f \text{ (of-real } x)) \text{ has-vector-derivative } f') (at a \text{ within } s)\)
using \text{has-derivative-compose}[of \text{of-real of-real } a - f (\*) f']
by (simp add: scaleR-conv-of-real ac-simps has-vector-derivative-def has-field-derivative-def)

lemma \text{has-vector-derivative-continuous}:
\((f \text{ has-vector-derivative } D) (at x \text{ within } s) \Rightarrow \text{continuous } (at x \text{ within } s) f\)
by (auto intro: has-derivative-continuous simp: has-vector-derivative-def)

lemma \text{continuous-on-vector-derivative}:
\((\lambda x. x \in S \Rightarrow (f \text{ has-vector-derivative } f' x) (at x \text{ within } S)) \Rightarrow \text{continuous-on } S f\)
by (auto simp: continuous-on-iff-continuous-within intro!: has-vector-derivative-continuous)

lemma \text{has-vector-derivative-mult-right}[derivative-intros]:
\(\text{fixes } a :: \text{a::real-normed-algebra}\)
shows \((f + g)'(x) = f'(x) + g'(x)\) for all \(x\) in \(S\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-minus[derivative-intros]:
\[(f - g)'(x) = f'(x) - g'(x)\] for all \(x\) in \(S\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-add[derivative-intros]:
\[(f + g)'(x) = f'(x) + g'(x)\] for all \(x\) in \(S\) 
by (rule has-derivative-imp-has-field-derivative)

corollary DERIV-add:
\((f + g)'(x) = f'(x) + g'(x)\) for all \(x\) in \(S\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-minus[derivative-intros]:
\[(-f)'(x) = -f'(x)\] for all \(x\) in \(S\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-div[derivative-intros]:
\[\left(\frac{f}{g}\right)'(x) = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}\] for all \(x\) in \(S\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-mult[derivative-intros]:
\[(fg)'(x) = f'(x)g(x) + f(x)g'(x)\] for all \(x\) in \(S\) 
by (rule has-derivative-imp-has-field-derivative)

corollary DERIV-mult:
\((fg)'(x) = f'(x)g(x) + f(x)g'(x)\) for all \(x\) in \(S\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-sub[derivative-intros]:
\[(f - g)'(x) = f'(x) - g'(x)\] for all \(x\) in \(S\) 
by (rule has-derivative-imp-has-field-derivative)

lemma DERIV-D: \(\lambda h. (f(x + h) - f(x))/h\to D\) 
by (rule has-derivative-imp-has-field-derivative)

lemma has-field-derivativeD:
\[(f + D)(x) = f(x)\] for all \(x\) in \(S\) 
by (rule has-derivative-imp-has-field-derivative)

lemma DERIV-const [simp, derivative-intros]: \(\lambda x. k\) has-field-derivative 0 
by (rule has-derivative-imp-has-field-derivative)

lemma DERIV-ident [simp, derivative-intros]: \(\lambda x. x\) has-field-derivative 1 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-add[derivative-intros]:
\[\lambda z. f(z) + g(z)\] has-field-derivative \(\lambda z. f(z) + g(z)\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-sub[derivative-intros]:
\[\lambda z. f(z) - g(z)\] has-field-derivative \(\lambda z. f(z) - g(z)\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-mult[derivative-intros]:
\[\lambda z. f(z)g(z)\] has-field-derivative \(\lambda z. f(z)g(z)\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-div[derivative-intros]:
\[\lambda z. f(z)/g(z)\] has-field-derivative \(\lambda z. f(z)/g(z)\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-sub[derivative-intros]:
\[\lambda z. f(z) - g(z)\] has-field-derivative \(\lambda z. f(z) - g(z)\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-mult[derivative-intros]:
\[\lambda z. f(z)g(z)\] has-field-derivative \(\lambda z. f(z)g(z)\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-div[derivative-intros]:
\[\lambda z. f(z)/g(z)\] has-field-derivative \(\lambda z. f(z)/g(z)\) 
by (rule has-derivative-imp-has-field-derivative)

lemma field-differentiable-sub[derivative-intros]:
\[\lambda z. f(z) - g(z)\] has-field-derivative \(\lambda z. f(z) - g(z)\) 
by (rule has-derivative-imp-has-field-derivative)
corollary DERIV-minus:
(f has-field-derivative D) (at x within s) \implies
((\lambda x. - f x) has-field-derivative -D) (at x within s)
by (rule field-differentiable-minus)

lemma field-differentiable-diff[derivative-intros]:
(f has-field-derivative \ f') F \implies
(g has-field-derivative \ g') F \implies
((\lambda z. f z - g z) has-field-derivative \ f' - \ g') F
by (simp only: diff-conv-add-uminus field-differentiable-add field-differentiable-minus)

corollary DERIV-diff:
(f has-field-derivative D) (at x within s) \implies
(g has-field-derivative E) (at x within s) \implies
((\lambda x. f x - g x) has-field-derivative D - E) (at x within s)
by (rule field-differentiable-diff)

lemma DERIV-continuous: (f has-field-derivative D) (at x within s) \implies
continuous (at x within s) f
by (drule has-derivative-continuous[OF has-field-derivative-imp-has-derivative])
simp

lemma DERIV-isCont: DERIV f x := D \implies
isCont f x
by (rule DERIV-continuous)

lemma DERIV-atLeastAtMost-imp-continuous-on:
assumes \(\forall x. \[a \leq x; x \leq b\] \implies \exists y. \text{DERIV} f x := y\)
shows continuous-on \{a..b\} f
by (meson DERIV-isCont assms atLeastAtMost-iff continuous-at-imp-continuous-at-within
continuous-on-eq-continuous-within)

lemma DERIV-continuous-on:
(\(\forall x. x \in s \implies (f has-field-derivative (D x)) (at x within s)\)) \implies
continuous-on s f
unfolding continuous-on-eq-continuous-within
by (intro continuous-at-imp-continuous-on ballI DERIV-continuous)

lemma DERIV-mult':
(f has-field-derivative D) (at x within s) \implies
(g has-field-derivative E) (at x within s) \implies
((\lambda x. f x * g x) has-field-derivative f x * E + D * g x) (at x within s)
by (rule has-derivative-imp-has-field-derivative[OF has-field-derivative-mul])
(auto simp: field-simps mult-commute-abs dest: has-field-derivative-imp-has-derivative)

lemma DERIV-mult[derivative-intros]:
(f has-field-derivative Da) (at x within s) \implies
(g has-field-derivative Db) (at x within s) \implies
((\lambda x. f x * g x) has-field-derivative Da * g x + Db * f x) (at x within s)
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-mul])
THEORY "Deriv"

(auto simp: field-simps dest: has-field-derivative-imp-has-derivative)

Derivative of linear multiplication

lemma DERIV-cmult:
  \((f \text{ has-field-derivative } D) \ (at \ x \ within \ s) \implies
  ((\lambda x. \ c \ast f \ x) \text{ has-field-derivative } c \ast D) \ (at \ x \ within \ s)\)
by (drule DERIV-mult'[OF DERIV-const]) simp

lemma DERIV-cmult-right:
  \((f \text{ has-field-derivative } D) \ (at \ x \ within \ s) \implies
  ((\lambda x. \ f \ x \ast c) \text{ has-field-derivative } D \ast c) \ (at \ x \ within \ s)\)
using DERIV-cmult by (auto simp add: ac-simps)

lemma DERIV-cdivide:
  \((f \text{ has-field-derivative } D) \ (at \ x \ within \ s) \implies
  ((\lambda x. \ f \ x / c) \text{ has-field-derivative } D / c) \ (at \ x \ within \ s)\)
using DERIV-cmult-right[of f D x s 1 / c] by simp

lemma DERIV-unique: \(\exists \leq 1 \ D. \ \text{DERIV } f \ x : D \implies \text{DERIV } f \ x : E \implies D = E\)
unfold DERIV-def by (rule LIM-unique)

lemma DERIV-Uniq: \(\exists \leq 1 \ D. \ \text{DERIV } f \ x : D\)
by (simp add: DERIV-unique Uniq-def)

lemma DERIV-sum[derivative-intros]:
  \(\bigwedge n. \ n \in S \implies ((\lambda x. \ f \ x \ n) \text{ has-field-derivative } (f' \ x \ n)) \ F) \implies
  ((\lambda x. \ \text{sum } (f \ x) \ S) \text{ has-field-derivative } \text{sum } (f' \ x) \ S) \ F\)
by (rule has-derivative-imp-has-field-derivative [OF has-derivative-sum])
(auto simp: sum-distrib-left mult-commute-abs dest: has-field-derivative-imp-has-derivative)

lemma DERIV-inverse[derivative-intros]:
  assumes \((f \text{ has-field-derivative } D) \ (at \ x \ within \ s)\)
  and \(f x \neq 0\)
  shows \(((\lambda x. \ \text{inverse } (f \ x)) \text{ has-field-derivative } - (\text{inverse } (f \ x) \ast D \ast \text{inverse } (f \ x)))\)
  \(\ (at \ x \ within \ s)\)
proof -
have \((f \text{ has-derivative } (\lambda x. \ x \ast D)) = (f \text{ has-derivative } (\ast) \ D)\)
  by (rule arg-cong[of \lambda x. \ x \ast D]) (simp add: fun-comp)
with \(\text{assms}\) have \((f \text{ has-derivative } (\lambda x. \ x \ast D)) \ (at \ x \ within \ s)\)
  by (auto dest!: has-field-derivative-imp-has-field-derivative)
then show \("\text{thesis}\ using \(f x \neq 0)\",
  by (auto intro: has-field-derivative-imp-has-field-derivative has-field-derivative-inverse)
qed

Power of \(-1\)
lemma DERIV-inverse:
x \neq 0 \implies ((\lambda x. \inverse(x)) \text{ has-field-derivative } - (\inverse x ^ \ Suc (Suc 0))) \ (at x \ within s)
by (drule DERIV-inverse' \ [OF DERIV-ident]) simp

Derivative of inverse

lemma DERIV-inverse-fun:
(\ (f \ has-field-derivative \ d) \ (at x \ within s) \ \Rightarrow \ f x \neq 0 \ \Rightarrow
(\ (\lambda x. \ inverse (f x)) \ has-field-derivative \ (\ - (d * inverse(f x ^ Suc (Suc 0)))))
(at x \ within s)
by (drule (1) DERIV-inverse') (simp add: ac-simps nonzero-inverse-mult-distrib)

Derivative of quotient

lemma DERIV-divide[derivative-intros]:
(\ (f \ has-field-derivative \ D) \ (at x \ within s) \ \Rightarrow
(\ (g \ has-field-derivative \ E) \ (at x \ within s) \ \Rightarrow \ g x \neq 0 \ \Rightarrow
((\lambda x. \ f x / g x) \ has-field-derivative \ (D * g x - f x * E) / (g x * g x)) \ (at x \ within s))
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-divide])
(auto dest: has-field-derivative-imp-has-derivative simp: field-simps)

lemma DERIV-quotient:
(\ (f \ has-field-derivative \ D) \ (at x \ within s) \ \Rightarrow
(\ (g \ has-field-derivative \ E) \ (at x \ within s) \ \Rightarrow \ g x \neq 0 \ \Rightarrow
((\lambda y. \ f y / g y) \ has-field-derivative \ (D * g x - (e * f x)) / (g x ^ Suc (Suc 0)))
(at x \ within s)
by (drule (2) DERIV-divide) (simp add: mult.commute)

lemma DERIV-power-Suc:
(\ (f \ has-field-derivative \ D) \ (at x \ within s) \ \Rightarrow
((\lambda x. \ f x ^ Suc n) \ has-field-derivative \ (1 + of-nat n) * (D * f x ^ n)) \ (at x \ within s))
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-power])
(auto simp: has-field-derivative-def)

lemma DERIV-power[derivative-intros]:
(\ (f \ has-field-derivative \ D) \ (at x \ within s) \ \Rightarrow
((\lambda x. \ f x ^ n) \ has-field-derivative \ of-nat n * (D * f x ^ (n - Suc 0))) \ (at x \ within s))
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-power])
(auto simp: has-field-derivative-def)

lemma DERIV-pow: ((\lambda x. \ x ^ n) \ has-field-derivative \ real n * (x ^ (n - Suc 0)))
(at x \ within s)
using DERIV-power [OF DERIV-ident] by simp

lemma DERIV-power-int [derivative-intros]:
  assumes [derivative-intros]: (\ (f \ has-field-derivative \ d) \ (at x \ within s) \ and [simp]: f x \neq 0
THEORY “Deriv”  

shows  \((\lambda x. \text{power-int} (f x) n) \text{ has-field-derivative} \)  
\((\text{af-int} n * \text{power-int} (f x) (n - 1) * d)) \ (at x within s)  

proof  \((\text{cases n rule: int-cases4})\)  
\(\text{case (nonneg n)}\)  
\(\text{thus ?thesis by (auto intro!: derivative-eq-intros simp: field-simps power-int-diff simp flip: power-Suc power-Suc2 power-add)}\)  
next  
\(\text{case (neg n)}\)  
\(\text{thus ?thesis by (auto intro!: derivative-eq-intros simp: field-simps power-int-minus simp flip: power-Suc power-Suc2 power-add)}\)  
qed  

lemma DERIV-chain': (\(f \text{ has-field-derivative } D\) \ (at x within s) \implies \text{DERIV} g \ (f x) :> E)  
\(\implies \((\lambda x. g (f x)) \text{ has-field-derivative } E * D\) \ (at x within s)\)  
using has-derivative-compose[of f (*) D x s g (*) E]  
by (simp only: has-field-derivative-def mult-commute-abs ac-simps)  
corollary DERIV-chain2: DERIV f \ (g x) :> Da \implies \((g \text{ has-field-derivative } Db) \ (at x within s)\)  
by (rule DERIV-chain')  

Standard version  

lemma DERIV-chain:  
\(\text{DERIV} f \ (g x) :> Da \implies \ (g \text{ has-field-derivative } Db) \ (at x within s) \implies \)  
\(\ (f \circ g \text{ has-field-derivative } Da * Db) \ (at x within s)\)  
by (drule (1) DERIV-chain', simp add: o-def mult.commute)  

lemma DERIV-image-chain:  
\(f \text{ has-field-derivative } Da) \ (at (g x) \text{ within } (g' s)) \implies \)  
\(f \circ g \text{ has-field-derivative } Da * Db) \ (at x within s) \implies \)  
using has-derivative-in-compose [of g (*) Db x s f (*) Da ]  
by (simp add: has-field-derivative-def o-def mult-commute-abs ac-simps)  

lemma DERIV-chain-s:  
assumes \((\forall x. x \in s \implies \text{DERIV} g x :> g'(x))\)  
and \(\text{DERIV} f x :> f'\)  
and \(f x \in s\)  
shows \(\text{DERIV} (\lambda x. g(f x)) \ x :> f' * g'(f x)\)  
by (metis (full-types) DERIV-chain' mult.commute assms)  

lemma DERIV-chain3:  
assumes \((\forall x. \text{DERIV} g x :> g'(x))\)
and DERIV f x := f′
shows DERIV (λx. g(f x)) x := f′ ∗ g′(f x)
by (metis UNIV-I DERIV-chain-s [of UNIV] assms)

Alternative definition for differentiability

lemma DERIV-LIM-iff:
  fixes f :: 'a::{real-normed-vector,inverse} ⇒ 'a
  shows ((λh. (f (a + h) − f a) / h) −0→ D) = ((λx. (f x − f a) / (x − a)) −a→ D) (is ?lhs = ?rhs)
proof
  assume ?lhs
  then have ((λx. (f (a + (x + − a)) − f a) / (x + − a)) −0 − a→ D)
    by (rule LIM-offset)
  then show ?rhs
    by simp
next
  assume ?rhs
  then have ((λx. (f (x+a) − f a) / ((x+a) − a)) −a−a→ D)
    by (rule LIM-offset)
  then show ?lhs
    by (simp add: add.commute)
qed

lemma has-field-derivative-cong-ev:
  assumes x = y and ∗: eventually (λx. x ∈ S → f x = g x) (nhds x)
  and u = v S = t x ∈ S
  shows (f has-field-derivative u) (at x within S) = (g has-field-derivative v) (at y within t)
unfolding has-field-derivative-iff
proof (rule filterlim-cong)
  from assms have f y = g y
    by (auto simp: eventually-nhds)
  with ∗ show ∀F z in at x within S. (f z − f x) / (z − x) = (g z − g y) / (z − y)
    unfolding eventually-at-filter
    by eventually-elim (auto simp: assms f y = g y)
qed (simp-all add: assms)

lemma has-field-derivative-cong-eventually:
  assumes eventually (λx. f x = g x) (at x within S) f x = g x
  shows (f has-field-derivative u) (at x within S) = (g has-field-derivative u) (at x within S)
unfolding has-field-derivative-iff
proof (rule tendsto-cong)
  show ∀F y in at x within S. (f y − f x) / (y − x) = (g y − g x) / (y − x)
    using assms by (auto elim: eventually-mono)
qed
lemma DERIV-cong-ev:
\[ x = y \implies \text{eventually } (\lambda x. f x = g x) \text{ (nhds } x) \implies u = v \implies \]
\[ \text{DERIV } f x :> u \longleftrightarrow \text{DERIV } g y :> v \]
by (rule has-field-derivative-cong-ev) simp-all

lemma DERIV-mirror: (\text{DERIV } f (x :>) y) \longleftrightarrow (\text{DERIV } (\lambda x. f (\neg x)) x :> \neg y) 
for \( f :: \text{real} \implies \text{real} \) and \( x y :: \text{real} \)
by (simp add: DERIV-def filterlim-at-split filterlim-at-left-to-right 
tendsto-minus-cancel-left field-simps conj-commute)

lemma DERIV-shift:
\[(f \text{ has-field-derivative } y) \ (at \ (x + z)) = ((\lambda x. f (x + z)) \text{ has-field-derivative } y)\ (at \ x)\]
by (simp add: DERIV-def field-simps)

lemma DERIV-at-within-shift-lemma: 
assumes (\text{f has-field-derivative } y) \ (at \ (z + x) \text{ within } (+) \ z ' S) 
shows (\text{f o (+)z has-field-derivative } y) \ (at \ x \text{ within } S) 
proof - 
have ((+)z \text{ has-field-derivative } 1) \ (at \ x \text{ within } S) 
by (rule derivative-eq-intros | simp)+
with assms DERIV-image-chain show ?thesis 
by (metis mult.right-neutral)
qed

lemma DERIV-at-within-shift: 
\[(f \text{ has-field-derivative } y) \ (at \ (x + z) \text{ within } (+) \ z ' S) \longleftrightarrow \]
\[((\lambda x. f (x + z)) \text{ has-field-derivative } y) \ (at \ x \text{ within } S)\] (is ?lhs = ?rhs)
proof 
assume ?lhs then show ?rhs 
using DERIV-at-within-shift-lemma unfolding o-def by blast
next 
have [simp]: (\lambda x. x - z) ' (+) z ' S = S 
by force
assume R: ?rhs 
have (f o (+) z o (+) (\neg z) \text{ has-field-derivative } y) \ (at \ (z + x) \text{ within } (+) \ z ' S) 
by (rule DERIV-at-within-shift-lemma) (use R in simp add: o-def)
then show ?lhs 
by (simp add: o-def)
qed

lemma floor-has-real-derivative: 
\text{fixes } f :: \text{real} \implies 'a::{floor-ceiling,order-topology} 
\text{assumes isCont } f x 
and f x \notin \mathbb{Z} 
\text{shows } ((\lambda x. \text{floor } (f x)) \text{ has-real-derivative } 0) \ (at \ x) 
proof (subst DERIV-cong-ev[OF refl - refl])
show ((\lambda x. \text{floor } (f x)) \text{ has-real-derivative } 0) \ (at \ x)
by simp
have ∀ y in at x. [f y] = [f x]
  by (rule eventually-floor-eq[OF assms unfolded continuous-at])
then show ∀ y in nhds x. real-of-int [f y] = real-of-int [f x]
  unfolding eventually-at-filter
  by eventually-elim auto
qed

lemmas has-derivative-floor[derivative-intros] =
  floor-has-real-derivative[THEN DERIV-compose-FDERIV]

lemma continuous-floor:
  fixes x::real
  shows x /∈ ℤ ⇒ continuous (at x) (real-of-int ◦ floor)
  using floor-has-real-derivative[where f=id]
  by (auto simp add:o-def has-field-derivative-def intro:has-derivative-continuous)

lemma continuous-frac:
  fixes x::real
  assumes x /∈ ℤ
  shows continuous (at x) frac
  proof
    have isCont (λx. real-of-int ⌊x⌋) x
      using continuous-floor[OF assms]
      by (simp add:o-def)
    then show ∗: continuous (at x) (λx. x - real-of-int ⌊x⌋)
      by (intro continuous-intros)
    moreover have ∀ x in nhds x. frac x = x - real-of-int ⌊x⌋
      by (simp add:frac-def)
    ultimately show ?thesis
      by (simp add:LIM-imp-LIM frac-def isCont-def)
  qed

Caratheodory formulation of derivative at a point

lemma CARAT-DERIV:
  (DERIV f x :> l) ←→ (∃ g. (∀z. f z - f x = g z * (z - x)) ∧ isCont g x ∧ g x = l)
  (is ?lhs = ?rhs)
  proof
    assume ?lhs
    show ∃ g. (∀z. f z - f x = g z * (z - x)) ∧ isCont g x ∧ g x = l
      proof (intro ezl conjI)
        let ?g = (λz. if z = x then l else (f z - f x) / (z - x))
        show ∀z. f z - f x = ?g z * (z - x)
          by simp
        show isCont ?g x
          using (?lhs) by (simp add:isCont-iff DERIV-def cong:LIM-equal [rule-format])
        show ?g x = l
          by simp
      qed
  qed
next
  assume ?rhs
  then show ?lhs
    by (auto simp add: isCont-iff DERIV-def cong: LIM-cong)
qed

111.8 Local extrema

If \( (0::'a) < f' x \) then \( x \) is Locally Strictly Increasing At The Right.

lemma has-real-derivative-pos-inc-right:
  fixes \( f :: \text{real} \Rightarrow \text{real} \)
  assumes \( \text{der}: (f \text{ has-real-derivative } l) \ (\text{at } x \text{ within } S) \)
  and \( l: 0 < l \)
  shows \( \exists d > 0. \ \forall h > 0. \ x + h \in S \rightarrow h < d \rightarrow f x < f (x + h) \)
  using assms
proof
  from \( \text{der} \ [\text{THEN has-field-derivativeD, THEN tendstoD}, \ OF \ l, \ unfolded \ eventually-at] \)
  obtain \( s \) where \( s: 0 < s \)
    and \( \text{all}: \ \exists x. \ xa \in S \Rightarrow xa \neq x \wedge \text{dist } xa x < s \rightarrow \left| (f xa - f x) / (xa - x) - l \right| < l \)
    by (auto simp: dist-real-def pos-less-divide-eq split: if-split_asm)
  then show \( \exists d > 0. \ \forall h > 0. \ h < d \rightarrow f x < f (x + h) \)
    using assms
thesis
proof (intro exI conjI strip)
  show \( 0 < s \) by (rule s)
next
  fix \( h :: \text{real} \)
  assume \( 0 < h \ h < s x + h \in S \)
  with \( \text{all } [\text{of } x + h] \) show \( f x < f (x + h) \)
proof
  (simp add: abs_if dist-real-def pos-less-divide-eq split: if-split_asm)
  assume \( = (f (x + h) - f x) / h < l \) and \( h: 0 < h \)
  with \( l \) have \( 0 < f (x + h) - f x) / h \)
    by arith
  then show \( f x < f (x + h) \)
  by (simp add: pos-less-divide-eq h)
qed
qed

lemma DERIV-pos-inc-right:
  fixes \( f :: \text{real} \Rightarrow \text{real} \)
  assumes \( \text{der}: \text{DERIV } f x :> l \)
  and \( l: 0 < l \)
  shows \( \exists d > 0. \ \forall h > 0. \ h < d \rightarrow f x < f (x + h) \)
  using has-real-derivative-pos-inc-right[OF assms]
by auto

lemma has-real-derivative-neg-dec-left:
  fixes \( f :: \text{real} \Rightarrow \text{real} \)

assumes \( \text{der} : (f \text{ has-real-derivative } l) \text{ (at } x \text{ within } S) \)
and \( l < 0 \)
shows \( \exists d > 0. \forall h > 0. x - h \in S \implies h < d \implies f x < f (x - h) \)

proof —
from \( l < 0 \) have \( l : -l > 0 \)
by simp
from \( \text{der} \) [\( \text{THEN field-derivativeD, THEN tendstoD, OF } l, \text{ unfolded eventually-at} \)]
obtain \( s \) where \( s : 0 < s \)
and all: \( \forall xa. xa \in S \implies xa \neq x \land \text{dist } xa \ x < s \implies |(f xa - f x)/(xa - x)| < -l \)
by (auto simp: dist-real-def)
then show \( ?\text{thesis} \)
proof (intro exI conjI strip)
show \( 0 < s \) by (rule s)
next
fix \( h :: \text{real} \)
assume \( 0 < h \) \( h < s \) \( x - h \in S \)
with all [of \( x - h \)] show \( f x < f (x - h) \)
proof (simp add: abs-if pos-less-divide-eq dist-real-def split_if_split_asm)
assume \( -((f (x - h) - f x)/h) < l \) and \( l : 0 < h \)
with \( l \) have \( 0 < (f (x - h) - f x)/h \)
by arith
then show \( f x < f (x - h) \)
by (simp add: pos-less-divide-eq h)
qed
qed

lemma \( \text{DERIV-neg-dec-left} \):
fixes \( f :: \text{real } \Rightarrow \text{real} \)
assumes \( \text{der} : \text{DERIV } f x > l \)
and \( l : l < 0 \)
shows \( \exists d > 0. \forall h > 0. h < d \implies f x < f (x - h) \)
using \( \text{has-real-derivative-neg-dec-left} \) [OF assms]
by auto

lemma \( \text{has-real-derivative-pos-inc-left} \):
fixes \( f :: \text{real } \Rightarrow \text{real} \)
shows \( (f \text{ has-real-derivative } l) \text{ (at } x \text{ within } S) \implies 0 < l \implies \exists d > 0. \forall h > 0. x - h \in S \implies h < d \implies f (x - h) < f x \)
by (rule \( \text{has-real-derivative-neg-dec-left} \) [of \( \lambda x. - f x - l x S \), simplified])
(auto simp: \( \text{DERIV-minus} \))

lemma \( \text{DERIV-pos-inc-left} \):
fixes \( f :: \text{real } \Rightarrow \text{real} \)
shows \( \text{DERIV } f x > l \implies 0 < l \implies \exists d > 0. \forall h > 0. h < d \implies f (x - h) < f x \)
using \( \text{has-real-derivative-pos-inc-left} \)
by blast

lemma has-real-derivative-neg-dec-right:
  fixes $f : \mathbb{R} \rightarrow \mathbb{R}$
  shows $(f \text{ has-real-derivative } l) \ (\text{at } x \text{ within } S) \implies l < 0 \implies$
  \exists d > 0. \forall h > 0. x + h \in S \rightarrow h < d \rightarrow f(x + h) > f(x)
  by (rule has-real-derivative-pos-inc-right [of $\lambda x. - f x - l x S$, simplified])
  (auto simp add: DERIV-minus)

lemma DERIV-neg-dec-right:
  fixes $f : \mathbb{R} \rightarrow \mathbb{R}$
  shows DERIV $f x : l \implies l < 0 \implies \exists d > 0. \forall h > 0. h < d \rightarrow f(x + h) > f(x)$
  using has-real-derivative-neg-dec-right by blast

lemma DERIV-local-max:
  fixes $f : \mathbb{R} \rightarrow \mathbb{R}$
  assumes $\text{der: } \text{DERIV } f x : l$
  and $d : 0 < d$
  and $\text{le: } \forall y. |x - y| < d \rightarrow f y \leq f x$
  shows $l = 0$
  proof (cases rule: linorder-cases [of l 0])
    case equal
    then show ?thesis .
    next
    case less
    from DERIV-neg-dec-left [OF $\text{der less}$]
    obtain $d'$ where $d' : 0 < d'$ and $\text{lt: } \forall h > 0. h < d' \rightarrow f(x) < f(x + h)$
      by blast
    obtain $e$ where $0 < e \land e < d \land e < d'$
      using field-lbound-gt-zero [OF $d d'$] ..
    with $\text{lt le: } \text{THEN spec [where x=x-e]}$ show ?thesis
      by (auto simp add: abs-if)
    next
    case greater
    from DERIV-pos-inc-right [OF $\text{der greater}$]
    obtain $d'$ where $d' : 0 < d'$ and $\text{lt: } \forall h > 0. h < d' \rightarrow f(x + h) < f(x)$
      by blast
    obtain $e$ where $0 < e \land e < d \land e < d'$
      using field-lbound-gt-zero [OF $d d'$] ..
    with $\text{lt le: } \text{THEN spec [where x=x+e]}$ show ?thesis
      by (auto simp add: abs-if)
  qed

Similar theorem for a local minimum

lemma DERIV-local-min:
  fixes $f : \mathbb{R} \rightarrow \mathbb{R}$
  shows $\text{DERIV } f x : l \implies 0 < d \implies \forall y. |x - y| < d \rightarrow f y \leq f x \implies l = 0$
  by (drule DERIV-minus [THEN DERIV-local-max]) auto
In particular, if a function is locally flat

**lemma** DERIV-local-const:

fixes \( f : \text{real} \Rightarrow \text{real} \)

shows \( \text{DERIV } f \ x : l \Rightarrow 0 < d \Rightarrow \forall y. |x - y| < d \Rightarrow f \ x = f \ y \Rightarrow l = 0 \)

by (auto dest: DERIV-local-max)

111.9 Rolle’s Theorem

**Lemma** about introducing open ball in open interval

**lemma** lemma-interval-lt:

fixes \( a \ b \ x : \text{real} \)

assumes \( a < x \ x < b \)

shows \( \exists d. 0 < d \land (\forall y. |x - y| < d \Rightarrow a < y \land y < b) \)

using linorder-linear [of \( x - a \ b - x \)]

**proof**

assume \( x - a \leq b - x \)

with assms show ?thesis by (rule-tac \( x\) = \( x - a \) in exI) auto

next

assume \( b - x \leq x - a \)

with assms show ?thesis by (rule-tac \( x\) = \( b - x \) in exI) auto

qed

**lemma** lemma-interval:

\( a < x \Rightarrow x < b \Rightarrow \exists d. 0 < d \land (\forall y. |x - y| < d \Rightarrow a \leq y \land y \leq b) \)

for \( a \ b \ x : \text{real} \)

by (force dest: lemma-interval-lt)

Rolle’s Theorem. If \( f \) is defined and continuous on the closed interval \([a,b]\)
and differentiable on the open interval \((a,b)\), and \( f \ a = f \ b \), then there exists \( x0 \in (a,b) \) such that \( f' \ x0 = (0::'a) \)

**theorem** Rolle-deriv:

fixes \( f : \text{real} \Rightarrow \text{real} \)

assumes \( a < b \)

and fab: \( f \ a = f \ b \)

and contf: continuous-on \{a..b\} \( f \)

and derf: \( \forall x. [a < x; x < b] \Rightarrow (f \text{-derivative } f' \ x) \) (at \( x \))

shows \( \exists z. a < z \land z < b \land f' \ z = (\lambda x. 0) \)

**proof**

have le: \( a \leq b \)

using \( \langle a < b \rangle \) by simp

have \( (a + b) / 2 \in \{a..b\} \)

using assms(1) by auto

then have *: \( \{a..b\} \neq \{\} \)

by auto

obtain \( x \) where \( x\text{-max}: \forall z. a \leq z \land z \leq b \Rightarrow f \ z \leq f \ x \) and \( a \leq x x \leq b \)

using continuous-attains-sup[OF compact-Icc * contf]
by (meson atLeastAtMost-iff)

obtain \( x' \) where \( x' \text{-}\min: \forall z. a \leq z \land z \leq b \rightarrow f x' \leq f z \) and \( a \leq x' \leq b \)

using continuous-attains-inf[OF compact-Icc * contf] by (meson atLeastAtMost-iff)

consider \( a < x < b \mid x = a \lor x = b \)

using \( \langle a \leq x, x \leq b \rangle \) by arith

then show \( \neg \)thesis

proof cases

next

case 2

then have \( f x = f b \) by (auto simp add: fab)

consider \( a < x' < b \mid x' = a \lor x' = b \)

using \( \langle a \leq x', x' \leq b \rangle \) by arith

then show \( \neg \)thesis

proof cases

next

case 2

then have \( f x' = f b \) by (auto simp: fab)

from dense[OF \( \langle a < b \rangle \)] obtain \( r \) where \( r: a < r < b \) by blast

obtain \( d \) where \( d: 0 < d \) and bound:\( \forall y. |x' - y| < d \rightarrow a \leq y \land y \leq b \)

using lemma-interval[OF \( r \)] by blast

have \( l = 0 \) by (rule DERIV-local-min[OF \( \langle a \leq x', x' \leq b \rangle \)])

then show \( \neg \)thesis using \( \langle a \leq x', x' \leq b \rangle \) by (metis has-derivative-unique has-field-derivative-def mult-zero-left)
proof (rule order-antisym)
  show \( f z \leq f b \) by (simp add: fx x-max that)
  show \( f b \leq f z \) by (simp add: fx' x'-min that)
qed

have bound': \( \forall y. |r - y| < d \rightarrow f r = f y \)
proof (intro strip)
  fix y :: real
  assume lt': \( |r - y| < d \)
  then have \( f y = f b \) by (simp add: eq-fb bound)
  then show \( f r = f y \) by (simp add: eq-fb r order-less-imp-le)
qed

obtain l where der: DERIV f r = l
  using derf differentiable-def r (1) r (2) real-differentiable-def
  by blast
have l = 0
  by (rule DERIV-local-const [OF der d bound'])
— the derivative of a constant function is zero
with r der derf [of r]
show ?thesis
  using f' has-derivative-imp-has-field-derivative by fastforce
qed

111.10 Mean Value Theorem

theorem mvt:
fixes f :: real 
assumes a < b
and contf: continuous-on {a..b} f
and def: \( \forall x. [a < x; x < b] \Rightarrow (f \text{ has-derivative } f' x) \text{ (at } x) \)
obtains \( \exists z. a < z \land z < b \land \text{DERIV } f \text{ z } > 0 \)
proof –
  obtain f' where f': \( \forall x. [a < x; x < b] \Rightarrow (f \text{ has-derivative } f' x) \text{ (at } x) \)
    using def unfolding differentiable-def by metis
  then have \( \exists z. a < z \land z < b \land f' z = (\lambda v. 0) \)
    by (metis Rolle-deriv[OF ab])
  then show ?thesis
    using f' has-derivative-imp-has-field-derivative by fastforce
qed
show \((\lambda x. f x - (f b - f a) / (b - a) * x)\)
has-derivative \((\lambda y. f' x y - (f b - f a) / (b - a) * y))\) \(\text{at } x\)
by \((\text{intro derivative-intros } \text{derf}[OF \ x])\)
qed \(\text{use assms in }\langle\text{auto intro: continuous-intros simp: field-simps}\rangle\)
then show \(?thesis\)
by \((\text{smt (verit, ccfv-SIG) pos-le-divide-eq pos-less-divide-eq that)}\)
qed

theorem MVT:
fixes \(a\ b\ ::\ \text{real}\)
assumes \(\text{lt: } a < b\)
and \(\text{contf: } \text{continuous-on } \{a..b\} f\)
shows \(\exists l\ z\ ::\ \text{real}.\ a < z \land z < b \land \text{DERIV } f z :: l \land f b - f a = (b - a) * l\)
proof –
obtain \(f' ::\ \text{real} \Rightarrow \text{real} \Rightarrow \text{real}\)
where \(\text{derf: } \forall x. [a < x \Rightarrow x < b \Rightarrow (f \text{ has-derivative } f' x) \text{ at } x]\)
using \(\text{dif} \\text{ unfolding differentiable-def by metis}\)
then obtain \(z\) where \(a < z < b \land f b - f a = (f' z) (b - a)\)
using \(\text{mvt}[OF \ \text{lt contf}]\) by blast
then show \(?thesis\)
by \((\text{simp add: ac-simps})\)
\((\text{metis derf dif has-derivative-unique has-field-derivative-imp-has-derivative real-differentiable-def})\)
qed

corollary MVT2:
assumes \(a < b\) and \(\text{der: } \forall x. [a \leq x; x \leq b] \Rightarrow \text{DERIV } f x :: f' x\)
shows \(\exists z::\ \text{real}.\ a < z \land z < b \land (f b - f a = (b - a) * f' z)\)
proof –
have \(\exists l\ z.\ a < z \land z < b \land (f \text{ has-real-derivative } l) \text{ at } z\) \land f b - f a = (b - a) * l
proof \(\text{rule MVT}[OF \ \text{lt}]\)\)
show \(\text{continuous-on } \{a..b\} f\)
by \((\text{meson DERIV-continuous atLeastAtMost-iff continuous-at-imp-continuous-on der})\)
show \(\forall x. [a < x; x < b] \Rightarrow f \text{ differentiable } \text{at } x\)
using \(\text{assms by (force dest: order-less-imp-le simp add: real-differentiable-def)}\)
qed
with \(\text{assms show } \text{thesis}\)
by \((\text{blast dest: DERIV-unique order-less-imp-le})\)
qed

lemma pos-derivative-imp-strict-mono:
assumes \(\forall x. (f \text{ has-real-derivative } f' x) \text{ at } x\)
assumes \(\forall x. f' x > 0\)
shows \(\text{strict-mono } f\)
proof (rule strict-monoI)
fix x y :: real
from assms and xy have \( \exists z > x. z < y \land f y - f x = (y - x) \cdot f' z \)
  by (intro MVT2) (auto dest: connectedD-interval)
then obtain z where \( z > x \) \( z < y \) \( f y - f x = (y - x) \cdot f' z \) by blast
also have \( (y - x) \cdot f' z > 0 \) using xy assms by (intro mult-pos-pos) auto
finally show \( f x < f y \) by simp
qed

proposition deriv-nonneg-imp-mono:
assumes deriv: \( \forall x. x \in \{a..b\} \Rightarrow (g \text{ has-real-derivative } g' x) \ (at x) \)
assumes nonneg: \( \forall x. x \in \{a..b\} \Rightarrow g' x \geq 0 \)
assumes ab: \( a \leq b \)
shows \( g a = g b \)
proof (cases a < b)
  assume a < b
  from deriv have \( \forall x. [a \leq x; x \leq b] \Rightarrow (g \text{ has-real-derivative } g' x) \ (at x) \) by simp
  with MVT [OF \( a < b \)] and deriv
  obtain \( \xi \) where \( \xi \text{-ab: } \xi > a \) \( \xi < b \) and \( g \text{-ab: } g b - g a = (b - a) \cdot g' \xi \) by blast
  from \( \xi \text{-ab } a \text{ and nonneg have } (b - a) \cdot g' \xi \geq 0 \) by simp
  with \( g \text{-ab } a \) show \( \text{thesis } \) by simp
qed (insert ab, simp)

111.10.1 A function is constant if its derivative is 0 over an interval.

lemma DERIV-isconst-end:
fixes f :: real \( \Rightarrow \) real
assumes a < b and conf: \( \text{continuous-on } \{a..b\} \) \( f \)
and \( 0: \forall x. [a < x; x < b] \Rightarrow \text{DERIV } f x > 0 \)
shows \( f b = f a \)
using MVT [OF \( a < b \)] \( 0 \) DERIV-unique conf real-differentiable-def
by (fastforce simp: algebra-simps)

lemma DERIV-isconst2:
fixes f :: real \( \Rightarrow \) real
assumes a < b and conf: \( \text{continuous-on } \{a..b\} \) \( f \) and derf: \( \forall x. [a < x; x < b] \Rightarrow \text{DERIV } f x > 0 \)
and \( a \leq x \leq b \)
shows \( f x = f a \)
proof (cases a < x)
  case True
  have *: \( \text{continuous-on } \{a..x\} \) \( f \)
  using \( x \leq b \) conf continuous-on-subset by fastforce
  show \( \text{thesis } \)
    by (rule DERIV-isconst-end [OF True *]) (use \( x \leq b \) derf in auto)
qed (use \( \langle a \leq x \rangle \) in auto)

lemma \( \text{DERIV-isconst3} \):
  fixes \( a \ b \ x \ y \) :: real
  assumes \( a < b \)
    and \( x \in \{ a <..< b \} \)
    and \( y \in \{ a <..< b \} \)
    and derivable: \( \forall x. x \in \{ a <..< b \} \implies \text{DERIV} f x > 0 \)
  shows \( f x = f y \)
proof (cases \( x = y \))
  case False
  let \( ?a = \min x y \)
  let \( ?b = \max x y \)
  have \( \star \): \( \text{DERIV} f z > 0 \) if \( ?a \leq z \leq ?b \) for \( z \)
  proof
    have \( a < z \) and \( z < b \)
      using that \( x \in \{ a <..< b \} \), \( y \in \{ a <..< b \} \) by auto
    then have \( z \in \{ a <..< b \} \) by auto
    then show \( \text{DERIV} f z > 0 \) by (rule dericable)
  qed
  have \( \text{isCont} \):
    continuous-on \( \{ ?a .. ?b \} \) \( f \)
  by (meson \( \star \) DERIV-continuous-on atLeastAtMost-iff has-field-derivative-at-within)
  have \( \text{DERIV}: \forall z. \[ ?a < z; z < ?b \] \implies \text{DERIV} f z > 0 \)
  using \( \star \) by auto
  have \( ?a < ?b \) using \( x \neq y \) by auto
  from \( \text{DERIV-isconst2} \[ \text{OF this isCont DERIV, of } x \] \) and \( \text{DERIV-isconst2} \[ \text{OF this isCont DERIV, of } y \] \)
  show \( ?\text{thesis} \) by auto
qed auto

lemma \( \text{DERIV-isconst-all} \):
  fixes \( f \) :: real \( \Rightarrow \) real
  shows \( \forall x. \text{DERIV} f x > 0 \imp f x = f y \)
  apply (rule linorder-cases \[ \text{of } x \ y \] )
  apply (metis DERIV-continuous DERIV-unique continuous-at-imp-continuous-on)+
  done

lemma \( \text{DERIV-const-ratio-const} \):
  fixes \( f \) :: real \( \Rightarrow \) real
  assumes \( a \neq b \) and \( \text{df}: \forall x. \text{DERIV} f x > k \)
  shows \( f b - f a = (b - a) \* k \)
proof (cases \( a \ b \) rule: linorder-cases)
  case less
  show \( ?\text{thesis} \)
    using MVT \[ \text{OF less} \] \( \text{df} \)
    by (metis DERIV-continuous DERIV-unique continuous-at-imp-continuous-on real-differentiable-def)
next
case greater
Lemma \( \text{DERIV-const-ratio-const2} \):

fixes \( f :: \mathbb{R} \Rightarrow \mathbb{R} \)

assumes \( a \neq b \) and \( \forall x. \text{DERIV } f \ x :> k \)

shows \( (f \ b - f \ a) / (b - a) = k \)

using \( \text{DERIV-const-ratio-const} [\text{OF assms}] \)

Lemma \( \text{real-average-minus-first} \) [simp]: \( (a + b) / 2 - a = (b - a) / 2 \)

for \( a \) and \( b :: \mathbb{R} \)

by simp

Lemma \( \text{real-average-minus-second} \) [simp]: \( (b + a) / 2 - a = (b - a) / 2 \)

for \( a \) and \( b :: \mathbb{R} \)

by simp

Galileo’s ‘trick’: average velocity = av. of end velocities.

Lemma \( \text{DERIV-const-average} \):

fixes \( v :: \mathbb{R} \Rightarrow \mathbb{R} \)

and \( a, b :: \mathbb{R} \)

assumes \( \text{neq}: a \neq b \)

and \( \text{der: } \forall x. \text{DERIV } v \ x :> k \)

shows \( v (((a + b) / 2) - v \ a) / ((a + b) / 2 - a) = k \)

proof (cases rule: linorder-cases [of \( a, b \)])

case equal

with \( \text{neq show } \) [simp]

next

case less

have \( (v \ b - v \ a) / (b - a) = k \)

by (rule \( \text{DERIV-const-ratio-const2} [\text{OF } \text{der}] \))

then have \( (b - a) * ((v \ b - v \ a) / (b - a)) = (b - a) * k \)

by simp

moreover have \( (v (((a + b) / 2) - v \ a) / ((a + b) / 2 - a) = k \)

by (rule \( \text{DERIV-const-ratio-const2} [\text{OF } \text{- der}] \) [simp add: \( \text{neq} \)]

ultimately show \( \) [simp]

using \( \text{neq by force} \)

next

case greater

have \( (v \ b - v \ a) / (b - a) = k \)

by (rule \( \text{DERIV-const-ratio-const2} [\text{OF } \text{neq } \text{der}] \))

then have \( (b - a) * ((v \ b - v \ a) / (b - a)) = (b - a) * k \)

by simp

moreover have \( (v (((b + a) / 2) - v \ a) / ((b + a) / 2 - a) = k \)
by (rule DERIV-const-ratio-const2 [OF - der]) (simp add: neq)
ultimately show ?thesis
  using neq by (force simp add: add.commute)
qed

111.10.2 A function with positive derivative is increasing

A simple proof using the MVT, by Jeremy Avigad. And variants.

lemma DERIV-pos-imp-increasing-open:
  fixes a b :: real
  and f :: real ⇒ real
  assumes a < b
  and \( \forall x. a < x \implies x < b \implies (\exists y. \text{DERIV } f x > y \land y > 0) \)
  and con: continuous-on \{a..b\} f
  shows f a < f b
proof (rule ccontr)
  assume \( \neg \)thesis
  have \( \exists l z. a < z \land z < b \land \text{DERIV } f z > l \land f b - f a = (b - a) \ast l \)
    by (rule MVT) (use assms real-differentiable-def in \( \text{force} + \))
  then obtain l z where z: a < z < b \( \text{DERIV } f z > l \) and f b - f a = (b - a) \( \ast l \)
    by auto
  with assms f have \( \neg l > 0 \)
    by (metis linorder-not-le mult-le-0-iff diff-le-0-iff-le)
  with assms z show False
    by (metis DERIV-unique)
qed

lemma DERIV-pos-imp-increasing:
  fixes a b :: real and f :: real ⇒ real
  assumes a < b
  and \( \forall x. [a \leq x; x \leq b] \implies (\exists y. \text{DERIV } f x > y \land y > 0) \)
  shows f a < f b
by (metis less-le-not-le DERIV-atLeastAtMost-imp-continuous-on DERIV-pos-imp-increasing-open [OF \( a < b \) \( \text{der} \)])

lemma DERIV-nonneg-imp-nondecreasing:
  fixes a b :: real
  and f :: real ⇒ real
  assumes a ≤ b
  and \( \forall x. [a \leq x; x \leq b] \implies (\exists y. \text{DERIV } f x > y \land y \geq 0) \)
  shows f a ≤ f b
proof (rule ccontr, cases \( a = b \))
  assume \( \neg \)thesis and \( a = b \)
  then show False by auto
next
  assume \( \ast. \neg \)thesis
  assume a ≠ b
  with \( a \leq b \) have a < b
by linarith
moreover have continuous-on \{a..b\} f
by (meson DERIV-isCont assms(2) atLeastAtMost-iff continuous-at-imp-continuous-on)
ultimately have \(\exists l \ z. a < z \wedge z < b \wedge DERIV f z : l \land f b - f a = (b - a)\)
* l
using assms MVT [OF \(\langle a < b, of f \rangle\) real-differentiable-def less-eq-real-def by blast
then obtain l z where l: a < z z < b DERIV f z : l and **: f b - f a = (b - a) * l
by auto
with * have \(a < b f b < f a\) by auto
with ** have \(l \geq 0\) by (auto simp add: not-le algebra-simps
(metis * add-le-cancel-right assms(1) less-eq-real-def mult-right-mono add-left-mono linear order-refl)
with assms lz show False
by (metis DERIV-unique order-less-imp-le)
qed

lemma DERIV-neg-imp-decreasing-open:
fixes a b :: real 
  and f :: real \Rightarrow real
  assumes a < b 
  and \(\lambda x. a < x \Rightarrow x < b \Rightarrow \exists y. DERIV f x : y \wedge y < 0\)
  and con: continuous-on \{a..b\} f 
  shows f a > f b 
proof
  have \((\lambda x. - f x) a < (\lambda x. - f x) b\)
  proof (rule DERIV-pos-imp-increasing-open [of a b])
  show \(\lambda x. [a < x; x < b] \Rightarrow \exists y. ((\lambda x. - f x) has-real-derivative y) (at x) \wedge 0 < y\)
    using assms 
    by simp (metis field-differentiable-minus neg-0-less-iff-less)
  show continuous-on \{a..b\} (\lambda x. - f x)
    using con continuous-on-minus by blast
  qed (use assms in auto)
  then show ?thesis
    by simp
qed

lemma DERIV-neg-imp-decreasing:
fixes a b :: real and f :: real \Rightarrow real
  assumes a < b 
  and der: \(\lambda x. [a \leq x; x \leq b] \Rightarrow \exists y. DERIV f x : y \wedge y < 0\)
  shows f a > f b 
by (metis less-le-not-le DERIV-atLeastAtMost-continuous-on DERIV-neg-imp-decreasing-open [OF \(\langle a < b, der \rangle\) der])

lemma DERIV-nonpos-imp-nonincreasing:
fixes a b :: real
and \( f :: \text{real} \Rightarrow \text{real} \)
assumes \( a \leq b \)
and \( \forall x. [a \leq x; x \leq b] \implies \exists y. \text{DERIV} f x :> y \wedge y \leq 0 \)
shows \( f a \geq f b \)
proof
- have \((\lambda x. -f x) a \leq (\lambda x. -f x) b\)
  using \text{DERIV-nonneg-imp-nonincreasing} \[ a \leq b \]
by fastforce
then show \(?\text{thesis}\)
  by simp
qed

lemma \text{DERIV-pos-imp-increasing-at-bot}:
fixes \( f :: \text{real} \Rightarrow \text{real} \)
assumes \( \forall x. x \leq b \implies (\exists y. \text{DERIV} f x :> y \wedge y > 0) \)
and \( \text{lim} : (f ---\rightarrow \lim) \text{ at-bot} \)
shows \( \lim < f b \)
proof
- have \( \exists N. \forall n \leq N. f n \leq f (b - 1) \)
  by (rule-tac \( x=b - 2 \) in \( \text{exI} \)) \((\text{force intro: order.strict-implies-order DERIV-pos-imp-increasing assms})\)
then have \( \lim \leq f (b - 1) \)
  by (auto simp: eventually-at-bot-linorder tendsto-upperbound \[ \text{OF} \lim \])
also have \( \ldots < f b \)
  by (force intro: \text{DERIV-pos-imp-increasing} \[ \text{where} f=f \] assms)
finally show \(?\text{thesis}\).
qed

lemma \text{DERIV-neg-imp-decreasing-at-top}:
fixes \( f :: \text{real} \Rightarrow \text{real} \)
assumes \( \exists y. \text{DERIV} f x :> y \wedge y < 0 \)
and \( \text{lim} : (f ---\rightarrow \lim) \text{ at-top} \)
shows \( \lim < f b \)
proof
apply (rule \text{DERIV-pos-imp-increasing-at-bot} \[ \text{where} f=\lambda i. f (-i) \] \( \text{and} \) \( b = -b \), simplified)
apply (metis \text{DERIV-mirror} \text{der} \text{le-minus-iff} \text{neg-0-less-iff-less})
apply (metis filterlim-at-top-mirror \text{lim})
done

Derivative of inverse function

lemma \text{DERIV-inverse-function}:
fixes \( f g :: \text{real} \Rightarrow \text{real} \)
assumes \( \text{der}: \text{DERIV} f (g x) :> D \)
and \( \text{neg}: D \neq 0 \)
and \( x: a < x x < b \)
and \( \text{inj}: \forall y. [a < y; y < b] \implies f (g y) = y \)
and \( \text{cont}: \text{isCont} g x \)
shows \( \text{DERIV} g x :> \text{inverse} D \)
unfolding \text{has-field-derivative-iff}
proof (rule LIM-equal2)
  show $0 < \min (x - a) (b - x)$
  using $x$ by arith
next
fix $y$
assume $\text{norm} (y - x) < \min (x - a) (b - x)$
then have $a < y$ and $y < b$
  by (simp-all add: abs-less-iff)
then show $(g y - g x) / (y - x) = \text{inverse} \left((f (g y) - x) / (g y - g x)\right)$
  by (simp add: inj)
next
have $(\lambda z. (f z - f (g x)) / (z - g x)) - g x \rightarrow D$
  by (rule der [unfolded has-field-derivative-iff])
then have $1: (\lambda z. (f z - x) / (z - g x)) - g x \rightarrow D$
  using inj $x$ by simp
have $2: \exists d>0. \forall y. y \neq x \land \text{norm} (y - x) < d \rightarrow g y \neq g x$
proof (rule exI, safe)
  show $0 < \min (x - a) (b - x)$
  using $x$ by simp
next
fix $y$
assume $\text{norm} (y - x) < \min (x - a) (b - x)$
then have $y: a < y \land y < b$
  by (simp-all add: abs-less-iff)
assume $g y = g x$
then have $f (g y) = f (g x)$ by simp
then have $y = x$ using inj $y x$ by simp
also assume $y \neq x$
finally show $\text{False}$ by simp
qed
have $(\lambda y. (f (g y) - x) / (g y - g x)) -x \rightarrow D$
  using cont 1 2 by (rule isCont-LIM-compose2)
then show $(\lambda y. \text{inverse} \left((f (g y) - x) / (g y - g x)\right)) -x \rightarrow \text{inverse} D$
  using neq by (rule tendsto-inverse)
qed

111.11 Generalized Mean Value Theorem

theorem GMVT:
fixes $a \ b :: \text{real}$
assumes $\text{alb}: a < b$
  and $\text{fc}: \forall x. a \leq x \land x \leq b \rightarrow \text{isCont} f x$
  and $\text{fd}: \forall x. a < x \land x < b \rightarrow f \text{ differentiable} \ (at x)$
  and $\text{gc}: \forall x. a \leq x \land x \leq b \rightarrow \text{isCont} g x$
  and $\text{gd}: \forall x. a < x \land x < b \rightarrow g \text{ differentiable} \ (at x)$
sshows $\exists g'c \ f'c \ c$.
  $\text{DERIV} g \ c \Rightarrow g'c \land \text{DERIV} f \ c \Rightarrow f'c \land a < c \land c < b \land (f b - f a) * g'c = (g b - g a) * f'c$
proof –
let \( \forall h = \lambda x. (f b - f a) * g x - (g b - g a) * f x \)

have \( \exists l. a < z \land z < b \land \text{DERIV } \forall h z : l \land \forall h b - \forall h a = (b - a) * l \)

proof (rule MVT)
  from assms show \( a < b \) by simp
  show continuous-on \( \{a..b\} \) \( \forall h \)
    by (simp add: continuous-at-imp-continuous-on fc gc)
  show \( \forall x. [a < x; x < b] \implies \forall h \) differentiable \( (at x) \)
    using fd gd by simp
  qed
  then obtain \( l \) where \( l : \exists z. a < z \land z < b \land \text{DERIV } \forall h z : l \land \forall h b - \forall h a = (b - a) * l \)
  then obtain \( c \) where \( c : a < c \land c < b \land \text{DERIV } \forall h c : l \land \forall h b - \forall h a = (b - a) * l \)

from \( c \) have \( cint: a < c \land c < b \) by auto
then obtain \( g'c \) where \( g'c: \text{DERIV } g \ c : g'c \)
  using gd real-differentiable-def by blast
from \( c \) have \( a < c \land c < b \) by auto
then obtain \( f'c \) where \( f'c: \text{DERIV } f \ c : f'c \)
  using fd real-differentiable-def by blast

from \( c \) have \( \text{DERIV } \forall h c : l \) by auto
moreover have \( \text{DERIV } \forall h c : g'c \ast (f b - f a) = f'c \ast (g b - g a) \)
  using \( g'c \) \( f'c \) by (auto intro!: derivative-eq-intros)
ultimately have \( \text{leq: } l = g'c \ast (f b - f a) - f'c \ast (g b - g a) \) by (rule DERIV-unique)

have \( \forall h b - \forall h a = (b - a) \ast (g'c \ast (f b - f a) - f'c \ast (g b - g a)) \)
proof
  from \( c \) have \( \forall h b - \forall h a = (b - a) \ast l \) by auto
  also from \( \text{leq} \) have \( \ldots = (b - a) \ast (g'c \ast (f b - f a) - f'c \ast (g b - g a)) \) by simp
  finally show \( ?\text{thesis} \) by simp
  qed
moreover have \( \forall h b - \forall h a = 0 \)
proof
  have \( \forall h b - \forall h a = \)
    \( ((f b) \ast (g b) - (f a) \ast (g b) - (g b) \ast (f b) - (g a) \ast (f b)) -\)
    \( (f b) \ast (g a) - (f a) \ast (g a) - (g a) \ast (f a) + (g a) \ast (f a)) \)
  by (simp add: algebra-simps)
  then show \( ?\text{thesis} \) by auto
  qed
ultimately have \( (b - a) \ast (g'c \ast (f b - f a) - f'c \ast (g b - g a)) = 0 \) by auto
with \( \text{abb} \) have \( g'c \ast (f b - f a) = f'c \ast (g b - g a) \) by simp
then have \( g'c \ast (f b - f a) = f'c \ast (g b - g a) \) by simp
then have \( (f b - f a) \ast g'c = (g b - g a) \ast f'c \) by (simp add: ac-simps)
with \( g'c \) \( f'c \) \( \text{cint} \) show \( ?\text{thesis} \) by auto
qed
lemma GMVT':
fixes f g :: real ⇒ real
assumes a < b
and isCont-f: ∀z. a ≤ z → z ≤ b → isCont f z
and isCont-g: ∀z. a ≤ z → z ≤ b → isCont g z
and DERIV-g: ∀z. a < z → z < b → DERIV g z :> (g' z)
and DERIV-f: ∀z. a < z → z < b → DERIV f z :> (f' z)
shows ∃c. a < c ∧ c < b ∧ (f b - f a) * g' c = (g b - g a) * f' c
proof –
have ∃g'c f'c c. DERIV g c :> g'c ∧ DERIV f c :> f'c ∧
  a < c ∧ c < b ∧ (f b - f a) * g' c = (g b - g a) * f' c
  using assms by (intro GMVT) (force simp: real-differentiable-def)+
then obtain c where a < c c < b ∧ (f b - f a) * g' c = (g b - g a) * f' c
  using DERIV-f DERIV-g by (force dest: DERIV-unique)
then show ?thesis
  by auto
qed

111.12 L'Hopitals rule

lemma isCont-If-ge:
fixes a :: 'a :: linorder-topology
assumes continuous (at-left a) g and f: (f −→ g a) (at-right a)
shows isCont (λx. if x ≤ a then g x else f x) a (isCont ?gf a)
proof –
have g: (g −→ g a) (at-left a)
  using assms continuous-within by blast
show ?thesis
  unfolding isCont-def continuous-within
  proof (intro filterlim-split-at; simp)
  show (?gf −→ g a) (at-left a)
    by (subst filterlim-cong[OF refl refl, where g=g]) (simp-all add: eventually-at-filter less-le g)
  show (?gf −→ g a) (at-right a)
    by (subst filterlim-cong[OF refl refl, where g=f]) (simp-all add: eventually-at-filter less-le f)
  qed
qed

lemma lhopital-right-0:
fixes f0 g0 :: real ⇒ real
assumes f-0: (f0 −→ 0) (at-right 0)
  and g-0: (g0 −→ 0) (at-right 0)
  and ev:
    eventually (λx. g0 x ≠ 0) (at-right 0)
    eventually (λx. g' x ≠ 0) (at-right 0)
    eventually (λx. DERIV f0 x :> f' x) (at-right 0)
    eventually (λx. DERIV g0 x :> g' x) (at-right 0)
  and lim: filterlim (λx. (f' x / g' x)) F (at-right 0)
shows filterlim (λ x. f0 x / g0 x) F (at-right 0)

proof –
define f where [abs-def]: f x = (if x ≤ 0 then 0 else f0 x) for x
then have f 0 = 0 by simp

define g where [abs-def]: g x = (if x ≤ 0 then 0 else g0 x) for x
then have g 0 = 0 by simp

have eventually (λ x. g0 x ≠ 0 ∧ g' x ≠ 0 ∧
  DERIV f0 x :> (f' x) ∧ DERIV g0 x :> (g' x)) (at-right 0)
using ev by eventually-elim auto
then obtain a where [arith]: 0 < a
and g0-neq-0: ∀ x. 0 < x ⇒ x < a ⇒ g0 x ≠ 0
and g'-neq-0: ∀ x. 0 < x ⇒ x < a ⇒ g' x ≠ 0
and f0: ∀ x. 0 < x ⇒ x < a ⇒ DERIV f0 x :> (f' x)
and g0: ∀ x. 0 < x ⇒ x < a ⇒ DERIV g0 x :> (g' x)
unfolding eventually-at by (auto simp: dist-real-def)

have g-neq-0: ∀ x. 0 < x ⇒ x < a ⇒ g x ≠ 0
using g0-neq-0 by (simp add: g-def)

have f: DERIV f x :> (f' x) if x: 0 < x x < a for x
using that
by (intro DERIV-cong-ev[THEN iffD1, OF - - - f0[OF x]])
(auto simp: f-def eventually-nhds-metric dist-real-def intro!: exI[of - x])

have g: DERIV g x :> (g' x) if x: 0 < x x < a for x
using that
by (intro DERIV-cong-ev[THEN iffD1, OF - - - g0[OF x]])
(auto simp: g-def eventually-nhds-metric dist-real-def intro!: exI[of - x])

have isCont f 0
unfolding f-def by (intro isCont-If-ge f-0 continuous-const)

have isCont g 0
unfolding g-def by (intro isCont-If-ge g-0 continuous-const)

have ∃ ζ. ∀ x ∈ {0 <..< a}. 0 < ζ x ∧ ζ x < x ∧ f x / g x = f' (ζ x) / g' (ζ x)
proof (rule choice, rule ballI)
fix x
assume x ∈ {0 <..< a}
then have x[arith]: 0 < x x < a by auto
with g'-neq-0 g-neq-0 (g 0 = 0) have g': ∀ x. 0 < x ⇒ x < a ⇒ 0 ≠ g' x
  g 0 ≠ g x
  by auto
have f x[arith]: 0 ≤ x ⇒ x < a ⇒ isCont f x
  using isCont f 0 by (auto intro: DERIV-isCont simp: le-less)
moreover have g x[arith]: 0 ≤ x ⇒ x < a ⇒ isCont g x
  using isCont g 0 by (auto intro: DERIV-isCont simp: le-less)
ultimately have \( \exists c. \ 0 < c \land c < x \land (f x - f 0) \cdot g' c = (g x - g 0) \cdot f' c \)
using \( f g x < a \) by (intro GMVT') auto
then obtain \( c \) where \( *: 0 < c < c < x (f x - f 0) \cdot g' c = (g x - g 0) \cdot f' c \)
by blast
moreover
from \( \ast \cdot g'(1)[of c] \) \( g'(2) \) have \((f x - f 0) \cdot (g x - g 0) = f' c \cdot g' c \)
by (simp add: field-simps)
ultimately show \( \exists y. \ 0 < y \land y < x \land f x / g x = f' y / g' y \)
using \( f 0 = 0 \cdot g 0 = 0 \) by (auto intro: exI[of - a])
\qedsht
then have \( \zeta \) where \( \forall x \in \{0 \ldots a\}. \ 0 < \zeta x \land \zeta x < x \land f x / g x = f' (\zeta x) / g' (\zeta x) \)
.. then have \( \zeta \) eventually (\( \lambda x. \ 0 < \zeta x \land \zeta x < x \land f x / g x = f' (\zeta x) / g' (\zeta x) \)) (at-right 0)
unfolding eventually-at by (intro exI[of - a]) (auto simp: dist-real-def)
moreover
from \( \zeta \) have eventually (\( \lambda x. \ \text{norm} (\zeta x) \leq x \)) (at-right 0)
by eventually-elim auto
then have (\( \lambda x. \ \text{norm} (\zeta x) \)) (at-right 0)
by (rule-tac real-tendsto-sandwich[where \( f=\lambda x. \ 0 \) and \( h=\lambda x. \ x \]) auto
then have (\( \zeta \)) (at-right 0)
by (rule tendsto-norm-zero-cancel)
with \( \zeta \) have filterlim \( \zeta \) (at-right 0) (at-right 0)
by (auto elim!: eventually-mono simp: filterlim-at)
from this \( \text{lim} \) have filterlim (\( \lambda t. \ f' (\zeta t) / g' (\zeta t) \)) \( F \) (at-right 0)
by (rule-tac filterlim-compose[of - - - \( \zeta \)])
ultimately have filterlim (\( \lambda t. \ f t / g t \) \( F \)) (at-right 0) (is \( ?P \))
by (rule-tac filterlim-cong[THEN iffD1, OF refl refl])
(auto elim: eventually-mono)
also have \( ?P \iff ?\text{thesis} \)
by (rule filterlim-cong) (auto simp: f-def g-def eventually-at-filter)
finally show \( ?\text{thesis} \).
\qedsht

**Lemma lhospital-right:**
\( (f \longrightarrow 0) \) (at-right 0) (at-right 0) 
\( \longrightarrow (g \longrightarrow 0) \) (at-right 0) 
\( \longrightarrow \)
eventually (\( \lambda x. \ g x \neq 0 \) ) (at-right 0) 
\( \longrightarrow \)
eventually (\( \lambda x. \ g' x \neq 0 \) ) (at-right 0) 
\( \longrightarrow \)
eventually (\( \lambda x. \ \text{DERIV} f x \cdot f' x \) ) (at-right 0) 
\( \longrightarrow \)
eventually (\( \lambda x. \ \text{DERIV} g x \cdot g' x \) ) (at-right 0) 
\( \longrightarrow \)
filterlim (\( \lambda x. \ f x / g x \)) \( F \) (at-right 0) 
\( \longrightarrow \)
filterlim (\( \lambda x. \ f x / g x \)) \( F \) (at-right 0) 
\( \longrightarrow \)
for \( x :: \text{real} \)
unfolding eventually-at-right-to-0[of - x] filterlim-at-right-to-0[of - x] DE-RIV-shift 
by (rule lhospital-right-0)

**Lemma lhospital-left:**
\( (f \longrightarrow 0) \) (at-left 0) \( \longrightarrow (g \longrightarrow 0) \) (at-left 0) 
\( \longrightarrow \)
lemma lhospitall-0-at-top:
fixes f g :: real ⇒ real
assumes g-0: LIM x at-right 0. g x := at-top
and ev:
  eventually (λx. g' x ≠ 0) (at-right 0)
  eventually (λx. DERIV f x := f' x) (at-right 0)
  eventually (λx. DERIV g x := g' x) (at-right 0)
  and lim: ((λ x. (f' x / g' x)) −→ x) (at-right 0)
shows ((λ x. f x / g x) −→ x) (at-right 0)
unfolding tendsto-iff

proof safe
  fix e :: real
  assume 0 < e
  with lim[unfolded tendsto-iff, rule-format, of e / 4]
  have eventually (λt. dist (f' t / g' t) x < e / 4) (at-right 0)
    by simp
  from eventually-conj[OF eventually-conj[OF ev(1) ev(2)] eventually-conj[OF ev(3) this]]
  obtain a where [arith]: 0 < a
    and g'-neg-0: λx. 0 < x ⇒ x < a ⇒ g' x ≠ 0
    and f0: λx. 0 < x ⇒ x ≤ a ⇒ DERIV f x := (f' x)
    and g0: λx. 0 < x ⇒ x ≤ a ⇒ DERIV g x := (g' x)
    and Df: λt. 0 < t ⇒ t < a ⇒ dist (f' t / g' t) x < e / 4
    unfolding eventually-at-le by (auto simp: dist-real-def)
  from Df have eventually (λt. t < a) (at-right 0) eventually (λt::real. 0 < t)
(at-right 0)

unfolding eventually-at by (auto intro!: exI[of - a] simp: dist-real-def)

moreover
have eventually (λt. 0 < t) (at-right 0) eventually (λt. g a < g t) (at-right 0)
  using g-0 by (auto elim: eventually-mono simp: filterlim-at-top-dense)

moreover
have inv-g: ((λx. inverse (g x)) →→ 0) (at-right 0)
  using tendsto-inverse-0 filterlimMonoOF g-0 at-top-le-at-infinity order-refl
  by (rule filterlim-compose)
then have ((λx. norm (1 − g a * inverse (g x))) →→ norm (1 − g a * 0)) (at-right 0)
  by (intro tendsto-intros)
then have ((λx. norm (1 − g a / g x)) →→ 1) (at-right 0)
  by (simp add: inverse-eq-divide)
from this[unfolded tendsto-iff, rule-format, of 1]
have eventually (λx. norm (1 − g a / g x) < 2) (at-right 0)
  by (auto elim!: eventually-mono simp: dist-real-def)

moreover
from inv-g have ((λt. norm ((f a − x * g a) * inverse (g t))) →→ norm ((f a − x * g a) * 0))
  (at-right 0)
  by (intro tendsto-intros)
then have ((λt. norm (f a − x * g a) / norm (g t)) →→ 0) (at-right 0)
  by (simp add: inverse-eq-divide)
from this[unfolded tendsto-iff, rule-format, of e / 2] (0 < e)
have eventually (λt. norm (f a − x * g a) / norm (g t) < e / 2) (at-right 0)
  by (auto simp: dist-real-def)

ultimately show eventually (λt. dist (f t / g t) x < e) (at-right 0)

proof (eventually-elim)
  fix t assume ℓ(arith): 0 < t t < a g a < g t 0 < g t
  assume ineq: norm (1 − g a / g t) < 2 norm (f a − x * g a) / norm (g t) < e / 2

  have ℓy. t < y ∧ y < a ∧ (g a − g t) * f' y = (f a − f t) * g' y
    using f0 g0 t(t,2) by (intro GMVT) (force intro!: DERIV-isCont)+
  then obtain y where ℓ[arith]: t < y y < a
    and D-eq0: (g a − g t) * f' y = (f a − f t) * g' y
    by blast
  from D-eq0 have D-eq: (f t − f a) / (g t − g a) = f' y / g' y
    using g a < g t g' -neq-0[of y] by (auto simp add: field-simps)
  have +: f t / g t − t = (f t − f a) / (g t − g a) − x * (1 − g a / g t) + (f a − x * g a) / g t
    by (simp add: field-simps)
  have norm (f t / g t − t) ≤
THEORY "Deriv"

norm (((f \ t - f a) / (g \ t - g a) - x) * (1 - g a / g t)) + norm ((f a - x / (g a) / g t))

unfolding by (rule norm-triangle-ineq)
also have \ldots = dist (f' y / g' y) x * norm (1 - g a / g t) + norm (f a - x / (g a) / g t)
by (simp add: abs-real D-eq dist-real-def)
also have \ldots < (e / 4) * 2 + e / 2
using lhopital-right-at-top filterlim-at-right-to-0[g a]
finally show dist (f t / g t) x < e
by (simp add: dist-real-def)
qed

lemma lhospital-right-at-top:
LIM x at-right x. (g::real \Rightarrow real) x \Rightarrow at-top \Rightarrow
  eventually (\lambda x. g' x \neq 0) (at-right x) \Rightarrow
  eventually (\lambda x. DERIV f x \Rightarrow f' x) (at-right x) \Rightarrow
  eventually (\lambda x. DERIV g x \Rightarrow g' x) (at-right x) \Rightarrow
  ((\lambda x. f' x / g' x) \Rightarrow y) (at-right x) \Rightarrow
  ((\lambda x. f x / g x) \Rightarrow y) (at-right x)
unfolding eventually-at-right-to-0[g a]
RIV-shift
by (rule lhospital-right-0-at-top)

lemma lhospital-left-at-top:
LIM x at-left x. g x \Rightarrow at-top \Rightarrow
  eventually (\lambda x. g' x \neq 0) (at-left x) \Rightarrow
  eventually (\lambda x. DERIV f x \Rightarrow f' x) (at-left x) \Rightarrow
  eventually (\lambda x. DERIV g x \Rightarrow g' x) (at-left x) \Rightarrow
  ((\lambda x. f' x / g' x) \Rightarrow y) (at-left x) \Rightarrow
  ((\lambda x. f x / g x) \Rightarrow y) (at-left x)
for x :: real
unfolding eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror
by (rule lhospital-right-at-top[where f'=\lambda x. - f' (- x)]) (auto simp: DERIV-mirror)

lemma lhospital-at-top:
LIM x at x. (g::real \Rightarrow real) x \Rightarrow at-top \Rightarrow
  eventually (\lambda x. g' x \neq 0) (at x) \Rightarrow
  eventually (\lambda x. DERIV f x \Rightarrow f' x) (at x) \Rightarrow
  eventually (\lambda x. DERIV g x \Rightarrow g' x) (at x) \Rightarrow
  ((\lambda x. f' x / g' x) \Rightarrow y) (at x) \Rightarrow
  ((\lambda x. f x / g x) \Rightarrow y) (at x)
unfolding eventually-at-split filterlim-at-split
by (auto intro: lhospital-right-at-top[g x g' f f'] lhospital-left-at-top[g x g' f f'])

lemma lhospital-at-top-at-top:
fixes f g :: real \Rightarrow real
assumes g-0: LIM x at-top. g x \Rightarrow at-top
and $g'$: eventually $(\lambda x. g' x \neq 0)$ at-top
and $Df$: eventually $(\lambda x. \text{DERIV } f x \mapsto f' x)$ at-top
and $Dg$: eventually $(\lambda x. \text{DERIV } g x \mapsto g' x)$ at-top
and $\lim$: $(\lambda x. (f' x / g' x)) \longrightarrow x$ at-top
shows $((\lambda x. f x / g x) \longrightarrow x)$ at-top
unfolding $\text{filterlim-at-top-to-right}$
proof (rule $\text{lhopital-right-0-at-top}$)
let $?F = \lambda x. f (\text{inverse } x)$
let $?G = \lambda x. g (\text{inverse } x)$
let $?R = \text{at-right } (0::real)$
let $?D = \lambda f' x. f' (\text{inverse } x) * -(\text{inverse } x \rightarrow Suc (Suc 0))$
show $\text{LIM } x \rightarrow ?R. \ ?G x :: \text{at-top}$
  using $g\cdot0$ unfolding $\text{filterlim-at-top-to-right}$ .
show eventually $(\lambda x. \text{DERIV } ?G x :: ?D g' x) ?R$
  unfolding eventually-at-right-to-top
  using $Dg$ eventually-ge-at-top[where $c=1$]
by eventually-elim (rule derivative-eq-intros DERIV-chain'[where $f=\text{inverse}$]
simp)+
| simp+
show eventually $(\lambda x. \text{DERIV } ?F x :: ?D f' x) ?R$
  unfolding eventually-at-right-to-top
  using $Df$ eventually-ge-at-top[where $c=1$]
by eventually-elim (rule derivative-eq-intros DERIV-chain'[where $f=\text{inverse}$]
simp)+
| simp+
show eventually $(\lambda x. ?D g' x \neq 0) ?R$
  unfolding eventually-at-right-to-top
  using $g'$ eventually-ge-at-top[where $c=1$]
by eventually-elim auto
show $((\lambda x. ?D f' x / ?D g' x) \longrightarrow x) ?R$
  unfolding $\text{filterlim-at-right-to-top}$
  apply (intro filterlim-conq THEN diffD2, OF refl refl - lim)
  using eventually-ge-at-top[where $c=1$]
by eventually-elim simp
qed

lemma $\text{lhopital-right-at-top-at-top}$:
fixes $f \ g :: \text{real } \Rightarrow \text{real}$
assumes $f\cdot0$: $\text{LIM } x \rightarrow \text{at-right } a. f x :: \text{at-top}$
assumes $g\cdot0$: $\text{LIM } x \rightarrow \text{at-right } a. g x :: \text{at-top}$
and ev:
  eventually $(\lambda x. \text{DERIV } f x \mapsto f' x)$ (at-right $a$)
  eventually $(\lambda x. \text{DERIV } g x \mapsto g' x)$ (at-right $a$)
and $\lim$: $\text{filterlim } (\lambda x. (f' x / g' x)) \text{ at-top } (\text{at-right } a)$
shows $\text{filterlim } (\lambda x. f x / g x) \text{ at-top } (\text{at-right } a)$
proof –
from $\text{lim}$ have pos: eventually $(\lambda x. f' x / g' x > 0)$ (at-right $a$)
  unfolding $\text{filterlim-at-top-dense}$ by blast
have $((\lambda x. g x / f x) \longrightarrow 0)$ (at-right $a$)
proof (rule lhopital-right-at-top)
  from pos show eventually $(\lambda x. f x / 0)$ (at-right $a$) by eventually-elim auto
from `tendsto-inverse-0-at-top[OF lim]`
  show \((\lambda x. g' x / f' x) \longrightarrow 0\) (at-right a) by simp
qed fact+
moreover from `f-0 g-0`
  have eventually \((\lambda x. f x > 0)\) (at-right a) eventually \((\lambda x. g x > 0)\) (at-right a)
  unfolding `filterlim-at-top-dense` by blast+
thence eventually \((\lambda x. g x / f x > 0)\) (at-right a) by eventually-elim simp
ultimately have `filterlim (\lambda x. inverse (g x / f x))` at-top (at-right a)
  by (rule `filterlim-inverse-at-top`)
thus `?thesis` by simp
qed

lemma `lhopital-right-at-top-at-bot`:
fixes `f` `g` :: `real` \Rightarrow `real`
assumes `f-0`: `LIM x at-right a. f x >` at-top
assumes `g-0`: `LIM x at-right a. g x >` at-bot
  and ev:
    eventually \((\lambda x. DERIV f x > f' x)\) (at-right a)
    eventually \((\lambda x. DERIV g x > g' x)\) (at-right a)
  and `lim`: `filterlim (\lambda x. (f' x / g' x))` at-bot (at-right a)
shows `filterlim (\lambda x. f x / g x)` at-bot (at-right a)
proof –
  from `ev(2)` have `ev`: eventually \((\lambda x. DERIV (\lambda x. -g x)) x > -g' x\) (at-right a)
    by eventually-elim (auto intro: derivative-intros)
  have `filterlim (\lambda x. f x / (-g x))` at-top (at-right a)
    by (rule `lhopital-right-at-top-at-top[where f' = f' and g' = \lambda x. -g' x]`)
        (insert assms `ev`, auto simp: `filterlim-uminus-at-bot`)
  hence `filterlim (\lambda x. -(f x / g x))` at-top (at-right a) by simp
  thus `?thesis` by (simp add: `filterlim-uminus-at-bot`)
qed

lemma `lhopital-left-at-top-at-top`:
fixes `f` `g` :: `real` \Rightarrow `real`
assumes `f-0`: `LIM x at-left a. f x >` at-top
assumes `g-0`: `LIM x at-left a. g x >` at-top
  and ev:
    eventually \((\lambda x. DERIV f x > f' x)\) (at-left a)
    eventually \((\lambda x. DERIV g x > g' x)\) (at-left a)
  and `lim`: `filterlim (\lambda x. (f' x / g' x))` at-top (at-left a)
shows `filterlim (\lambda x. f x / g x)` at-top (at-left a)
  by (insert assms, unfold eventually-at-left-to-right `filterlim-at-left-to-right DERIV-mirror`,
      rule `lhopital-right-at-top-at-top[where f' = \lambda x. -f' (- x)]`
      (insert assms, auto simp: `DERIV-mirror`)

lemma `lhopital-left-at-top-at-bot`:
fixes `f` `g` :: `real` \Rightarrow `real`
assumes `f-0`: `LIM x at-left a. f x >` at-top
assumes \( g-0 \): \( \lim x \) at-left \( a \). \( g x :> \) at-bot

and ev:

\( \text{eventually } (\lambda x. \text{DERIV } f x :> f' x) \) (at-left \( a \))

\( \text{eventually } (\lambda x. \text{DERIV } g x :> g' x) \) (at-left \( a \))

and \( \text{lim: filterlim } (\lambda x. (f' x / g' x)) \) at-bot (at-left \( a \))

shows \( \text{filterlim } (\lambda x. f x / g x) \) at-bot (at-left \( a \))

by (insert assms, unfold eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror),

\( \text{rule lhopital-right-at-top-at-bot}\) (where \( f' = \lambda x. f' (- x) \))

(insert assms, auto simp: DERIV-mirror)

lemma lhopital-at-top-at-top:

\( \text{fixes } f g :: \text{real } \Rightarrow \text{real} \)

assumes \( f-0 \): \( \lim x \) at \( a \). \( f x :> \) at-top

assumes \( g-0 \): \( \lim x \) at \( a \). \( g x :> \) at-top

and ev:

\( \text{eventually } (\lambda x. \text{DERIV } f x :> f' x) \) (at \( a \))

\( \text{eventually } (\lambda x. \text{DERIV } g x :> g' x) \) (at \( a \))

and \( \text{lim: filterlim } (\lambda x. (f' x / g' x)) \) at-top (at \( a \))

shows \( \text{filterlim } (\lambda x. f x / g x) \) at-top (at \( a \))

using assms unfolding eventually-at-split filterlim-at-split

by (auto intro: lhopital-right-at-top-at-top[of \( f \) \( a \) \( g \) \( f' \) \( g' \)])

lhopital-left-at-top-at-top[of \( f \) \( a \) \( g \) \( f' \) \( g' \)])

lemma lhopital-at-top-at-bot:

\( \text{fixes } f g :: \text{real } \Rightarrow \text{real} \)

assumes \( f-0 \): \( \lim x \) at \( a \). \( f x :> \) at-top

assumes \( g-0 \): \( \lim x \) at \( a \). \( g x :> \) at-bot

and ev:

\( \text{eventually } (\lambda x. \text{DERIV } f x :> f' x) \) (at \( a \))

\( \text{eventually } (\lambda x. \text{DERIV } g x :> g' x) \) (at \( a \))

and \( \text{lim: filterlim } (\lambda x. (f' x / g' x)) \) at-bot (at \( a \))

shows \( \text{filterlim } (\lambda x. f x / g x) \) at-bot (at \( a \))

using assms unfolding eventually-at-split filterlim-at-split

by (auto intro: lhopital-right-at-top-at-bot[of \( f \) \( a \) \( g \) \( f' \) \( g' \)])

lhopital-left-at-top-at-bot[of \( f \) \( a \) \( g \) \( f' \) \( g' \)])

end

112 Nth Roots of Real Numbers

theory NthRoot

imports Deriv

begin

112.1 Existence of Nth Root

Existence follows from the Intermediate Value Theorem
lemma realpow-pos-nth:
  fixes a :: real  
  assumes n: 0 < n  
    and a: 0 < a  
  shows ∃r>0. r ^ n = a
proof -
  have ∃r≥0. r ≤ (max 1 a) ∧ r ^ n = a
  proof (rule IVT)
    show 0 ≤ r ≤ a
      using n a by (simp add: power-0-left)
    show 0 ≤ max 1 a
      by simp
    from n have n1: 1 ≤ n
      by simp
    have a ≤ max 1 a ^ 1
      by simp
    also have max 1 a ^ 1 ≤ max 1 a ^ n
      using n1 by (rule power-increasing) simp
    finally show a ≤ max 1 a ^ n .
  show ∀r. 0 ≤ r ∧ r ≤ max 1 a → isCont (λx. x ^ n) r
    by simp
  qed
then obtain r where r: 0 ≤ r ∧ r ^ n = a
  by fast
with n a have r ≠ 0
  by (auto simp add: power-0-left)
with r have 0 < r ∧ r ^ n = a
  by simp
then show ?thesis ..
qed

lemma realpow-pos-nth2: (0::real) < a ⇒ ∃r>0. r ^ Suc n = a
  by (blast intro: realpow-pos-nth)

Uniqueness of nth positive root.

lemma realpow-pos-nth-unique: 0 < n ⇒ 0 < a ⇒ ∃!r. 0 < r ∧ r ^ n = a for
  a :: real  
  by (auto intro!: realpow-pos-nth simp: power-eq-iff-eq-base)

112.2 Nth Root

We define roots of negative reals such that root n (- x) = - root n x. This
allows us to omit side conditions from many theorems.

lemma inj-sgn-power:
  assumes 0 < n
  shows inj (λy. sgn y * |y|^n :: real)
  (is inj ?f)
proof (rule injI)
  have \( x \coloneqq (0 < a \land b < 0) \lor (a < 0 \land 0 < b) \implies a \neq b \)
  for \( a \cdot b :: \mathbb{R} \)
  by auto

fix \( x \cdot y \)
assume \( \exists f \cdot x = y \)
with \( \text{power-eq-iff-eq-base[of } n \cdot \abs{x} \cdot \abs{y} \cdot \langle 0 < n \rangle \cdot \text{show } x = y \)
  (cases rule: linorder-cases[of 0 x, case-product linorder-cases[of 0 y]])
  (simp-all add: x)
qed

lemma sgn-power-injE:
  \( \text{sgn } a \cdot \abs{a} \cdot \hat{n} = x \implies x = \text{sgn } b \cdot \abs{b} \cdot \hat{n} \implies 0 < n \implies a = b \)
  for \( a \cdot b :: \mathbb{R} \)
  using \( \text{inj-sgn-power}[\text{THEN injD, of } a \cdot b] \) by simp

definition root :: nat \Rightarrow \mathbb{R} \Rightarrow \mathbb{R} where
  \( \text{root } n \cdot x \coloneqq (\text{if } n = 0 \text{ then } 0 \text{ else } \text{the-inv(\lambda y. \text{sgn } y \cdot \abs{y}^\hat{n}) } x) \)

lemma root-0 [simp]; root 0 \( x = 0 \)
  by (simp add: root-def)

lemma root-sgn-power; 0 < n \implies \text{root } n \cdot (\text{sgn } y \cdot \abs{y}^\hat{n}) = y
  using \( \text{the-inv-f-f[OF inj-sgn-power]} \) by (simp add: root-def)

lemma sgn-power-root:
  assumes 0 < n
  shows \( \text{sgn } (\text{root } n \cdot x) \cdot \abs{(\text{root } n \cdot x)} \cdot \hat{n} = x \)
  (is \( \exists f \cdot (\text{root } n \cdot x) = x \))
  proof (cases x = 0)
  case True then show \?thesis by simp
  next
  case False
  with \( \text{realpow-pos-nth[OF } 0 < n \cdot \text{, of } \abs{x}] \)
  obtain r \where \( 0 < r \cdot r \cdot \hat{n} = \abs{x} \)
  by auto
  with \( \langle x \neq 0 \rangle \cdot \text{have } S: x \in \text{range } \exists f \)
  (intro image-eqI[of - \text{ sgn } x \cdot r])
  (auto simp: abs-mult sgn-mult power-mult-distrib abs-sgn-eq mult-sgn-abs)
  from \( 0 < n \cdot \text{f-the-inv-into-f[OF inj-sgn-power[OF } 0 < n \cdot \text{ this]} \) \text{ show } \?thesis
  by (simp add: root-def)
qed

lemma split-root; \( P \cdot (\text{root } n \cdot x) \longleftrightarrow (n = 0 \longrightarrow P \cdot 0) \land (0 < n \longrightarrow (\forall y. \text{sgn } y \cdot \abs{y}^\hat{n} = x \longrightarrow P \cdot y)) \)
proof (cases n = 0)
  case True then show \?thesis by simp
next
  case False
  then show ?thesis
    by simp (metis root-sgn-power sgn-power-root)
qed

lemma real-root-zero [simp]: root n 0 = 0
  by (simp split: split-root add: sgn-zero-iff)

lemma real-root-minus: root n (− x) = − root n x
  by (clarsimp split: split-root elim: sgn-power-injE simp: sgn-minus)

lemma real-root-less-mono: 0 < n ⇒ x < y ⇒ root n x < root n y
  proof (clarsimp split: split-root)
    have ∗: 0 < b ⇒ a < 0 ⇒ ¬ a > b for a b :: real
      by auto
    fix a b :: real
    assume 0 < n sgn a * |a| ^ n < sgn b * |b| ^ n
    then show a < b
      using power-less-imp-less-base[of a b]
      power-less-imp-less-base[of − b n − a]
      by (simp add: sgn-real-def * [of a ^ n − ((− b) ^ n)]
      split: if-split-asm)
qed

lemma real-root-gt-zero: 0 < n ⇒ 0 < x ⇒ 0 < root n x
  using real-root-less-mono[of n 0 x] by simp

lemma real-root-ge-zero: 0 ≤ x ⇒ 0 ≤ root n x
  using real-root-gt-zero[of n x]
  by (cases n = 0) (auto simp add: le-less)

lemma real-root-pow-pos: 0 < n ⇒ 0 < x ⇒ root n x ^ n = x
  using sgn-power-root[of n x] real-root-gt-zero[of n x] by simp

lemma real-root-pow-pos2 [simp]: 0 < n ⇒ 0 ≤ x ⇒ root n x ^ n = x
  by (auto simp add: order-ge-less real-root-pow-pos)

lemma sgn-root: 0 < n ⇒ sgn (root n x) = sgn x
  by (auto split: split-root simp: sgn-real-def)

lemma odd-real-root-pow: odd n ⇒ root n x ^ n = x
  using sgn-power-root[of n x]
  by (simp add: odd-pos sgn-real-def split: if-split-asn)

lemma real-root-power-cancel: 0 < n ⇒ 0 ≤ x ⇒ root n (x ^ n) = x
  using root-sgn-power[of n x] by (auto simp add: le-less power-0-left)

lemma odd-real-root-power-cancel: odd n ⇒ root n (x ^ n) = x
Roots of multiplication and division.

**lemma** real-root-mult: $\text{root } n (x \ast y) = \text{root } n x \ast \text{root } n y$

by (auto split: split-root elim!: sgn-power-injE)

Root function is strictly monotonic, hence injective.

**lemma** real-root-pos-unique: $0 < n \Rightarrow 0 \leq y \Rightarrow y \sim n = x \Rightarrow \text{root } n x = y$

**using** root-sgn-power[of n x] by (simp add: odd-pos sgn-real-def power-0-left split: if-split-asm)

**lemma** real-root-eq-1-iff: $y \sim n = x \iff \text{root } n x = y$

by (erule subst)

**lemma** real-root-eq-0-iff: $y \sim 0 = x \iff \text{root } n x = y$

by (auto simp add: root-sgn-power)

**lemma** real-root-ge-1-iff: $y \sim n = x \iff \text{root } n x \leq y$

by auto

**lemma** real-root-le-1-iff: $y \sim n = x \iff \text{root } n x \geq y$

by auto

**lemma** real-root-le-0-iff: $y \sim 0 = x \iff \text{root } n x \leq y$

by auto

**lemma** real-root-ge-0-iff: $y \sim 0 = x \iff \text{root } n x \geq y$

by auto

**lemma** real-root-lt-1-iff: $y \sim n = x \iff \text{root } n x > y$

by (auto simp add: root-sgn-power)

**lemma** real-root-gt-1-iff: $y \sim n = x \iff \text{root } n x < y$

by (auto simp add: root-sgn-power)

**lemma** real-root-lt-0-iff: $y \sim 0 = x \iff \text{root } n x < y$

by (auto simp add: root-sgn-power)

**lemma** real-root-gt-0-iff: $y \sim 0 = x \iff \text{root } n x > y$

by (auto simp add: root-sgn-power)

Roots of multiplication and division.

**lemma** real-root-mult: $\text{root } n (x \ast y) = \text{root } n x \ast \text{root } n y$

by (auto split: split-root elim!: sgn-power-injE)
THEORY “NthRoot”

```
simp: sgn-mult abs-mult power-mult-distrib)

lemma real-root-inverse: root n (inverse x) = inverse (root n x)
by (auto split: split-root elim!: sgn-power-injE
  simp: power-inverse)

lemma real-root-divide: root n (x / y) = root n x / root n y
by (simp add: divide-inverse real-root-mult real-root-inverse)

lemma real-root-abs: 0 < n ⇒ root n |x| = |root n x|
by (simp add: abs-if real-root-minus)

lemma root-abss: n > 0 ⇒ abs (root n (y ^ n)) = abs y
using root-sgn-power [of n]
by (metis abs-ge-zero power-abs real-root-abs real-root-power-cancel)

lemma real-root-power: 0 < n ⇒ root n (x ^ k) = root n x ^ k
by (induct k) (simp-all add: real-root-mult)

Roots of roots.

lemma real-root-Suc-0 [simp]: root (Suc 0) x = x
by (simp add: add-real-root-unique)

lemma real-root-mult-exp: root (m * n) x = root m (root n x)
by (auto split: split-root elim!: sgn-power-injE
  simp: sgn-zero-iff sgn-mult power-mult [symmetric]
  abs-mult power-mult-distrib abs-sgn-equ)

lemma real-root-commute: root m (root n x) = root n (root m x)
by (simp add: real-root-mult-exp [symmetric] mult.commute)

Monotonicity in first argument.

lemma real-root-strict-decreasing:
  assumes 0 < n n < N 1 < x
  shows root N x < root n x
proof –
  from assms have root n (root N x) ^ n < root N (root n x) ^ n
  by (simp add: real-root-commute power-strict-increasing del: real-root-pow-pos2)
  with assms show ?thesis by simp
qed

lemma real-root-strict-increasing:
  assumes 0 < n n < N 0 < x x < 1
  shows root n x < root N x
proof –
  from assms have root N (root n x) ^ n < root n (root N x) ^ n
  by (simp add: real-root-commute power-strict-decreasing del: real-root-pow-pos2)
  with assms show ?thesis by simp
qed
```
THEORY “NthRoot”

lemma real-root-decreasing: $0 < n \implies n \leq N \implies 1 \leq x \implies \text{root } N \ x \leq \text{root } n \ x$
by (auto simp add: order-le-less real-root-strict-decreasing)

lemma real-root-increasing: $0 < n \implies n \leq N \implies 0 \leq x \implies x \leq 1 \implies \text{root } n \ x \leq \text{root } N \ x$
by (auto simp add: order-le-less real-root-strict-increasing)

Continuity and derivatives.

lemma isCont-real-root: $\text{isCont } (\text{root } n) \ x$
proof (cases $n > 0$
  case True
  let $?f = \lambda y :: \text{real}. \ sgn \ y \ast |y|^n$
  have continuous-on $\{0..\} \cup \{..0\}$ $(\lambda x. \text{if } 0 < x \text{ then } x \ast n \text{ else } ((-x) \ast n)$ :: real
  using True by (intro continuous-on-If continuous-intros) auto
  then have continuous-on UNIV $?f$
  by (rule continuous-on-cong[THEN iffD1, rotated 2]) (auto simp: not-less le-less True)
  then have [simp]: $\text{isCont } ?f \ x$ for $x$
  by (simp add: continuous-on-eq-continuous-at)
  have $\text{isCont } (\text{root } n) \ (\text{if } \text{root } n \ x)$
  by (rule isCont-inverse-function [where $f=\text{if } d=1\text{ then } f\text{ else } f$]) (auto simp: root-sgn-power True)
  then show ?thesis
  by (simp add: sgn-power-root True)
next
  case False
  then show ?thesis
  by (simp add: root-def[abs-def])
qed

lemma tendsto-real-root [tendsto-intros]:
$(f \longrightarrow x) \ F \implies ((\lambda x. \text{root } n \ (f \ x)) \longrightarrow \text{root } n \ x) \ F$
using isCont-tendsto-compose[OF isCont-real-root, of $f \ x \ F$].

lemma continuous-real-root [continuous-intros]:
$\text{continuous } F \ f \implies \text{continuous } F \ (\lambda x. \text{root } n \ (f \ x))$
unfolding continuous-def by (rule tendsto-real-root)

lemma continuous-on-real-root [continuous-intros]:
$\text{continuous-on } s \ f \implies \text{continuous-on } s \ (\lambda x. \text{root } n \ (f \ x))$
unfolding continuous-on-def by (auto intro: tendsto-real-root)

lemma DERIV-real-root:
assumes $n: 0 < n$
and $x: 0 < x$
shows $\text{DERIV } (\text{root } n) \ x :\ast \text{inverse } (\text{real } n \ast \text{root } n \ x \ast (n - \text{Suc } 0))$
proof (rule DERIV-inverse-function)
show $0 < x$
  using $x$.
show $x < x + 1$
  by simp
show $\text{DERIV} (\lambda x. x ^ n) (\text{root n} \ x) :> \text{real n} * \text{root n} \ x ^ {(n - \text{Suc 0})}$
  by (rule DERIV-pow)
show $\text{real n} * \text{root n} \ x ^ {(n - \text{Suc 0})} \neq 0$
  using $n \ x$ by simp
show $\text{isCont} (\text{root n}) \ x$
  by (rule isCont-real-root)
qed (use $n$ in auto)

lemma $\text{DERIV-odd-real-root}$:
assumes $n$: odd $n$
and $x$: $x \neq 0$
shows $\text{DERIV} (\text{root n} \ x) :> \text{inverse} (\text{real n} * \text{root n} \ x ^ {(n - \text{Suc 0})})$
proof (rule DERIV-inverse-function)
show $x - 1 < x < x + 1$
  by auto
show $\text{DERIV} (\lambda x. x ^ n) (\text{root n} \ x) :> \text{real n} * \text{root n} \ x ^ {(n - \text{Suc 0})}$
  by (rule DERIV-pow)
show $\text{real n} * \text{root n} \ x ^ {(n - \text{Suc 0})} \neq 0$
  using odd-pos [OF $n$] $x$ by simp
show $\text{isCont} (\text{root n}) \ x$
  by (rule isCont-real-root)
qed (use $n$ odd-real-root-pow in auto)

lemma $\text{DERIV-even-real-root}$:
assumes $n$: $0 < n$
and even $n$
and $x$: $x < 0$
shows $\text{DERIV} (\text{root n} \ x) :> \text{inverse} (- \text{real n} * \text{root n} \ x ^ {(n - \text{Suc 0})})$
proof (rule DERIV-inverse-function)
show $x - 1 < x$
  by simp
show $x < 0$
  using $x$.
show $- (\text{root n} \ y ^ n) = y$ if $x - 1 < y$ and $y < 0$ for $y$
proof -
  have $\text{root n} \ (-y) ^ n = -y$
    using (that $0 < n$) by simp
  with real-root-minus and (even $n$)
  show $- (\text{root n} \ y ^ n) = y$ by simp
qed
show $\text{DERIV} (\lambda x. - (x ^ n)) (\text{root n} \ x) :> - \text{real n} * \text{root n} \ x ^ {(n - \text{Suc 0})}$
  by (auto intro!: derivative-eq-intros)
show $- \text{real n} * \text{root n} \ x ^ {(n - \text{Suc 0})} \neq 0$
  using $n \ x$ by simp
show $\text{isCont} (\text{root n}) \ x$
by (rule isCont-real-root)

qed

lemma DERIV-real-root-generic:
assumes \( 0 < n \)
and \( x \neq 0 \)
and even \( n \) \( \implies \) \( 0 < x \implies D = \text{inverse} (\text{real } n \times \text{root } n \times (n - \text{Suc } 0)) \)
and even \( n \) \( \implies \) \( x < 0 \implies D = -\text{inverse} (\text{real } n \times \text{root } n \times (n - \text{Suc } 0)) \)
and odd \( n \) \( \implies \) \( D = \text{inverse} (\text{real } n \times \text{root } n \times (n - \text{Suc } 0)) \)
shows \( \text{DERIV} (\text{root } n) x :\to D \)
using assms
by (cases even \( n \), cases \( 0 < x \))
(auto intro: \text{DERIV}-real-root[THEN \text{DERIV-cong}]
\text{DERIV-odd-real-root}[THEN \text{DERIV-cong}]
\text{DERIV-even-real-root}[THEN \text{DERIV-cong}])

lemma power-tendsto-0-iff [simp]:
fixes \( f : 'a \to \text{real} \)
assumes \( n > 0 \)
shows \( ((\lambda x. f \times x ^ n) \to \to 0) F \iff (f \to \to 0) F \)
proof -
have \( ((\lambda x. |\text{root } n \times (f \times x ^ n)|) \to \to 0) F \iff (f \to \to 0) F \)
by (auto simp: assms root-abs-power tendsto-rabs-zero-iff)
then have \( ((\lambda x. f \times x ^ n) \to \to 0) F \iff (f \to \to 0) F \)
by (metis \text{tendsto-}real-root \text{abs-0} \text{real-root-zero} tendsto-rabs)
with assms show \( ?\text{thesis} \)
by (auto simp: \text{tendsto-null-power})
qed

112.3 Square Root

definition \text{sqrt} :: \text{real} \to \text{real} 
where \text{sqrt} = \text{root } 2

lemma pos2: \( 0 < (2::\text{nat}) \)
by simp

lemma real-sqrt-unique: \( y ^ 2 = x \implies 0 \leq y \implies \text{sqrt } x = y \)
unfolding \text{sqrt-def} by (rule \text{real-root-pos-unique} [OF \text{pos2}])

lemma real-sqrt-abs [simp]: \text{sqrt} \( (x ^ 2) = |x| \)
by (metis power2-abs-abs-ge-zero real-sqrt-unique)

lemma real-sqrt-pow2 [simp]: \( 0 \leq x \implies (\text{sqrt } x) ^ 2 = x \)
unfolding \text{sqrt-def} by (rule \text{real-root-pow2} [OF \text{pos2}])

lemma real-sqrt-pow2-iff [simp]: \( (\text{sqrt } x) ^ 2 = x \iff 0 \leq x \)
by (metis real-sqrt-pow2 zero-le-power2)
lemma real-sqrt-zero [simp]: \( \sqrt{0} = 0 \)
unfolding sqrt-def by (rule real-root-zero)

lemma real-sqrt-one [simp]: \( \sqrt{1} = 1 \)
unfolding sqrt-def by (rule real-root-one [OF pos2])

lemma real-sqrt-four [simp]: \( \sqrt{4} = 2 \)
using real-sqrt-abs[of 2] by simp

lemma real-sqrt-minus: \( \sqrt{-x} = -\sqrt{x} \)
unfolding sqrt-def by (rule real-root-minus)

lemma real-sqrt-mult: \( \sqrt{xy} = \sqrt{x} \cdot \sqrt{y} \)
unfolding sqrt-def by (rule real-root-mult)

lemma real-sqrt-mult-self [simp]: \( \sqrt{a} \cdot \sqrt{a} = |a| \)

lemma real-sqrt-inverse: \( \sqrt{\frac{1}{x}} = \frac{1}{\sqrt{x}} \)
unfolding sqrt-def by (rule real-root-inverse)

lemma real-sqrt-divide: \( \sqrt{\frac{x}{y}} = \sqrt{x} / \sqrt{y} \)
unfolding sqrt-def by (rule real-root-divide)

lemma real-sqrt-power: \( \sqrt{x^k} = \sqrt{x}^k \)
unfolding sqrt-def by (rule real-root-power [OF pos2])

lemma real-sqrt-gt-zero: \( 0 < x \Rightarrow 0 < \sqrt{x} \)
unfolding sqrt-def by (rule real-root-gt-zero [OF pos2])

lemma real-sqrt-ge-zero: \( 0 \leq x \Rightarrow 0 \leq \sqrt{x} \)
unfolding sqrt-def by (rule real-root-ge-zero)

lemma real-sqrt-less-mono: \( x < y \Rightarrow \sqrt{x} < \sqrt{y} \)
unfolding sqrt-def by (rule real-root-less-mono [OF pos2])

lemma real-sqrt-le-mono: \( x \leq y \Rightarrow \sqrt{x} \leq \sqrt{y} \)
unfolding sqrt-def by (rule real-root-le-mono [OF pos2])

lemma real-sqrt-less-iff [simp]: \( \sqrt{x} < \sqrt{y} \iff x < y \)
unfolding sqrt-def by (rule real-root-less-iff [OF pos2])

lemma real-sqrt-le-iff [simp]: \( \sqrt{x} \leq \sqrt{y} \iff x \leq y \)
unfolding sqrt-def by (rule real-root-le-iff [OF pos2])

lemma real-sqrt-eq-iff [simp]: \( \sqrt{x} = \sqrt{y} \iff x = y \)
unfolding sqrt-def by (rule real-root-eq-iff [OF pos2])

lemma real-less-lsqr: \( 0 \leq y \Rightarrow x < y^2 \Rightarrow \sqrt{x} < y \)
using real-sqrt-less-iff[of x y^2] by simp

lemma real-le-lsqrt: 0 ≤ y → x ≤ y^2 → sqrt x ≤ y
using real-sqrt-le-iff[of x y^2] by simp

lemma real-sqrt-le-iff[of x y^2] by simp

lemma real-le-rsqrt: x^2 ≤ y =⇒ x ≤ sqrt y
using real-sqrt-le-mono[of x^2 y] by simp

lemma real-less-rsqrt: x^2 < y =⇒ x < sqrt y
using real-sqrt-less-mono[of x^2 y] by simp

lemma real-sqrt-power-even:
assumes even n x ≥ 0
shows sqrt x ^ n = x ^ (n div 2)
proof −
  from assms obtain k where n = 2 * k by (auto elim!: evenE)
  with assms show ?thesis by (simp add: power-mult)
qed

lemma sqrt-le-D: sqrt x ≤ y =⇒ x ≤ y^2
by (meson not-le real-less-rsqrt)

lemma sqrt-ge-absD: |x| ≤ sqrt y =⇒ x^2 ≤ y
using real-sqrt-le-iff[of x^2] by simp

lemma sqrt-even-pow2:
assumes n: even n
shows sqrt (2 ^ n) = 2 ^ (n div 2)
proof −
  from n obtain m where m: n = 2 * m ..
  from m have sqrt (2 ^ n) = sqrt ((2 ^ m)^2)
    by (simp only: power-mult[symmetric] mult.commute)
  then show ?thesis
    using m by simp
qed

lemmas real-sqrt-gt-0-iff [simp] = real-sqrt-less-iff [where x=0, unfolded real-sqrt-zero]
lemmas real-sqrt-lt-0-iff [simp] = real-sqrt-less-iff [where y=0, unfolded real-sqrt-zero]
lemmas real-sqrt-ge-0-iff [simp] = real-sqrt-le-iff [where x=0, unfolded real-sqrt-zero]
lemmas real-sqrt-le-0-iff [simp] = real-sqrt-le-iff [where y=0, unfolded real-sqrt-zero]
lemmas real-sqrt-eq-0-iff [simp] = real-sqrt-eq-iff [where y=0, unfolded real-sqrt-zero]
lemmas real-sqrt-gt-1-iff [simp] = real-sqrt-less-iff [where x=1, unfolded real-sqrt-one]
lemmas real-sqrt-lt-1-iff [simp] = real-sqrt-less-iff [where y=1, unfolded real-sqrt-one]
lemmas real-sqrt-ge-1-iff [simp] = real-sqrt-le-iff [where x=1, unfolded real-sqrt-one]
lemmas real-sqrt-le-1-iff [simp] = real-sqrt-le-iff [where y=1, unfolded real-sqrt-one]
lemmas real-sqrt-eq-1-iff [simp] = real-sqrt-eq-iff [where y=1, unfolded real-sqrt-one]

lemma sqrt-add-le-add-sqrt:
assumes $0 \leq x \leq y$
shows $\sqrt{x + y} \leq \sqrt{x} + \sqrt{y}$
by (rule power2-le-imp-le) (simp-all add: power2-sum assms)

lemma isCont-real-sqrt: isCont $\sqrt{x}$
  unfolding sqrt-def by (rule isCont-real-root)

lemma tendsto-real-sqrt [tendsto-intros]:
  $(f \longrightarrow x) \Longrightarrow ((\lambda x. \sqrt{f x}) \longrightarrow \sqrt{x}) F$
  unfolding sqrt-def by (rule tendsto-real-root)

lemma continuous-real-sqrt [continuous-intros]:
  continuous $F f \Longrightarrow$ continuous $F (\lambda x. \sqrt{f x})$
  unfolding sqrt-def by (rule continuous-real-root)

lemma continuous-on-real-sqrt [continuous-intros]:
  continuous-on $s f \Longrightarrow$ continuous-on $s (\lambda x. \sqrt{f x})$
  unfolding sqrt-def by (rule continuous-on-real-root)

lemma DERIV-real-sqrt-generic:
  assumes $x \neq 0$
  and $x > 0 \Longrightarrow D = \text{inverse} (\sqrt{x}) / 2$
  and $x < 0 \Longrightarrow D = -\text{inverse} (\sqrt{x}) / 2$
  shows DERIV $\sqrt{x} := D$
  using assms unfolding sqrt-def
  by (auto intro: DERIV-real-root-generic)

lemma DERIV-real-sqrt: $0 < x \Longrightarrow$ DERIV $\sqrt{x} := \text{inverse} (\sqrt{x}) / 2$
  using DERIV-real-sqrt-generic by simp

declare DERIV-real-sqrt-generic[THEN DERIV-chain2, derivative-intros]
DERIV-real-root-generic[THEN DERIV-chain2, derivative-intros]

lemmas has-derivative-real-sqrt[derivative-intros] = DERIV-real-sqrt[THEN DERIV-compose-FDERIV]

lemma not-real-square-gt-zero [simp]: $\neg 0 < x \ast x \longleftrightarrow x = 0$
  for $x :: \text{real}$
  apply auto
  using linorder-less-linear [where $x = x$ and $y = 0$]
  apply (simp add: zero-less-mult-iff)
done

lemma real-sqrt-abs2 [simp]: $\sqrt{x \ast x} = |x|$
  apply (subst power2-eq-square [symmetric])
  apply (rule real-sqrt-abs)
done
lemma real-inv-sqrt-pow2: $0 < x \Longrightarrow (inverse (sqrt x))^2 = inverse x$
  by (simp add: power-inverse)

lemma real-sqrt-eq-zero-cancel: $0 \leq x \Longrightarrow sqrt x = 0 \Longrightarrow x = 0$
  by simp

lemma real-sqrt-ge-one: $1 \leq x \Longrightarrow 1 \leq sqrt x$
  by simp

lemma sqrt-divide-self-eq:
  assumes nneg: $0 \leq x$
  shows $sqrt x / x = inverse (sqrt x)$
proof (cases x = 0)
  case True
  then show ?thesis by simp
next
  case False
  then have pos: $0 < x$ using nneg by arith
  show ?thesis
  proof
    (rule right-inverse-eq [THEN iffD1, symmetric])
      show $sqrt x / x \neq 0$
        by (simp add: divide-inverse nneg False)
      show $inverse (sqrt x) / (sqrt x / x) = 1$
        by (simp add: divide-inverse mult.assoc [symmetric]
          power2-eq-square [symmetric] real-inv-sqrt-pow2 pos False)
  qed
qed

lemma real-div-sqrt:
  $0 \leq x \Longrightarrow x / sqrt x = sqrt x$
  by (cases x = 0) (simp-all add: sqrt-divide-self-eq [of x] field-simps)

lemma real-divide-square-eq [simp]: $(r * a) / (r * r) = a / r$
  for a r :: real
  by (cases r = 0) (simp-all add: divide-inverse ac-simps)

lemma lemma-real-divide-sqrt-less: $0 < u \Longrightarrow u / sqrt 2 < u$
  by (simp add: divide-less-eq)

lemma four-x-squared: $4 * x^2 = (2 * x)^2$
  for x :: real
  by (simp add: power2-eq-square)

lemma sqrt-at-top: LIM x at-top. sqrt x :: real ::> at-top
  by (rule filterlim-at-top-at-top[where Q=\lambda x. True and P=\lambda x. 0 < x and g=power2])
    (auto intro: eventually-gt-at-top)
112.4 Square Root of Sum of Squares

lemma sum-squares-bound: \(2 \times x \times y \leq x^2 + y^2\)
for \(x\) \(y\) :: 'a::linordered-field
proof –
  have \((x - y)^2 = x \times x - 2 \times x \times y + y \times y\)
  by algebra
  then have \(0 \leq x^2 - 2 \times x \times y + y^2\)
  by (metis sum-power2-ge-zero zero-le-double-add-iff-zero-le-single-add power2-eq-square)
  then show \(?thesis
  by arith
qed

lemma arith-geo-mean:
  fixes \(u\) :: 'a::linordered-field
  assumes \(u^2 = x \times y\) \(x \geq 0\) \(y \geq 0\)
  shows \(u \leq (x + y)/2\)
  apply (rule power2-le-imp-le)
  using sum-squares-bound assms
  apply (auto simp: zero-le-mult-iff)
  apply (auto simp: algebra-simps power2-eq-square)
  done

lemma arith-geo-mean-sqrt:
  fixes \(x\) :: real
  assumes \(x \geq 0\) \(y \geq 0\)
  shows \(\sqrt{x \times y} \leq (x + y)/2\)
  apply (rule arith-geo-mean)
  using assms
  apply (auto simp: zero-le-mult-iff)
  done

lemma real-sqrt-sum-squares-mult-ge-zero [simp]: \(0 \leq \sqrt{(x^2 + y^2) \times (xa^2 + ya^2)}\)
  by (metis real-sqrt-ge-0-iff split-mult-pos-le sum-power2-ge-zero)

lemma real-sqrt-sum-squares-mult-squared-eq [simp]: \((\sqrt{(x^2 + y^2) \times (xa^2 + ya^2)})^2 = (x^2 + y^2) \times (xa^2 + ya^2)\)
  by (simp add: zero-le-mult-iff)

lemma real-sqrt-sum-squares-eq-cancel: \(\sqrt{x^2 + y^2} = x \implies y = 0\)
  by (drule arg-cong [where \(f = \lambda x. x^2\)]) simp

lemma real-sqrt-sum-squares-eq-cancel2: \(\sqrt{x^2 + y^2} = y \implies x = 0\)
  by (drule arg-cong [where \(f = \lambda x. x^2\)]) simp

lemma real-sqrt-sum-squares-eq1 [simp]: \(x \leq \sqrt{x^2 + y^2}\)
  by (rule power2-le-imp-le) simp-all

lemma real-sqrt-sum-squares-eq2 [simp]: \(y \leq \sqrt{x^2 + y^2}\)
by (rule power2-le-imp-le) simp-all

lemma real-sqrt-ge-abs1 [simp]: |x| ≤ sqrt (x² + y²)
  by (rule power2-le-imp-le) simp-all

lemma real-sqrt-ge-abs2 [simp]: |y| ≤ sqrt (x² + y²)
  by (rule power2-le-imp-le) simp-all

lemma le-real-sqrt-sumsq [simp]: x ≤ sqrt (x² + y²)
  by (rule power2-le-imp-le) simp-all

lemma sqrt-sum-squares-le-sum-abs:
  [0 ≤ x; 0 ≤ y] ⇒ sqrt (x² + y²) ≤ |x| + |y|
  by (rule power2-le-imp-le) (simp-all add: power2-sum)

lemma sqrt-sum-squares-less:
  |x| < u / sqrt 2 ⇒ |y| < u / sqrt 2 ⇒ sqrt (x² + y²) < u
  apply (rule power2-less-imp-less)
  apply simp
  apply (drule power-strict-mono [OF - abs-ge-zero pos2])
  apply (drule power-strict-mono [OF - abs-ge-zero pos2])
  apply (rule power2-le-imp-le) simp-all
  then have (a + c)² + (b + d)² ≤ (sqrt (a² + b²))² + (sqrt (c² + d²))²
    by (simp add: power2-sum)
  then show ?thesis
    by (auto intro: power2-le-imp-le)
  qed
apply (simp add: power-divide)
apply (erule order-le-less-trans [OF abs-ge-zero])
apply (simp add: zero-less-divide-iff)
done

lemma sqrt2-less-2: sqrt 2 < (2 :: real)
  by (metis not-less not-less-iff-gr-or-eq numeral-less-iff real-sqrt-four
       real-sqrt-le-iff semiring-norm(75) semiring-norm(78) semiring-norm(85))

lemma sqrt-sum-squares-half-less:
x < u / 2 =⇒ y < u / 2 =⇒ 0 ≤ x =⇒ 0 ≤ y =⇒ sqrt (x^2 + y^2) < u
  apply (rule real-sqrt-sum-squares-less)
  apply (auto simp add: abs-if field-simps)
  apply (rule le-less-trans [where y = x * 2])
  using less-eq-real-def sqrt2-less-2
  apply force
  apply assumption
  apply (rule le-less-trans [where y = y * 2])
  using less-eq-real-def sqrt2-less-2
  mult-le-cancel-left
  apply auto
  done

lemma LIMSEQ-root: (λ n. root n n) −−−−→ 1
proof
  define x where x n = root n n - 1
  have x −−−−→ sqrt 0
    proof (rule tendsto-sandwich[of - - tendsto-const])
      show (λ x. sqrt (2 / x)) −−−−→ sqrt 0
        by (intro tendsto-intros tendsto-divide-0[of tendsto-const]
             filterlim-mono[of filterlim-real-sequentially])
        (simp-all add: at-infinity-eq-at-top-bot)
      have x n ≤ sqrt (2 / real n) if 2 < n for n :: nat
        proof
          have 1 + (real (n - 1) * n) / 2 * (x n)^2 = 1 + of-nat (n choose 2) * (x n)^2
            by (auto simp add: choose-two field-char-0-class.of-nat-div mod-eq-0-iff-dvd)
          also have ... ≤ (∑ k∈{0, 2}. of-nat (n choose k) * x n ^ k)
            by (simp add: x-def)
          also have ... ≤ (∑ k≤n. of-nat (n choose k) * x n ^ k)
            using (2 < n)
            by (intro sum-mono2) (auto intro!: mult-nonneg-nonneg zero-le-power simp:
                x-def le-diff-eq)
          also have ... = (x n + 1) ^ n
            by (simp add: binomial-ring)
          also have ... = n
            using (2 < n) by (simp add: x-def)
          finally have real (n - 1) * (real n / 2 * (x n)^2) ≤ real (n - 1) * 1
            by simp
          then have (x n)^2 ≤ 2 / real n
            using (2 < n) unfolding mult-le-cancel-left
            by (simp add: field-simps)
          from real-sqrt-le-mono[of this] show ?thesis
by simp

qed

ten show eventually (\( \lambda n. \ x \ n \leq \sqrt{2 / \text{real } n} \)) sequentially

by (auto intro!: exI[of - 3] simp: eventually-sequentially)

show eventually (\( \lambda n. \ \sqrt{0} \leq x \ n \)) sequentially

by (auto intro!: exI[of - 1] simp: eventually-sequentially le-diff-eq x-def)

qed

from tendsto-add[OF this tendsto-const[of 1]]

show ?thesis

by (simp add: x-def)

qed
A theorem about the factorial function on the reals.

**Lemma** square-fact-le-2-fact: fact n * fact n ≤ (fact (2 * n) :: real)

**Proof** (induct n)

- case 0
  - then show ?case by simp

- next
  - case (Suc n)
    - have (fact (Suc n)) * (fact (Suc n)) = of-nat (Suc n) * of-nat (Suc n) * (fact n * fact n :: real)
      - by (simp add: field-simps)
    - also have ... ≤ of-nat (Suc n) * of-nat (Suc n) * fact (2 * n)
      - by (rule mult-left-mono [OF Suc]) simp
    - also have ... ≤ of-nat (Suc (Suc (2 * n))) * of-nat (Suc (2 * n)) * fact (2 * n)
      - by (rule mult-right-mono) + (auto simp: field-simps)
    - also have ... = fact (2 * Suc n) by (simp add: field-simps)
    - finally show ?case.

qed
lemma fact-in-Reals: fact \( n \in \mathbb{R} \)
by (induction \( n \)) auto

lemma of-real-fact [simp]: of-real (fact \( n \)) = fact \( n \)
by (metis of-nat-fact of-real-of-nat-eq)

lemma pochhammer-of-real: pochhammer (of-real \( x \)) \( n \) = of-real (pochhammer \( x \) \( n \))
by (simp add: pochhammer-prod)

lemma norm-fact [simp]: norm (fact \( n \)) = fact \( n \)
proof
  have (fact \( n \)) = of-real (fact \( n \))
  also have norm . . . = fact \( n \)
  finally show ?thesis.
qed

lemma root-test-convergence:
  fixes \( f \) :: nat \( \Rightarrow \) real
  assumes \( f \) : (\( \lambda n \). root \( n \) (norm (f \( n \)))) \( \longrightarrow \) \( x \) — could be weakened to lim sup
  and \( x < 1 \)
  shows \( \text{summable } f \)
proof
  have \( 0 \leq x \)
  also have \( \text{norm } \leq \text{z}^n \) sequentially
  finally show ?thesis.
qed

113.1 More facts about binomial coefficients

These facts could have been proven before, but having real numbers makes the proofs a lot easier.
lemma central-binomial-odd:
odd n \Rightarrow n \choose (Suc \ (n \ div \ 2)) = n \choose (n \ div \ 2)
proof –
  assume odd n
  hence Suc (n \ div \ 2) \leq n by presburger
  hence n \choose (Suc (n \ div \ 2)) = n \choose (n - Suc (n \ div \ 2))
    by (rule binomial-symmetric)
  also from \langle odd n \rangle have n - Suc (n \ div \ 2) = n \ div \ 2 by presburger
  finally show thesis.
qed

lemma binomial-less-binomial-Suc:
assumes k: k < n \ div \ 2
shows n \choose k < n \choose (Suc k)
proof –
  from k have k': k \leq Suc k \leq n by simp-all
  from k' have real (n \ choose \ k) = fact n / (fact k * fact \ (n - k))
    by (simp add: binomial-fact)
  also from k' have n - k = Suc (n - Suc k) by simp
  also from k' have fact ... = (real n - real k) * fact \ (n - Suc k)
    by (subst fact-Suc) (simp-all add: of-nat-diff)
  also have fact n / (fact (Suc k) / (real k + 1) * ((real n - real k) * fact \ (n - Suc k))) =
    (n \ choose \ (Suc k)) * ((real k + 1) / (real n - real k))
    using k by (simp add: field-split-simps binomial-fact)
  also from assms have (real k + 1) / (real n - real k) < 1 by simp
  finally show thesis using k by (simp add: mult-less-cancel-left)
qed

lemma binomial-strict-mono:
assumes k < k' 2*k' \leq n
shows n \choose k < n \choose k'
proof –
  from assms have k \leq k' - 1 by simp
  thus thesis
proof (induction rule: inc-induct)
  case base
    with assms binomial-less-binomial-Suc[of k' - 1 n]
    show ?case by simp
next
  case (step k)
    from step.prems step.hyps assms have n \ choose \ k < n \ choose \ (Suc k)
      by (intro binomial-less-binomial-Suc) simp-all
    also have ... < n \ choose \ k' by (rule step.IH)
    finally show ?case.
qed

qed
lemma binomial-mono:
  assumes $k \leq k' \leq n$
  shows $\binom{n}{k} \leq \binom{n}{k'}$
  using assms binomial-strict-mono[of $k\ k'\ n$] by (cases $k = k'$) simp-all

lemma binomial-strict-antimono:
  assumes $k < k' \leq n$
  shows $\binom{n}{k} > \binom{n}{k'}$
  proof
    from assms have $(n - k) > (n - k')$
      by (intro binomial-strict-mono) (simp-all add: algebra-simps)
    with assms show ?thesis
      by (simp add: binomial-symmetric[symmetric])
  qed

lemma binomial-antimono:
  assumes $k \leq k' \leq \frac{n}{2}$
  shows $\binom{n}{k} \geq \binom{n}{k'}$
  proof (cases $k = k'$)
    case False
    note not-eq = False
    show ?thesis
      proof (cases $n \leq odd\ n$)
        case False
        with assms have $2k \leq n$
          by presburger
        with not-eq assms binomial-strict-antimono[of $k\ k'\ n$]
          show ?thesis
            by simp
      next
      case True
      have $k' \leq \frac{n}{2}$
        by (simp add: central-binomial-odd)
      proof (cases $k' = \frac{n}{2}$)
        case False
        with assms True not-eq have $\binom{n}{\frac{n}{2}} < k'$
          by simp
        with assms binomial-strict-antimono[of $\frac{n}{2}\ k'\ n$] True
          show ?thesis
            by auto
        qed simp-all
      also from True have ... = $\binom{n}{k}$
        by (simp add: central-binomial-odd)
      finally show ?thesis .
    qed
    qed simp-all

lemma binomial-maximum: $\binom{n}{k} \leq \binom{n}{\frac{n}{2}}$
  proof
    have $k \leq \frac{n}{2} \iff 2k \leq n$
      by linarith
    consider $2k \leq n$ | $2k \geq n$
      by linarith
    thus ?thesis
      proof cases
      case 1
      thus ?thesis
        by (intro binomial-mono) linarith
    next
  qed
THEORY “Transcendental” 2339

lemma powser-zero: (\sum n. f n * 0 ^ n) = f 0
  
  for f :: nat => 'a::real-normed-algebra-1

lemma powser-sums-zero: (\sum n. a n * 0 ^ n) sums a 0
for $a :: \mathbb{nat} \Rightarrow 'a::\mathbb{real-normed-div-algebra}$
using sums-finite [of $\{0\}$ $\lambda n. a n * 0 ^ {\sim} n$]
by simp

lemma powser-sums-zero-iff [simp]: $(\lambda n. a n * 0 ^ {\sim} n)$ sums $x \longleftrightarrow a 0 = x$
for $a :: \mathbb{nat} \Rightarrow 'a::\mathbb{real-normed-div-algebra}$
using powser-sums-zero sums-unique2 by blast

Power series has a circle or radius of convergence: if it sums for $x$, then it
sums absolutely for $z$ with $|z| < |x|$.

lemma powser-insidea:
fixes $x z :: 'a::\mathbb{real-normed-div-algebra}$
assumes $1$: summable $(\lambda n. f n * x ^ {\sim} n)$
and $2$: norm $z < \text{norm } x$
shows summable $(\lambda n. \text{norm } (f n * z ^ {\sim} n))$
proof −
from $2$ have $x-neq-0$: $x \neq 0$ by clarsimp
from $1$ have $(\lambda n. f n * x ^ {\sim} n) \limmap 0$
by (rule summable-LIMSEQ-zero)
then have convergent $(\lambda n. f n * x ^ {\sim} n)$
by (rule convergentI)
then have Cauchy $(\lambda n. f n * x ^ {\sim} n)$
by (rule convergent-Cauchy)
then have $Bseq (\lambda n. f n * x ^ {\sim} n)$
by (rule Cauchy-Bseq)
then obtain $K$ where $3$: $0 < K$ and $4$: $\forall n. \text{norm } (f n * z ^ {\sim} n) \leq K * \text{norm } (x ^ {\sim} n)$
by (auto simp: $Bseq$-def)
have $\exists N. \forall n \geq N. \text{norm } (f n * z ^ {\sim} n) \leq K * \text{norm } (x ^ {\sim} n)$
proof (intro exI allI impI)
fix $n :: \mathbb{nat}$
assume $0 \leq n$
have $\text{norm } (\text{norm } (f n * z ^ {\sim} n)) * \text{norm } (x ^ {\sim} n) =$
$\text{norm } (f n * x ^ {\sim} n) * \text{norm } (z ^ {\sim} n)$
by (simp add: norm-mult abs-mult)
also have $\ldots \leq K * \text{norm } (x ^ {\sim} n)$
by (simp only: mult-right-mono $4$ norm-ge-zero)
also have $\ldots = K * \text{norm } (z ^ {\sim} n) * (\text{inverse } (\text{norm } (x ^ {\sim} n)) * \text{norm } (x ^ {\sim} n))$
by (simp add: x-neq-0)
also have $\ldots = K * \text{norm } (z ^ {\sim} n) * \text{inverse } (\text{norm } (x ^ {\sim} n)) * \text{norm } (x ^ {\sim} n)$
by (simp only: mult.assoc)
finally show $\text{norm } (\text{norm } (f n * z ^ {\sim} n)) \leq K * \text{norm } (z ^ {\sim} n) * \text{inverse } (\text{norm } (x ^ {\sim} n))$
by (simp add: mult-le-cancel-right x-neq-0)
qed
moreover have summable $(\lambda n. K * \text{norm } (z ^ {\sim} n) * \text{inverse } (\text{norm } (x ^ {\sim} n)))$
proof −
from $2$ have $\text{norm } (\text{norm } (z * \text{inverse } x)) < 1$
using x-neq-0
by (simp add: norm-mult nonzero-norm-inverse divide-inverse [where 'a=real, symmetric])

then have summable (λn. norm (z * inverse x) ^ n)
  by (rule summable-geometric)
then have summable (λn. K * norm (z * inverse x) ^ n)
  by (rule summable-mult)
then show summable (λn. K * norm (z ^ n) * inverse (norm (x ^ n)))
  using x-neq-0
  by (simp add: norm-mult nonzero-norm-inverse power-mult-distrib
       power-inverse norm-power mult.assoc)
qed

ultimately show summable (λn. norm (f n * z ^ n))
  by (rule summable-comparison-test)
qed

lemma powser-inside:
  fixes f :: 'a::{real-normed-div-algebra,banach}
  shows summable (λn. f n * (x^n)) =⇒ norm z < norm x =⇒ summable (λn. f n * (z ^ n))
  by (rule powser-insidea [THEN summable-norm-cancel])

lemma powser-times-n-limit-0:
  fixes x :: 'a::{real-normed-div-algebra,banach}
  assumes norm x < 1
  shows (λn. of-nat n * x ^ n) −−−−→ 0
  proof
    have norm x / (1 - norm x) ≥ 0
      using assms by (auto simp: field-split-simps)
    moreover obtain N where N: norm x / (1 - norm x) < of-int N
      using ex-le-of-int by (meson ex-less-of-int)
    ultimately have N0: N>0
      by auto
    then have *: real-of-int (N + 1) * norm x / real-of-int N < 1
      using N assms by (auto simp: field-simps)
    have **: real-of-int N * (norm x * (real-of-nat (Suc n) * norm (x ^ n))) ≤
      real-of-int n * (norm x * ((1 + N) * norm (x ^ n))) if N ≤ int n for n :: nat
    proof
      from that have real-of-int N * real-of-int (Suc n) ≤ real-of-int n * real-of-int (1 + N)
        by (simp add: algebra-simps)
      then have (real-of-int N * real-of-int (Suc n)) * (norm x * norm (x ^ n)) ≤
        (real-of-int n * (1 + N)) * (norm x * norm (x ^ n))
        using N0 mult-mono by fastforce
      then show ?thesis
        by (simp add: algebra-simps)
    qed
    show ?thesis using *
      by (rule summable-LIMSEQ-zero [OF summable-ratio-test, where N1=nat N])
corollary  \( \lim n \over p^n : \) 

fixes \( x :: 'a::(real-normed-field,banach) \)

shows \( 1 < \text{norm } x \implies ((\lambda n. \text{of-nat } n / x^n) \longrightarrow 0) \) sequentially

using powser-times-n-limit-0 \([\text{of inverse } x] \)

by (simp add: norm-divide field-split-simps)

lemma sum-split-even-odd:

fixes \( f :: \text{n} \Rightarrow \text{real} \)

shows \((\sum i < 2 \ast n. \text{if even } i \text{ then } f i \text{ else } g i) = (\sum i < n. f (2 \ast i)) + (\sum i < n. g (2 \ast i + 1)) \)

proof (induct \( n \))

case \( 0 \)
then show \(?case\) by simp

next

case \((\text{Suc } n)\)

have \((\sum i < 2 \ast \text{Suc } n. \text{if even } i \text{ then } f i \text{ else } g i) = \)

\((\sum i < n. f (2 \ast i)) + (\sum i < n. g (2 \ast i + 1)) + f(2 \ast n) + g(2 \ast n + 1))\)

using Suc.hyps unfolding One-nat-def by auto

also have \(\ldots = (\sum i < \text{Suc } n. f (2 \ast i)) + (\sum i < \text{Suc } n. g (2 \ast i + 1))\)

by auto

finally show \(?case\).

qed
case True
then show ?thesis
  by (auto simp: even-two-times-div-two)
next
case False
then have eq: Suc (2 * (m div 2)) = m by simp
then have even (2 * (m div 2)) using odd m by auto
have ?SUM m = ?SUM (Suc (2 * (m div 2))) unfolding eq ..
also have ... = ?SUM (2 * (m div 2)) using even (2 * (m div 2)) by auto
finally show ?thesis by auto
qed
ultimately show ?thesis by auto
qed
then show ∃ no. ∀ m ≥ no. norm (?SUM m − x) < r
by blast
qed

lemma sums-if:
  fixes g :: nat ⇒ real
  assumes g sums x and f sums y
  shows (λ n. if even n then f (n div 2) else g ((n − 1) div 2)) sums (x + y)
proof —
  let ?s = λ n. if even n then 0 else f ((n − 1) div 2)
  have if-sum: (if B then 0 else E) + (if B then T else 0) = (if B then T else E)
    for B T E
    by (cases B)
  have g-sums: (λ n. if even n then 0 else g ((n − 1) div 2)) sums x
    using sums-if[OF g sums x]
  have if-eq: \B T E. (if \¬ B then T else E) = (if B then E else T)
    by auto
  have ?s sums y using sums-if[OF f sums y]
    from this[unfolded sums-def, THEN LIMSEQ-Suc]
  have (λ n. if even n then f (n div 2) else 0) sums y
    by (simp add: lessThan-Suc-insert-0 sum.atLeast1-atMost-eq image-Suc-lessThan
    if-eq sums-def cong del: if-weak-cong)
  from sums-add[OF g sums this] show ?thesis
    by (simp only: if-sum)
qed

113.3 Alternating series test / Leibniz formula

lemma sums-alternating-upper-lower:
  fixes a :: nat ⇒ real
  assumes mono: \A n. a (Suc n) ≤ a n
  and a-pos: \A n. 0 ≤ a n
  and a ⟹ 0
  shows ∃ l. ((∀ n. (\sum i<2*n. (− 1) ^ i * a i) ≤ l) \A (λ n. ∑ i<2*n. (− 1) ^ i * a i)
proof (rule nested-sequence-unique)
  have fg-diff: \(\forall n. \text{if } f n - g n = -a \times n \text{ by auto}\)
  show \(\forall n. \text{if } f n \leq g n \text{ for } n\)
    using mono[of 2*\(n\)] by auto
  qed
  show \(\forall n. \text{if } g (\text{Suc } n) \leq g n \text{ for } n\)
    using mono[of Suc (2*n)] by auto
  qed
  show \(\forall n. \text{if } f n \leq g n \text{ for } n\)
    using fg-diff a-pos by auto
  qed
  show \(\lambda n. \text{if } f n - g n \longrightarrow 0\)
  unfolding fg-diff
  proof (rule LIMSEQ-I)
    fix \(r : \text{real}\)
    assume \(0 < r\)
    with \(\alpha \longrightarrow 0\)[THEN LIMSEQ-D] obtain \(N \text{ where } \bigwedge n. n \geq N \implies \text{norm } (a \times n - 0) < r\)
      by auto
    then have \(\forall n \geq N. \text{norm } (a \times (2 \times n) - 0) < r\)
      by auto
    then show \(\exists N. \forall n \geq N. \text{norm } (a \times (2 \times n) - 0) < r\)
      by auto
  qed
  qed

lemma summable-Leibniz' :
fixes \(a : \text{nat} \Rightarrow \text{real}\)
assumes a-zero: \(a \longrightarrow 0\)
and a-pos: \(\bigwedge n. 0 \leq a n\)
and a-monotone: \(\bigwedge n. a (\text{Suc } n) \leq a n\)
shows summable: summable \((\lambda n. (-1)^{n \times a n})\)
  and \(\bigwedge n. \text{(sum } i<2\times n. (-1)^{\text{i} \times a i}) \leq \text{(sum } i. (-1)^{\text{i} \times a i})\)
  and \(\bigwedge n. \text{(sum } i<2\times n. (-1)^{\text{i} \times a i}) \longrightarrow \text{(sum } i. (-1)^{\text{i} \times a i})\)
  and \(\bigwedge n. \text{(sum } i<2\times n+1. (-1)^{\text{i} \times a i}) \leq \text{(sum } i \times 2\times n+1. (-1)^{\text{i} \times a i})\)
  and \(\bigwedge n. \text{(sum } i<2\times n+1. (-1)^{\text{i} \times a i}) \longrightarrow \text{(sum } i. (-1)^{\text{i} \times a i})\)
proof –
  let \(S = \lambda n. (-1)^{n \times a n}\)
let \( \mathcal{P} = \lambda n. \sum_{i < n} \mathcal{S} i \)
let \( \mathcal{F} = \lambda n. \mathcal{P} (2 \ast n) \)
let \( \mathcal{G} = \lambda n. \mathcal{P} (2 \ast n + 1) \)

obtain \( l :: \text{real} \)
where below-l: \( \forall n. \mathcal{F} n \leq l \)
and \( \mathcal{F} \rightarrow l \)
and above-l: \( \forall n. l \leq \mathcal{G} n \)
and \( \mathcal{G} \rightarrow l \)

using sums-alternating-upper-lower[OF a-monotone a-pos a-zero] by blast

let \( \mathcal{S} a = \lambda m. \sum_{n < m} \mathcal{S} n \)

have \( \mathcal{S} a \rightarrow l \)
proof (rule LIMSEQ-I)

fix \( r :: \text{real} \)
assume \( 0 < r \)

with \( \mathcal{F} \rightarrow l \) [THEN LIMSEQ-D]

obtain \( f \)-no where \( f: \bigwedge n. n \geq f\)-no \( \Rightarrow \) \( \text{norm} (\mathcal{F} n - l) < r \)
by auto

from \( 0 < r \) \( \& \& \mathcal{G} \rightarrow l \) [THEN LIMSEQ-D]

obtain \( g \)-no where \( g: \bigwedge n. n \geq g\)-no \( \Rightarrow \) \( \text{norm} (\mathcal{G} n - l) < r \)
by auto

have \( \text{norm} (\mathcal{S} a n - l) < r \) \( \text{if} \ n \geq \max (2 \ast f\)-no \( \& \& 2 \ast g\)-no) \) \( \text{for} \ n \)
proof

from that have \( n \geq 2 \ast f\)-no \( \& \& n \geq 2 \ast g\)-no by auto

show \( \exists \)thesis
proof (cases even \( n \))

  case True
  then have \( n\)-eq: \( 2 \ast (n \div 2) = n \)
  by (simp add: even-two-times-div-two)

  with \( n \geq 2 \ast f\)-no have \( n \div 2 \geq f\)-no
  by auto

  from \( f \)[OF this] show \( \exists \)thesis

  unfolding \( n\)-eq \text{atLeastLessThanSuc-atLeastAtMost}.

next

  case False

  then have \( \text{even} (n - 1) \) by simp

  then have \( n\)-eq: \( 2 \ast ((n - 1) \div 2) = n - 1 \)
  by (simp add: even-two-times-div-two)

  then have range-eq: \( n - 1 + 1 = n \)
  using odd-pos[OF False] by auto

  from \( n\)-eq \( n \geq 2 \ast g\)-no have \( n - 1 \div 2 \geq g\)-no
  by auto

  from \( g \)[OF this] show \( \exists \)thesis

  by (simp only: \( n\)-eq range-eq)

  qed

  qed

then show \( \exists n. \forall n \geq n. \text{norm} (\mathcal{S} a n - l) < r \) by blast

qed

then have sums-l: \( (\lambda i. (-1)^i \ast a \ i) \) \( \text{sums} \ l \)
by (simp only; sums-def)
then show summable ?S by (auto simp: summable-def)

have l = suminf ?S by (rule sums-unique[OF sums-l])

fix n
show suminf ?S ≤ ?g n
unfolding sums-unique[OF sums-l, symmetric] using above-l by auto
show ?f n ≤ suminf ?S
unfolding sums-unique[OF sums-l, symmetric] using below-l by auto
show ?g −→ suminf ?S
using (?g −→ b ⟨l = suminf ?S⟩) by auto
show ?f −→ suminf ?S
using (?f −→ b ⟨l = suminf ?S⟩) by auto
qed

theorem summable-Leibniz:
fixes a :: nat ⇒ real
assumes "a-zero: a −→ 0"
and monoseq a
shows summable (λ n. (−1)^n * a n) (is summable)
and 0 < a 0 −→
(∀ n. (∑ i. (−1)^i * a i) ∈ { ∑ i<2*n. (−1)^i * a i .. ∑ i<2*n+1. (−1)^i * a i }) (is ?pos)
and a 0 < 0 −→
(∀ n. (∑ i. (−1)^i * a i) ∈ { ∑ i<2*n+1. (−1)^i * a i .. ∑ i<2*n+2. (−1)^i * a i }) (is ?neg)
and (λn. ∑ i<2*n+1. (−1)^i * a i) −→ (∑ i. (−1)^i * a i) (is ?f)
and (λn. ∑ i<2*n+2. (−1)^i * a i) −→ (∑ i. (−1)^i * a i) (is ?g)

proof –
have ?summable ∧ ?pos ∧ ?neg ∧ ?f ∧ ?g
proof (cases "∀ n. 0 ≤ a n" ∧ "∀ m. ∀ n≥m a n ≤ a m")
case True
then have ord: (∀ n m. m ≤ n −→ a n ≤ a m)
and ge0: (∀ n. 0 ≤ a n)
by auto
have monono: a (Suc n) ≤ a n for n
using ord[where n=Suc n and m=n] by auto
note leibniz = summable-Leibniz[OF ⟨a −→ 0⟩ ge0]
from leibniz[OF monoseq]
show ?thesis using ⟨0 ≤ a 0⟩ by auto

next
let ?a = λn. − a n

case False
with monoseq-le[OF ⟨monoseq a⟩ ⟨a −→ 0⟩]
have (∀ n. a n ≤ 0) ∧ (∀ m. ∀ n≥m. a m ≤ a n) by auto
then have ord: (∀ n m. m ≤ n −→ ?a n ≤ ?a m and ge0: (∀ n. 0 ≤ ?a n)
by auto
have monotone: \(a (\text{Suc} \ n) \leq a n\) for \(n\)

using ord/where \(n=\text{Suc} \ n\) and \(m=n\) by auto

note leibniz =
summable-Leibniz[OF - ge0, of \(\lambda x. \ x\),
OF tendsto-minus[OF \(\langle a \longrightarrow 0\rangle\), unfolded minus-zero] monotone]

have summable (\(\lambda n. (-1) n \ast a n\)) using leibniz(1) by auto

then obtain \(l\) where (\(\lambda n. (-1) n \ast a n\)) sums \(l\)

unfolding summable-def by auto

from this[THEN sums-minus] have (\(\lambda n. (-1) n \ast a n\)) sums \(-l\)
by auto
then have \(?\)summable by (auto simp: summable-def)

moreover have \(|-a - -b| = |a - b|\) for \(a \ b::\text{real}\)

unfolding minus-diff-minus by auto

from suminf-minus[OF leibniz(1), unfolded mult-minus-right minus-minus]

have move-minus: (\(\sum n. ((-1) n \ast a n)\)) = -(\(\sum n. (-1) n \ast a n\))
by auto

have \(?\)pos using (\(0 \leq \langle a \rangle 0\)) by auto

moreover have \(?\)neg
using leibniz(2,4)
unfolding mult-minus-right sum-negf move-minus neg-le iff-le
by auto

moreover have \(?f\) and \(?g\)
using leibniz(3,5)[unfolded mult-minus-right sum-negf move-minus, THEN
tendsto-minus-cancel]
by auto
ultimately show \(?\)thesis by auto
qed
then show \(?\)summable and \(?\)pos and \(?\)neg and \(?f\) and \(?g\)
by safe
qed

113.4 Term-by-Term Differentiability of Power Series

definition \(\text{diffs} :: (\text{nat} \Rightarrow 'a::\text{ring-1}) \Rightarrow \text{nat} \Rightarrow 'a\)
where \(\text{diffs} \ c = (\lambda n. \text{of-nat} (\text{Suc} \ n) \ast c (\text{Suc} \ n))\)

Lemma about distributing negation over it.

lemma \(\text{diffs-minus}: \text{diffs} \ (\lambda n. - c n) = (\lambda n. - \text{diffs} \ c n)\)
by (simp add: diffs-def)

lemma \(\text{diffs-equiv}::\)
fixes \(x::\{\text{real-normed-vector}, \text{ring-1}\}\)
shows summable (\(\lambda n. \text{of-nat} \ast c n \ast x' \ n\)) sums (\(\sum n. \text{diffs} \ c n \ast x' \ n\))

unfolding diffs-def
by (simp add: summable-sums sums-Suc-imp)

lemma lemma-termdiff1:
  fixes z :: 'a :: {monoid_mult, comm_ring}
  shows \((\sum_{p < m} \cdot ((z + h) \cdot (m - p)) \cdot (z \cdot p)) - (z \cdot m)\) =
  \((\sum_{p < m} \cdot (z \cdot p) \cdot (((z + h) \cdot (m - p)) - (z \cdot (m - p))))\)
  by (auto simp: algebra-simps power-add [symmetric])

lemma lemma-termdiff2: sum f \{..<n\} - of-nat n * r = (\sum_{i<n} f \cdot i - r)
  for r :: 'a :: ring_1
  by (simp add: sum-subtractf)

lemma lemma-termdiff3:
  fixes h :: 'a :: field
  assumes h: h \neq 0
  shows \(((z + h) \cdot \sum_{n<Suc k} n \cdot (z \cdot (k - n)) \cdot (h + z) \cdot (z \cdot n))\) =
  \((\sum_{j<Suc k} \cdot h \cdot ((h + z) \cdot (j \cdot z) \cdot (x + k - j)))\)
  by (auto simp add: power-add [symmetric] mult.commute intro: sum.cong)

have \*: \((\sum_{i<m} \cdot z \cdot (z + h) \cdot (m - i) - z \cdot (m - i))\) =
  \((\sum_{i<m} \cdot \sum_{j=m-i} \cdot h \cdot ((z + h) \cdot (j \cdot z) \cdot (m - Suc j)))\)
  by (force simp add: less_iff_Suc_add sum_distrib_left diff_power_eq_sum ac_simps)

  case (Suc m)
  have 0: \(\forall x. \left(\sum_{n<Suc k} n \cdot (z \cdot x) \cdot (z \cdot (k - n)) \cdot (h + z) \cdot (z \cdot n))\right)\) =
    \((\sum_{j<Suc k} \cdot h \cdot ((h + z) \cdot (j \cdot z) \cdot (x + k - j)))\)
  by (auto simp add: power-add [symmetric] mult.commute intro: sum.cong)

  have \*: \((\sum_{i<m} \cdot z \cdot (z + h) \cdot (m - i) - z \cdot (m - i))\) =
    \((\sum_{i<m} \cdot \sum_{j=m-i} \cdot h \cdot ((z + h) \cdot (j \cdot z) \cdot (m - Suc j)))\)
  by (force simp add: less_iff_Suc_add sum_distrib_left diff_power_eq_sum ac_simps)

  case (Suc m)
  have 0: \(\forall x. \left(\sum_{n<Suc k} n \cdot (z \cdot x) \cdot (z \cdot (k - n)) \cdot (h + z) \cdot (z \cdot n))\right)\) =
    \((\sum_{j<Suc k} \cdot h \cdot ((h + z) \cdot (j \cdot z) \cdot (x + k - j)))\)
  by (auto simp add: power-add [symmetric] mult.commute intro: sum.cong)

  have \*: \((\sum_{i<m} \cdot z \cdot (z + h) \cdot (m - i) - z \cdot (m - i))\) =
    \((\sum_{i<m} \cdot \sum_{j=m-i} \cdot h \cdot ((z + h) \cdot (j \cdot z) \cdot (m - Suc j)))\)
  by (force simp add: less_iff_Suc_add sum_distrib_left diff_power_eq_sum ac_simps)

  then show \*: \(\forall x. \left(\sum_{n<Suc k} n \cdot (z \cdot x) \cdot (z \cdot (k - n)) \cdot (h + z) \cdot (z \cdot n))\right)\) =
    \((\sum_{j<Suc k} \cdot h \cdot ((h + z) \cdot (j \cdot z) \cdot (x + k - j)))\)
  by (auto simp add: power-add [symmetric] mult.commute intro: sum.cong)

  finally have \*: \(\forall x. \left(\sum_{n<Suc k} n \cdot (z \cdot x) \cdot (z \cdot (k - n)) \cdot (h + z) \cdot (z \cdot n))\right)\) =
    \((\sum_{j<Suc k} \cdot h \cdot ((h + z) \cdot (j \cdot z) \cdot (x + k - j)))\)
  by (auto simp add: power-add [symmetric] mult.commute intro: sum.cong)

  qed auto

lemma real-sum-nat-ivl-bounded2:
fixes $K :: 'a::linordered-semidom$
assumes $f: \forall p::nat. p < n \Rightarrow f p \leq K$ and $K: 0 \leq K$
shows $\sum f \{..<n-k\} \leq \text{of-nat} n * K$

proof –
  have $\sum f \{..<n-k\} \leq (\sum i<n-k. K)$
  by (rule sum-mono \{OF f\}) auto
  also have \ldots \leq \text{of-nat} n * K
  by (auto simp: mult-right-mono K)
finally show \text{?thesis}.
qed

lemma lemma-termdiff3:
fixes $h, z :: 'a::real-normed-field$
assumes $1: h \neq 0$
and $2: \text{norm} z \leq K$
and $3: \text{norm} (z + h) \leq K$
shows $\text{norm} (((z + h) \sim n - z \sim n) / h - \text{of-nat} n * z \sim (n - \text{Suc} 0)) \leq$
\text{of-nat} n * of-nat \((n - \text{Suc} 0) * K \sim (n - 2) * \text{norm} h$

proof –
  have $\text{norm} (((z + h) \sim n - z \sim n) / h - \text{of-nat} n * z \sim (n - \text{Suc} 0)) =$
  $\text{norm} (\sum p<n - \text{Suc} 0. \sum q<n - \text{Suc} 0 - p. (z + h) \sim q * z \sim (n - 2 - q))$
  * \text{norm} h
  by (metis (lifting, no-types) lemma-termdiff2 \{OF 1\} mult.commute norm-mult)
  also have \ldots \leq \text{of-nat} n * (\text{of-nat} \((n - \text{Suc} 0) * K \sim (n - 2)) * \text{norm} h$
  proof (rule mult-right-mono \{OF - \text{norm-ge-zero}\})
    from \text{norm-ge-zero} 2 have $K: 0 \leq K$
    by (rule order-trans)
    have le-Kn: $\text{norm} (((z + h) \sim i * z \sim j) \leq K \sim n$ if $i + j = n$ for $i j n$
    proof –
      have $\text{norm} (z + h) \sim i * \text{norm} z \sim j \leq K \sim i * K \sim j$
      by (intro mult-mono power-mono 2 3 \text{norm-zero-le-power} K)
      also have \ldots = $K^n$
      by (metis power-add that)
      finally show \text{?thesis}
      by (simp add: norm-mult norm-power)
    qed
    then have $\forall p q. [p < n; q < n - \text{Suc} 0] \Rightarrow norm (((z + h) \sim q * z \sim (n - 2 - q)) \leq K \sim$
    \((n - 2))$
    by (simp del: subst-all)
    then
    show $\text{norm} (\sum p<n - \text{Suc} 0. \sum q<n - \text{Suc} 0 - p. (z + h) \sim q * z \sim (n - 2 - q)) \leq$
    of-nat n * (of-nat \((n - \text{Suc} 0) * K \sim (n - 2))$
    by (intro order-trans \{OF \text{norm-sum}\}\text{real-sum-nat-idl-bounded2 mult-nonneg-nonneg of-nat-0-le-iff zero-le-power} K)
    qed
    also have \ldots = \text{of-nat} n * \text{of-nat} \((n - \text{Suc} 0) * K \sim (n - 2) * \text{norm} h$
by (simp only: mult.assoc)
finally show ?thesis .
qed

lemma lemma-termdiff4:
  fixes f :: 'a::real-normed-vector ⇒ 'b::real-normed-vector
  and k :: real
  assumes k: 0 < k
  and le: λh. h ≠ 0 ⟹ norm h < k ⟹ norm (f h) ≤ K * norm h
  shows f −0→ 0
proof (rule tendsto-norm-zero-cancel)
  show (λh. norm (f h)) −0→ 0
proof (rule real-tendsto-sandwich)
  show eventually (λh. 0 ≤ norm (f h)) (at 0)
  by simp
  show eventually (λh. norm (f h) ≤ K * norm h) (at 0)
  using k by (auto simp: eventually-at dist-norm le)
  show (λh. 0) −(0::'a)→ (0::real)
  by (rule tendsto-const)
  have (λh. K * norm h) −(0::'a)→ K * norm (0::'a)
  by (intro tendsto-intros)
  then show (λh. K * norm h) −(0::'a)→ 0
  by simp
qed

lemma lemma-termdiff5:
  fixes g :: 'a::real-normed-vector ⇒ nat ⇒ 'b::banach
  and k :: real
  assumes k: 0 < k
  and f: summable f
  and le: λh n. h ≠ 0 ⟹ norm h < k ⟹ norm (g h n) ≤ f n * norm h
  shows (λh. suminf (g h)) −0→ 0
proof (rule lemma-termdiff4 [OF k])
  fix h :: 'a
  assume h ≠ 0 and norm h < k
  then have 1: ∀n. norm (g h n) ≤ f n * norm h
  by (simp add: le)
  then have 3: ∀n≥N. norm (norm (g h n)) ≤ f n * norm h
  by simp
  moreover from f have 2: summable (λn. f n * norm h)
  by (rule summable-mult2)
  ultimately have 3: summable (λn. norm (g h n))
  by (rule summable-comparison-test)
  then have norm (suminf (g h)) ≤ (∑n. norm (g h n))
  by (rule summable-norm)
  also from 1 3 2 have (∑n. norm (g h n)) ≤ (∑n. f n * norm h)
  by (simp add: suminf-le)
  also from f have (∑n. f n * norm h) = suminf f * norm h
by \(\text{rule suminf-mult2 \ [symmetric]}\)
finally show \(\text{norm (suminf (g h))} \leq \text{suminf f} \ast \text{norm h} \).
qed

lemma termdiffs-aux:
fixes \(x :: 'a :: \{\text{real-normed-field}, \text{banach}\}\)
assumes 1: \(\text{summable (}\lambda n. \text{diffs (difs c) n} \ast K \sim n)\)
and 2: \(\text{norm x} < \text{norm K}\)
shows \(\lambda h. \sum n. c n \ast (((x + h) \sim n - x \sim n) / h \ast \text{of-nat n} \ast x \sim (n - \text{Suc 0}))) \rightarrow 0\)
proof
from dense \(\text{OF 2}\) obtain \(r\) where \(r1: \text{norm x} < r\) and \(r2: r < \text{norm K}\)
by fast
from \(\text{norm-ge-zero r1}\)
have \(r: 0 < r\)
by \(\text{rule order-le-less-trans}\)
then have \(r-neq-0: r \neq 0\)
by simp
show \(?\text{thesis}\)
proof \(\text{rule lemma-termdiff5}\)
show \(0 < r \sim \text{norm x}\)
using \(r1\) by simp
from \(r\ \text{r2}\) have \(\text{norm (of-real r ::'a)} < \text{norm K}\)
by simp
with \(1\) have \(\text{summable (}\lambda n. \text{norm (difs (difs c) n) (of-real r \sim n))}\)
by \(\text{rule power-inside}\)
then have \(\text{summable (}\lambda n. \text{difs (difs (}\lambda n. \text{norm (c n))) n \ast r \sim n)\)}\)
using \(r\) by \(\text{simp add: diffs-def norm-mult norm-power del: of-nat-Suc}\)
then have \(\text{summable (}\lambda n. \text{of-nat n} \ast \text{difs (}\lambda n. \text{norm (c n))) n \ast r \sim (n - \text{Suc 0})\)}\)
by \(\text{rule diffs-equiv [THEN sums-summable]}\)
also have \(\text{summable (}\lambda n. \text{of-nat n} \ast \text{difs (}\lambda n. \text{norm (c n))) n \ast r \sim (n - \text{Suc 0})\) = \(\lambda n. \text{difs (}\lambda m. \text{of-nat (m - Suc 0) * norm (c m) * inverse r) n} \ast (r \sim n))\)
by \(\text{simp add: diffs-def r-neq-0 fun-eq-iff split: nat-diff-split}\)
finally have \(\text{summable (}\lambda n. \text{of-nat n} \ast (\text{of-nat (n - Suc 0) * norm (c n) * inverse r) \ast r \sim (n - Suc 0)})\)
by \(\text{rule diffs-equiv [THEN sums-summable]}\)
also have \(\text{summable (}\lambda n. \text{of-nat n} \ast (\text{of-nat (n - Suc 0) * norm (c n) * inverse r) * r \sim (n - Suc 0)}) = \(\lambda n. \text{norm (c n) * of-nat n} \ast \text{of-nat (n - Suc 0) * inverse r) * r \sim (n - Suc 0)}\))\)
by \(\text{rule ext} (\text{simp add: r-neq-0 split: nat-diff-split})\)
finally show \(\text{summable (}\lambda n. \text{norm (c n) * of-nat n} \ast \text{of-nat (n - Suc 0) * r \sim (n - 2)})\)
next
fix \(h :: 'a \text{ and n}\)
assume \( h: h \neq 0 \)
assume \( \text{norm } h < r - \text{norm } x \)
then have \( \text{norm } x + \text{norm } h < r \) by simp
with \( \text{norm-triangle-ineq} \)
have \( \text{xh: } \text{norm } (x + h) < r \)
by (rule order-le-less-trans)
have \( \text{norm } (((x + h)^n - x^n) / h - \text{of-nat } n * x^n (n - \text{Suc } 0)) \)
\( \leq \text{real } n * (\text{real } (n - \text{Suc } 0) * (r^\omega (n - 2) * \text{norm } h)) \)
by (metis (mono-tags, lifting) \( h \) mult.assoc lemma-termdiff3 less-eq-real-def
\( \text{r1 xh} \))
then show \( \text{norm } (c * ((x + h)^n - x^n) / h - \text{of-nat } n * x^n (n - \text{Suc } 0)) \) \( \leq \)
\( \text{norm } (c * \text{of-nat } n * \text{of-nat } (n - \text{Suc } 0) * r^\omega (n - 2) * \text{norm } h) \)
by (simp only: norm-mult mult.assoc mult-left-mono [OF - norm-ge-zero])
qed

lemma termdiffs:
fixes \( K x :: a:\{\text{real-normed-field,banach} \}
assumes 1: \text{summable } (\lambda n. c \cdot n * K^n) \nand 2: \text{summable } (\lambda n. (\text{diffs } c) \cdot n * K^n) \nand 3: \text{summable } (\lambda n. (\text{diffs } (\text{diffs } c)) \cdot n * K^n) \nand 4: \text{norm } x < \text{norm } K \nshows \( \text{DERIV } (\lambda x. \sum n. c \cdot n * x^n) x : (\sum n. (\text{diffs } c) \cdot n * x^n) \)

unfolding DERIV-def
proof (rule LIM-zero-cancel)
show \( \lambda h. (\text{suminf } (\lambda n. c \cdot n * (x + h)^n) - \text{suminf } (\lambda n. c \cdot n * x^n)) / h \)
\( - \text{suminf } (\lambda n. \text{diffs } c \cdot n * x^n)) \to 0 \)
proof (rule LIM-equal2)
show \( \theta < \text{norm } K - \text{norm } x \)
using 4 by (simp add: less-diff-eq)

next
fix \( h :: 'a \)
assume \( \text{norm } (h - 0) < \text{norm } K - \text{norm } x \)
then have \( \text{norm } (x + h) < \text{norm } K \) by simp
then have 5: \( \text{norm } (x + h) < \text{norm } K \)
by (rule norm-triangle-ineq [THEN order-le-less-trans])
have \( \text{summable } (\lambda n. c \cdot n * x^n) \)
and \( \text{summable } (\lambda n. c \cdot n * (x + h)^n) \)
and \( \text{summable } (\lambda n. \text{diffs } c \cdot n * x^n) \)
using 1 2 4 5 by (auto elim: pouser-inside)
then have \( ((\sum n. c \cdot n * (x + h)^n) - (\sum n. c \cdot n * x^n)) / h - (\sum n. \text{diffs } c \cdot n * x^n) = \)
\( (\sum n. (c \cdot n * (x + h)^n - c \cdot n * x^n) / h - \text{of-nat } n * c \cdot n * x^n (n - \text{Suc } 0)) \)
by (intro sums-unique sums-diff sums-divide diffs-eqv summable-sums)
then show \( ((\sum n. c \cdot n * (x + h)^n - (\sum n. c \cdot n * x^n)) / h - (\sum n. \text{diffs } c \cdot n * x^n) = \)
\( (\sum n. c \cdot n * (((x + h)^n - x^n) / h - \text{of-nat } n * x^n (n - \text{Suc } 0))) \)
by (simp add: algebra-simps)
next
show \((\lambda h. \sum n. \ c \ n \ast ((x + h) \sim n \sim x \sim n) / h - of-nat n \ast x \sim (n - Suc 0))) \sim \theta \rightarrow 0\)
  by (rule termdiffs-aux [OF 3 4])
qed

113.5 The Derivative of a Power Series Has the Same Radius of Convergence

lemma termdiff-converges:
fixes \(x \:: 'a::{real-normed-field,banach}\)
assumes \(K: \text{norm } x < K\)
and \(sm: \bigwedge x. \text{norm } x < K \Rightarrow \text{summable}(\lambda n. c \ n \ast x \sim n)\)
shows \(\text{summable}(\lambda n. \text{diffs } c \ n \ast x \sim n)\)
proof (cases \(x = 0\))
case True
  then show \(?\text{thesis}\)
    using powser-sums-zero sums-summable by auto
next
case False
  then have \(K > 0\)
    using \(K \text{ less-trans zero-less-norm-iff}\) by blast
  then obtain \(r::\text{real}\)
    where \(r: \text{norm } x < \text{norm } r \text{ norm } r < K \text{ } r > 0\)
    using \(K \text{ False}\)
    by (auto simp: field-simps abs-less-iff add-pos-pos intro: that [of (norm } x + K) / 2])
  have \(to0: (\lambda n. \text{of-nat } n \ast (x / \text{of-real } r) \sim n) \sim \sim 0\)
    using \(r\) by (simp add: norm-divide powser-times-n-limit-0 [of x / of-real r])
  obtain \(N\) where \(N: \bigwedge n. n \geq N \Rightarrow \text{real-of-nat } n \ast \text{norm } x \sim n < r \sim n\)
    using \(r \text{ LIMSEQ-D [OF to0, of 1]}\)
    by (auto simp: norm-mult norm-power field-simps)
  have \(\text{summable}(\lambda n. (\text{of-nat } n \ast c \ n) \ast x \sim n)\)
    proof (rule summable-comparison-test')
      show \(\text{summable}(\lambda n. \text{norm } (c \ n \ast \text{of-real } r \sim n))\)
        apply (rule powser-insidea [OF sm [of of-real ((r+K)/2)])]
        using \(N\) \(r \text{ norm-of-real [of } r + K, \text{ where } 'a = 'a]\) by auto
      show \(\bigwedge n. N \leq n \Rightarrow \text{norm } (\text{of-nat } n \ast c \ n \ast x \sim n) \leq \text{norm } (c \ n \ast \text{of-real } r \sim n)\)
        using \(N \ast r\) by (fastforce simp add: norm-mult norm-power less-eq-real-def)
    qed
  then have \(\text{summable}(\lambda n. (\text{of-nat } (Suc \ n) \ast c(Suc \ n)) \ast x \sim Suc \ n)\)
    using summable-iff-shift [of \(\lambda n. \text{of-nat } n \ast c \ n \ast x \sim n 1\] by simp
  then have \(\text{summable}(\lambda n. (\text{of-nat } (Suc \ n) \ast c(Suc \ n)) \ast x \sim n)\)
    using False summable-mult2 [of \(\lambda n. (\text{of-nat } (Suc \ n) \ast c(Suc \ n) \ast x \sim n) \ast x\]
    inverse x]
    by (simp add: mult.assoc (auto simp: ac-simps)
then show \( ?\text{thesis} \)
\[ \text{by (simp add: \text{diffs-def})} \]
\[ \text{qed} \]

**lemma** \text{termdiff-converges-all}: 
\[ \text{fixes } x :: \text{a}::\{\text{real-normed-field, banach}\} \]
\[ \text{assumes } \bigwedge x. \text{summable } (\lambda n. \ c\ n * x ^ \sim n) \]
\[ \text{shows } \text{summable } (\lambda n. \ \text{diffs } c\ n * x ^ \sim n) \]
\[ \text{by (rule \text{termdiff-converges [where } K = 1 + \text{norm } x])} \ (\text{use assms in auto}) \]

**lemma** \text{termdiffs-strong}: 
\[ \text{fixes } K x :: \text{a}::\{\text{real-normed-field, banach}\} \]
\[ \text{assumes } sm: \text{summable } (\lambda n. \ c\ n * K ^ \sim n) \]
\[ \text{and } K: \text{norm } x < \text{norm } K \]
\[ \text{shows } \text{DERIV } (\lambda x. \sum n. \ c\ n * x ^ \sim n) \ x :> (\sum n. \ \text{diffs } c\ n * x ^ \sim n) \]
\[ \text{proof} \]
\[ \text{have } \text{norm } K + \text{norm } x < \text{norm } K + \text{norm } K \]
\[ \text{using } K \text{ by force} \]
\[ \text{then have } K2: \text{norm } ((\text{of-real } (\text{norm } K) + \text{of-real } (\text{norm } x)) / 2 :: 'a) < \text{norm } K \]
\[ \text{by (auto simp: norm-triangle-lt norm-divide field-simps)} \]
\[ \text{then have } [\text{simp}: \text{norm } ((\text{of-real } (\text{norm } K) + \text{of-real } (\text{norm } x)) :: 'a) < \text{norm } K * 2 \]
\[ \text{by simp} \]
\[ \text{have } \text{summable } (\lambda n. \ c\ n * (\text{of-real } (\text{norm } x + \text{norm } K) / 2) ^ \sim n) \]
\[ \text{by (metis K2 summable-norm-cancel [OF powser-insidea [OF sm]] add.commute of-real-add)} \]
\[ \text{moreover have } \bigwedge x. \text{norm } x < \text{norm } K \Rightarrow \text{summable } (\lambda n. \ \text{diffs } c\ n * x ^ \sim n) \]
\[ \text{by (blast intro: sm \text{termdiff-converges powser-inside)} \]
\[ \text{moreover have } \bigwedge x. \text{norm } x < \text{norm } K \Rightarrow \text{summable } (\lambda n. \ \text{diffs } (\text{diffs } c\ n) n * x ^ \sim n) \]
\[ \text{by (blast intro: sm \text{termdiff-converges powser-inside)} \]
\[ \text{ultimately show } ?\text{thesis} \]
\[ \text{by (rule \text{termdiffs [where } K = \text{of-real } (\text{norm } x + \text{norm } K) / 2])} \]
\[ \text{(use } K \text{ in } \langle \text{auto simp: field-simps simp flip: of-real-add)} \rangle \]
\[ \text{qed} \]

**lemma** \text{termdiffs-strong-everywhere}: 
\[ \text{fixes } K x :: 'a::\{\text{real-normed-field, banach}\} \]
\[ \text{assumes } \bigwedge y. \text{summable } (\lambda n. \ c\ n * y ^ \sim n) \]
\[ \text{shows } ((\lambda x. \sum n. \ c\ n * x ^ \sim n) \ \text{has-field-derivative } (\sum n. \ \text{diffs } c\ n * x ^ \sim n)) \ (\text{at } x) \]
\[ \text{using \text{termdiffs-strong}[\text{OF assms[of of-real (norm x + 1)], of } x]} \]
\[ \text{by (force simp del: of-real-add)} \]

**lemma** \text{termdiffs-strong'}: 
\[ \text{fixes } z :: 'a :: \{\text{real-normed-field, banach}\} \]
\[ \text{assumes } \bigwedge z. \text{norm } z < K \Rightarrow \text{summable } (\lambda n. \ c\ n * z ^ \sim n) \]
\[ \text{assumes } \text{norm } z < K \]
\[ \text{shows } ((\lambda z. \sum n. \ c\ n * z ^ \sim n) \ \text{has-field-derivative } (\sum n. \ \text{diffs } c\ n * z ^ \sim n)) \ (\text{at } z) \]
proof (rule termdiffs-strong)

define L :: real where L = (norm z + K) / 2

have 0 ≤ norm z by simp
also note (norm z < K)
finally have K: K ≥ 0 by simp

from assms K have L: L ≥ 0 norm z < L L < K by (simp-all add: L-def)
from L show norm z < norm (of-real L :: 'a) by simp
from L show summable (λn. c n * of-real L ^ n) by (intro assms(1)) simp-all

qed

lemma termdiffs-sums-strong:

fixes z :: 'a :: {banach,real-normed-field}
assumes sums: (∃z. norm z < K ⇒ (λn. c n * z ^ n) sum f z)
assumes deriv: (f has-field-derivative f') (at z)
assumes norm: norm z < K
shows (λn. diffs c n * z ^ n) sums f'

proof

have summable: summable (λn. diffs c n * z ^ n)
  by (intro termdiff-converges[OF norm] sums-summable[OF sums])
from norm have eventually (λz. z ∈ norm - 'Μ..<K) (nhds z)
  by (intro eventually-nhds-in-open-open-vimage)
  (simp-all add: continuous-on-norm)

hence eq: eventually (λz. (∑n. c n * z ^ n) = f z) (nhds z)
  by eventually-elim (insert sums, simp add: sums-iff)

have ((λz. ∑n. c n * z ^ n) has-field-derivative (∑n. diffs c n * z ^ n)) (at z)
  by (intro termdiffs-strong[OF - norm] sums-summable[OF sums])

hence (f has-field-derivative (∑n. diffs c n * z ^ n)) (at z)
  by (subst (asm) DERIV-cong-ev[OF refl eq refl])

from this and deriv have (∑n. diffs c n * z ^ n) = f' by (rule DERIV-unique)
with summable show ?thesis by (simp add: sums-iff)

qed

lemma isCont-powser:

fixes K x :: 'a::{real-normed-field,banach}
assumes summable: (λn. c n * K ^ n)
assumes norm x < norm K
shows isCont (λx. ∑n. c n * x ^ n) x
using termdiffs-strong[OF assms] by (blast intro!: DERIV-isCont)

lemmas isCont-powser' = isCont-o2[OF - isCont-powser]

lemma isCont-powser-converges-everywhere:

fixes K x :: 'a::{real-normed-field,banach}
assumes ∃y. summable (λn. c n * y ^ n)
shows isCont (λx. ∑n. c n * x ^ n) x
using termdiffs-strong[OF assms[of of-real (norm x + 1), of x]]
by (force intro!: DERIV-isCont simp del: of-real-add)
lemma powser-limit-0:
fixes a :: nat ⇒ 'a::{real-normed-field,banach}
assumes s: 0 < s
and sm: \A. x. norm x < s ⟹ (λn. a n * x ^ n) sums (f x)
shows (f ----> a 0) (at 0)
proof –
have norm (of-real s / 2 :: 'a) < s
  using s by (auto simp: norm-divide)
then have summable (λn. a n * (of-real s / 2) ^ n)
  by (rule sums-summable [OF sm])
then have ((λx. ∑n. a n * x ^ n) has-field-derivative (∑n. diffs a n * 0 ^ n)) (at 0)
  by (rule termdiffs-strong) (use s in ‹auto simp: norm-divide›)
then have isCont (λx. ∑n. a n * x ^ n) 0
  by (blast intro: DERIV-continuous)
then have (λx. f x − (∑n. a n * x ^ n)) −−−→ 0 (at 0)
  apply (clarsimp simp: LIM-eq)
  apply (rule-tac x = s in exI)
  using s sm sums-unique by fastforce
ultimately show ?thesis
by (rule Lim-transform)
qed

lemma powser-limit-0-strong:
fixes a :: nat ⇒ 'a::{real-normed-field,banach}
assumes s: 0 < s
and sm: \A. x. x ≠ 0 ⟹ norm x < s ⟹ (λn. a n * x ^ n) sums (f x)
shows (f ----> a 0) (at 0)
proof –
have *: ((λx. if x = 0 then a 0 else f x) ----> a 0) (at 0)
  by (rule powser-limit-0 [OF s]) (auto simp: powser-sums-zero sm)
show ?thesis
  using * by (auto cong: Lim-cong-within)
qed

113.6  Derivability of power series

lemma DERIV-series:
fixes f :: real ⇒ nat ⇒ real
assumes DERIV-f: \A. n. DERIV (λx. f x n) x0 :> (f' x0 n)
and allf-summable: \A. x. x ∈ {a <..< b} ⟹ summable (f x)
and x0-in-I: x0 ∈ {a <..< b}
and summable (f' x0)
and summable L
and L-def: \A. x y. x ∈ {a <..< b} ⟹ y ∈ {a <..< b} ⟹ |f x n − f y n| ≤ L n * |x − y|
shows DERIV (λx. suminf (f x)) x0 :> (suminf (f' x0))
unfolding DERIV-def
proof (rule LIM-I)
  fix r :: real
  assume \( \theta < r \) then have \( \theta < r/3 \) by auto

obtain N-L where N-L: \( \land \) n. N-L \leq n \implies \sum i. L (i + n) \leq r/3
  using suminf-exist-split[OF \( \theta < r/3 \), summable L] by auto

obtain N-f' where N-f': \( \land \) n. N-f' \leq n \implies \sum i. f' x \cdot (i + n) \leq r/3
  using suminf-exist-split[OF \( \theta < r/3 \), summable (f' x)] by auto

let \(?N = Suc \( \max N-L N-f' \)
  have \( \sum i. f' x \cdot (i + \uparrow N) \leq r/3 \) (is \( \text{iff-part} < r/3 \)
    and L-estimate: \( \sum i. L (i + \uparrow N) \leq r/3 \)
    using N-L[of \(?N \)] and N-f'[of \(?N \)] by auto

let \(?diff = \lambda i x. (f (x + x) - i - f x) / x \)

let \(?r = r / (3 * real \(?N \))
  from \( \theta < r \) have \( \theta < \uparrow r \) by simp

let \(?s = \lambda n. \text{SOME } s. 0 < s \land (\forall x. x \neq 0 \land |x| < s \implies |\text{diff } n x - f' x| \leq \theta \)
  n \leq \uparrow \theta \)
  define S' where S' = Min \( \{ \uparrow s \cdot \{..<\uparrow N \} \} \)

have \( \theta < S' \)
  unfolding S'-def
proof (rule iffD2[OF OF Min-gr-iff])
  show \( \forall x \in \{ \uparrow s \cdot \{..<\uparrow N \} \}. \theta < x \)
    proof
      fix x
      assume x \in \( \{ \uparrow s \cdot \{..<\uparrow N \} \}
      then obtain n where x = \( \uparrow s \) n and n \in \{..<\uparrow N \}
      using image-iff[OF THEN iffD1] by blast
      from DERIV-D[OF DERIV-f'[where n=n], THEN LIM-D, OF \( \theta < \uparrow r \),
        unfolded real-norm-def]
      obtain s where s-bound: \( 0 < s \land (\forall x. x \neq 0 \land |x| < s \implies |\text{diff } n x - f' x| \leq \theta \)
        n \leq \uparrow \theta \)
        by auto
      have \( \theta < \uparrow s \) n
      by (rule someI2[where a=s]) (auto simp; s-bound simp del: of-nat-Suc)
      then show \( \theta < x \) by (simp only: \( x = \uparrow s \) n)
  qed
  qed auto

define S where S = min \( \min (x0 - a) (b - x0) \) S'
then have \( \theta < S \) and S-a: \( S \leq x0 - a \) and S-b: \( S \leq b - x0 \)
and \( S \leq S' \) using x0-in-I and \( \theta < S' \)
  by auto
have \(|(\text{suminf } (f \,(x_0 + x)) - \text{suminf } (f \,x_0)) / x - \text{suminf } (f' \,x_0)| < r\)
if \(x \neq 0\) and \(|x| < S\) for \(x\)

**proof**

- from that have \(x\text{-in-I}: x_0 + x \in \{a <..< b\}\)
  - using \(S\text{-a} \, S\text{-b} \, S\text{-c}\) by auto

  \[\text{note} \, \text{diff-smbl} = \text{summable-diff}[\text{OF allf-summable}[\text{OF } x\text{-in-I}]]\]
  \[\text{note} \, \text{div-smbl} = \text{summable-divide}[\text{OF diff-smbl}]\]
  \[\text{note} \, \text{all-smbl} = \text{summable-diff}[\text{OF div-smbl} \, \langle \text{summable } (f' \,x_0) \rangle]\]
  \[\text{note} \, \text{ign} = \text{summable-ignore-initial-segment}[\text{where } k=\text{?N}]\]
  \[\text{note} \, \text{diff-shift-smbl} = \text{summable-diff}[\text{OF ign}[\text{OF allf-summable}[\text{OF } x\text{-in-I}]]]\]
  \[\text{note} \, \text{div-shift-smbl} = \text{summable-divide}[\text{OF diff-shift-smbl}]\]
  \[\text{note} \, \text{all-shift-smbl} = \text{summable-diff}[\text{OF div-smbl} \, \text{ign}[\text{OF } \langle \text{summable } (f' \,x_0) \rangle]]\]

have \(I:\, |(\text{diff } (n \,\mapsto \,N) \, x)| \leq L \,(n \,\mapsto \,N)\) for \(n\)

**proof**

- have \(|(\text{diff } (n \,\mapsto \,N) \, x)| \leq L \,(n \,\mapsto \,N) \ast \|(x_0 + x) - x_0| / |x|\)
  - using \(\text{divide-right-mono}[\text{OF } \langle \text{L-def}[\text{OF } x\text{-in-I} \, x_0\text{-in-I}] \, \text{abs-ge-zero} \rangle]\)
    - by (simp only: \(\text{abs-divide}\))
      - with \(\langle x \neq 0 \rangle\) show \(?\text{thesis} by auto\)

**qed**

**note** \(2 = \text{summable-rabs-comparison-test}[\text{OF } \text{ign}[\text{OF } \langle \text{summable } L_i \rangle]]\)

from \(l\) have \(|\sum i. \, \text{diff } (i \,\mapsto \,N) \, x| \leq (\sum i. \, L \,(i \,\mapsto \,N))\)

- by (metis (lifting) \(\text{abs-idempotent}\))
  - order-trans[\(\text{OF summable-rabs}[\text{OF } 2 \text{ \, \text{suminf-le}[\text{OF } \text{2 \, \text{ign}[\text{OF } \langle \text{summable } L_i \rangle]]]\)]

then have \(|\sum i. \, \text{diff } (i \,\mapsto \,N) \, x| \leq r / 3 \,(\text{is } \langle L\text{-part } \leq r/3 \rangle)\)

- using \(\langle L\text{-estimate} by auto \rangle\)

have \(|\sum n<\text{?N}. \, \text{diff } n \text{ } x - f' \,x_0 \, n| \leq (\sum n<\text{?N}. \, |\text{diff } n \text{ } x - f' \,x_0 \, n|)\) ..

also have \(\ldots \leq (\sum n<\text{?N}. \, ?r)\)

**proof** (rule \(\text{sum-strict-mono}\))

- fix \(n\)
  - assume \(n \in \{\ldots \,\mapsto \,\text{?N}\}\)
  - have \(|x| < S\) using \(|x| < S\).
  - also have \(S \leq S'\) using \(\langle S \leq S' \rangle\).
  - also have \(S' \leq ?s\) \(n\)
    - unfolding \(S'i\)-def
      - proof (rule \(\text{Min-le-iff}[\text{THEN } \text{iffD2}]\))
        - have \(?s \, n \in \{?s \,' \, \langle \ldots \,\mapsto \,\text{?N} \rangle\} \wedge ?s \, n \leq ?s\)
          - using \(\langle n \in \{\ldots \,\mapsto \,\text{?N}\} \rangle\) by auto
        - then show \(\exists a \in \{?s \,' \, \langle \ldots \,\mapsto \,\text{?N} \rangle\}, \, a \leq ?s \)
          - by blast
      - qed auto
    - finally have \(|x| < ?s \, n\).
from DERIV-D[OF DERIV-f[where n=n], THEN LIM-D, OF \$0 < \?r\$, unfolded real-norm-def diff-0-right, unfolded some-eq-ex[symmetric], THEN conjunct2]

have \(\forall x. x \neq 0 \land |x| < \$s \cdot n \rightarrow |\$\text{diff} n x - f' x0 n| < \$r\)
with \(\langle x \neq 0 \rangle\) and \(\langle |x| < \$s \cdot n\rangle\) show \(|\$\text{diff} n x - f' x0 n| < \$r\)
by blast
qed auto
also have \(\ldots = \text{of-nat (card \{..<\$N\}\} \ast \$r\)
by (rule sum-constant)
also have \(\ldots = \text{real \$N \ast \$r\}
by simp
also have \(\ldots = r/3\)
by (auto simp del: of-nat-Suc)
finally have \(|\sum n<\$N. \$\text{diff} n x - f' x0 n| < r / 3\) (is \$\text{diff-part} < r / 3\).

from suminf-diff[OF allf-summable[OF x-in-I] allf-summable[OF x0-in-I]]
have \(|(\text{suminf} (f (x0 + x)) - (\text{suminf} (f x0))) / x - \text{suminf} (f' x0)| =
|\sum n. \$\text{diff} n x - f' x0 n|\)
unfolding suminf-diff[OF div-smbl (summable (f' x0)), symmetric]
using suminf-dl[allf-smbl, symmetric] by auto
also have \(\ldots = \text{diff-part} + |(\sum n. \$\text{diff} (n + \$N\} x) - (\sum n. f' x0 (n + \$N))|\)
unfolding suminf-shift-initial-segment[allf-smbl, where k=\$N\]
unfolding suminf-diff[OF div-smbl ign[allf-summable (f' x0)]]
apply (simp only: add.commute)
using abs-triangle-ineq by blast
also have \(\ldots = \text{diff-part} + \$L$-part + \$f'$-part\)
using abs-triangle-ineq by auto
also have \(\ldots < r / 3 + r/3 + r/3\)
using \(\$\text{diff-part} < r/3\), \$L$-part \(\leq r/3\) and \$f'$-part \(\leq r/3\)
by (rule add-less-le-monotone)
finally show \$\text{thesis}\)
by auto
qed

then show \(\exists s > 0. \forall x. x \neq 0 \land \text{norm} (x - 0) < s \rightarrow
\text{norm} \(((\sum n. f (x0 + x) n) - (\sum n. f x0 n)) / x - (\sum n. f' x0 n)) < r\)
using \(\langle 0 < S \rangle\) by auto
qed

lemma DERIV-power-series':
fixes f :: nat \Rightarrow real
assumes converges: \(\land. \exists x \in \{-R <..< R\} \Rightarrow \text{summable} (\lambda n. f n \ast \text{real} (\text{Suc n}) \ast x^n)\)
and x0-in-I: x0 \(\in \{-R <..< R\}\)
and \(0 < R\)
shows DERIV \((\lambda x. (\sum n. f n \ast x^n) (\text{Suc n}))) x0 :> (\sum n. f n \ast \text{real} (\text{Suc n}) \ast x0^n)\)
(is DERIV \((\lambda x. \text{suminf} (\$f x\}) x0 :> \text{suminf} (\$f' x0)\))
proof –
have for-subinterval: DERIV (λx. suminf (?f x)) x0 := suminf (?f' x0)

if 0 < R' and R' < R and -R' < x0 and x0 < R' for R'

proof –

from that have x0 ∈ {-R' <= R'} and R' ∈ {-R <..< R} and x0 ∈ {-R <..< R}

by auto

show thesis

proof (rule DERIV-series')

show summable (λ n. |f n * real (Suc n) * R'~ n|)

proof –

have (R' + R) / 2 < R and 0 < (R' + R) / 2

using :0 < R' by (auto simp: field-simps)

then have in-Rball: (R' + R) / 2 ∈ {-R <..< R}

by auto

have norm R' < norm ((R' + R) / 2)

using :0 < R' by (auto simp: field-simps)

from powser-insidea[OF converges[OF in-Rball this]] show  thesis

by auto

qed

next

fix n x y

assume x ∈ {-R' <= R'} and y ∈ {-R' <= R'}

show |f x n - f y n| ≤ |f n * real (Suc n) * R'~ n| * |x-y|

proof –

have |f n * x ~ (Suc n) - f n * y ~ (Suc n)| =

(|f n| * |x-y|) * (∑p<Suc n. x ~ p * y ~ (n - p))

unfolding right-diff-distrib[symmetric] diff-power-eq-sum abs-mul

by auto

also have ... ≤ (|f n| * |x-y|) * (|real (Suc n)| * |R' ~ n|)

proof (rule mult-left-mono)

have (∑p<Suc n. x ~ p * y ~ (n - p)) ≤ (∑p<Suc n. |x ~ p * y ~ (n - p)|)

by (rule sum-abs)

also have ... ≤ (∑p<Suc n. R' ~ n)

proof (rule sum-mono)

fix p

assume p ∈ {...<Suc n}

then have p ≤ n by auto

have |x ~ n| ≤ R'~ n if x ∈ {-R'<=R'} for n and x :: real

proof –

from that have |x| ≤ R' by auto

then show thesis

unfolding power-abs by (rule power-mono) auto

qed

from mult-mono[OF this[OF this x ∈ {-R'<=R'}], of p] this[OF y ∈ {-R'<=R'}], of n-p]...
then show $|x^p \cdot y^{(n-p)}| \leq R^{n}$
  unfolding power-add[symmetric] using $(p \leq n)$ by auto
qed
also have $\ldots = \text{real} (\text{Suc } n) \cdot R^{2}$
unfolding sum-constant card-atLeastLessThan by auto
finally show $|\sum_{p<\text{Suc } n} x^p \cdot y^{(n-p)}| \leq |\text{real} (\text{Suc } n) \cdot R^{2}|$
unfolding abs-of-nonneg[of zero-le-power[of less-imp-le[of $0 < R'$]]]
by linarith
show $0 \leq |f n| \cdot |x-y|$
unfolding abs-mult[_symmetric] by auto
qed

let $?R = (R + |x0|) / 2$
have $|x0| < ?R$
  using assms by (auto simp: field-simps)
then have $- ?R < x0$
proof (cases $x0 < 0$)
  case True

then have \(- x_0 < \bar{R}\)
using \(|x_0| < \bar{R}\) by auto
then show ?thesis
unfolding neg-less-iff-less[symmetric, of \(- x_0\)] by auto

next
  case False
  have \(- \bar{R} < 0\) using assms by auto
  also have \(\ldots \leq x_0\) using False by auto
  finally show ?thesis.

qed

then have \(0 < \bar{R}\) \(\bar{R} < R - \bar{R} < x_0\) and \(x_0 < \bar{R}\)
using assms by (auto simp: field-simps)
from for-subinterval[OF this] show ?thesis.

qed

lemma geometric-deriv-sums:
fixes \(z::'a::{real-normed-field,banach}\)
assumes norm \(z < 1\)
shows \((\lambda n.\ of-nat(Suc n)\ast z^n)\)\ sums \((1 / (1 - z)^2)\)
proof
  have \((\lambda n. \text{diffs} (\lambda n. 1)\ast z^n)\)\ sums \((1 / (1 - z)^2)\)
  proof (rule termdiffs-sums-strong)
    fix \(z::'a\) assume \(\text{norm } z < 1\)
    thus \((\lambda n. 1\ast z^n)\)\ sums \((1 / (1 - z))\) by (simp add: geometric-sums)
  qed (insert assms, auto intro: derivative-eq-intros simp: power2-eq-square)
thus ?thesis unfolding diffs-def by simp
qed

lemma isCont-pochhammer [continuous-intros]: isCont \((\lambda z. \text{pochhammer } z n)\) for \(z::'a::\text{real-normed-field}\)
by (induct n) (auto simp: pochhammer-rec)

lemma continuous-on-pochhammer [continuous-intros]: continuous-on \(A\) \((\lambda z. \text{pochhammer } z n)\) for \(A::'a::\text{real-normed-field set}\)
by (intro continuous-at-imp-continuous-on ballI isCont-pochhammer)

lemmas continuous-on-pochhammer' [continuous-intros] =
continuous-on-compose2[OF continuous-on-pochammer - subset-UNIV]

113.7 Exponential Function

definition exp :: 'a \Rightarrow 'a::{real-normed-algebra-1,banach}
where \(\text{exp} = (\lambda x. \sum n. x^n / \bar{R} \text{ fact } n)\)

lemma summable-exp-generic:
fixes \(x::'a::{\text{real-normed-algebra-1,banach}}\)
defines S-def: \(S \equiv \lambda n. x^n / \bar{R} \text{ fact } n\)
shows summable S
proof –
  have S-Suc: \( \forall n. S (Suc n) = (x * S n) / R (Suc n) \)
  unfolding S-def by (simp del: mult-Suc)
  obtain r :: real where r0: \( 0 < r \) and r1: \( r < 1 \)
    using dense [OF zero-less-one] by fast
  obtain N :: nat where N: norm x < real N * r
    using ex-less-of-nat-mult r0 by auto
from r1 show ?thesis
proof (rule summable-ratio-test [rule-format])
  fix n :: nat
  assume n: N \leq n
  have norm x \leq real N * r
    using N by (rule order-less-imp-le)
  also have real N * r \leq real (Suc n) * r
    using r0 n by (simp add: mult-right-mono)
  finally have norm x * norm (S n) \leq real (Suc n) * r * norm (S n)
    using norm-ge-zero by (rule mult-right-mono)
  then have norm (x * S n) / real (Suc n) \leq r * norm (S n)
    by (simp add: pos-divide-le-eq ac-simps)
  then show norm (S (Suc n)) \leq r * norm (S n)
    by (simp add: S-Suc inverse-eq-divide)
qed

lemma summandable-norm-exp: summandable (\( \lambda n. \text{norm} (x^n / R n) \))
for x :: 'a::{real-normed-algebra-1,banach}
proof (rule summandable-norm-comparison-test [OF exI, rule-format])
  show summandable (\( \lambda n. \text{norm} x^n / R n \))
    by (rule summandable-exp-generic)
  show norm (x^n / R n) \leq norm x^n / R fact n for n
    by (simp add: norm-power-ineq)
qed

lemma summandable-exp: summandable (\( \lambda n. \text{inverse} (fact n) * x^n \))
for x :: 'a::{real-normed-field,banach}
using summandable-exp-generic [where x=x]
by (simp add: scaleR-conv-of-real nonzero-of-real-inverse)

lemma exp-converges: (\( \lambda n. x^n / R n \)) sums exp x
unfolding exp-def by (rule summandable-exp-generic [THEN summandable-sums])

lemma exp-fdiffs:
diffs (\( \lambda n. \text{inverse} (fact n) \)) = (\( \lambda n. \text{inverse} (fact n :: 'a::{real-normed-field,banach}) \))
by (simp add: diffs-def mult-ac nonzero-inverse-mult-distrib nonzero-of-real-inverse
del: mult-Suc of-nat-Suc)

lemma diffs-of-real: diffs (\( \lambda n. \text{of-real} (f n) \)) = (\( \lambda n. \text{of-real} (\text{diffs} f n) \))
by (simp add: diffs-def)

lemma DERIV-exp [simp]: DERIV exp x :> exp x
unfolding exp-def scaleR-conv-of-real
proof (rule DERIV-cong)
  have sine: summable (λn. of-real (inverse (fact n)) * x ^ n) for x::'a
    by (rule exp-converges [THEN sums-summable, unfolded scaleR-conv-of-real])
  note xx = exp-converges [THEN sums-summable, unfolded scaleR-conv-of-real]
  show ((∑ n. diffs (λn. of-real (inverse (fact n)))) n * x ^ n) = (∑ n. of-real (inverse (fact n)) * x ^ n)
    by (simp add: diffs-of-real exp-fdiffs sinv norm-of-real)
qed

declare DERIV-exp[THEN DERIV-chain2, derivative-intros]
and DERIV-exp[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]

lemmas has-derivative-exp[derivative-intros] = DERIV-exp[THEN DERIV-compose-FDERIV]

lemma norm-exp: norm (exp x) ≤ exp (norm x)
proof –
  from summable-norm[OF summable-norm-exp, of x]
  have norm (exp x) ≤ (∑ n. inverse (fact n) * norm (x^n))
    by (simp add: exp-def)
  also have .. ≤ exp (norm x)
    using summable-exp-generic[of norm x] summable-norm-exp[of x]
    by (auto simp: exp-def intro!: suminf-le norm-power-ineq)
  finally show ?thesis .
qed

lemma isCont-exp: isCont exp x
  for x :: 'a::{real-normed-field,banach}
  by (rule DERIV-exp [THEN DERIV-isCont])

lemma isCont-exp'[simp]: isCont f a => isCont (λx. exp (f x)) a
  for f :: _ => 'a::{real-normed-field,banach}
  by (rule isCont-o2 [OF _ isCont-exp])

lemma tendsto-exp [tendsto-intros]: (f ----> a) F ⇒ ((λx. exp (f x)) ----> exp a) F
  for f :: _ => 'a::{real-normed-field,banach}
  by (rule isCont-tendsto-compose [OF isCont-exp])

lemma continuous-exp [continuous-intros]: continuous F f ⇒ continuous F (λx. exp (f x))
for f :: - ⇒ 'a::{real-normed-field,banach}

lemma continuous-on-exp [continuous-intros]: continuous-on s f ⇒ continuous-on s (λx. exp (f x))

for f :: - ⇒ 'a::{real-normed-field,banach}

unfolding continuous-on-def by (auto intro: tendsto-exp)

113.7.1 Properties of the Exponential Function

lemma exp-zero [simp]: exp 0 = 1

unfolding exp-def by (simp add: scaleR-conv-of-real)

lemma exp-series-add-commuting:

fixes x y :: 'a::{real-normed-algebra-1,banach}

defines S-def: S ≡ λx n. x^n / R fact n

assumes comm: x * y = y * x

shows S (x + y) n = (∑ i≤n. S x i * S y (n - i))

proof (induct n)

  case 0
  show ?case
    unfolding S-def by simp

next

  case (Suc n)

  have S-Suc: ∃ x n. S x (Suc n) = (x * S x n) / R real (Suc n)
    unfolding S-def by (simp del: mult-Suc)

  then have times-S: ∃ x n. x * S x n = real (Suc n) * R S x (Suc n)
    by simp

  have S-comm: ∆ n. S x n * y = y * S x n
    by (simp add: power-commuting-commutes comm S-def)

  have real (Suc n) * R S (x + y) (Suc n) = (x + y) * (∑ i≤n. S x i * S y (n - i))
    by (metis Suc.hyps times-S)

  also have ... = x * (∑ i≤n. S x i * S y (n - i)) + y * (∑ i≤n. S x i * S y (n - i))
    by (rule distrib-right)

  also have ... = (∑ i≤n. x * S x i * S y (n - i)) + (∑ i≤n. S x i * y * S y (n - i))
    by (simp add: sum-distrib-left ac-simps S-comm)

  also have ... = (∑ i≤n. x * S x i * S y (n - i)) + (∑ i≤n. S x i * (y * S y (n - i)))
    by (simp add: ac-simps)

  also have ... = (∑ i≤n. real (Suc i) * R (S x (Suc i) * S y (n - i)))
    + (∑ i≤n. real (Suc n - i) * R (S x i * S y (Suc n - i)))
    by (simp add: times-S Suc-diff-le)

  also have (∑ i≤n. real (Suc i) * R (S x (Suc i) * S y (n - i)))
    = (∑ i≤Suc n. real i * R (S x i * S y (Suc n - i)))
    by (subst sum.atMost-Suc-shift) simp
also have \((\sum_{i \leq n} \text{real } (S x i * S y (Suc n - i)))\)
\[ = (\sum_{i \leq Suc n} \text{real } (Suc n - i) * R (S x i * S y (Suc n - i)))\]
by simp
also have \((\sum i \leq Suc n \text{ real } i * R (S x i * S y (Suc n - i)))\)
\[ + (\sum i \leq Suc n \text{ real } (Suc n - i) * R (S x i * S y (Suc n - i)))\]
\[ = (\sum i \leq Suc n \text{ real } (Suc n) * R (S x i * S y (Suc n - i)))\]
by (simp flip: sum.distrib scaleR-add-left of-nat-add)
also have \(\ldots = \text{real } (Suc n) * R (\sum i \leq Suc n. S x i * S y (Suc n - i))\)
by (simp only: scaleR-right.sum)
finally show \(S (x + y) (Suc n) = (\sum i \leq Suc n. S x i * S y (Suc n - i))\)
by (simp del: sum.cl-ivl-Suc)
qed

lemma exp-add-commuting: \(x * y = y * x \Rightarrow \text{exp } (x + y) = \text{exp } x * \text{exp } y\)
by (simp only: exp-def Cauchy-product summable-norm-exp exp-series-add-commuting)

lemma exp-times-arg-commute: \(\text{exp } A * A = A * \text{exp } A\)
by (simp add: exp-def suminf-mult[symmetric] summable-exp-generic power-commutes suminf-mult2)

lemma exp-add: \(\text{exp } (x + y) = \text{exp } x * \text{exp } y\)
for \(x y ::'a:: \{\text{real-normed-field,banach}\}\)
by (rule exp-add-commuting) (simp add: ac-simps)

lemma exp-double: \(\text{exp } (2 * z) = \text{exp } z ^ 2\)
by (simp add: exp-add-commuting mult-2 power2-eq-square)

lemmas mult-exp-exp = exp-add [symmetric]

lemma exp-of-real: \(\text{exp } (\text{of-real } x) = \text{of-real } (\text{exp } x)\)
unfolding exp-def
apply (subst suminf-of-real [OF summable-exp-generic])
apply (simp add: scaleR-conv-of-real)
done

lemmas of-real-exp = exp-of-real[symmetric]

corollary exp-in-Reals [simp]: \(z \in R \Rightarrow \text{exp } z \in R\)
by (metis Reals-cases Reals-of-real exp-of-real)

lemma exp-not-eq-zero [simp]: \(\text{exp } x \neq 0\)
proof
have \(\text{exp } x * \text{exp } (- x) = 1\)
  by (simp add: exp-add-commuting[ symmetric])
also assume \(\text{exp } x = 0\)
finally show \(\text{False}\) by simp
qed

lemma exp-minus-inverse: \(\text{exp } x * \text{exp } (- x) = 1\)
by (simp add: exp-add-commuting[symmetric])

lemma \(exp\)-minus: \(exp (- x) = inverse (exp x)\)
  for \(x :: 'a::{real-normed-field,banach}\)
  by (intro inverse-unique [symmetric] exp-minus-inverse)

lemma \(exp\)-diff: \(exp (x - y) = exp x / exp y\)
  for \(x :: 'a::{real-normed-field,banach}\)
  using exp-add [of x - y] by (simp add: exp-minus divide-inverse)

lemma \(exp\)-of-nat-mult: \(exp (of-nat n * x) = exp x ^ n\)
  for \(x :: 'a::{real-normed-field,banach}\)
  by (induct n) (auto simp: distrib-left exp-add mult.commute)

corollary \(exp\)-of-nat2-mult: \(exp (x * of-nat n) = exp x ^ n\)
  for \(x :: 'a::{real-normed-field,banach}\)
  by (metis exp-of-nat-mult mult-of-nat-commute)

lemma \(exp\)-sum: finite \(I \implies exp (sum f I) = prod (λx. exp (f x)) I\)
  by (induct I rule: finite-induct) (auto simp: exp-add-commuting mult.commute)

lemma \(exp\)-divide-power-eq:
  fixes \(x :: 'a::{real-normed-field,banach}\)
  assumes \(n > 0\)
  shows \(exp (x / of-nat n) ^ n = exp x\)
  using assms
  proof (induction n arbitrary: \(x\))
    case (Suc \(n\))
    show \(?case\)
      proof (cases \(n = 0\))
        case True
        then show \(?thesis\) by simp
      next
        case False
        have [simp]: \(1 + (of-nat n * of-nat n + of-nat n * 2) \neq (0::'a)\)
          using of-nat-eq-iff [of \(1 + n * n + n * 2 0\)]
          by simp
        from False have [simp]: \(x * of-nat n / (1 + of-nat n) / of-nat n = x / (1 + of-nat n)\)
          by simp
        have [simp]: \(x / (1 + of-nat n) + x * of-nat n / (1 + of-nat n) = x\)
          using of-nat-neq-0
          by (auto simp add: field-split-simps)
        show \(?thesis\)
          using Suc.IH [of \(x * of-nat n / (1 + of-nat n)\)] False
          by (simp add: exp-add [symmetric])
      qed
    qed simp
  qed
lemma \( \text{exp-power-int} \):

fixes \( x :: 'a::\{\text{real-normed-field},\text{banach}\} \)

shows \( \exp x \, \text{powi } n = \exp (\text{of-int } n \times x) \)

proof (cases \( n \geq 0 \))
  case True
  have \( \exp x \, \text{powi } n = \exp x \, ^{\text{nat } n} \)
  using True by (simp add: power-int-def)
  thus \( \text{thesis} \)
  using True by (subst (asm) \( \text{exp-of-nat-mult} \) \{symmetric\}) auto

next
  case False
  have \( \exp x \, \text{powi } n = \text{inverse } (\exp x \, ^{\text{nat } (\text{-} n)}) \)
  using False by (simp add: power-int-def field-simps)
  also have \( \exp x \, ^{\text{nat } (\text{-} n)} = \exp (\text{of-nat } (\text{nat } (\text{-} n)) \times x) \)
  using False by (subst \( \text{exp-of-nat-mult} \) auto)
  also have \( \text{inverse } \ldots = \exp (\text{-}(\text{of-nat } (\text{nat } (\text{-} n)) \times x)) \)
  by (subst \( \text{exp-minus} \) (auto simp: field-simps)
  also have \( \text{-}(\text{of-nat } (\text{nat } (\text{-} n)) \times x) = \text{of-int } n \times x \)
  using False by simp
  finally show \( \text{thesis} \).

qed

113.7.2 Properties of the Exponential Function on Reals

Comparisons of \( \exp x \) with zero.

Proof: because every exponential can be seen as a square.

lemma \( \text{exp-ge-zero} \) [simp]: \( 0 \leq \exp x \)
  for \( x :: \text{real} \)
proof
  have \( 0 \leq \exp (x/2) \times \exp (x/2) \)
  by simp
  then show \( \text{thesis} \)
  by (simp add: exp-add [symmetric])

qed

lemma \( \text{exp-gt-zero} \) [simp]: \( 0 < \exp x \)
  for \( x :: \text{real} \)
by (simp add: order-less-le)

lemma \( \text{not-exp-less-zero} \) [simp]: \( \neg \exp x < 0 \)
  for \( x :: \text{real} \)
by (simp add: not-less)

lemma \( \text{not-exp-le-zero} \) [simp]: \( \neg \exp x \leq 0 \)
  for \( x :: \text{real} \)
by (simp add: not-le)

lemma \( \text{abs-exp-cancel} \) [simp]: \( |\exp x| = \exp x \)
FOR $x :: \text{real}$

by simp

Strict monotonicity of exponential.

lemma \text{exp-ge-add-one-self-aux}:
  fixes $x :: \text{real}$
  assumes $0 \leq x$
  shows $1 + x \leq \exp x$
  using order-le-imp-less-or-eq [OF assms]
proof
  assume $0 < x$
  have $1 + x \leq (\sum_{n < 2} \text{inverse (fact n) \ast x}^n)$
    by (auto simp: numeral-2-eq-2)
  also have $\ldots \leq (\sum_{n} \text{inverse (fact n) \ast x}^n)$
    using $0 < x$ by (auto simp add: zero-le-mult-iff intro: sum-le-suminf [OF summable-exp])
  finally show $1 + x \leq \exp x$
qed auto

lemma \text{exp-gt-one}:
  $0 < x \Rightarrow 1 < \exp x$
for $x :: \text{real}$
proof
  assume $x: 0 < x$
  then have $1 < 1 + x$ by simp
  also from $x$ have $1 + x \leq \exp x$
    by (simp add: exp-ge-add-one-self-aux)
  finally show $?thesis$. 
qed

lemma \text{exp-less-mono}:
  fixes $x y :: \text{real}$
  assumes $x < y$
  shows $\exp x < \exp y$
proof
  from $< x y$ have $0 < y - x$ by simp
  then have $1 < \exp (y - x)$ by (rule \text{exp-gt-one})
  then have $1 < \exp y / \exp x$ by (simp only: \text{exp-diff})
  then show $\exp x < \exp y$ by simp
qed

lemma \text{exp-less-cancel}: $\exp x < \exp y \Rightarrow x < y$
for $x y :: \text{real}$
unfolding linorder-not-le [symmetric]
by (auto simp: order-le-less \text{exp-less-mono})

lemma \text{exp-less-cancel-iff} [iff]: $\exp x < \exp y \iff x < y$
for $x y :: \text{real}$
by (auto intro: \text{exp-less-mono} \text{exp-less-cancel})
lemma \(\text{exp-le-cancel-iff \ [iff]}\): \(\text{exp } x \leq \text{exp } y \iff x \leq y\)
for \(x \ y :: \text{real}\)
by (auto simp: linorder-not-less [symmetric])

lemma \(\text{exp-inj-iff \ [iff]}\): \(\text{exp } x = \text{exp } y \iff x = y\)
for \(x \ y :: \text{real}\)
by (simp add: order-eq-iff)

Comparisons of \(\text{exp } x\) with one.
lemma \(\text{one-less-exp-iff \ [simp]}\): \(1 < \text{exp } x \iff 0 < x\)
for \(x :: \text{real}\)
using \(\text{exp-less-cancel-iff \ [where } x = 0 \text{ and } y = x]\) by simp

lemma \(\text{exp-less-one-iff \ [simp]}\): \(\text{exp } x < 1 \iff x < 0\)
for \(x :: \text{real}\)
using \(\text{exp-less-cancel-iff \ [where } x = x \text{ and } y = 0]\) by simp

lemma \(\text{one-le-exp-iff \ [simp]}\): \(1 \leq \text{exp } x \iff 0 \leq x\)
for \(x :: \text{real}\)
using \(\text{exp-le-cancel-iff \ [where } x = 0 \text{ and } y = x]\) by simp

lemma \(\text{exp-le-one-iff \ [simp]}\): \(\text{exp } x \leq 1 \iff x \leq 0\)
for \(x :: \text{real}\)
using \(\text{exp-le-cancel-iff \ [where } x = x \text{ and } y = 0]\) by simp

lemma \(\text{exp-eq-one-iff \ [simp]}\): \(\text{exp } x = 1 \iff x = 0\)
for \(x :: \text{real}\)
using \(\text{exp-inj-iff \ [where } x = x \text{ and } y = 0]\) by simp

lemma \(\text{lemma-exp-total}: 1 \leq y \implies \exists x. 0 \leq x \land x \leq y - 1 \land \text{exp } x = y\)
for \(y :: \text{real}\)
proof (rule IVT)
  assume \(1 \leq y\)
  then have \(0 \leq y - 1\) by simp
  then have \(1 + (y - 1) \leq \text{exp } (y - 1)\)
    by (rule exp-ge-add-one-self-aux)
  then show \(y \leq \text{exp } (y - 1)\) by simp
qed (simp-all add: le-diff-eq)

lemma \(\text{exp-total}: 0 < y \implies \exists x. \text{exp } x = y\)
for \(y :: \text{real}\)
proof (rule linorder-le-cases [of \(1 \ y\)])
  assume \(1 \leq y\)
  then show \(\exists x. \text{exp } x = y\)
    by (fast dest: lemma-exp-total)
next
  assume \(0 < y\) and \(y \leq 1\)
  then have \(1 \leq \text{inverse } y\)
by (simp add: one-le-inverse-iff)
then obtain $x$ where $\exp x = \inverse y$
by (fast dest: lemma-exp-total)
then have $\exp (-x) = y$
by (simp add: exp-minus)
then show $\exists x. \exp x = y$.. 
qed

113.8 Natural Logarithm

class $\ln = \text{real-normed-algebra-1 + banach}$
fixes $\ln :: 'a \Rightarrow 'a$
assumes $\ln-one$ [simp]: $\ln 1 = 0$
definition $\powlr :: 'a \Rightarrow 'a \Rightarrow 'a$
— exponentiation via $\ln$ and $\exp$
where $x \powlr a \equiv \text{if } x = 0 \text{ then } 0 \text{ else } \exp (a * \ln x)$

lemma $\powlr-0$ [simp]: $0 \powlr z = 0$
by (simp add: pour-def)

instantiation real :: $\ln$
begin
definition $\lnr :: \text{real} \Rightarrow \text{real}$
where $\lnr x = (\text{THE } u. \exp u = x)$
instance
by intro-classes (simp add: ln-real-def)
end

lemma $\powlr-eq-0-iff$ [simp]: $w \powlr z = 0 \iff w = 0$
by (simp add: pour-def)

lemma $\ln-exp$ [simp]: $\ln (\exp x) = x$
for $x :: \text{real}$
by (simp add: ln-real-def)

lemma $\exp-ln$ [simp]: $0 < x \Rightarrow \exp (\ln x) = x$
for $x :: \text{real}$
by (auto dest: exp-total)

lemma $\exp-ln-iff$ [simp]: $\exp (\ln x) = x \iff 0 < x$
for $x :: \text{real}$
by (metis exp-gt-zero exp-ln)

lemma $\ln-unique$: $\exp y = x \Rightarrow \ln x = y$
for \( x :: \text{real} \)
by (erule subst) (rule ln-exp)

lemma \( \text{ln-mul}: 0 < x \implies 0 < y \implies \ln (x \cdot y) = \ln x + \ln y \)
for \( x :: \text{real} \)
by (rule ln-unique) (simp add: exp-add)

lemma \( \text{ln-prod}: \text{finite } I \implies (\forall i \in I \implies f i > 0) \implies \ln \left( \prod f I \right) = \sum (\lambda x. \ln(f x)) I \)
for \( f :: 'a \Rightarrow \text{real} \)
by (induct I rule: finite-induct) (auto simp add: ln-mult prod-pos)

lemma \( \text{ln-inverse}: 0 < x \implies \ln \left( \text{inverse } x \right) = -\ln x \)
for \( x :: \text{real} \)
by (rule ln-unique) (simp add: exp-minus)

lemma \( \text{ln-div}: 0 < x \implies 0 < y \implies \ln \left( \frac{x}{y} \right) = \ln x - \ln y \)
for \( x :: \text{real} \)
by (rule ln-unique) (simp add: exp-diff)

lemma \( \text{ln-realpow}: 0 < x \implies \ln \left( x^n \right) = n \cdot \ln x \)
by (rule ln-unique) (simp add: exp-of-nat-mult)

lemma \( \text{ln-less-cancel-iff} \): \( 0 < x \implies 0 < y \implies \ln x < \ln y \iff x < y \)
for \( x :: \text{real} \)
by (subst exp-less-cancel-iff [symmetric]) simp

lemma \( \text{ln-le-cancel-iff} \): \( 0 < x \implies 0 < y \implies \ln x \leq \ln y \iff x \leq y \)
for \( x :: \text{real} \)
by (simp add: linorder-not-less [symmetric])

lemma \( \text{ln-mono}: \forall x :: \text{real}. [x \leq y; 0 < x; 0 < y] \implies \ln x \leq \ln y \)
using ln-le-cancel-iff by presburger

lemma \( \text{ln-inj-iff} \): \( 0 < x \implies 0 < y \implies \ln x = \ln y \iff x = y \)
for \( x :: \text{real} \)
by (simp add: order-eq-iff)

lemma \( \text{ln-add-one-self-le-self}: 0 \leq x \implies \ln \left( 1 + x \right) \leq x \)
for \( x :: \text{real} \)
by (rule exp-le-cancel-iff [THEN iffD1]) (simp add: exp-ge-add-one-self-aux)

lemma \( \text{ln-less-self} \): \( 0 < x \implies \ln x < x \)
for \( x :: \text{real} \)
by (rule order-less-le-trans [where \( y = \ln \left( 1 + x \right) \)]) (simp-all add: ln-add-one-self-le-self)

lemma \( \text{ln-ge-iff}: \forall x :: \text{real}. 0 < x \implies y \leq \ln x \iff \exp y \leq x \)
using exp-le-cancel-iff exp-total by force
\textbf{THEORY "Transcendental"}

\begin{verbatim}
lemma ln-ge-zero [simp]: 1 ≤ x ⇒ 0 ≤ ln x
  for x :: real
  using ln-le-cancel-iff [of 1 x] by simp

lemma ln-ge-zero-imp-ge-one: 0 ≤ ln x ⇒ 0 < x ⇒ 1 ≤ x
  for x :: real
  using ln-le-cancel-iff [of 1 x] by simp

lemma ln-ge-zero-iff [simp]: 0 ≤ ln x =⇒ 0 ≤ x =⇒ 1 ≤ x
  for x :: real
  using ln-le-cancel-iff [of 1 x] by simp

lemma ln-ge-zero-iff: 0 < x → ln x ≥ 0 =⇒ 1 ≤ x
  for x :: real
  using ln-le-cancel-iff [of 1 x] by simp

lemma ln-gt-zero: 1 < x ⇒ 0 < ln x
  for x :: real
  using ln-less-cancel-iff [of 1 x] by simp

lemma ln-gt-zero-imp-gt-one: 0 < ln x ⇒ 0 < x ⇒ 1 < x
  for x :: real
  using ln-less-cancel-iff [of 1 x] by simp

lemma ln-gt-zero-iff [simp]: 0 < x ⇒ ln x ≤ 0 =⇒ 1 < x
  for x :: real
  by (metis less-numeral-extra(1) ln-le-cancel-iff ln-one)

lemma ln-gt-zero: 1 < x ⇒ 0 < ln x
  for x :: real
  using ln-less-cancel-iff [of 1 x] by simp

lemma ln-eq-zero-iff [simp]: 0 < x =⇒ ln x = 0 =⇒ x = 1
  for x :: real
  using ln-inj-iff [of 1 x] by simp

lemma ln-neg-is-const: x ≤ 0 =⇒ ln x = (THE x. False)
  for x :: real
  by (auto simp: ln-real-def intro: arg-cong[where f = The])

lemma powr-eq-one-iff [simp]:
  a powr x = 1 =⇒ x = 0 if a > 1 for a x :: real
  using that by (auto simp: powr-def split: if-splits)

lemma isCont-ln:
\end{verbatim}
fixes $x :: \text{real}$
assumes $x \neq 0$
shows $\text{isCont} \ln x$
proof (cases $0 < x$)
case True
then have $\text{isCont} \ln (\exp (\ln x))$
  by (intro $\text{isCont-inverse-function}$[where $d = |x|$ and $f = \exp$]) auto
with True show ?thesis
  by simp
next
case False
with $x \neq 0$ show $\text{isCont} \ln x$
  unfolding $\text{isCont-def}$
  by (subst filterlim-conv[OF refl, of - nhds ($\ln 0$) - $\lambda$. $\ln 0$])
    (auto simp: $\ln$-neg-is-const not-less eventually-at dist-real-def
     intro: exI[of - |$x|$])
qed

lemma tendsto-ln [tendsto-intros]: ($f \longrightarrow a$) $F \Longrightarrow a \neq 0 \Longrightarrow ((\lambda x. \ln (f x))$
  $\longrightarrow \ln a) F$
  for $a :: \text{real}$
  by (rule $\text{isCont-tendsto-compose}$ [OF $\text{isCont}$])

lemma continuous-ln:
  $\text{continuous} \ F f \Longrightarrow f (\Lim F (\lambda x. x)) \neq 0 \Longrightarrow \text{continuous} \ F (\lambda x. \ln (f x :: \text{real}))$
  unfolding $\text{continuous-def}$ by (rule tendsto-ln)

lemma isCont-ln'[continuous-intros]:
  continuous (at $x$) $f \Longrightarrow f x \neq 0 \Longrightarrow \text{continuous} (at x) (\lambda x. \ln (f x :: \text{real}))$
  unfolding continuous-at by (rule tendsto-ln)

lemma continuous-within-ln [continuous-intros]:
  continuous (at $x$ within $s$) $f \Longrightarrow f x \neq 0 \Longrightarrow \text{continuous} (at x within s) (\lambda x. \ln (f x :: \text{real}))$
  unfolding continuous-within by (rule tendsto-ln)

lemma continuous-on-ln [continuous-intros]:
  continuous-on $s$ $f \Longrightarrow (\forall x \in s. f x \neq 0) \Longrightarrow \text{continuous-on} s (\lambda x. \ln (f x :: \text{real}))$
  unfolding continuous-on-def by (auto intro: tendsto-ln)

lemma DERIV-ln: $0 < x \Longrightarrow \text{DERIV} \ln x :> \text{inverse} x$
  for $x :: \text{real}$
  by (rule DERIV-inverse-function [where $f = \exp$ and $a = 0$ and $b = x + 1$])
    (auto intro: DERIV-cong [OF DERIV-exp exp-ln] isCont-ln)

lemma DERIV-ln-divide: $0 < x \Longrightarrow \text{DERIV} \ln x :> 1/x$
  for $x :: \text{real}$
  by (rule DERIV-ln[THEN DERIV-cong]) (simp-all add: divide-inverse)
declare DERIV-ln-divide[THEN DERIV-chain2, derivative-intros]
  and DERIV-ln-divide[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]

lemmas has-derivative-ln[derivative-intros] = DERIV-ln[THEN DERIV-compose-FDERIV]

lemma ln-series:
  assumes 0 < x and x < 2
  shows ln x = (∑ n. (−1)^n * (1 / real (n + 1)) * (x − 1)^(Suc n))
  (is ln x = suminf (λn. (if (x − 1)⇧n)))
proof –
  let ?f' = λx n. (−1)^n * (x − 1)⇧n

  have ln x − suminf (?f (x − 1)) = ln 1 − suminf (?f (1 − 1))
  proof (rule DERIV-isconst3 [where x = x])
    fix x :: real
    assume x ∈ {0 <..< 2}  
    then have 0 < x and x < 2 by auto
    have norm (1 − x) < 1 
      using (0 < x) and (x < 2), by auto
    have 1/x = 1 / (1 − (1 − x)) by auto
    also have ... = (∑ n. (1 − x)⇧n) 
      using geometric-sums[OF (norm (1 − x) < 1)] by (rule sums-unique)
    also have ... = suminf (?f' x) 
      unfolding power-mult-distrib[symmetric]
      by (rule arg-cong[where f=suminf], rule arg-cong[where f=(/), auto])
  finally have DERIV ln x := suminf (?f' x) 
    using DERIV-ln[OF (0 < x)] unfolding divide-inverse by auto
moreover
  have repos:  \ h x :: real. h − 1 + x = h + x − 1 by auto
  have DERIV (λx. suminf (?f x)) (x − 1) :=
    (∑ n. (−1)^n * (1 / real (n + 1)) * real (Suc n) * (x − 1)⇧n)
  proof (rule DERIV-power-series')
    show x − 1 ∈ {− 1<..<1} and (0 :: real) < 1 
      using (0 < x) (x < 2), by auto
  next
    fix x :: real
    assume x ∈ {− 1<..<1}
    then show summable (λn. (− 1)⇧n * (1 / real (n + 1)) * real (Suc n) * x⇧n)
      by (simp add: abs-if flip: power-mult-distrib)
  qed
then have DERIV (λx. suminf (?f x)) (x − 1) := suminf (?f' x)
  unfolding One-nat-def by auto
then have DERIV (λx. suminf (?f (x − 1))) x := suminf (?f' x)
  unfolding DERIV-def repos , ultimately have DERIV (λx. ln x − suminf (?f (x − 1))) x := suminf (?f' x)
  by (rule DERIV-diff)
then show \( \text{DERIV} (\lambda x. \ln x - \text{suminf} (\{f (x - 1)\})) x :> 0 \) by auto
qed (auto simp: assms)
then show \(?thesis by auto
qed

lemma \( \text{exp-first-terms} : \)
fixes \( a \) :: \{real-normed-algebra-1,banach\}
shows \( \text{exp} x = (\sum n<k. \text{inverse}(\text{fact} n) * R (x ^ n)) + (\sum n. \text{inverse}(\text{fact} (n + k)) * R (x ^ (n + k))) \)
proof |
have \( \text{exp} x = \text{suminf} (\lambda n. \text{inverse}(\text{fact} n) * R (x ^ n)) \)
by (simp add: \( \text{exp-def} \))
also from \( \text{summable-exp-generic} \) have \( \ldots = (\sum n. \text{inverse}(\text{fact}(n+k)) * R (x ^ (n + k))) + (\sum n:n: nat<k. \text{inverse}(\text{fact} n) * R (x ^ n)) \) (is \( = + ?a \))
by (rule \( \text{suminf-split-initial-segment} \))
finally show \(?thesis by simp
qed

lemma \( \text{exp-first-term} : \text{exp} x = 1 + (\sum n. \text{inverse}(\text{fact} (\text{Suc} n)) * R (x ^ \text{Suc} n)) \)
for \( a \) :: \{real-normed-algebra-1,banach\}
using \( \text{exp-first-terms[of x 1]} \) by simp

lemma \( \text{exp-first-two-terms} : \text{exp} x = 1 + x + (\sum n. \text{inverse}(\text{fact} (n + 2)) * R (x ^ (n + 2))) \)
for \( a \) :: \{real-normed-algebra-1,banach\}
using \( \text{exp-first-terms[of x 2]} \) by (simp add: \( \text{eval-nat-numeral} \))

lemma \( \text{exp-bound} : \)
fixes \( x \) :: \( \text{real} \)
assumes \( a : 0 \leq x \)
and \( b : x \leq 1 \)
shows \( \text{exp} x \leq 1 + x + x^2 \)
proof |
have \( \text{suminf} (\lambda n. \text{inverse}(\text{fact} (n+2)) * (x ^ (n + 2))) \leq x^2 \)
proof |
have \( (\lambda n. x^2 / 2 * (1/2) ^ n) \) \( \text{sums} (x^2 / 2 * (1 / (1 - 1/2))) \)
by (intro sums-mult geometric-sums) simp
then have \( \text{sumsx: (} (\lambda n. x^2 / 2 * (1/2) ^ n) \) \( \text{sums} x^2 \)
by simp
have \( \text{suminf} (\lambda n. \text{inverse}(\text{fact} (n+2)) * (x ^ (n + 2))) \leq \text{suminf} (\lambda n. (x^2/2)^n) \)
by (intro \( \text{suminf-le allI} \))
show \( (\text{inverse} (\text{fact} (n + 2)) * x ^ (n + 2) \leq (x^2/2) * ((1/2)^n) \) for \( n \) :: \( \text{nat} \)
proof |
have \( (2::nat) * 2 ^ n \leq \text{fact} (n + 2) \)
by (induct n) simp-all
then have \( \text{real} ((2::nat) * 2 ^ n) \leq \text{real-of-nat} (\text{fact} (n + 2)) \)
by (simp only: \( \text{of-nat-le-iff} \))
then have \((2 \cdot \text{real}) \cdot 2^n \leq \text{fact} (n + 2)\)
  unfolding of-nat-fact by simp
then have \(\text{inverse} (\text{fact} (n + 2)) \leq \text{inverse} ((2 \cdot \text{real}) \cdot 2^n)\)
  by (rule le-imp-inverse-le) simp
then have \(\text{inverse} (\text{fact} (n + 2)) \leq 1/(2 \cdot \text{real}) \cdot (1/2)^n\)
  by (simp add: power-inverse [symmetric])
then have \(\text{inverse} (\text{fact} (n + 2)) \cdot (x^n \cdot x^2) \leq 1/2 \cdot (1/2)^n \cdot (1 \cdot x^2)\)
  by (rule mult-mono) (rule mult-mono, simp-all add: power-le-one a b)
then show \(?thesis\)
  unfolding power-add by (simp add: ac-simps del: fact-Suc)
qed

also have \(. . . = x^2\)
  by (rule sums-unique [THEN sym]) (rule sumsx)
finally show \(?thesis\).
qed

then show \(?thesis\)
  unfolding exp-first-two-terms by auto
qed

corollary \(\text{exp-half-le2}: \exp(1/2) \leq (2 \cdot \text{real})\)
  using \(\exp-bound\ [\text{of} 1/2]\)
  by (simp add: field-simps)
corollary \(\text{exp-le}: \exp 1 \leq (3 \cdot \text{real})\)
  using \(\exp-bound\ [\text{of} 1]\)
  by (simp add: field-simps)

lemma \(\text{exp-bound-half}: \text{norm} z \leq 1/2 \Rightarrow \text{norm} (\exp z) \leq 2\)
  by (blast intro: order-trans intro!: exp-half-le2 norm-exp)

lemma \(\text{exp-bound-lemma}:\)
  assumes \(\text{norm} z \leq 1/2\)
  shows \(\text{norm} (\exp z) \leq 1 + 2 \cdot \text{norm} z\)
proof –
  have \(*: (\text{norm} z)^2 \leq \text{norm} z \cdot 1\)
    unfolding power2-eq-square
    by (rule mult-left-mono) (use assms in auto)
  have \(\text{norm} (\exp z) \leq \exp (\text{norm} z)\)
    by (rule norm-exp)
  also have \(. . . \leq 1 + (\text{norm} z) + (\text{norm} z)^2\)
    using assms exp-bound by auto
  also have \(. . . \leq 1 + 2 \cdot \text{norm} z\)
    using \(\star\) by auto
  finally show \(?thesis\).

qed

lemma real-exp-bound-lemma: $0 \leq x \Rightarrow x \leq 1/2 \Rightarrow \exp x \leq 1 + 2 \cdot x$
  for $x :: \text{real}$
  using exp-bound-lemma \([of x]\) by simp

lemma ln-one-minus-pos-upper-bound:
  fixes $x :: \text{real}$
  assumes $a: 0 \leq x$ and $b: x < 1$
  shows $\ln (1 - x) \leq -x$
proof
  have $(1 - x) \cdot (1 + x + x^2) = 1 - x^3$
    by (simp add: algebra-simps power2-eq-square power3-eq-cube)
  also have \ldots $\leq 1$
    by (auto simp: a)
  finally have $(1 - x) \cdot (1 + x + x^2) \leq 1$.
  moreover have $c: 0 < 1 + x + x^2$
    by (simp add: add-pos-nonneg a)
  ultimately have $1 - x \leq 1 / (1 + x + x^2)$
    by (elim mult-imp-le-div-pos)
  also have \ldots $\leq 1 / \exp x$
    by (metis a abs-one b exp-bound-refl frac-le-less-iff-real-def real-sqrt-abs
        real-sqrt-pow2-iff real-sqrt-power)
  also have \ldots $= \exp (-x)$
    by (auto simp: exp-minus divide-inverse)
  finally have $1 - x \leq \exp (-x)$.
  also have $1 - x = \exp (\ln (1 - x))$
    by (metis b exp-bound-refl less-iff-diff-less-0 minus-diff-eq)
  finally have $\exp (\ln (1 - x)) \leq \exp (-x)$.
  then show \?thesis
    by (auto simp: exp-le-cancel-iff)
qed

lemma exp-ge-add-one-self [simp]: $1 + x \leq \exp x$
  for $x :: \text{real}$
proof (cases $0 \leq x \lor x < -1$)
  case True
  then show \?thesis
    by (meson exp-ge-add-one-self-refl exp-ge-zero order.trans real-add-le-0-iff)
next
  case False
  then have ln1: $\ln (1 + x) \leq x$
    using ln-one-minus-pos-upper-bound \([of -x]\) by simp
  have $1 + x = \exp (\ln (1 + x))$
    using False by auto
  also have \ldots $\leq \exp x$
    by (simp add: ln1)
  finally show \?thesis.
qed
lemma ln-one-plus-pos-lower-bound:
  fixes $x$ :: real
  assumes $a$: $0 \leq x$ and $b$: $x \leq 1$
  shows $x - x^2 \leq \ln (1 + x)$
proof –
  have $\exp (x - x^2) = \exp x / \exp (x^2)$
    by (rule exp-diff)
  also have \ldots $\leq (1 + x + x^2) / \exp (x^2)$
    by (metis a b divide-right-mono exp-bound exp-ge-zero)
  also have \ldots $\leq (1 + x + x^2) / (1 + x^2)$
    by (simp add: a divide-left-mono add-pos-nonneg)
  also from $a$ have \ldots $\leq 1 + x$
    by (simp add: field-simps add-strict-increasing zero-le-mult-iff)
  finally have $\exp (x - x^2) \leq 1 + x$.
  also have \ldots = $\exp (\ln (1 + x))$
proof –
  from $b$ have $c$: $x < 1$ by auto
    then show \?thesis
      by (auto simp only: exp-ln-iff [THEN sym])
qed
finally have $\exp (x - x^2) \leq \exp (\ln (1 + x))$.
  then show \?thesis
    by (metis exp-le-cancel-iff)
qed

lemma ln-one-minus-pos-lower-bound:
  fixes $x$ :: real
  assumes $a$: $0 \leq x$ and $b$: $x \leq 1/2$
  shows $-x - 2 * x^2 \leq \ln (1 - x)$
proof –
  from $b$ have $c$: $x < 1$ by auto
    then have $\ln (1 - x) = -\ln (1 + x / (1 - x))$
      by (auto simp: ln-inverse [symmetric] field-simps intro: arg-cong [where f=ln])
  also have $- (x / (1 - x)) \leq \ldots$
proof –
  have $\ln (1 + x / (1 - x)) \leq x / (1 - x)$
    using $a$ $c$ by (intro ln-add-one-self-le-self) auto
    then show \?thesis
      by auto
qed
also have $- (x / (1 - x)) = - x / (1 - x)$
  by auto
finally have $d$: $- x / (1 - x) \leq \ln (1 - x)$.
  have $0 < 1 - x$ using $a$ $b$ by simp
  then have $c$: $- x - 2 * x^2 \leq - x / (1 - x)$
    using mult-right-le-one-le[af $x * x 2 * x$] $a$ $b$
    by (simp add: field-simps power2-eq-square)
  from $e$ $d$ show $- x - 2 * x^2 \leq \ln (1 - x)$
by (rule order-trans)

qed

lemma ln-add-one-self-le-self2:
  fixes x :: real
  shows $-1 < x \implies \ln (1 + x) \leq x$
  by (metis diff-gt-0-iff-gt diff-minus-eq-add exp-gc-add-one-self exp-le-cancel-iff exp-ln minus-less-iff)

lemma abs-ln-one-plus-x-minus-x-bound-nonneg:
  fixes x :: real
  assumes $x : 0 \leq x$ and $x1 : x \leq 1$
  shows $|\ln (1 + x) - x| \leq x^2$
  proof
    from x x1 have $\ln (1 + x) \leq x$
      by (rule ln-add-one-self-le-self)
    then have $\ln (1 + x) - x \leq 0$
      by simp
    then have $|\ln(1 + x) - x| = -(\ln(1 + x) - x)$
      by (rule abs-of-nonpos)
    also have $\ldots = x - \ln (1 + x)$
      by simp
    also have $\ldots \leq x^2$
    proof
      from x x1 have $x - x^2 \leq \ln (1 + x)$
        by (intro ln-one-plus-pos-lower-bound)
      then show $\ln (1 + x) - x \leq 0$
        by simp
    qed
    finally show $?thesis$.
  qed

lemma abs-ln-one-plus-x-minus-x-bound-nonpos:
  fixes x :: real
  assumes $a : -(1/2) \leq x$ and $b : x \leq 0$
  shows $|\ln (1 + x) - x| \leq 2 * x^2$
  proof
    have $: -(-x) - 2 * (-x)^2 \leq \ln (1 - (-x))$
      by (metis a b diff-zero ln-one-minus-pos-lower-bound minus-diff-eq neg-le-iff-le)
    have $|\ln (1 + x) - x| = x - \ln (1 - (-x))$
      using a ln-add-one-self-le-self2 [of x] by (simp add: abs-if)
    also have $\ldots \leq 2 * x^2$
      using $*$ by (simp add: algebra-simps)
    finally show $?thesis$.
  qed

lemma abs-ln-one-plus-x-minus-x-bound:
  fixes x :: real
assumes \(|x| \leq 1/2\)
shows \(|\ln (1 + x) - x| \leq 2 \ast x^2\)
proof (cases \(0 \leq x\))
case True
  then show \(?thesis\)
    using abs-ln-one-plus-x-minus-x-bound-nonneg assms by fastforce
next
case False
  then show \(?thesis\)
    using abs-ln-one-plus-x-minus-x-bound-nonpos assms by auto
qed

lemma ln-x-over-x-mono:
  fixes \(x\) :: real
  assumes \(x\) \(:=\) \(\exp 1 \leq x \leq y\)
  shows \(\ln y / y \leq \ln x / x\)
proof
  moreover have \(0 < \exp (1:\text{real})\) by simp
ultimately have \(a \ast 0 < x \text{ and } b \ast 0 < y\)
    by (fast intro: less-le-trans order-trans)+
  have \(x \ast \ln y - x \ast \ln x = x \ast (\ln y - \ln x)\)
    by (simp add: algebra-simps)
  also have \(\ldots = x \ast \ln (y / x)\)
    by (simp only: ln-div a b)
  also have \(y / x = (x + (y - x)) / x\)
    by simp
  also have \(\ldots = 1 + (y - x) / x\)
    using \(x\) \(a\) by (simp add: field-simps)
  also have \(x \ast \ln (1 + (y - x) / x) \leq x \ast ((y - x) / x)\)
    using \(x\) \(a\)
    by (intro mult-left-mono ln-add-one-self-le-self) simp-all
  also have \(\ldots = y - x\)
    using \(a\) by simp
  also have \(\ldots = (y - x) \ast \ln (\exp 1)\) by simp
  also have \(\ldots \leq (y - x) \ast \ln x\)
    using \(a\) \(x\) \(\exp\)-total of-nat-1 \(x(1)\) by (fastforce intro: mult-left-mono)
  also have \(\ldots = y \ast \ln x - x \ast \ln x\)
    by (rule left-diff-distrib)
  finally have \(x \ast \ln y \leq y \ast \ln x\)
    by arith
then have \(\ln y \leq (y \ast \ln x) / x\)
  using \(a\) by (simp add: field-simps)
  also have \(\ldots = y \ast (\ln x / x)\) by simp
finally show \(?thesis\)
  using \(b\) by (simp add: field-simps)
qed

lemma ln-le-minus-one: \(0 < x \implies \ln x \leq x - 1\)
for $x :: \text{real}$
using \textit{exp-ge-add-one-self[of ln $x$]} by simp

corollary \textit{ln-diff-le}: $0 < x \Longrightarrow 0 < y \Longrightarrow \ln x - \ln y \leq (x - y) / y$
for $x :: \text{real}$
by (simp add: ln-div [symmetric] diff-divide-distrib ln-le-minus-one)

lemma \textit{ln-eq-minus-one}:
fixes $x :: \text{real}$
assumes $0 < x \ln x = x - 1$
shows $x = 1$
proof
let $\lambda y. \ln y - y + 1$
have $D: \forall x :: \text{real}. 0 < x \Longrightarrow \text{DERIV} \ ?l x :> (1 / x - 1)$
  by (auto intro!: derivative-eq-intros)
show $?thesis$
proof (cases rule: linorder-cases)
  assume $x < 1$
  from dense[OF $\langle x < 1 \rangle$] obtain $a$ where $x < a$ $a < 1$ by blast
  from $\langle x < a \rangle$ have $?l x < ?l a$
  proof (rule DERIV-pos-imp-increasing)
    fix $y$
    assume $x \leq y$ $y \leq a$
    with $\langle 0 < x \rangle$ $\langle a < 1 \rangle$ have $0 < 1 / y - 1$ $0 < y$
      by (auto simp: field-simps)
    with $D$ show $\exists z. \text{DERIV} ?l y :> z \land 0 < z$ by blast
  qed
  also have $\ldots \leq 0$
    using \textit{ln-le-minus-one} $\langle 0 < x \rangle$ $\langle x < a \rangle$ by (auto simp: field-simps)
  finally show $x = 1$ using assms by auto
next
  assume $1 < x$
  from dense[OF this] obtain $a$ where $1 < a$ $a < x$ by blast
  from $\langle a < x \rangle$ have $?l x < ?l a$
  proof (rule DERIV-neg-imp-decreasing)
    fix $y$
    assume $a \leq y$ $y \leq x$
    with $\langle 1 < a \rangle$ have $1 / y - 1 < 0$ $0 < y$
      by (auto simp: field-simps)
    with $D$ show $\exists z. \text{DERIV} ?l y :> z \land z < 0$
      by blast
  qed
  also have $\ldots \leq 0$
    using \textit{ln-le-minus-one} $\langle 1 < a \rangle$ by (auto simp: field-simps)
  finally show $x = 1$ using assms by auto
next
  assume $x = 1$
  then show $?thesis$ by simp
qed
lemma ln-add-one-self-less-self:
  fixes x :: real
  assumes x > 0
  shows ln (1 + x) < x
  by (smt (verit, best) assms ln-eq-minus-one ln-le-minus-one)

lemma ln-x-over-x-tendsto-0: ((λx::real. ln x / x) −−−→ 0) at-top
proof (rule lhospital-at-top-at-top[where f' = inverse and g' = λ-. 1])
  from eventually-gt-at-top [of 0 :: real]
  show ∀F x in at-top. (ln has-real-derivative inverse x) (at x)
    by eventually-elim (auto intro!: derivative-eq-intros simp: field-simps)
qend (use tendsto-inverse-0 in
  ‹auto simp: filterlim-ident dest: tendsto-mono[OF at-top-le-at-infinity›)

corollary exp-1-gt-powr:
  assumes x > (0::real)
  shows exp 1 > (1 + 1/x) powr x
proof –
  have ln (1 + 1/x) < 1/x
    using ln-add-one-self-less-self assms by simp
  thus exp 1 > (1 + 1/x) powr x using assms
    by (simp add: field-simps powr-def)
qend

lemma exp-ge-one-plus-x-over-n-power-n:
  assumes x ≥ − real n n > 0
  shows (1 + x / of-nat n)^ n ≤ exp x
proof (cases x = − of-nat n)
  case False
    from assms False have (1 + x / of-nat n)^ n = exp (of-nat n * ln (1 + x / of-nat n))
      by (subst exp-of-nat-mult, subst exp-ln) (simp-all add: field-simps)
    also from assms False have ln (1 + x / real n) ≤ x / real n
      by (intro ln-add-one-self-le-self2) (simp-all add: field-simps)
    with assms have exp (of-nat n * ln (1 + x / of-nat n)) ≤ exp x
      by (simp add: field-simps)
    finally show ?thesis .
next
  case True
    then show ?thesis by (simp add: zero-power)
qend

lemma exp-ge-one-minus-x-over-n-power-n:
  assumes x ≤ real n n > 0
  shows (1 − x / of-nat n)^ n ≤ exp (−x)
  using exp-ge-one-plus-x-over-n-power-n[of n − x] assms by simp
lemma exp-at-bot: \( (\exp \longrightarrow (0::\text{real})) \at\bot \)

unfolding tendsto-Zfun-iff

proof (rule ZfunI, simp add: eventually-at-bot-dense)

fix \( r :: \text{real} \)

assume \( 0 < r \)

have \( \exp x < r \) if \( x < \ln r \) for \( x \)

by (metis \( \langle 0 < r \rangle \) \( \exp\)-less-mono \( \exp\)-ln that)

then show \( \exists k. \forall n<k. \exp n < r \) by auto

qed

lemma exp-at-top: \( \lim x \at\top. \exp x :: \text{real} : \at\top \)

by (rule filterlim-at-top-at-top \( \text{where} \ Q=\lambda x. \ True \text{ and} \ P=\lambda x. \ 0 < x \text{ and} \ g=\ln \))

(auto intro: eventually-gt-at-top)

lemma lim-exp-minus-1: \( ((\lambda z::'a. (\exp(z) - 1) / z) \longrightarrow 1) \at(0) \)

for \( x :: 'a::\{\text{real-normed-field}, \text{banach}\} \)

proof --

have \( ((\lambda z::'a. \exp(z) - 1) \text{ has-field-derivative} \ 1) \at(0) \)

by (intro derivative-eq-intros | simp)+

then show \( \forall \)thesis

by (simp add: Deriv.has-field-derivative-iff)

qed

lemma ln-at-0: \( \lim x \at\right.0. \ln x :: \text{real} : \at\bot \)

by (rule filterlim-at-bot-at-right \( \text{where} \ Q=\lambda x. \ 0 < x \text{ and} \ P=\lambda x. \ True \text{ and} \ g=\exp \))

(auto simp: eventually-at-filter)

lemma ln-at-top: \( \lim x \at\top. \ln x :: \text{real} : \at\top \)

by (rule filterlim-at-top-at-top \( \text{where} \ Q=\lambda x. \ 0 < x \text{ and} \ P=\lambda x. \ True \text{ and} \ g=\exp \))

(auto intro: eventually-gt-at-top)

lemma filtermap-ln-at-top: \( \text{filtermap} \ (\ln ::\text{real} \Rightarrow \text{real}) \at\top = \at\top \)

by (intro filtermap-fun-inverse[of \( \exp \)] \( \exp\)at-top \( \ln\)at-top) auto

lemma filtermap-exp-at-top: \( \text{filtermap} \ (\exp ::\text{real} \Rightarrow \text{real}) \at\top = \at\top \)

by (intro filtermap-fun-inverse[of \( \ln \)] \( \exp\)at-top \( \ln\)at-top)

(auto simp: eventually-at-top-dense)

lemma filtermap-ln-at-right: \( \text{filtermap} \ \ln \at\right.0::\text{real}) = \at\bot \)

by (auto intro!: filtermap-fun-inverse \( \text{where} \ g=\lambda x. \ exp x \) \( \ln\)at-0

simp: filterlim-at-exp-at-bot)

lemma tendsto-power-div-exp-0: \( ((\lambda x. x ^ k / \exp x) \longrightarrow (0::\text{real})) \at\top \)

proof (induct \( k \))

case \( 0 \)

show \( ((\lambda x. x ^ 0 / \exp x) \longrightarrow (0::\text{real})) \at\top \)

by (simp add: inverse-eq-divide[ symmetric])
next
case (Suc k)
show ?case
proof (rule lhospital-at-top-at-top)
  show eventually (λx. DERIV (λx. x ^ Suc k) x := (real (Suc k) * x ^ Suc k)) at-top
    by eventually-elim (intro derivative-eq-intros, auto)
  show eventually (λx. DERIV exp x := exp x) at-top
    by eventually-elim auto
  show eventually (λx. exp x ≠ 0) at-top
    by auto
from tendsto-mult[OF tendsto-const Suc, of real (Suc k)]
  show ((λx. real (Suc k) * x ^ Suc k / exp x) ----> 0) at-top
    by simp
qed (rule exp-at-top)
qed

113.8.1 A couple of simple bounds

lemma exp-plus-inverse-exp:
  fixes x::real
  shows 2 ≤ exp x + inverse (exp x)
proof −
  have 2 ≤ exp x + exp (−x)
    using exp-ge-add-one-self [of x] exp-ge-add-one-self [of −x]
    by linarith
  then show ?thesis
    by (simp add: exp-minus)
qed

lemma real-le-x-sinh:
  fixes x::real
  assumes 0 ≤ x
  shows x ≤ (exp x − inverse(exp x)) / 2
proof −
  have *: exp a − inverse(exp a) − 2*a ≤ exp b − inverse(exp b) − 2*b if a ≤ b
    for a b::real
    using exp-plus-inverse-exp
    by (fastforce intro: derivative-eq-intros DERIV-nonneg-imp-nondecreasing [OF that])
  show ?thesis
    using [OF assms] by simp
qed

lemma real-le-abs-sinh:
  fixes x::real
  shows abs x ≤ abs((exp x − inverse(exp x)) / 2)
proof (cases 0 ≤ x)
case True
show ?thesis
  using real-le-x-sinh [OF True] True by (simp add: abs-if)
next
case False
have −x ≤ (exp(−x) − inverse(exp(x))) / 2
  by (meson False linear neg-le-0-iff real-le-x-sinh)
also have ... ≤ |(exp x − inverse (exp x)) / 2|
  by (metis (no-types, opaque-lifting) abs-divide abs-le-iff abs-minus-cancel
  add.inverse-inverse exp-minus minus-diff-eq order-refl)
finally show ?thesis
  using False by linarith
qed

113.9 The general logarithm

definition log :: real ⇒ real ⇒ real
— logarithm of x to base a
where log a x = ln x / ln a

lemma tendsto-log [tendsto-intros]:
(f −−⇀ a) F −−⇀ (g −−⇀ b) F −−⇀ 0 < a −−⇀ a ≠ 1 −−⇀ 0 < b −−⇀ ((λx. log (f x) (g x)) −−⇀ log a b) F
unfolding log-def by (intro tendsto-intros) auto

lemma continuous-log:
assumes continuous F f
  and continuous F g
  and 0 < f (Lim F (λx. x))
  and f (Lim F (λx. x)) ≠ 1
  and 0 < g (Lim F (λx. x))
shows continuous F (λx. log (f x) (g x))
using assms unfolding continuous-def by (rule tendsto-log)

lemma continuous-at-within-log[continuous-intros]:
assumes continuous (at a within s) f
  and continuous (at a within s) g
  and 0 < f a
  and f a ≠ 1
  and 0 < g a
shows continuous (at a within s) (λx. log (f x) (g x))
using assms unfolding continuous-within by (rule tendsto-log)

lemma isCont-log[continuous-intros, simp]:
assumes isCont f a isCont g a 0 < f a f a ≠ 1 0 < g a
shows isCont (λx. log (f x) (g x)) a
using assms unfolding continuous-at by (rule tendsto-log)

lemma continuous-on-log[continuous-intros]:
assumes continuous-on s f continuous-on s g
and ∀x∈s. 0 < f x ∀x∈s. f x ≠ 1 ∀x∈s. 0 < g x
shows continuous-on s (λx. log (f x) (g x))
using assms unfolding continuous-on-def by (fast intro; tendsto-log)

lemma exp-powr-real:
  fixes x::real shows exp x powr y = exp (x*y)
  by (simp add: powr-def)

lemma powr-one-eq-one [simp]: 1 powr a = 1
  by (simp add: powr-def)

lemma powr-zero-eq-one [simp]: x powr 0 = (if x = 0 then 0 else 1)
  by (simp add: powr-def)

lemma powr-one-gt-zero-iff [simp]: x powr 1 = x ←→ 0 ≤ x
  for x :: real
  by (auto simp add: powr-def)

declare powr-one-gt-zero-iff [THEN iffD2, simp]

lemma powr-mult:
  0 ≤ x =⇒ 0 ≤ y =⇒ (x * y) powr a = (x powr a) * (y powr a)
  for a x y :: real
  by (simp add: powr-def exp-add [symmetric] ln-mult distrib-left)

lemma prod-powr-distrib:
  fixes x :: 'a ⇒ real
  assumes ∃i. i∈I =⇒ x i ≥ 0
  shows (prod x I) powr r = (∏i∈I. x i powr r)
  using assms
  by (induction I rule: infinite-finite-induct) (auto simp add: powr-mult prod-nonneg)

lemma powr-ge-pzero [simp]: 0 ≤ x powr y
  for x y :: real
  by (simp add: powr-def)

lemma powr-non-neg[simp]: ¬a powr x < 0 for a :: real
  using powr-ge-pzero[of a x] by arith

lemma inverse-powr: ∀x::real. 0 ≤ y =⇒ inverse y powr a = inverse (y powr a)
  by (simp add: exp-minus ln-inverse powr-def)

lemma powr-divide: [0 ≤ x; 0 ≤ y] =⇒ (x / y) powr a = (x powr a) / (y powr a)
  for a b x :: real

by (simp add: divide-inverse powr-mult inverse-powr)

lemma powr-add: \( x \, \text{powr} \, (a + b) = (x \, \text{powr} \, a) \, \ast \, (x \, \text{powr} \, b) \)
for \( a, b, x :: 'a::\{\text{ln, real-normed-field}\} \)
by (simp add: powr-def exp-add [symmetric] distrib-right)

lemma powr-mult-base: \( 0 \leq x \implies x \, \ast \, x \, \text{powr} \, y = x \, \text{powr} \, (1 + y) \)
for \( x :: \text{real} \)
by (auto simp: powr-add)

lemma powr-mult-base: \( (x \, \text{powr} \, a) \, \text{powr} \, b = x \, \text{powr} \, (a \, \ast \, b) \)
for \( a, b, x :: \text{real} \)
by (simp add: powr-def powr-add)

lemma powr-sum: \( x \neq 0 \implies \text{finite } A \implies x \, \text{powr} \, \text{sum } f \, A = (\prod y \in A. \, x \, \text{powr} \, f \, y) \)
for \( a, b, c :: \text{real} \)
by (simp add: powr-def powr-mult-commute)

lemma powr-less-mono: \( a < b \implies 1 < x \implies x \, \text{powr} \, a < x \, \text{powr} \, b \)
for \( a, b, x :: \text{real} \)
by (simp add: powr-def powr-less-cancel)

lemma powr-less-cancel-iff [simp]: \( 1 < x \implies x \, \text{powr} \, a < x \, \text{powr} \, b \iff a < b \)
for \( a, b, x :: \text{real} \)
by (blast intro: powr-less-cancel powr-less-mono)
lemma powr-le-cancel-iff [simp]: \(1 < x \implies x \text{ powr } a \leq x \text{ powr } b \iff a \leq b\)
for \(a, b, x :: \text{real}\)
by (simp add: linorder-not-less [symmetric])

lemma powr-realpow: \(0 < x \implies x \text{ powr } (\text{real } n) = x \sim n\)
by (induction \(n\)) (simp-all add: ac-simps power-add)

lemma powr-realpow\': \((z :: \text{real}) \geq 0 \implies n \neq 0 \implies z \text{ powr } \text{of-nat } n = z \sim n\)
by (cases \(z = 0\)) (auto simp: powr-realpow)

lemma powr-real-of-int\':
assumes \(x \geq 0 \land x \neq 0 \lor n > 0\)
shows \(x \text{ powr } \text{of-int } n = \text{power-int } x \cdot n\)
by (metis assms exp-ln-iff exp-power-int nless-le power-int-eq-0-iff powr-def)

lemma exp-minus-ge:
fixes \(x :: \text{real}\)
shows \(1 - x \leq \exp(-x)\)
by (smt (verit) exp-ge-add-one-self)

lemma exp-minus-greater:
fixes \(x :: \text{real}\)
shows \(1 - x < \exp(-x) \iff x \neq 0\)
by (smt (verit) exp-minus-ge exp-eq-zero iff exp-eq-one-iff exp-gt-zero ln-eq-minus-one ln-exp)

lemma log-ln: \(\ln x = \log(\exp(1)) \cdot x\)
by (simp add: log-def)

lemma DERIV-log:
assumes \(x > 0\)
shows \(\text{DERIV } (\lambda y. \log b \cdot y) \cdot x \cdot l' / (\ln b \cdot x)\)
proof
  define \(lb\) where \(lb = 1 / \ln b\)
moreover have \(\text{DERIV } (\lambda y. \ln b \cdot y) \cdot x \cdot lb / x\)
  using \(x > 0\)
  by (auto intro: derivative-eq-intros)
ultimately show ?thesis
  by (simp add: log-def)
qed

lemmas DERIV-log\[\text{THEN } \text{DERIV-chains2, derivative-intros}\]
and DERIV-log\[\text{THEN } \text{DERIV-chains2, unfolded has-field-derivative-def, derivative-intros}\]

lemma powr-log-cancel [simp]: \(0 < a \implies a \neq 1 \implies 0 < x \implies x \text{ powr } (\log a \cdot x) = x\)
by (simp add: powr-def log-def)

lemma log-powr-cancel [simp]: \(0 < a \implies a \neq 1 \implies \log a \cdot (a \text{ powr } x) = x\)
by (simp add: log-def powr-def)
lemma \textit{powr-eq-iff}: \[ y>0; a>1 \] \implies a \powr x = y \iff \log a y = x 
by auto

lemma \textit{log-mul}: 
\[ 0 < x \implies 0 < y \implies \log a (x \cdot y) = \log a x + \log a y \]
by (simp add: log-def ln-mult divide-inverse distrib-right)

lemma \textit{log-eq-div-ln-mult-log}: 
\[ 0 < b \implies b \neq 1 \implies 0 < x \implies \log a x = (\ln b/\ln a) \cdot \log b x \]
by (simp add: log-def divide-inverse)

Base 10 logarithms

lemma \textit{log-base-10-eq1}: \[ 0 < x \implies \log 10 x = (\ln (\exp 1) / \ln 10) \cdot \ln x \]
by (simp add: log-def)

lemma \textit{log-base-10-eq2}: \[ 0 < x \implies \log 10 x = (\log 10 (\exp 1)) \cdot \ln x \]
by (simp add: log-def)

lemma \textit{log-one}: \[ \log a 1 = 0 \]
by (simp add: log-def)

lemma \textit{log-eq-one}: \[ 0 < a \implies a \neq 1 \implies \log a a = 1 \]
by (simp add: log-def)

lemma \textit{log-inverse}: \[ 0 < x \implies \log a (\inverse x) = -\log a x \]
using \textit{ln-inverse} log-def by auto

lemma \textit{log-divide}: \[ 0 < x \implies 0 < y \implies \log a (x/y) = \log a x - \log a y \]
by (simp add: log-mult divide-inverse log-inverse)

lemma \textit{powr-gt-zero}: \[ 0 < x \ \powr a \not\equiv 0 \ \iff a \not\equiv 1 \]
for a x :: real 
by (simp add: powr-def)

lemma \textit{powr-nonneg-iff}: \[ a \powr x \leq 0 \iff a = 0 \]
for a x::real 
by (meson not-less powr-gt-zero)

lemma \textit{log-add-eq-powr}: \[ 0 < b \implies b \neq 1 \implies 0 < x \implies \log b x + y = \log b (x \cdot b \powr y) \]
and \textit{add-log-eq-powr}: \[ 0 < b \implies b \neq 1 \implies 0 < x \implies y + \log b x = \log b (b \powr y \cdot x) \]
and \textit{log-minus-eq-powr}: \[ 0 < b \implies b \neq 1 \implies 0 < x \implies \log b x - y = \log b (x \cdot b \powr -y) \]
and \textit{minus-log-eq-powr}: \[ 0 < b \implies b \neq 1 \implies 0 < x \implies y - \log b x = \log b (b \powr y / x) \]
by (simp-all add: log-mult log-divide)

lemma \textit{log-less-cancel-iff}: \[ 1 < a \implies 0 < x \implies 0 < y \implies \log a x < \log a \]
y \leftrightarrow x < y

by (metis less-eq-real-def less-trans not-le zero-less-one)

lemma log-inj:
  assumes 1 < b
  shows inj-on (log b) {0 <..}
proof (rule inj-onI, simp)
  fix x y
  assume pos: 0 < x 0 < y and *: log b x = log b y
  show x = y
  proof (cases rule: linorder-cases)
    assume x = y
    then show * by simp
  next
    assume x < y
    then have log b x < log b y
    using log-less-cancel-iff[of 1 < b] pos by simp
    then show * using * by simp
  next
    assume y < x
    then have log b y < log b x
    using log-less-cancel-iff[of 1 < b] pos by simp
    then show * using * by simp
  qed
qed

lemma log-le-cancel-iff [simp]: 1 < a \Rightarrow 0 < x \Rightarrow 0 < y \Rightarrow log a x \leq log a y
\leftrightarrow x \leq y
by (simp flip: linorder-not-less)

lemma log-mono: 1 < a \Rightarrow 0 < x \Rightarrow x \leq y \Rightarrow log a x \leq log a y
by simp

lemma log-less: 1 < a \Rightarrow 0 < x \Rightarrow x < y \Rightarrow log a x < log a y
by simp

lemma zero-less-log-cancel-iff[simp]: 1 < a \Rightarrow 0 < x \Rightarrow 0 < log a x \leftrightarrow 1 < x
using log-less-cancel-iff[of a 1 x] by simp

lemma zero-le-log-cancel-iff[simp]: 1 < a \Rightarrow 0 < x \Rightarrow 0 \leq log a x \leftrightarrow 1 \leq x
using log-le-cancel-iff[of a 1 x] by simp

lemma log-less-zero-cancel-iff[simp]: 1 < a \Rightarrow 0 < x \Rightarrow log a x < 0 \leftrightarrow x < 1
using log-less-cancel-iff[of a x 1] by simp

lemma log-le-zero-cancel-iff[simp]: 1 < a \Rightarrow 0 < x \Rightarrow log a x \leq 0 \leftrightarrow x \leq 1
using log-le-cancel-iff[of a x 1] by simp

lemma one-less-log-cancel-iff[simp]: 1 < a ⇒ 0 < x ⇒ 1 < log a x ↔ a < x
using log-less-cancel-iff[of a a x] by simp

lemma one-le-log-cancel-iff[simp]: 1 < a ⇒ 0 < x ⇒ log a x ≤ 1 ↔ x < a
using log-le-cancel-iff[of a a x] by simp

lemma log-less-cancel-iff[simp]: 1 < a ⇒ 0 < x ⇒ log a x < 1 ↔ x < a
using log-less-cancel-iff[of a a x] by simp

lemma log-le-cancel-iff[simp]: 1 < a ⇒ 0 < x ⇒ 1 ≤ log a x ↔ a ≤ x
using one-le-log-cancel-iff[of a a x] by simp

lemma le-log-iff:
fixes b x y :: real
assumes 1 < b x
shows y ≤ log b x ≤ b powr y
using assms by (metis zero_less_one less-log-cancel
powr-le-cancel-iff powr-log-cancel
less-irrefl powr-less-cancel-iff)

lemma less-log-iff:
assumes 1 < b x
shows y < log b x < b powr y
by (metis assms dual-order.strict-trans
powr-less-cancel-iff
powr-log-cancel
less-irrefl powr-less-cancel-iff)

lemma le-log-iff:
assumes 1 < b x
shows log b x ≤ y ≤ b powr y
and log-le-iff: log b x ≤ y ↔ x ≤ b powr y
using le-log-iff[of OF assms, of y] less-log-iff[of OF assms, of y]
by auto

lemmas powr-le-iff = le-log-iff[symmetric]
and powr-less-iff = less-log-iff[symmetric]
and less-powr-iff = log-less-iff[symmetric]
and le-powr-iff = log-le-iff[symmetric]

lemma le-log-of-power:
assumes b ^ n ≤ m 1 < b
shows n ≤ log b m
proof
from assms have 0 < m by (metis less-trans zero-less-power
less-le-trans zero-less-one)
thus ?thesis using assms by (simp add: le-log-iff powr-realpow)
qed

lemma le-log2-of-power: 2 ^ n ≤ m ⇒ n ≤ log 2 m for m n :: nat
using le-log-of-power[of 2] by simp
lemma \text{log-of-power-le}: \[ m \leq b \cdot n; b > 1; m > 0 \] \implies \log b (\text{real } m) \leq n \\
by \text{(simp add: log-le-iff powr-realpow)}

lemma \text{log2-of-power-le}: \[ m \leq 2 \cdot n; m > 0 \] \implies \log 2 m \leq n \\
for m n :: \text{nat} \by \text{simp}

lemma \text{log-of-power-less}: \[ m < b \cdot n; b > 1; m > 0 \] \implies \log b (\text{real } m) < n \\
by \text{(simp add: log-less-iff powr-realpow)}

lemma \text{log2-of-power-less}: \[ m < 2 \cdot n; m > 0 \] \implies \log 2 m < n \\
for m n :: \text{nat} \by \text{simp}

lemma \text{less-log-of-power}: \smallskip
assumes \[ b \cdot n < m \] \AND \[ 1 < b \] \\
shows \[ n < \log b m \] \proof 
  have \[ 0 < m \] \by \text{(metis assms less-trans zero-less-power zero-less-one)}
  thus \[ ?thesis \] \by \text{(simp add: less-log-iff powr-realpow)}
\qed

lemma \text{less-log2-of-power}: \smallskip
assumes \[ 2 \cdot n < m \] \\
shows \[ n < \log 2 m \] \by \text{simp}

lemma \text{gr-one-powr}: \smallskip
fixes x y :: \text{real} \\
shows \[ x > 1; y > 0 \] \implies 1 < x \text{ powr } y \\
by \text{simp}

lemma \text{log-pow-cancel}: \smallskip
assumes \[ a > 0 \] \implies \[ a \neq 1 \] \implies \[ \log a (a \cdot b) = b \] \\
by \text{(simp add: ln-realpow log-def)}

lemma \text{floor-log-eq-powr-iff}: \smallskip
assumes \[ x > 0 \] \implies \[ b > 1 \] \implies \[ \lfloor \log b x \rfloor = k \] \IFF \[ b^k \leq x \AND x < b^{k+1} \] \\
by \text{(auto simp: floor-eq-iff powr-le-iff less-power-iff)}

lemma \text{floor-log-nat-eq-powr-iff}: \smallskip
assumes \[ b \cdot n \cdot k :: \text{nat} \] \\
shows \[ b \cdot n \leq k \] \AND \[ k < b^{(n+1)} \] \\

lemma \text{floor-log-nat-eq-if}: \smallskip
assumes \[ b \cdot n \leq k \] \AND \[ k < b^{(n+1)} \] \\
shows \[ \lfloor \log b (\text{real } k) \rfloor = n \] \\
by \text{—}
have \( k \geq 1 \)

using assms linorder-le-less-linear by force
with assms show \(?thesis\)
by (simp add: floor-log-nat-eq-powr-iff)

qed

lemma ceiling-log-eq-powr-iff:
\[ [ x > 0; b > 1 ] \implies \lceil \log b x \rceil = \text{int} k + 1 \iff b \text{ powr} k < x \land x \leq b \text{ powr} (k + 1) \]
by (auto simp: ceiling-eq-iff powr-less-iff le-powr-iff)

lemma ceiling-log-nat-eq-powr-iff:
fixes \( b, n, k \) :: nat
shows \[ [ b \geq 2; k > 0 ] \implies \lceil \log b (\text{real} k) \rceil = \text{int} n + 1 \iff (b^n < k \land k \leq b^{(n+1)}) \]
using ceiling-log-eq-powr-iff

lemma ceiling-log-nat-eq-if:
fixes \( b, n, k \) :: nat
assumes \( b^n < k \land k \leq b^{(n+1)} \land b \geq 2 \)
shows \( \lceil \log (\text{real} b) (\text{real} k) \rceil = \text{int} n + 1 \)
using assms ceiling-log-nat-eq-powr-iff
by force

lemma floor-log2-div2:
fixes \( n \) :: nat
assumes \( n \geq 2 \)
shows \( \lfloor \log 2 (\text{real} n) \rfloor = \lfloor \log 2 (n \text{ div} 2) \rfloor + 1 \)
proof cases
assumes \( \text{n=2} \) thus \(?thesis\) by simp
next
let \( ?m = n \text{ div} 2 \)
assume \( n \neq 2 \)
hence \( i \leq ?m \) using assms by arith
then obtain \( i \) where \( i \cdot 2 \leq ?m \land ?m < 2 \cdot (i + 1) \)
using ex-power-int1[of 2 \(?m\)] by auto
have \( 2^\cdot (i+1) \leq 2 \cdot ?m \) using \( i(1)\) by simp
also have \( 2 \cdot ?m \leq n \) by arith
finally have \(*: 2^\cdot (i+1) \leq \ldots \)
have \( n < 2^\cdot (i+1+1) \) using \( i(2)\) by simp
from floor-log-nat-eq-if[OF \(*\) this] floor-log-nat-eq-if[OF \(*\)
shows \(?thesis\) by simp

qed

lemma ceiling-log2-div2:
assumes \( n \geq 2 \)
shows \( \lceil \log 2 (\text{real} n) \rceil = \lceil \log 2 ((n-1) \text{ div} 2 + 1) \rceil + 1 \)
proof cases
  assume \( n=2 \) thus \(?\)thesis by simp
next
  let \(?m = (n-1) \div 2 + 1 \)
  assume \( n\neq 2 \)
  hence \( 2 \leq ?m \) using assms by arith
  then obtain \( i \) where \( i \cdot 2 \leq i < ?m \) \( ?m \leq 2 \cdot (i + 1) \)
    using ex-power-int2[of 2 ?m] by auto
  have \( n \leq 2 \cdot ?m \) by arith
  also have \( 2 \cdot ?m \leq 2 \cdot ((i+1)+1) \) using \( i(2) \)
    by simp
  finally have \( i: n \leq \ldots \).
  have \( 2 \cdot (i+1) < n \) using \( i(1) \)
    by (auto simp: less-Suc-eq-0-disj)
  from ceiling-log-nat-if[OF this \( \ast \) ceiling-log-nat-if[OF \( i \)]]
    show \(?\)thesis by simp
qed

lemma pourr-real-of-int:
  \( x > 0 \implies x \text{ pourr } \text{real-of-int } n = \) 
  \( (\text{if } n \geq 0 \text{ then } x \cdot \text{nat } n \text{ else } \text{inverse } (x \cdot \text{nat } (\neg n))) \)
  using pourr-powr[of \( x \text{ nat } n \)]
  powr-powr[of \( x \text{ nat } (\neg n) \)]
  by (auto simp: field-simps powr-minus)

lemma pourr-numeral [simp]: \( 0 \leq x \implies x \text{ pourr } (\text{numeral } n :: \text{real}) = x \cdot (\text{numeral } n) \)
  by (metis less-le power-zero-numeral powr-0 of-nat-numeral powr-realpow)

lemma pourr-int:
  assumes \( x > 0 \)
  shows \( x \text{ pourr } i = (\text{if } i \geq 0 \text{ then } x \cdot \text{nat } i \text{ else } 1/x \cdot \text{nat } (-i)) \)
  by (simp add: assms inverse-eq-divide powr-real-of-int)

lemma power-of-nat-log-le:
  assumes \( b > 1 \)
  shows \( b \cdot \text{nat } [\text{log } b \ x] \leq x \)
  by (smt (verit) less-log-of-power of-nat-ceiling)

lemma power-of-nat-log-ge:
  assumes \( b > 1 \)
  shows \( b \cdot \text{nat } [\text{log } b \ x] \geq x \)
  by (smt (verit) less-log-of-power of-nat-ceiling)

lemma power-of-nat-log-le:
  assumes \( b > 1 \)
  shows \( b \cdot \text{nat } [\text{log } b \ x] \leq x \)
  proof
    have \( [\text{log } b \ x] \geq 0 \)
      using assms by auto
    then show \(?\)thesis
      by (smt (verit) assms le-log-iff of-int-floor-le powr-int)
  qed

definition pourr :: \( \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \)
  where \( \text{code-abbrev, simp: } \text{pourr } = \text{Transcendental.pourr} \)

lemma compute-pourr [code]:
  pourr \( b \) \( i \) =
THEORY "Transcendental"

(if \(b \leq 0\) then Code.abort (STR "powr-real with nonpositive base") (λ-. powr-real b i)
else if \([i] = i\) then (if \(0 \leq i\) then \(b \sim nat [i]\) else \(1 / b \sim nat \sim i\))
else Code.abort (STR "powr-real with non-integer exponent") (λ-. powr-real b i))

for \(b i :: real\)
by (auto simp: powr-int)

lemma powr-one: \(0 \leq x \Rightarrow x \sim powr 1 = x\)
for \(x :: real\)
using powr-realpow [of x 1] by simp

lemma powr-neg-one: \(0 < x \Rightarrow x \sim powr - 1 = 1 / x\)
for \(x :: real\)
using powr-int [of x - 1] by simp

lemma powr-neg-numeral: \(0 < x \Rightarrow x \sim powr \sim numeral n = 1 / x \sim numeral n\)
for \(x :: real\)
using powr-int [of x - numeral n] by simp

lemma root-powr-inverse: \(0 < n \Rightarrow 0 < x \Rightarrow \sqrt[n]{x} = x \sim powr \sim 1 / n\)
by (rule real-root-pos-unique) (auto simp: powr-realpow[symmetric] powr-powr)

lemma ln-powr: \(x \neq 0 \Rightarrow \ln (x \sim powr y) = y \ast \ln x\)
for \(x :: real\)
by (simp add: powr-def)

lemma ln-root: \(n > 0 \Rightarrow b > 0 \Rightarrow \ln (\sqrt[n]{b}) = \ln b / n\)
by (simp add: powr-root-inv ln-powr)

lemma ln-sqrt: \(0 < x \Rightarrow \ln (\sqrt{x}) = \ln x / 2\)
by (simp add: ln-powr ln-powr[symmetric] mult.commute)

lemma log-root: \(n > 0 \Rightarrow a > 0 \Rightarrow \log_b (\sqrt[n]{a}) = \log_b a / n\)
by (simp add: log-def ln-root)

lemma log-powr: \(x \neq 0 \Rightarrow \log_b (x \sim powr y) = y \ast \log_b x\)
by (simp add: log-def ln-powr)

lemma log-nat-power: \(0 < x \Rightarrow \log_b (x \sim n) = \text{real } n \ast \log_b x\)
by (simp add: log-powr powr-realpow[symmetric])

lemma log-of-power-eq:
assumes \(m = b \sim n\) \(b > 1\)
shows \(n = \log_b (\text{real } m)\)
proof —
have \(n = \log_b (b \sim n)\) using assms(2) by (simp add: log-nat-power)
also have \(\ldots = \log_b m\) using assms by simp
finally show \textit{?thesis}.

qed

lemma log2-of-power-eq: \(\forall m n. m = 2^n \iff n = \log 2 m\) for \(m n :: \mathbb{N}\)
using \textit{log-of-power-eq[of 2]} by simp

lemma log-base-change: \(0 < a \iff a \neq 1 \implies \log_b x = \log a x / \log a b\)
by (simp add: \textit{log-def})

lemma log-base-pow: \(0 < a \implies \log (a^n) x = \log a x / n\)
by (simp add: \textit{log-def ln-realpow})

lemma log-base-powr: \(a \neq 0 \implies \log (a^{powr} b) x = \log a x / b\)
by (simp add: \textit{log-def ln-powr})

lemma log-base-root: \(n > 0 \implies b > 0 \implies \log (\sqrt[n]{b}) x = n (\log b x)\)
by (simp add: \textit{log-def ln-root})

lemma ln-bound: \(0 < x \implies \ln x \leq x\) for \(x :: \mathbb{R}\)
using \textit{ln-le-minus-one} by force

lemma powr-less-one:
fixes \(x :: \mathbb{R}\)
assumes \(1 < x < 0\)
shows \(x powr y < 1\)
using \textit{assms less-log-iff} by force

lemma powr-le-one-le:
\(\forall x y :: \mathbb{R}. 0 < x \implies x \leq 1 \implies 1 \leq y \implies x powr y \leq x\)
by (smt (verit) \textit{ln-gt-zero-imp-gt-one} \textit{ln-cancel-iff} \textit{ln-powr} \textit{mult-le-cancel-right2})

lemma powr-mono:
fixes \(x :: \mathbb{R}\)
assumes \(a \leq b\) and \(1 \leq x\)
shows \(x powr a \leq x powr b\)
using \textit{assms less-eq-real-def} by auto

lemma ge-one-powr-ge-zero: \(1 \leq x \implies 0 \leq a \implies 1 \leq x powr a\)
for \(x :: \mathbb{R}\)
using \textit{powr-mono} by fastforce

lemma powr-less-mono2: \(0 < a \implies 0 \leq x \leq y \implies x powr a < y powr a\)
for \(x :: \mathbb{R}\)
by (simp add: \textit{powr-def})

lemma powr-less-mono2-neg: \(a < 0 \implies 0 < x \leq y \implies y powr a < x powr a\)
for \(x :: \mathbb{R}\)
by (simp add: \textit{powr-def})

lemma powr-mono2: \(x powr a \leq y powr a\) if \(0 \leq a \leq x x \leq y\)
for $x :: \text{real}$
using less-eq-real-def powr-less-mono2 that by auto

lemma powr01-less-one:
fixes $a :: \text{real}$
assumes $0 < a < 1$
shows $a \text{ powr } e < 1 \iff e > 0$
proof
  show $a \text{ powr } e < 1 \implies e > 0$
    using assms not-less-iff-gr-or-eq powr-less-mono2_neg by fastforce
  show $e > 0 \implies a \text{ powr } e < 1$
    by (metis assms less-eq-real-def powr-less-mono2 powr-one-eq-one)
qed

lemma powr-le1:
$0 \leq a \implies 0 \leq x \implies x \leq 1 \implies x \text{ powr } a \leq 1$
for $x :: \text{real}$
using powr-mono2 by fastforce

lemma powr-mono2'::
fixes $a \ x \ y :: \text{real}$
assumes $a \leq 0 \ x > 0 \ x \leq y$
shows $x \text{ powr } a \geq y \text{ powr } a$
proof
  from assms have $x \text{ powr } -a \leq y \text{ powr } -a$
    by (intro powr-mono2) simp-all
  with assms show ?thesis
    by (auto simp: powr-minus field-simps)
qed

lemma powr-mono-both:
fixes $x :: \text{real}$
assumes $0 \leq a \ a \leq b \ 1 \leq x \ x \leq y$
shows $x \text{ powr } a \leq y \text{ powr } b$
by (meson assms order.trans powr-mono powr-mono2 zero-le-one)

lemma powr-mono-both'::
fixes $x :: \text{real}$
assumes $a \geq b \ b \geq 0 \ 0 < x \ x \leq y \ y \leq 1$
shows $x \text{ powr } a \leq y \text{ powr } b$
by (meson assms nless-le order.trans powr-mono' powr-mono2)

lemma powr-less-mono':
assumes $(x :: \text{real}) > 0 \ x < 1 \ a < b$
shows $x \text{ powr } b < x \text{ powr } a$
by (metis assms log-powr-cancel order.strict-iff-order powr-mono')
lemma powr-inj: \( 0 < a \implies a \neq 1 \implies a \text{ powr } x = a \text{ powr } y \iff x = y \)
  for \( x \::\: \text{real} \)
  unfolding powr-def exp-inj-iff by simp

lemma powr-half-sqrt: \( 0 \leq x \implies x \text{ powr } (1/2) = \sqrt{x} \)
  by (simp add: powr-half-sqrt powr-def exp-inj-iff)

lemma powr-half-sqrt-powr: \( 0 \leq x \implies x \text{ powr } (a/2) = \sqrt{x \text{ powr } a} \)
  by (metis divide-inverse mult_left_neutral powr-ge-pzero powr-half-sqrt powr-powr)

lemma square-powr-half [simp]:
  fixes \( x \::\: \text{real} \)
  shows \( x^2 \text{ powr } (1/2) = |x| \)
  by (simp add: powr-half-sqrt)

lemma ln-powr-bound: \( 1 \leq x \implies 0 < a \implies \ln x \leq (x \text{ powr } a) / a \)
  for \( x \::\: \text{real} \)
  by (metis exp_gt_zero linear ln_eq_zero_iff ln_less_self ln_powr mult_commute
      mult_imp_le_div_pos not_less powr_gt_zero)

lemma ln-powr-bound2:
  fixes \( x \::\: \text{real} \)
  assumes \( 1 < x \) and \( 0 < a \)
  shows \( (\ln x) \text{ powr } a \leq (a \text{ powr } a) * x \)
  proof (from assms)
    have \( \ln x \leq (x \text{ powr } (1/a)) / (1/a) \)
      by (metis less_eq_real_def ln_powr_bound zero_less_divide_1_iff)
    also have \( \ldots = a * (x \text{ powr } (1/a)) \)
      by simp
    finally have \( (\ln x) \text{ powr } a \leq (a * (x \text{ powr } (1/a))) \text{ powr } a \)
      by (metis assms_less_imp_le ln_gt_zero powr_mono2)
    also have \( \ldots = (a \text{ powr } a) * ((x \text{ powr } (1/a)) \text{ powr } a) \)
      using assms powr_mult by auto
    also have \( \ldots = x \text{ using assms} \)
      by auto
    finally show \( ?thesis \).
  qed

lemma tendsto-powr:
  fixes \( a \) \( b \::\: \text{real} \)
  assumes \( f: (f \longrightarrow a) \ F \)
    and \( g: (g \longrightarrow b) \ F \)
    and \( a: a \neq 0 \)
  shows \( ((\lambda x. f \text{ powr } g \ x) \longrightarrow a \text{ powr } b) \ F \)
  unfolding powr-def
  proof (rule filterlim-If)
    from \( f \) show \( ((\lambda x. 0) \longrightarrow (if a = 0 \text{ then } 0 \text{ else } \exp (b * \ln a))) \ (\inf F \ (\text{principal})
\{x. f x = 0\})

\begin{align*}
&\text{by simp (auto simp: filterlim-iff eventually-inf-principal elim: eventually-mono dest: t1-space-nhds)} \\
&\text{from } f \text{ g a show } ((\lambda x. \exp (g x \ast \ln (f x))) \longrightarrow (\text{if } a = 0 \text{ then } 0 \text{ else } \exp (b \ast \ln a))) \\
&\quad (\inf F (\text{principal } \{x. f x \neq 0\})) \\
&\quad \text{by (auto intro!: tendsto-intros intro: tendsto-mono inf-le1)}
\end{align*}

qed

lemma \textit{tendsto-powr}[tendsto-intros]:
\begin{align*}
&\text{fixes } a :: \text{real} \\
&\text{assumes } f : (f \longrightarrow a) F \\
&\quad \text{and } g : (g \longrightarrow b) F \\
&\quad \text{and } a : a \neq 0 \lor (b > 0 \land \text{eventually } (\lambda x. f x \geq 0)) F \\
&\text{shows } ((\lambda x. f x \text{ powr } g x) \longrightarrow a \text{ powr } b) F \\
&\text{proof} \quad - \\
&\text{from } a \text{ consider } a \neq 0 | a = 0 \ b > 0 \text{ eventually } (\lambda x. f x \geq 0) F \\
&\quad \text{by auto} \\
&\text{then show } \textit{?thesis} \\
&\quad \text{proof cases} \\
&\quad \text{case } 1 \\
&\quad \quad \text{with } f \text{ g show } \textit{?thesis by (rule tendsto-powr)} \\
&\quad \text{next} \\
&\quad \text{case } 2 \\
&\quad \quad \text{have } ((\lambda x. \text{if } f x = 0 \text{ then } 0 \text{ else } \exp (g x \ast \ln (f x))) \longrightarrow 0) F \\
&\quad \quad \text{proof (intro filterlim-If)} \\
&\quad \quad \quad \text{have filterlim } f (\text{principal } \{0<..\}) (\inf F (\text{principal } \{z. f z \neq 0\})) \\
&\quad \quad \quad \quad \text{using } (\text{eventually } (\lambda x. f x \geq 0)) F \\
&\quad \quad \quad \quad \text{by (auto simp: filterlim-iff eventually-inf-principal} \\
&\quad \quad \quad \quad \quad \text{eventually-principal elim: eventually-mono}) \\
&\quad \quad \quad \text{moreover have filterlim } f (\text{nhds } a) (\inf F (\text{principal } \{z. f z \neq 0\})) \\
&\quad \quad \quad \quad \text{by (rule tendsto-mono[OF - f]) simp-all} \\
&\quad \quad \quad \text{ultimately have } f : \text{filterlim } f (\text{at-right } 0) (\inf F (\text{principal } \{x. f x \neq 0\})) \\
&\quad \quad \quad \quad \text{by (simp add: at-within-def filterlim-inf \{a = 0\})} \\
&\quad \quad \quad \text{have } g : (g \longrightarrow b) (\inf F (\text{principal } \{z. f z \neq 0\})) \\
&\quad \quad \quad \quad \text{by (rule tendsto-mono[OF - g]) simp-all} \\
&\quad \quad \quad \text{show } ((\lambda x. \exp (g x \ast \ln (f x))) \longrightarrow 0) (\inf F (\text{principal } \{x. f x \neq 0\})) \\
&\quad \quad \quad \quad \text{by (rule filterlim-compose[OF exp-at-bot] filterlim-tendsto-pos-mult-at-bot} \\
&\quad \quad \quad \quad \quad \text{filterlim-compose[OF ln-at-0] } f \ g \ \{b > 0\}+) \\
&\quad \quad \quad \quad \text{qed simp-all} \\
&\quad \quad \quad \text{with } \{a = 0\} \text{ show } \textit{?thesis} \\
&\quad \quad \quad \quad \text{by (simp add: powr-def)} \\
&\quad \quad \text{qed}
\end{align*}

qed

lemma \textit{continuous-powr}:
\begin{align*}
&\text{assumes } \text{continuous } F f \\
&\quad \text{and } \text{continuous } F g \\
&\quad \text{and } f (\text{Lim } F (\lambda x. x)) \neq 0
\end{align*}
shows continuous $F$ $(\lambda x. (f x) \text{ powr} (g x :: \text{real}))$
using assms unfolding continuous-def by (rule tendsto-powr)

lemma continuous-at-within-powr[continuous-intros]:
fixes $f \ g :: - \Rightarrow \text{real}$
assumes continuous (at $a$ within $s$) $f$
and continuous (at $a$ within $s$) $g$
and $f a \neq 0$
shows continuous (at $a$ within $s$) $(\lambda x. (f x) \text{ powr} (g x))$
using assms unfolding continuous-within by (rule tendsto-powr)

lemma isCont-powr[continuous-intros, simp]:
fixes $f \ g :: - \Rightarrow \text{real}$
assumes isCont $f$ $a$ isCont $g$ $a$ $f a \neq 0$
shows isCont $(\lambda x. (f x) \text{ powr} (g x))$ $a$
using assms unfolding continuous-at by (rule tendsto-powr)

lemma continuous-on-powr[continuous-intros]:
fixes $f \ g :: - \Rightarrow \text{real}$
assumes continuous-on $s$ $f$ continuous-on $s$ $g$ and $\forall x \in s. f x \neq 0$
shows continuous-on $s$ $(\lambda x. (f x) \text{ powr} (g x))$
using assms unfolding continuous-on-def by (fast intro: tendsto-powr)

lemma tendsto-powr2:
fixes $a :: \text{real}$
assumes $f :: (f \lim a) F$
and $g :: (g \lim b) F$
and $\forall x \in F. 0 \leq f x$
and $b : 0 < b$
shows $((\lambda x. f x \text{ powr} g x) \lim a \text{ powr} b) F$
using tendsto-powr[if f a F g b] assms by auto

lemma has-derivative-powr[derivative-intros]:
assumes $g$[derivative-intros]: $(g$ has-derivative $g')$ $(at \ x \ within \ X)$
and $f$[derivative-intros]:$(f$ has-derivative $f')$ $(at \ x \ within \ X)$
assumes $pos: 0 < g x$ and $x \in X$
shows $((\lambda x. g x \text{ powr} f x :: \text{real})$ has-derivative $(\lambda h. (g x \text{ powr} f x) * (f' h * \text{ln} (g x) + g' h * f x / g x)))$ $(at \ x \ within \ X)$

proof –
have $\forall x \in at x \ within \ X. g x > 0$
by (rule order-tendstoD[OF $\text{pos}$])
(rule has-derivative-continuous[OF $g$, unfolded continuous-within])
then obtain $d$ where $d > 0$ and $pos': \forall x'. x' \in X \Rightarrow \text{dist} x' x < d \Rightarrow 0 < g x'$
using pos unfolding eventually-at by force
have $((\lambda x. \text{exp} (f x * \text{ln} (g x)))$ has-derivative
$(\lambda h. (g x \text{ powr} f x) * (f' h * \text{ln} (g x) + g' h * f x / g x)))$ $(at \ x \ within \ X)$
using pos
by (auto intro!: derivative-eq-intros simp: field-split-simps powr-def)
then show ?thesis
  by (rule has-derivative-transform-within[OF - (d > 0) (x ∈ X)]) (auto simp: powr-def dest: pos')
qed

lemma has-derivative-const-powr [derivative-intros]:
  assumes f: (f has-derivative (λx. a powr (f x))) (at x)
  shows ((λx. a powr (f x)) has-derivative (λy. f' y * ln a * a powr (f x))) (at x)
  using assms
  apply (simp add: powr-def)
  apply (rule assms derivative-eq-intros refl)+
done

lemma has-real-derivative-const-powr [derivative-intros]:
  assumes f: (f has-real-derivative (λx. a powr (f x))) (at x)
  shows ((λx. a powr (f x)) has-real-derivative (λy. f' y * ln a * a powr (f x))) (at x)
  using assms
  apply (simp add: powr-def)
  apply (rule assms derivative-eq-intros refl | simp)+
done

lemma DERIV-powr:
  fixes r :: real
  assumes g: DERIV g x :> m
        and pos: g x > 0
        and f: DERIV f x :> r
  shows DERIV (λx. g x powr f x) x :> (g x powr f x) * (r * ln (g x) + m * f x / g x)
  using assms
  by (auto intro!: derivative-eq-intros ext simp: has-field-derivative-def algebra-simps)

lemma DERIV-fun-powr:
  fixes r :: real
  assumes g: DERIV g x :> m
        and pos: g x > 0
  shows DERIV (λx. (g x) powr r) x :> r * (g x) powr (r - of-nat 1) * m
  using DERIV-powr[OF g pos DERIV-const, of r] pos
  by (simp add: pour-diff field-simps)

lemma has-real-derivative-powr:
  assumes z > 0
  shows ((λz. z powr r) has-real-derivative r * z powr (r - 1)) (at z)
proof (subst DERIV-cong-ev[OF refl - refl])
  from assms have eventually (λz. z ≠ 0) (nhds z)
    by (intro t1-space-nhds auto)
  then show eventually (λz. z powr r = exp (r * ln z)) (nhds z)
    unfolding powr-def by eventually-elim simp
  from assms show ((λz. exp (r * ln z)) has-real-derivative r * z powr (r - 1))
by (auto intro!: derivative-eq-intros simp: powr-def field-simps exp-diff)

qed

declare has-real-derivative-powr[THEN DERIV-chain2, derivative-intros]

A more general version, by Johannes Hölzl

lemma has-real-derivative-powr':
  fixes f g :: real ⇒ real
  assumes (f has-real-derivative f') (at x)
  assumes (g has-real-derivative g') (at x)
  assumes f x > 0
  defines h ≡ λx. f x powr g x * (g' * ln (f x) + f' * g x / f x)
  shows ((λx. f x powr g x) has-real-derivative h x) (at x)
proof (subst DERIV-cong-ev [OF refl - refl])
  from assms have isCont f x by (simp add: DERIV-continuous)
  hence f −−→ f x by (simp add: continuous-at)
  with (f x > 0) have eventually (λx. f x > 0) (nhds x)
    by (auto simp: tendsto-at-iff-tendsto-nhds dest: order-tendstoD)
  thus eventually (λx. f x powr g x = exp (g x * ln (f x))) (nhds x)
    by eventually-elim (simp add: powr-def)
next
  from assms show ((λx. exp (g x * ln (f x))) has-real-derivative h x) (at x)
    by (auto intro!: derivative-eq-intros simp: h-def powr-def)
qed

lemma tendsto-zero-powrI:
  assumes (f −−→ (0 :: real)) F (g −−→ b) F ∀ x in F. 0 ≤ f x 0 < b
  shows ((λx. f x powr g x) −−→ 0) F
using tendsto-powr2 [OF assms] by simp

lemma continuous-on-powr':
  fixes f g :: - ⇒ real
  assumes continuous-on s f continuous-on s g
      and ∀ x∈s. f x ≥ 0 ∧ (f x = 0 −→ g x > 0)
  shows continuous-on s (λx. (f x) powr (g x))
unfolding continuous-on-def
proof
  fix x
  assume x: x∈s
  from assms x show ((λx. f x powr g x) −−→ f x powr g x) (at x within s)
proof (cases f x = 0)
    case True
    from assms(3) have eventually (λx. f x ≥ 0) (at x within s)
      by (auto simp: at-within-def eventually-inf-principal)
    with True x assms show ?thesis
      by (auto intro!: tendsto-zero-powrI[of f - g g x] simp: continuous-on-def)
next
theory "Transcendental"

  case False
  with assms x show ?thesis
    by (auto intro: tendsto-powr' simp: continuous-on-def)
  qed

lemma tendsto-neg-powr:
  assumes s < 0
  and f: LIM x F. f x :> at-top
  shows ((λx. f x powr s) −→ (0::real)) F
proof
  have ((λx. exp (s * ln (f x))) −→ (0::real)) F (is ?X)
    by (auto intro: filterlim-compose[OF exp-at-bot] filterlim-compose[OF ln-at-top]
      filterlim-tendsto-neg-mult-at-bot assms)
  also have ?X ⇔ ((λx. f x powr s) −→ (0::real)) F
    using f filterlim-at-top-dense[of F] by (intro filterlim-cong[OF refl refl])
  finally show ?thesis.
  qed

lemma tendsto-exp-limit-at-right: ((λy. (1 + x * y) powr (1 / y)) −→ exp x)
  (at-right 0)
  for x :: real
proof (cases x = 0)
  case True
  then show ?thesis by simp
next
  case False
  have ((λy. ln (1 + x * y)::real) has-real-derivative 1 + x) (at 0)
    by (auto intro: derivative-eq-intros)
  then have ((λy. ln (1 + x * y) / y) −→ x) (at 0)
    by (auto simp: has-field-derivative-def has-derivative-at)
  then have *: ((λy. exp (ln (1 + x * y) / y)) −→ exp x) (at 0)
    by (rule tendsto-intros)
  then show ?thesis
proof (rule filterlim-mono-eventually)
  show eventually (λxa. exp (ln (1 + x * xa) / xa) = (1 + x * xa) powr (1 / xa)) (at-right 0)
    unfolding eventually-at-right[OF zero-less-one]
  using False
    by (intro exI[of - 1 / |x|]) (auto simp: field-simps powr-def abs_if add-nonneg-eq-0-iff)
  qed (simp-all add: at-eq-sup-left-right)
  qed

lemma tendsto-exp-limit-at-top: ((λy. (1 + x / y) powr y) −→ exp x) at-top
  for x :: real
  by (simp add: filterlim-at-top-to-right inverse-eq-divide tendsto-exp-limit-at-right)
THEORY “Transcendental”

lemma tendsto-exp-limit-sequentially: $(\lambda n. (1 + x / n)^n) \rightarrow\!\!\rightarrow \exp x$
for $x :: \real$
proof (rule filterlim-mono-eventually)
from reals-Archimedean2 [of $|x|$] obtain $n :: \nat$ where $*: \real n > |x| ..$
then have eventually $(\lambda n :: \nat. 0 < 1 + x / \real n)$ at-top
  by (intro eventually-sequentiallyI [of $n$]) (auto simp: field-split-simps)
then show eventually $(\lambda n :: \nat. (1 + x / \real n)^n)$ at-top
  by (rule eventually-mono) (erule powr-realpow)
show $(\lambda n. (1 + x / \real n)^n) \rightarrow\!\!\rightarrow \exp x$
  by (rule filterlim-compose [OF tendsto-exp-limit-at-top filterlim-real-sequentially])
qed auto

113.10 Sine and Cosine

definition $\sin-coeff :: \nat \Rightarrow \real$
  where $\sin-coeff = (\lambda n. \text{if even } n \text{ then } 0 \text{ else } (-1)^{((n - Suc 0) \div 2)} / (\text{fact } n))$

definition $\cos-coeff :: \nat \Rightarrow \real$
  where $\cos-coeff = (\lambda n. \text{if even } n \text{ then } (-1)^{(n \div 2)} / (\text{fact } n) \text{ else } 0)$

definition $\sin :: \'a \Rightarrow \'a::\{\text{real-normed-algebra-1,banach}\}$
  where $\sin = (\lambda x. \sum n. \sin-coeff n * \real^x^n)$

definition $\cos :: \'a \Rightarrow \'a::\{\text{real-normed-algebra-1,banach}\}$
  where $\cos = (\lambda x. \sum n. \cos-coeff n * \real^x^n)$

lemma $\sin-coeff-0$ [simp]: $\sin-coeff 0 = 0$
unfolding $\sin-coeff-def$ by simp

lemma $\cos-coeff-0$ [simp]: $\cos-coeff 0 = 1$
unfolding $\cos-coeff-def$ by simp

lemma $\sin-coeff-Suc$: $\sin-coeff (Suc n) = \cos-coeff n / \real (Suc n)$
unfolding $\cos-coeff-def \sin-coeff-def$
by (simp del: mult-Suc)

lemma $\cos-coeff-Suc$: $\cos-coeff (Suc n) = - \sin-coeff n / \real (Suc n)$
unfolding $\cos-coeff-def \sin-coeff-def$
by (simp del: mult-Suc) (auto elim: oddE)

lemma summable-norm-sin: summable $(\lambda n. \text{norm } (\sin-coeff n * \real^x^n))$
for $x :: \'a::\{\text{real-normed-algebra-1,banach}\}$
proof (rule summable-comparison-test [OF - summable-norm-exp])
  show $\exists N. \forall n \geq N. \text{norm } (\text{norm } (\sin-coeff n * \real^x^n)) \leq \text{norm } (x^n / \real \text{ fact } n)$
    unfolding $\sin-coeff-def$
  by (auto simp: divide-inverse abs-mult power-abs [symmetric] zero-le-mult-iff)
qed
lemma summable-norm-cos: summable (\(\lambda n. \text{norm} (\text{cos-coeff} n * R x^n)\))
for \(x :: 'a::{\text{real-normed-algebra-1,banach}}\)
proof (rule summable-comparison-test [OF - summable-norm-exp])
show \(\exists N. \forall n \geq N. \text{norm} (\text{cos-coeff} n * R x^n) \leq \text{norm} (x^n / R \text{fact} n)\)
  unfolding cos-coeff-def
  by (auto simp: divide-inverse abs-mult power-abs [symmetric] zero-le-mult-iff)
qed

lemma sin-converges: (\(\lambda n. \text{sin-coeff} n * R x^n\)) sums \(\text{sin} x\)
unfolding sin-def
by (metis (full-types) summable-norm-cancel summable-norm-sin summable-sums)

lemma cos-converges: (\(\lambda n. \text{cos-coeff} n * R x^n\)) sums \(\text{cos} x\)
unfolding cos-def
by (metis (full-types) summable-norm-cancel summable-norm-cos summable-sums)

lemma sin-of-real: \(\text{sin} (\text{of-real} x) = \text{of-real} (\text{sin} x)\)
for \(x :: \text{real}\)
proof
  have (\(\lambda n. \text{of-real} (\text{sin-coeff} n * R x^n)\)) = (\(\lambda n. \text{sin-coeff} n * R (\text{of-real} x)^n\))
  proof
    show (\(\lambda n. \text{of-real} (\text{sin-coeff} n * R x^n)\)) = (\(\lambda n. \text{sin-coeff} n * R (\text{of-real} x)^n\)) for \(n\)
    by (simp add: scaleR-conv-of-real)
  qed
  also have \(\ldots\) sums (\(\text{sin} (\text{of-real} x)\))
  by (rule sin-converges)
  finally have (\(\lambda n. \text{of-real} (\text{sin-coeff} n * R x^n)\)) sums (\(\text{sin} (\text{of-real} x)\))
  then show \(?thesis\)
  using sums-unique2 sums-of-real [OF sin-converges] by blast
qed

corollary sin-in-Reals [simp]: \(z \in \mathbb{R} \implies \text{sin} z \in \mathbb{R}\)
by (metis Reals-cases Reals-of-real sin-of-real)

lemma cos-of-real: \(\text{cos} (\text{of-real} x) = \text{of-real} (\text{cos} x)\)
for \(x :: \text{real}\)
proof
  have (\(\lambda n. \text{of-real} (\text{cos-coeff} n * R x^n)\)) = (\(\lambda n. \text{cos-coeff} n * R (\text{of-real} x)^n\))
  proof
    show (\(\lambda n. \text{of-real} (\text{cos-coeff} n * R x^n)\)) = (\(\lambda n. \text{cos-coeff} n * R (\text{of-real} x)^n\)) for \(n\)
    by (simp add: scaleR-conv-of-real)
  qed
  also have \(\ldots\) sums (\(\text{cos} (\text{of-real} x)\))
  by (rule cos-converges)
  finally have (\(\lambda n. \text{of-real} (\text{cos-coeff} n * R x^n)\)) sums (\(\text{cos} (\text{of-real} x)\))
  then show \(?thesis\)
THEORY "Transcendental"

using sums-unique2 sums-of-real [OF cos-converges]
  by blast
qed

corollary cos-in-Reals [simp]: \( z \in \mathbb{R} \implies \cos z \in \mathbb{R} \)
  by (metis Reals-cases Reals-of-real cos-of-real)

lemma diffs-sin-coeff: diffs \( \sin \cdot \) coeff = \( \cos \cdot \) coeff
  by (simp add: diffs-def sin-coeff-Suc del: of-nat-Suc)

lemma diffs-cos-coeff: diffs \( \cos \cdot \) coeff = (\( \lambda n \). \( - \cdot \) sin-coeff \( n \))
  by (simp add: diffs-def cos-coeff-Suc del: of-nat-Suc)

lemma sin-int-times-real: \( \sin (of-int m \cdot of-real x) = of-real (sin (of-int m \cdot x)) \)
  by (metis sin-of-real of-real-mult of-real-of-int-eq)

lemma cos-int-times-real: \( \cos (of-int m \cdot of-real x) = of-real (cos (of-int m \cdot x)) \)
  by (metis cos-of-real of-real-mult of-real-of-int-eq)

Now at last we can get the derivatives of \( \exp, \sin \) and \( \cos \).

lemma DERIV-sin [simp]: DERIV \( \sin x : \geq \cos x \)
  for \( x ::'a::\{\text{real-normed-field}, \text{banach}\} \)
  unfolding sin-def cos-def scaleR-conv-of-real
  apply (rule DERIV-cong)
    apply (rule termdiffs [where \( K=of-real (norm x) + 1 ::'a\)])
      apply (simp-all add: norm-less-p1 diffs-of-real diffs-sin-coeff diffs-cos-coeff
                      summable-minus-iff scaleR-conv-of-real [symmetric]
                      summable-norm-sin [THEN summable-norm-cancel]
                      summable-norm-cos [THEN summable-norm-cancel])
  done

declare DERIV-sin[THEN DERIV-chain2, derivative-intros]
and DERIV-sin[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]

lemmas has-derivative-sin[derivative-intros] = DERIV-sin[THEN DERIV-compose-FDERIV]

lemma DERIV-cos [simp]: DERIV \( \cos x : \geq -\sin x \)
  for \( x ::'a::\{\text{real-normed-field}, \text{banach}\} \)
  unfolding sin-def cos-def scaleR-conv-of-real
  apply (rule DERIV-cong)
    apply (rule termdiffs [where \( K=of-real (norm x) + 1 ::'a\)])
      apply (simp-all add: norm-less-p1 diffs-of-real diffs-minus summinf-minus
                      diffs-sin-coeff diffs-cos-coeff
                      summable-minus-iff scaleR-conv-of-real [symmetric]
                      summable-norm-sin [THEN summable-norm-cancel]
                      summable-norm-cos [THEN summable-norm-cancel])
  done
THEORY "Transcendental"

declare DERIV-cos[THEN DERIV-chain2, derivative-intros]
and DERIV-cos[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]

lemmas has-derivative-cos[derivative-intros] = DERIV-cos[THEN DERIV-compose-FDERIV]

lemma isCont-sin: isCont sin x
for x :: 'a::{real-normed-field,banach}
by (rule DERIV-sin[THEN DERIV-isCont])

lemma continuous-on-sin-real: continuous-on {a..b} sin for a::real
using continuous-at-imp-continuous-on isCont-sin by blast

lemma isCont-cos: isCont cos x
for x :: 'a::{real-normed-field,banach}
by (rule DERIV-cos[THEN DERIV-isCont])

lemma continuous-on-cos-real: continuous-on {a..b} cos for a::real
using continuous-at-imp-continuous-on isCont-cos by blast

context
fixes f :: 'a::t2-space ⇒ 'b::{real-normed-field,banach}
begin

lemma isCont-sin' [simp]: isCont f a ⟹ isCont (∑x. sin (f x)) a
by (rule isCont-o2[OF - isCont-sin])

lemma isCont-cos' [simp]: isCont f a ⟹ isCont (∑x. cos (f x)) a
by (rule isCont-o2[OF - isCont-cos])

lemma tendsto-sin [tendsto-intros]: (f ⟹ a) F ⟹ ((∑x. sin (f x)) ⟹ ∑a) F
by (rule isCont-tendsto-compose[OF isCont-sin])

lemma tendsto-cos [tendsto-intros]: (f ⟹ a) F ⟹ ((∑x. cos (f x)) ⟹ ∑a) F
by (rule isCont-tendsto-compose[OF isCont-cos])

lemma continuous-sin [continuous-intros]: continuous F f ⟹ continuous F (∑x. sin (f x))
unfolding continuous-def by (rule tendsto-sin)

lemma continuous-on-sin [continuous-intros]: continuous-on s f ⟹ continuous-on s (∑x. sin (f x))
unfolding continuous-on-def by (auto intro: tendsto-sin)

lemma continuous-cos [continuous-intros]: continuous F f ⟹ continuous F (∑x. cos (f x))
THEORY “Transcendental” 2409

unfolding continuous-def by (rule tendsto-cos)

lemma continuous-on-cos [continuous-intros]: continuous-on s f \implies\ continuous-on s (\lambda x. \cos (f x))
  unfolding continuous-on-def by (auto intro: tendsto-cos)
end

lemma continuous-within-sin: continuous (at z within s) sin
  for z :: 'a::{real-normed-field,banach}
  by (simp add: continuous-within tendsto-sin)
lemma continuous-within-cos: continuous (at z within s) cos
  for z :: 'a::{real-normed-field,banach}
  by (simp add: continuous-within tendsto-cos)

113.11 Properties of Sine and Cosine

lemma sin-zero [simp]: \(\sin 0 = 0\)
  by (simp add: \sin-def \sin-coeff-def scaleR-conv-of-real)
lemma cos-zero [simp]: \(\cos 0 = 1\)
  by (simp add: \cos-def \cos-coeff-def scaleR-conv-of-real)
lemma DERIV-fun-sin: DERIV g x \(\mapsto\) m \implies\ DERIV (\lambda x. \sin (g x)) x \(\mapsto\) cos (g x) \(\ast\) m
  by (fact derivative-intros)
lemma DERIV-fun-cos: DERIV g x \(\mapsto\) m \implies\ DERIV (\lambda x. \cos(g x)) x \(\mapsto\) \(-\sin (g x) \ast\) m
  by (fact derivative-intros)

113.12 Deriving the Addition Formulas

The product of two cosine series.

lemma cos-x-cos-y:
  fixes x :: 'a::{real-normed-field,banach}
  shows
    \((\lambda p. \sum n\leq p. \text{if even } p \land\ even\ n\ then\ ((-1)^{(p\ div\ 2)} \ast\ (p\ choose\ n) / (\text{fact } p) \ast_R (x\ ^n) \ast\ y\ ^{(p-n)}) \ else\ 0))\ sums\ (\cos x \ast\ \cos y)\)
  proof
    have \((\cos-coeff\ n \ast\ \cos-coeff\ (p\ -\ n)) \ast_R (x\ ^n \ast\ y\ ^{(p-n)}) =\)
      \((\text{if } even\ p \land\ even\ n\ then\ ((-1)^{(p\ div\ 2)} \ast\ (p\ choose\ n) / (\text{fact } p) \ast_R (x\ ^n) \ast\ y\ ^{(p-n)}) \ else\ 0))\)
      if \(n\ \leq\ p\) for \(n\ p::\ \text{nat}\)
proof
  from that have *: even $n \implies$ even $p \implies$
    $(-1)^{\frac{n}{2}} (n \div 2) * (-1)^{\frac{(p - n)}{2}} (p \div 2) = (-1 :: \text{real}) \cdot (-1) \div 2$
  by (metis div-add power-add le-add-diff-inverse odd-add)
with that show ?thesis
  by (auto simp: algebra-simps cos-coeff-def binomial-fact)
qed

then have $(\lambda p. \sum n \leq p. \text{if even } p \land \text{even } n$
  then $(-1)^{\frac{n}{2}} (p \div 2) * (p \text{ choose } n) / (\text{fact } p)) * R (x)^n * y^{\frac{n}{2}}(p-n)$
  else 0) =
    $(\lambda p. \sum n \leq p. (\text{cos-coeff } n * \text{cos-coeff } (p - n)) * R (x)^n * y^{\frac{n}{2}}(p-n)))$
  by simp
also have ... = $(\lambda p. \sum n \leq p. (\text{cos-coeff } n * R x^n) * (\text{cos-coeff } (p - n) * R y^{\frac{n}{2}}(p-n)))$
  by (simp add: algebra-simps)
also have ... sums $(\cos x * \cos y)$
using summable-norm-cos
by (auto simp: cos-def scaleR-conv-of-real intro: Cauchy-product-sums)
finally show ?thesis.
qed

The product of two sine series.

**lemma** sin-x-sin-y:
fixes $x :: \{\text{real-normed-field,banach}\}$
shows
  $(\lambda p. \sum n \leq p.$
    if even $p \land \text{odd } n$
      then $-((-1)^{\frac{n}{2}} (p \div 2) * (p \text{ choose } n) / (\text{fact } p)) * R (x)^{\frac{n}{2}} * y^{\frac{n}{2}}(p-n)$
      else 0)
    sums $(\sin x * \sin y)$
proof
  have $(\text{sin-coeff } n * \text{sin-coeff } (p - n)) * R (x)^{\frac{n}{2}} * y^{\frac{n}{2}}(p-n) =$
    if even $p \land \text{odd } n$
      then $-((-1)^{\frac{n}{2}} (p \div 2) * (p \text{ choose } n) / (\text{fact } p)) * R (x)^{\frac{n}{2}} * y^{\frac{n}{2}}(p-n)$
      else 0)
    if $n \leq p$ for $n p :: \text{nat}$
proof
  have $((-1)^{\frac{n}{2}} (n - Suc 0) \div 2) * (-1)^{\frac{n}{2}} (p - Suc n) \div 2) = -((-1 :: \text{real}) \cdot (-1) \div 2)$
    if $np \text{ odd } n \text{ even } p$
proof
  have $p > 0$
    using $\langle n \leq p \rangle \text{ neq0-conv that(1)}$ by blast
then have $\langle n :: \langle (-1 :: \text{real}) \cdot (-1) \div 2) \rangle$
    using $\langle \text{even } p \rangle$ by (auto simp add: dvd-def power-eq-if)
from $\langle n \leq p \rangle \text{ have } *: n - Suc 0 + (p - Suc n) = p - Suc (Suc 0) Suc (Suc 0) \leq p$
by arith+
have $(p - Suc (Suc 0)) \div 2 = p \div 2 - Suc 0$
by simp  
with \( n \leq p \) np  § * show ?thesis  
by (simp add: flip: div-add power-add)  
qed  
then show ?thesis  
using \( n \leq p \) by (auto simp add: algebra-simps sin-coeff-def binomial-fact)  
qed  
then have  
(\( \lambda p. \sum_{n \leq p} \) if even \( p \) \& odd \( n \)  
then \(-1 \) \( ^{(p \div 2)} \) \( \binom{p}{n} \) \( \cdot_R (x^n) \cdot y^{(p-n)} \)  
else 0)  
by simp  
also have ... = (\( \lambda p. \sum_{n\leq p} \) (sin-coeff \( n \) \( \cdot_R x^n \)) \( \cdot_R (sin-coeff (p-n) \cdot_R y^{(p-n)}) \))  
by (simp add: algebra-simps)  
also have ... sums (sin \( x \) \* sin \( y \))  
using summable-norm-sin  
by (auto simp: Cauchy-product-sums)  
finally show ?thesis .  
qed  

lemma sums-cos-x-plus-y:  
fixes \( x :: \'a::\{real-normed-field,banach\} \)  
shows  
(\( \lambda p. \sum_{n\leq p} \) if even \( p \)  
then \(-1 \) \( ^{(p \div 2)} \) \( \binom{p}{n} \) \( \cdot_R (x^n) \cdot y^{(p-n)} \)  
else 0)  
sums cos (\( x + y \))  
proof –  
have  
(\( \sum_{n\leq p} \) if even \( p \) then \(-1 \) \( ^{(p \div 2)} \) \( \binom{p}{n} \) \( \cdot_R (x^n) \cdot y^{(p-n)} \)  
else 0) = cos-coeff \( p \cdot_R ((x + y) \cdot^n) \)  
for \( p :: nat \)  
proof –  
have  
(\( \sum_{n\leq p} \) if even \( p \) then \(-1 \) \( ^{(p \div 2)} \) \( \binom{p}{n} \) \( \cdot_R (x^n) \cdot y^{(p-n)} \)  
else 0) = cos-coeff \( p \cdot_R ((x + y) \cdot^n) \)  
for \( p :: nat \)  
by simp  
also have ... =  
(if even \( p \)  
then of-real \(-1 \) \( ^{(p \div 2)} \) \( \binom{p}{n} \) \( \cdot_R (x^n) \cdot y^{(p-n)} \)  
else 0)  
by (auto simp: sum-distrib-left field-simps scaleR-conv-of-real nonzero-of-real-divide)  
also have ... = cos-coeff \( p \cdot_R ((x + y) \cdot^n) \)
by (simp add: cos-coeff-def binomial-ring [of x y] scaleR-conv-of-real atLeast0At-Most)

finally show ?thesis.
qed

then have
(λp. ∑ n≤p.
  if even p
    then ((-1) ^ (p div 2) * (p choose n) / (fact p)) *R (x ^ n) * y ^ (p-n)
    else 0) = (λp. cos-coeff p *R ((x+y) ^ p))
  by simp
also have ... sums cos (x + y)
  by (rule cos-converges)
finally show ?thesis.
qed

cos-add:

fixes x :: 'a::{real-normed_field,banach}
shows cos (x + y) = cos x * cos y - sin x * sin y

proof
  have
    (if even p ∧ even n
      then ((-1) ^ (p div 2) * int (p choose n) / (fact p)) *R (x ^ n) * y ^ (p-n)
      else 0) =
      (if even p then ((-1) ^ (p div 2) * int (p choose n) / (fact p)) *R (x ^ n) * y ^ (p-n)
      else 0)
      if n ≤ p for n p :: nat
      by simp
then have
(λp. ∑ n≤p. (if even p then ((-1) ^ (p div 2) * (p choose n) / (fact p)) *R (x ^ n) * y ^ (p-n) else 0))
  sums (cos x * cos y - sin x * sin y)
  using sums-diff [OF cos-x-cos-y [of x y] sin-x-sin-y [of x y]]
  by (simp add: sum-subtractf [symmetric])
then show ?thesis
  by (blast intro: sums-cos-x-plus-y sums-unique2)
qed

sin-minus-converges: (λn. - (sin-coeff n *R (-x) ^ n)) sums sin x

proof
  have [simp]: ∀n. - (sin-coeff n *R (-x) ^ n) = (sin-coeff n *R x ^ n)
    by (auto simp: sin-coeff-def elim!: oddE)
show ?thesis
  by (simp add: sin-def summable-norm-sin [THEN summable-norm-cancel, THEN summable-sums])
qed
theory "Transcendental"

lemma sin-minus [simp]: sin (- x) = - sin x
  for x :: 'a::{real-normed-algebra-1,banach}
  using sin-minus-converges [of x]
  by (auto simp: sin-def summable-norm-sin [THEN summable-norm-cancel]
    suminf-minus sums-iff equation-minus-iff)

lemma cos-minus-converges: (λn. (cos-coeff n *R (-x) ^ n)) sums cos x
proof -
  have [simp]: ∀n. (cos-coeff n *R (-x) ^ n) = (cos-coeff n *R x ^ n)
    by (auto simp: Transcendental.cos-coeff-def elim!: evenE)
  show ?thesis
    by (simp add: cos-def summable-norm-cos [THEN summable-norm-cancel, THEN summable-sums])
qed

lemma cos-minus [simp]: cos (-x) = cos x
  for x :: 'a::{real-normed-algebra-1,banach}
  using cos-minus-converges [of x]
  by (metis cos-def sums-unique)

lemma cos-abs-real [simp]: cos |x :: real| = cos x
  by (simp add: abs-if)

lemma sin-cos-squared-add [simp]: (sin x) ^ 2 + (cos x) ^ 2 = 1
  for x :: 'a::{real-normed-field,banach}
  using sin-cos-squared-add2 [of x - x]
  by (simp add: power2-eq-square algebra-simps
    subst add.commute, rule sin-cos-squared-add)

lemma sin-cos-squared-add2 [simp]: (cos x) ^ 2 + (sin x) ^ 2 = 1
  for x :: 'a::{real-normed-field,banach}
  by (simp add: power2-eq-square algebra-simps)

lemma sin-squared-eq: (sin x) ^ 2 = 1 - (cos x) ^ 2
  for x :: 'a::{real-normed-field,banach}
  unfolding eq-diff-eq by (rule sin-cos-squared-add)

lemma cos-squared-eq: (cos x) ^ 2 = 1 - (sin x) ^ 2
  for x :: 'a::{real-normed-field,banach}
  unfolding eq-diff-eq by (rule sin-cos-squared-add2)

lemma abs-sin-le-one [simp]: |sin x| ≤ 1
  for x :: real
  by (rule power2-le-imp-le) (simp-all add: sin-squared-eq)

lemma sin-ge-minus-one [simp]: - 1 ≤ sin x
  for x :: real
using abs-sin-le-one [of \(x\)] by (simp add: abs-le-iff)

lemma sin-le-one [simp]: \(\sin x \leq 1\)
for \(x::\text{real}\)
using abs-sin-le-one [of \(x\)] by (simp add: abs-le-iff)

lemma abs-cos-le-one [simp]: \(|\cos x| \leq 1\)
for \(x::\text{real}\)
by (rule power2-le-imp-le) (simp-all add: cos-squared-eq)

lemma cos-ge-minus-one [simp]: \(-1 \leq \cos x\)
for \(x::\text{real}\)
using abs-cos-le-one [of \(x\)] by (simp add: abs-le-iff)

lemma cos-le-one [simp]: \(\cos x \leq 1\)
for \(x::\text{real}\)
using abs-cos-le-one [of \(x\)] by (simp add: abs-le-iff)

lemma cos-diff: \(\cos (x - y) = \cos x \cdot \cos y + \sin x \cdot \sin y\)
for \(x::\{\text{real-normed-field}, \text{banach}\}\)
using cos-add [of \(x-y\)] by simp

lemma cos-double: \(\cos(2 \cdot x) = (\cos x)^2 - (\sin x)^2\)
for \(x::\{\text{real-normed-field}, \text{banach}\}\)
using cos-add [where \(x=x\) and \(y=x\)] by (simp add: power2-eq-square)

lemma sin-cos-le1: \(|\sin x \cdot \sin y + \cos x \cdot \cos y| \leq 1\)
for \(x::\text{real}\)
using cos-diff [of \(x y\)] by (metis abs-cos-le-one add.commute)

lemma DERIV-fun-pow: DERIV \(g x :: m\) \(\Rightarrow\) DERIV (\(\lambda x. (g x)^n\)) \(x :: real\)
\(n \cdot (g x)^{n - 1} \cdot m\)
by (auto intro!: derivative-eq-intros simp)

lemma DERIV-fun-exp: DERIV \(g x :: m\) \(\Rightarrow\) DERIV (\(\lambda x. \exp (g x)\)) \(x :: \exp\)
\((g x) \cdot m\)
by (auto intro!: derivative-intros)

113.13 The Constant Pi

definition pi :: real
where pi = \(2 \star (\text{THE } x. 0 \leq x \land x \leq 2 \land \cos x = 0)\)

Show that there’s a least positive \(x\) with \(\cos x = (0::'a)\); hence define pi.

lemma sin-paired: (\(\lambda n. (-1)^n / (\text{fact } (2 \star n + 1)) \cdot x^\sim (2 \star n + 1)\)) sums \(\sin x\)
for \(x::\text{real}\)
proof
have (\(\lambda n. \sum k = n \star 2..<n \star 2 + 2. \text{sin-coeff } k \star x^\sim k\)) sums \(\sin x\)
by (rule sums-group) (use sin-converges [of x, unfolded scaleR-conv-of-real] in auto)
then show ?thesis
by (simp add: sin-coeff-def ac-simps)
qed

lemma sin-gt-zero-02:
fixes x :: real
assumes 0 < x and x < 2
shows 0 < sin x
proof
let ?f = λn::nat. ∑k = n*2..<n*2+2. (−1) ^ k / (fact (2*k+1)) * x^(2*k+1)
have pos: ∀n. 0 < ?f n
proof
fix n :: nat
let ?k2 = real (Suc (Suc (4 * n)))
let ?k3 = real (Suc (Suc (Suc (Suc (4 * n)))))
have x * x < ?k2 * ?k3
  using assms by (intro mult-strict-mono’, simp-all)
then have x * x * x * x ^ (n * 4) < ?k2 * ?k3 * x * x ^ (n * 4)
  by (intro mult-strict-right-mono zero-less-power ;0 < x)
then show 0 < ?f n
by (simp add: ac-simps divide-less-eq)
qed

have sums: ∀f sums sin x
  by (rule sin-paired [THEN sums-group]) simp
show 0 < sin x
  (simp add: suminf-pos)
qed

lemma cos-double-less-one: 0 < x ⇒ x < 2 ⇒ cos (2 * x) < 1
for x :: real
using sin-gt-zero-02 [where x = x] by (auto simp: cos-squared-eq cos-double)

lemma cos-paired: (λn. (−1) ^ n / (fact (2 * n)) * x ^ (2 * n)) sums cos x
for x :: real
proof
have (λn. ∑k = n * 2..<n * 2 + 2. cos_coeff k * x ^ k) sums cos x
  by (rule sums-group) (use cos-converges [of x, unfolded scaleR-conv-of-real] in auto)
then show ?thesis
  by (simp add: cos-coeff-def ac-simps)
qed

lemma sum-pos-lt-pair:
fixes f :: nat ⇒ real
assumes f: summable f and fplus: ∀d. 0 < f (Suc(Suc 0) * d)) + f (k + ((Suc (Suc 0) * d) + 1))
shows \( \sum f \{..<k\} < \text{suminf} f \)

proof –

have \( (\lambda n. \sum n = n \ast \text{Suc} (\text{Suc} 0..<n \ast \text{Suc} (\text{Suc} 0) + \text{Suc} (\text{Suc} 0). f (n + k)) \)

proof (rule sums-group)

show \( (\lambda n. f (n + k)) \) sums \( \sum n. f (n + k) \)

by (simp add: \( f \) summable-iff-shift summable-sums)

qed auto

with \( \text{fplus} \) have \( 0 < (\sum n. f (n + k)) \)

apply (simp add: add.commute)

apply (metis (no-types, lifting) \( \text{suminf-pos} \) summable-def \( \text{sums-unique} \))

done

then show \( ?\text{thesis} \)

by (simp add: \( f \) summable-minus-initial-segment)

qed

lemma \( \text{cos-two-less-zero} \) \([simp] \): \( \cos 2 < (0::\text{real}) \)

proof –

note \( \text{fact-Suc} \) \([simp del] \)

from \( \text{sums-minus} \) \([OF cos-paired] \)

have \( * ((\lambda n. \text{Suc} (\text{Suc} 0..<n \ast \text{Suc} (\text{Suc} 0) + \text{Suc} (\text{Suc} 0). f (n + k)))) \)

by simp

then have \( \text{sm} = \) summable \( \lambda n. (\text{Suc} (\text{Suc} 0..<n \ast \text{Suc} (\text{Suc} 0) + \text{Suc} (\text{Suc} 0). f (n + k)))) \)

by (rule sums-summable)

have \( 0 < (\sum n. \text{Suc} (\text{Suc} 0). f (n + k)) \)

by (simp add: \( \text{fact-num-eq-if} \) \( \text{power-eq-if} \))

moreover have \( (\sum n. \text{Suc} (\text{Suc} 0). f (n + k)) < \)

\( \lambda n. (\text{Suc} (\text{Suc} 0..<n \ast \text{Suc} (\text{Suc} 0) + \text{Suc} (\text{Suc} 0). f (n + k))))) \)

proof –

\{ fix \( d \)

let \( ?six4d = \text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} 4 \ast d)))))) \)

have \( (4::\text{real}) \ast \text{fact} \( (?six4d)) \) \(< \text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (?six4d))))) \)

unfolding \( \text{of-nat-mult} \) \( \text{by} \) \( \text{rule} \) \( \text{mult-strict-mono} \) \( \text{simp-all add: fact-less-mono} \)

then have \( (4::\text{real}) \ast \text{fact} \( (?six4d)) \) \(< \text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (?six4d))))) \)

by (simp only: \( \text{fact-Suc} \) \([of \text{Suc} (\text{Suc} (\text{Suc} (?six4d))) \text{of-nat-mult} \text{of-nat-fact} \))

then have \( (4::\text{real}) \ast \text{inverse} \text{fact} \text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (?six4d))))) \) \(< \text{inverse fact} \text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (?six4d)))))) \)

by (simp add: inverse-eq-divide less-divide-eq)

\}

then show \( ?\text{thesis} \)

by (force \( \text{intro: sum-pos-lt-pair} \) \([OF \text{sm simp add: divide-inverse algebra-simps} \))

qed

ultimately have \( 0 < (\sum n. (\text{Suc} (\text{Suc} 0..<n \ast \text{Suc} (\text{Suc} 0) + \text{Suc} (\text{Suc} 0). f (n + k))))) \)
by (rule order-less-trans)
moreover from * have \(- \cos 2 = (\sum n. - ((- 1::real) ^ n * 2 ^ (2 * n) / (fact (2 * n))))\)
by (rule sums-unique)
ultimately have \((0::real) < - \cos 2\) by simp
then show \(?thesis by simp\)
qed

lemmas \(\text{cos-two-neq-zero [simp]} = \text{cos-two-less-zero [THEN less-imp-neq]}\)
lemmas \(\text{cos-two-le-zero [simp]} = \text{cos-two-less-zero [THEN order-less-imp-le]}\)

lemma \(\text{cos-is-zero}: \exists! x::\text{real} \cdot 0 \leq x \wedge x \leq 2 \wedge \cos x = 0\)
proof (rule ex-ex1I)
show \(\exists x::\text{real} \cdot 0 \leq x \wedge x \leq 2 \wedge \cos x = 0\)
by (rule IVT2 simp-all)
next
fix \(a b :: \text{real}\)
assume \(ab: 0 \leq a \wedge a \leq 2 \wedge \cos a = 0 \wedge b \leq 2 \wedge \cos b = 0\)
have \(\text{cosd}: \bigwedge x::\text{real} \cdot \cos \text{differentiable (at x)}\)
unfolding \(\text{real-differentiable-def}\) by (auto intro: DERIV-cos)
show \(a = b\)
proof (cases \(a b\) rule: linorder-cases)
case less
then obtain \(z\) where \(a < z < b\) (\(\cos\text{ has-real-derivative 0}\) (at \(z\))
using Rolle by (metis \(\text{cosd continuous-on-cos-real ab}\))
then have \(\sin z = 0\)
using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast
then show \(?thesis\)
by (metis \(\langle a < z \rangle \langle z < b \rangle \text{ ab order-less-le-trans less-le sin-gt-zero-02}\))
next
case greater
then obtain \(z\) where \(b < z < a\) (\(\cos\text{ has-real-derivative 0}\) (at \(z\))
using Rolle by (metis \(\text{cosd continuous-on-cos-real ab}\))
then have \(\sin z = 0\)
using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast
then show \(?thesis\)
by (metis \(\langle b < z \rangle \langle z < a \rangle \text{ ab order-less-le-trans less-le sin-gt-zero-02}\))
qed auto

qed

lemma \(\text{pi-half}: \pi/2 = (\text{THE } x. 0 \leq x \wedge x \leq 2 \wedge \cos x = 0)\)
by (simp add: pi-def)

lemma \(\text{cos-pi-half [simp]}: \cos (\pi/2) = 0\)
by (simp add: cos-is-zero [THEN theI'])

lemma \(\text{cos-of-real-pi-half [simp]}: \cos ((\text{of-real } \pi/2) :: 'a) = 0\)
if \(\text{SORT-CONSTRAINT('a::{real-field,banach,real-normed-algebra-1}\})\)
by (metis \(\text{cos-pi-half cos-of-real eq-numeral-simps(4)}\))
nonzero-of-real-divide of-real-0 of-real-numeral

lemma pi-half-gt-zero [simp]: \(0 < \pi/2\)
proof
have \(0 \leq \pi/2\)
by (simp add: pi-half cos-is-zero THEN theI')
then show \(?thesis\)
by (metis cos-pi-half cos-zero less-eq-real-def one-neq-zero)
qed

lemmas pi-half-neq-zero [simp] = pi-half-gt-zero THEN less-imp-neq, symmetric
lemmas pi-half-ge-zero [simp] = pi-half-gt-zero THEN order-less-imp-le

lemma pi-half-less-two [simp]: \(\pi/2 < 2\)
proof
have \(\pi/2 \leq 2\)
by (simp add: pi-half cos-is-zero THEN theI')
then show \(?thesis\)
by (metis cos-pi-half cos-two-neq-zero le-less)
qed

lemmas pi-half-neq-two [simp] = pi-half-less-two THEN less-imp-neq
lemmas pi-half-le-two [simp] = pi-half-less-two THEN order-less-imp-le

lemma pi-gt-zero [simp]: \(0 < \pi\)
using pi-half-gt-zero by simp

lemma pi-ge-zero [simp]: \(0 \leq \pi\)
by (rule pi-gt-zero THEN order-less-imp-le)

lemma pi-neq-zero [simp]: \(\pi \neq 0\)
by (rule pi-gt-zero THEN less-imp-neq, symmetric)

lemma pi-not-less-zero [simp]: \(\neg \pi < 0\)
by (simp add: linorder-not-less)

lemma minus-pi-half-less-zero: \(\neg(\pi/2) < 0\)
by simp

lemma m2pi-less-pi: \(\neg(2\pi) < \pi\)
by simp

lemma sin-pi-half [simp]: \(\sin(\pi/2) = 1\)
using sin-cos-squared-add2 [where \(x = \pi/2\)]
using sin-gt-zero-02 [OF pi-half-gt-zero pi-half-less-two]
by (simp add: power2-eq-1-iff)

lemma sin-of-real-pi-half [simp]: \(\sin ((\text{of-real } \pi/2 :: 'a)) = 1\)
if SORT-CONSTRAINT('a::{real-field,banach,real-normed-algebra-1})
using sin-pi-half
by (metis sin-pi-half eq-numeral-simps(4) nonzero-of-real-divide of-real-1 of-real-numeral
sin-of-real)

lemma sin-cos-eq: sin x = cos (of-real pi/2 - x)
for x :: 'a::{real-normed-field,banach}
by (simp add: cos-diff)

lemma minus-sin-cos-eq: - sin x = cos (x + of-real pi/2)
for x :: 'a::{real-normed-field,banach}
by (simp add: cos-add nonzero-of-real-divide)

lemma cos-sin-eq: cos x = sin (of-real pi/2 - x)
for x :: 'a::{real-normed-field,banach}
using sin-cos-eq [of of-real pi/2 - x] by simp

lemma sin-add: sin (x + y) = sin x * cos y + cos x * sin y
for x :: 'a::{real-normed-field,banach}
using sin-cos-eq [of of-real pi/2 - x - y] by (simp add: cos-sin-eq) (simp add: sin-cos-eq)

lemma sin-diff: sin (x - y) = sin x * cos y - cos x * sin y
for x :: 'a::{real-normed-field,banach}
using sin-add [of x - y] by simp

lemma sin-double: sin(2 * x) = 2 * sin x * cos x
for x :: 'a::{real-normed-field,banach}
using sin-add [where x=x and y=x] by simp

lemma cos-of-real-pi [simp]: cos (of-real pi) = -1
using cos-add [where x = pi/2 and y = pi/2] by (simp add: cos-of-real)

lemma sin-of-real-pi [simp]: sin (of-real pi) = 0
using sin-add [where x = pi/2 and y = pi/2] by (simp add: sin-of-real)

lemma cos-pi [simp]: cos pi = -1
using cos-add [where x = pi/2 and y = pi/2] by simp

lemma sin-pi [simp]: sin pi = 0
using sin-add [where x = pi/2 and y = pi/2] by simp

lemma sin-periodic-pi [simp]: sin (x + pi) = - sin x
by (simp add: sin-add)

lemma sin-periodic-pi2 [simp]: sin (pi + x) = - sin x
by (simp add: sin-add)
lemma cos-periodic-pi [simp]: \( \cos(x + \pi) = -\cos x \)
by (simp add: cos-add)

lemma cos-periodic-pi2 [simp]: \( \cos(\pi + x) = -\cos x \)
by (simp add: cos-add)

lemma sin-periodic [simp]: \( \sin(x + 2 \cdot \pi) = \sin x \)
by (simp add: sin-add sin-double cos-double)

lemma cos-periodic [simp]: \( \cos(x + 2 \cdot \pi) = \cos x \)
by (simp add: cos-add sin-double cos-double)

lemma cos-npi [simp]: \( \cos(n \cdot \pi) = (-1)^n \)
by (induct n) (auto simp: distrib-right)

lemma cos-npi2 [simp]: \( \cos(n \cdot \pi) = (-1)^n \)
by (metis cos-npi mult.commute)

lemma sin-npi [simp]: \( \sin(n \cdot \pi) = 0 \)
for \( n \in \mathbb{N} \)
by (induct n) (auto simp: distrib-right)

lemma sin-npi2 [simp]: \( \sin(n \cdot \pi) = 0 \)
for \( n \in \mathbb{N} \)
by (simp add: mult.commute [of \( \pi \)])

lemma cos-two-pi [simp]: \( \cos(2 \cdot \pi) = 1 \)
by (simp add: cos-double)

lemma sin-two-pi [simp]: \( \sin(2 \cdot \pi) = 0 \)
by (simp add: sin-double)

context
fixes w :: 'a::{real_normed_field,banach}
begin

lemma sin-times-sin: \( \sin w \cdot \sin z = (\cos(w - z) - \cos(w + z)) / 2 \)
by (simp add: cos-diff cos-add)

lemma sin-times-cos: \( \sin w \cdot \cos z = (\sin(w + z) + \sin(w - z)) / 2 \)
by (simp add: sin-add sin-diff)

lemma cos-times-sin: \( \cos w \cdot \sin z = (\sin(w + z) - \sin(w - z)) / 2 \)
by (simp add: sin-diff sin-add)

lemma cos-times-cos: \( \cos w \cdot \cos z = (\cos(w - z) + \cos(w + z)) / 2 \)
by (simp add: cos-diff cos-add)
lemma cos-double-cos: \(\cos(2 \cdot w) = 2 \cdot \cos w^2 - 1\)
by (simp add: cos-double sin-squared-eq)

lemma cos-double-sin: \(\cos(2 \cdot w) = 1 - 2 \cdot \sin w^2\)
by (simp add: cos-double sin-squared-eq)
end

lemma sin-plus-sin: \(\sin w + \sin z = 2 \cdot \sin \left(\frac{(w + z)}{2}\right) \cdot \cos \left(\frac{(w - z)}{2}\right)\)
for \(w::'a::\{\text{real-normed-field, banach}\}\)
apply (simp add: mult.assoc sin-times-cos)
apply (simp add: field-simps)
done

lemma sin-diff-sin: \(\sin w - \sin z = 2 \cdot \sin \left(\frac{(w - z)}{2}\right) \cdot \cos \left(\frac{(w + z)}{2}\right)\)
for \(w::'a::\{\text{real-normed-field, banach}\}\)
apply (simp add: mult.assoc sin-times-cos)
apply (simp add: field-simps)
done

lemma cos-plus-cos: \(\cos w + \cos z = 2 \cdot \cos \left(\frac{(w + z)}{2}\right) \cdot \cos \left(\frac{(w - z)}{2}\right)\)
for \(w::'a::\{\text{real-normed-field, banach, field}\}\)
apply (simp add: mult.assoc cos-times-cos)
apply (simp add: field-simps)
done

lemma cos-diff-cos: \(\cos w - \cos z = 2 \cdot \sin \left(\frac{(w + z)}{2}\right) \cdot \sin \left(\frac{(z - w)}{2}\right)\)
for \(w::'a::\{\text{real-normed-field, banach, field}\}\)
apply (simp add: mult.assoc sin-times-sin)
apply (simp add: field-simps)
done

lemma sin-pi-minus [simp]: \(\sin (\pi - x) = \sin x\)
by (metis sin-minus sin-periodic-pi minus-minus uminus-add-conv-diff)

lemma cos-pi-minus [simp]: \(\cos (\pi - x) = - (\cos x)\)
by (metis cos-minus cos-periodic-pi uminus-add-conv-diff)

lemma sin-minus-pi [simp]: \(\sin (x - \pi) = - (\sin x)\)
by (simp add: sin-diff)

lemma cos-minus-pi [simp]: \(\cos (x - \pi) = - (\cos x)\)
by (simp add: cos-diff)

lemma sin-2pi-minus [simp]: \(\sin (2 \cdot \pi - x) = - (\sin x)\)
by (metis sin-periodic-pi2 add-diff-eq mult-2 sin-pi-minus)

lemma cos-2pi-minus [simp]: \(\cos (2 \cdot \pi - x) = \cos x\)
by (metis (no-types, opaque-lifting) cos-add cos-minus cos-two-pi sin-minus sin-two-pi)
lemma sin-gt-zero2: \(0 < x \implies x < \pi/2 \implies 0 < \sin x\)
by (metis sin-gt-zero-02 order-less-trans pi-half-less-two)

lemma sin-less-zero:
  assumes \(-\pi/2 < x\) and \(x < 0\)
  shows \(\sin x < 0\)
proof
  have \(0 < \sin (-x)\)
  using assms by (simp only: sin-gt-zero2)
  then show \(?thesis\) by simp
qed

lemma pi-less-4: \(\pi < 4\)
using pi-half-less-two by auto

lemma cos-gt-zero:
  \(0 < x \implies x < \pi/2 \implies 0 < \cos x\)
by (simp add: cos-sin-eq sin-gt-zero2)

lemma cos-gt-zero-pi:
  \(-\pi/2 < x \implies x < \pi/2 \implies 0 < \cos x\)
using cos-gt-zero [of \(-x\)]
by (cases rule: linorder-cases [of \(x\) \(0\)]) auto

lemma cos-ge-zero:
  \(-\pi/2 \leq x \implies x \leq \pi/2 \implies 0 \leq \cos x\)
by (auto simp: order-le-less cos-gt-zero-pi)
  (metis cos-pi-half eq-divide-eq eq-numeral-simps(4))

lemma sin-gt-zero:
  \(0 < x \implies x < \pi \implies 0 < \sin x\)
by (simp add: sin-cos-eq cos-gt-zero-pi)

lemma sin-lt-zero:
  \(\pi < x \implies x < 2 \ast \pi \implies \sin x < 0\)
using sin-gt-zero [of \(x - \pi\)]
by (simp add: sin-diff)

lemma pi-ge-two: \(2 \leq \pi\)
proof (rule ccontr)
  assume \(?thesis\) \neg
  then have \(\pi < 2\) by auto
  have \(\exists y > \pi. y < 2 \land y < 2 \ast \pi\)
  proof (cases \(2 < 2 \ast \pi\))
    case True
    with dense[OF \(\pi < 2\)] show \(?thesis\) by auto
  next
    case False
    have \(\pi < 2 \ast \pi\) by auto
    from dense[OF this] and False show \(?thesis\) by auto
  qed
  then obtain \(y\) where \(\pi < y\) and \(y < 2\) and \(y < 2 \ast \pi\)
by blast
then have \(0 < \sin y\)
  using \(\sin gt zero 02\) by auto
moreover have \(\sin y < 0\)
  using \(\sin gt zero[of y - pi]\) \(\pi < y\) and \(y < 2 * \pi\) \(\sin periodic pi[of y - pi]\)
  by auto
ultimately show \(False\) by auto
qed

lemma \(\sin ge zero\): \(0 \leq x = \Rightarrow x \leq \pi = \Rightarrow 0 \leq \sin x\)
by (auto simp: order-le-less \(\sin gt zero\))

lemma \(\sin le zero\): \(\pi \leq x = \Rightarrow x < 2 * \pi = \Rightarrow \sin x \leq 0\)
using \(\sin ge zero\) [of \(x - \pi\)] by (simp add: \(\sin diff\))

lemma \(\sin pi divide n ge 0\) [simp]:
assumes \(n \neq 0\)
shows \(0 \leq \sin(\pi/real n)\)
by (rule \(\sin ge zero\)) (use \(\text{assms}\) in \(\text{simp all add: field split simp}\))

lemma \(\sin pi divide n gt 0\):
assumes \(2 \leq n\)
shows \(0 < \sin(\pi/real n)\)
by (rule \(\sin gt zero\)) (use \(\text{assms}\) in \(\text{simp all add: field split simp}\))

Proof resembles that of \(\cos is zero\) but with \(\pi\) for the upper bound

lemma \(\cos total\):
assumes \(y : -1 \leq y y \leq 1\)
shows \(\exists!x. 0 \leq x \land x \leq \pi \land \cos x = y\)
proof (rule \(\text{ex ex1 I}\))
  show \(\exists x::real. 0 \leq x \land x \leq \pi \land \cos x = y\)
    by (rule \(\text{IVT 2}\)) (simp all add: \(\text{y}\))
next
fix \(a b::real\)
assume \(ab: 0 \leq a \land a \leq \pi \land \cos a = y = 0 \leq b \land b \leq \pi \land \cos b = y\)
have \(\text{cosd}: \bigwedge x::real. \text{cos differentiable \(at x\)}\)
  unfolding \(\text{real differentiable def}\ by\ (auto intro: DERIV cos)\)
  show \(a = b\)
proof (cases \(a b\) rule: linorder cases)
  case less
  then obtain \(z\) where \(a < z z < b\) \(\cos has real derivative 0\) \(at z\)
    using Rolle by (metis cosd continuous on cos real \(ab\))
  then have \(\sin z = 0\)
    using DERIV cos DERIV unique neg equal 0 iff equal by blast
  then show ?thesis
    by (metis \(a < z\) \(z < b\) \(ab order less le trans less le \(\sin gt zero)\))
next
  case greater
then obtain \( z \) where \( b < z < a \) (cos has-real-derivative 0) (at \( z \))
using Rolle by (metis cosd continuous-on-cos-real ab)
then have \( \sin z = 0 \)
using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast
then show \( \exists \)thesis
by (metis \( \langle b < z \rangle \langle z < a \rangle \) ab order-less-le-trans less-le sin-gt-zero)
qed auto
qed

lemma \( \sin\)–total:
assumes \( y : -1 \leq y y \leq 1 \)
shows \( \exists ! x. - (pi/2) \leq x \land x \leq pi/2 \land \sin x = y \)
proof –
from cos-total [OF \( y \)]
obtain x where \( x : 0 \leq x x \leq \pi \cos x = y \)
and uniq: \( \forall x'. 0 \leq x' \Longrightarrow x' \leq \pi \Longrightarrow \cos x' = y \Longrightarrow x' = x \)
by blast
show \( \exists \)thesis
unfolding sin-cos-eq
proof (rule ex1I [where \( a = \pi/2 - x \)])
show \( - (pi/2) \leq z \land z \leq pi/2 \land \cos (of-real pi/2 - z) = y \Longrightarrow z = pi/2 - x \) for \( z \)
using uniq [of pi/2 -\( z \)] by auto
qed (use \( x \) in auto)
qed

lemma \( \cos\)–zero-lemma:
assumes \( 0 \leq x \cos x = 0 \)
shows \( \exists n. odd n \land x = of-nat n * (pi/2) \)
proof –
have xle: \( x < (1 + \text{real-of-int } \lfloor x/pi \rfloor) \ast pi \)
using floor-correct [of \( x/pi \)]
by (simp add: add.commute divide-less-eq)
obtain n where \( \text{real n } \ast \pi \leq x x < \text{real } (Suc n) \ast pi \)
proof
show \( \text{real } (nat } \lfloor x / pi \rfloor) \ast pi \leq x \)
using assms floor-divide-lower [of \( pi \) \( x \)] by auto
show \( x < \text{real } (Suc (nat } \lfloor x / pi \rfloor)) \ast pi \)
using assms floor-divide-upper [of \( pi \) \( x \)] by (simp add: xle)
qed
then have \( x: 0 \leq x - n \ast pi (x - n \ast pi) \leq pi \cos (x - n \ast pi) = 0 \)
by (auto simp: algebra-simps cos-diff assms)
then have \( \exists ! x. 0 \leq x \land x \leq \pi \land \cos x = 0 \)
by (auto simp: intro!: cos-total)
then obtain \( \vartheta \) where \( \vartheta : 0 \leq \vartheta \vartheta \leq \pi \cos \vartheta = 0 \)
and uniq: \( \forall \varphi. 0 \leq \varphi \Longrightarrow \varphi \leq \pi \Longrightarrow \cos \varphi = 0 \Longrightarrow \varphi = \vartheta \)
by blast
then have \( x - \text{real } n \ast pi = \vartheta \)
using \( x \) by blast
moreover have pi/2 = 0
  using pi-half-ge-zero uniq by fastforce
ultimately show ?thesis
  by (rule_tac x = Suc (2 * n) in exI) (simp add: algebra-simps)
qed

lemma sin-zero-lemma:
  assumes 0 ≤ x sin x = 0
  shows ∃ n::nat. even n ∧ x = real n * (pi/2)
proof −
  obtain n where odd n and n: x + pi/2 = of-nat n * (pi/2) n > 0
  using cos-zero-lemma [of x + pi/2] assms by (auto simp add: cos-add)
then have x = real (n - 1) * (pi/2)
  by (simp add: algebra-simps of-nat-diff)
then show ?thesis
  by (simp add: ‹odd n›)
qed

lemma cos-zero-iff:
  cos x = 0 ↔ ((∃ n. odd n ∧ x = real n * (pi/2)) ∨ (∃ n. odd n ∧ x = − (real n * (pi/2))))
(is ?lhs = ?rhs)
proof −
  have *: cos (real n * pi/2) = 0 if odd n for n :: nat
  proof −
    from that obtain m where n = 2 * m + 1 ..
    then show ?thesis
      by (simp add: field-simps) (simp add: cos-add add-divide-distrib)
  qed
  show ?thesis
  proof
    show ?rhs if ?lhs
      using that cos-zero-lemma [of x] cos-zero-lemma [of −x] by force
    show ?lhs if ?rhs
      using that by (auto dest: * simp del: eq-divide-eq-numeral1)
  qed
qed

lemma sin-zero-iff:
  sin x = 0 ↔ ((∃ n. even n ∧ x = real n * (pi/2)) ∨ (∃ n. even n ∧ x = − (real n * (pi/2))))
(is ?lhs = ?rhs)
proof −
  show ?rhs if ?lhs
    using that sin-zero-lemma [of x] sin-zero-lemma [of −x] by force
  show ?lhs if ?rhs
    using that by (auto elim: evenE)
qed
lemma \textit{sin-zero-pi-iff}:

fixes x::real
assumes |x| < pi
shows \( \sin x = 0 \iff x = 0 \)
proof
  show \( x = 0 \) if \( \sin x = 0 \)
    using \textit{that} \textit{assms} by (auto simp: \textit{sin-zero-iff})
qed auto

lemma \textit{cos-zero-iff-int}:

\( \cos x = 0 \iff (\exists i. \text{odd } i \land x = \text{of-int } i \ast (\pi/2)) \)
proof
  have \( 1: \all n. \text{odd } n \implies \exists i. \text{odd } i \land \text{real } n = \text{real-of-int } i \)
    by (metis even-of-nat-iff of-int-of-nat-eq)
  have \( 2: \all n. \text{odd } n \implies \exists i. \text{odd } i \land - (\text{real } n \ast \pi) = \text{real-of-int } i \ast \pi \)
    by (metis even-minus even-of-nat-iff mult.commute mult-minus-right of-int-minus of-int-of-nat-eq)
  have \( 3: [\text{odd } i; \forall n. \text{even } n \centerdot \text{real-of-int } i \neq -(\text{real } n)] \)
      \( \implies \exists n. \text{odd } n \land \text{real-of-int } i = \text{real } n \, \text{for } i \)
    by (cases i rule: \textit{int-cases2}) auto
  show \( \text{?thesis} \)
    by (force simp: \textit{cos-zero-iff} intro !:\ 1 2 3)
qed

lemma \textit{sin-zero-iff-int}: \( \sin x = 0 \iff (\exists i. \text{even } i \land x = \text{of-int } i \ast (\pi/2)) \)
proof safe
  assume \( \text{?lhs} \)
  then consider \( \text{plus } n \) where \( \text{even } n \, x = \text{real } n \ast (\pi/2) \) | \( \text{minus } n \) where \( \text{even } n \, x = - (\text{real } n \ast (\pi/2)) \)
    using \textit{sin-zero-iff} by auto
  then show \( \exists n. \text{even } n \land x = \text{of-int } n \ast (\pi/2) \)
  proof cases
    case \( \text{plus} \)
    then show \( \text{?rhs} \)
      by (metis even-of-nat-iff of-int-of-nat-eq)
  next
    case \( \text{minus} \)
    then show \( \text{?thesis} \)
      by (rule-tac \( x=- (\text{int } n) \) in \textit{exI}) simp
  qed
next
fix i :: int
assume \( \text{even } i \)
then show \( \sin (\text{of-int } i \ast (\pi/2)) = 0 \)
  by (cases i rule: \textit{int-cases2}, simp-all add: \textit{sin-zero-iff}]
qed

lemma \textit{sin-zero-iff-int2}: \( \sin x = 0 \iff (\exists i::\text{int}. x = \text{of-int } i \ast \pi) \)
proof –
have \( \sin x = 0 \iff (\exists\ i. \text{even } i \land x = \text{real-of-int } i \ast (\pi/2)) \)
by (auto simp: sin-zero-iff-int)
also have \( ... = (\exists\ j. x = \text{real-of-int } (2\ast j) \ast (\pi/2)) \)
using dvd-triv-left by blast
also have \( ... = (\exists\ i::\text{int}. x = \text{of-int } i \ast \pi) \)
by auto
finally show \( \text{thesis} \).
qed

lemma \( \text{cos-zero-iff-int2} \):
fixes \( x :: \text{real} \)
shows \( \cos x = 0 \iff (\exists\ n :: \text{int}. x = n \ast \pi + \pi/2) \)
using sin-zero-iff-int2[of \( -\pi/2 \)] unfolding sin-cos-eq
by (auto simp add: algebra-simps)

lemma \( \text{sin-npi-int} \):
shows \( \sin (\pi \ast \text{of-int } n) = 0 \)
by (simp add: sin-zero-iff-int2)

lemma \( \text{cos-monotone-0-pi} \):
assumes \( 0 \leq y \) and \( y < x \) and \( x \leq \pi \)
shows \( \cos x < \cos y \)
proof –
have \( -(x - y) < 0 \) using assms by auto
from MVT2[OF \( y < x \) DERIV-cos]
obtain \( z \) where \( y < z \) and \( z < x \) and \( \text{cos-diff}: \cos x - \cos y = (x - y) \ast -\sin z \)
by auto
then have \( 0 < z \) and \( z < \pi \)
using assms by auto
then have \( 0 < \sin z \)
using sin-gt-zero by auto
then have \( \cos x - \cos y < 0 \)
unfolding \( \text{cos-diff minus-mult-commute[symmetric]} \)
using \( -(x - y) < 0 \) by (rule mult-pos-neg2)
then show \( \text{thesis} \)
by auto
qed

lemma \( \text{cos-monotone-0-pi-le} \):
assumes \( 0 \leq y \) and \( y \leq x \) and \( x \leq \pi \)
shows \( \cos x \leq \cos y \)
proof (cases \( y < x \))
case True
show \( \text{thesis} \)
using cos-monotone-0-pi[of \( \theta \leq y \) True \( x \leq \pi \)] by auto
next
case False
then have \( y = x \)
using \( y \leq x \)
by auto
then show \( \text{thesis} \)
by auto
qed
lemma \( \text{cos-monotone-minus-pi-0} \):
assumes \(-\pi \leq y \) and \( y < x \) and \( x \leq 0 \)
shows \( \cos y < \cos x \)
proof
have \( 0 \leq -x \) and \(-x < -y \) and \(-y \leq \pi \)
using assms by auto
from \( \text{cos-monotone-0-pi[of this]} \) show ?thesis
unfolding \( \text{cos-minus} \).
qed

lemma \( \text{cos-monotone-minus-pi-0}' \):
assumes \(-\pi \leq y \) and \( y \leq x \) and \( x \leq 0 \)
shows \( \cos y \leq \cos x \)
proof (cases \( y < x \))
case True
show ?thesis using \( \text{cos-monotone-minus-pi-0[of \(-\pi \leq y \) True \( x \leq 0 \)]} \)
by auto
next
case False
then have \( y = x \)
using \( \text{y \leq x} \) by auto
then show ?thesis by auto
qed

lemma \( \text{sin-monotone-2pi} \):
assumes \(-\pi/2 \leq y \) and \( y < x \) and \( x \leq \pi/2 \)
shows \( \sin y < \sin x \)
unfolding \( \text{sin-cos-eq} \)
using assms by (auto intro: \( \text{cos-monotone-0-pi} \))

lemma \( \text{sin-monotone-2pi-le} \):
assumes \(-\pi/2 \leq y \) and \( y \leq x \) and \( x \leq \pi/2 \)
shows \( \sin y \leq \sin x \)
by (metis assms le-less \( \text{sin-monotone-2pi} \))

lemma \( \text{sin-x-le-x} \):
fixes \( x \) :: \( \text{real} \)
assumes \( x \geq 0 \)
shows \( \sin x \leq x \)
proof
let \( \mathcal{f} = \lambda x. x - \sin x \)
have \( \bigwedge u. [0 \leq u; u \leq x] \implies \exists y. (\mathcal{f} \text{ has-real-derivative } 1 - \cos u) \text{ (at } u) \)
by (auto intro!: \( \text{derivative-eq-intros simp: field-simps} \))
then have \( \mathcal{f} x \geq \mathcal{f} 0 \)
by (metis \( \text{cos-le-one diff-ge-0-iff-ge DERIV-nonneg-imp-nondecreasing} \) [OF assms])
then show \( \sin x \leq x \) by simp
qed
THEORY "Transcendental"

lemma sin-x-ge-neg-x:
  fixes x :: real
  assumes x: x ≥ 0
  shows sin x ≥ − x
proof –
  let f = λx. x + sin x
  have f: ∀u. [0 ≤ u; u ≤ x] ⟷ ∃y. (f has-real-derivative 1 + cos u) (at u)
    by (auto intro!: derivative-eq-intros simp: field-simps)
  have f x ≥ f 0
    by (rule DERIV-nonneg-imp-nondecreasing [OF assms]) (use f in force)
  then show sin x ≥ − x by simp
qed

lemma abs-sin-x-le-abs-x:
  |sin x| ≤ |x|
for x :: real
  using sin-x-ge-neg-x[of x] sin-x-le-x[of x] sin-x-ge-neg-x[of − x] sin-x-le-x[of − x]
  by (auto simp: abs-real-def)

subsection{113.14 More Corollaries about Sine and Cosine}

lemma sin-cos-npi [simp]: sin (real (Suc (2 * n)) * pi/2) = (−1) ^ n
proof –
  have sin ((real n + 1/2) * pi) = cos (real n * pi)
    by (auto simp: algebra-simps sin-add)
  then show ?thesis
    by (simp add: distrib-right add-divide-distrib add.commute mult.commute[of pi])
qed

lemma cos-2npi [simp]: cos (2 * real n * pi) = 1
for n :: nat
  by (cases even n) (simp-all add: cos-double mult.assoc)

lemma cos-3over2-pi [simp]: cos (3/2*pi) = 0
proof –
  have cos (3/2*pi) = cos (pi + pi/2)
    by simp
  also have ... = 0
    by (subst cos-add, simp)
  finally show ?thesis .
qed

lemma sin-2npi [simp]: sin (2 * real n * pi) = 0
for n :: nat
  by (auto simp: mult.assoc sin-double)

lemma sin-3over2-pi [simp]: sin (3/2*pi) = − 1
proof –
    have \(\sin (\frac{3}{2} \pi) = \sin (\pi + \pi/2)\)
        by simp
    also have \(\ldots = -1\)
        by (subst sin-add, simp)
    finally show \(?thesis\).
qed

lemma \(\cos-pi-eq-zero\) [simp]: \(\cos (\pi \times \text{real} (\text{Suc} (2 \times m)) / 2) = 0\)
    by (simp only: cos-add sin-add of-nat-Suc distrib-right distrib-left add-divide-distrib, auto)

lemma \(\text{DERIV-cos-add}\) [simp]: \(\text{DERIV} (\lambda x. \cos (x + k)) \ x a > -\sin (xa + k)\)
    by (auto intro!: derivative-eq-intros)

lemma \(\text{sin-zero-norm-cos-one}\):
    fixes \(x\) :: 'a::{real-normed-field, banach}
    assumes \(\sin x = 0\)
    shows \(\|\cos x\| = 1\)
    using sin-cos-squared-add [of \(x\), unfolded assms]
    by (simp add: square-norm-one)

lemma \(\text{sin-zero-abs-cos-one}\): \(\sin x = 0 \rightarrow |\cos x| = (1::real)\)
    using sin-zero-norm-cos-one by fastforce

lemma \(\text{cos-one-sin-zero}\):
    fixes \(x\) :: 'a::{real-normed-field, banach}
    assumes \(\cos x = 1\)
    shows \(\sin x = 0\)
    using sin-cos-squared-add [of \(x\), unfolded assms]
    by simp

lemma \(\text{sin-times-pi-eq-0}\): \(\sin (x \times \pi) = 0 \leftrightarrow x \in \mathbb{Z}\)
    by (simp add: sin-zero-iff-int2) (metis Ints-cases Ints-of-int)

lemma \(\text{cos-one-2pi}\): \(\cos x = 1 \leftrightarrow (\exists n::\text{nat}. \ x = n \times 2 \times \pi) \lor (\exists n::\text{nat}. \ x = - (n \times 2 \times \pi))\)
    (is \(?lhs = \ ?rhs\))
proof
    assume \(?lhs\)
    then have \(\sin x = 0\)
        by (simp add: cos-one-sin-zero)
    then show \(?rhs\)
        proof (simp only: sin-zero-iff, elim exE disjE conjE)
            fix \(n\) :: \(\text{nat}\)
            assume \(n\): even \(n x = \text{real} \ n \times (\pi/2)\)
            then obtain \(m\) where \(m\): \(n = 2 \times m\)
                using dvdE by blast
            then have \(\text{me: even } m\) using \(?lhs\) \(\times\) \(n\)
by (auto simp: field-simps) (metis one-neq-neg-one power-minus-odd power-one)

show ?rhs
  using m me n
  by (auto simp: field-simps elim!: evenE)

next

fix n :: nat

assume n: even n x = -(real n * (pi/2))

then obtain m where: m = 2 * n
  using dvdE by blast

then have me: even m
  using ‹?lhs› n
by (auto simp: field-simps)

show ?rhs
  using m me n
  by (auto simp: field-simps elim!: evenE)

qed

next

assume ?rhs

then show cos x = 1
  by (metis cos-one-2pi mult.commute mult-minus-right of-int-minus of-int-of-nat-eq)

qed

lemma cos-one-2pi-int: cos x = 1 ↔ (∃n::int. x = n * 2 * pi) (is ?lhs = ?rhs)

proof

assume cos x = 1

then show ?rhs
by (metis cos-one-2pi mult.commute mult-minus-right of-int-minus of-int-of-nat-eq)

next

assume ?rhs

then show cos x = 1
by (clarsimp simp add: cos-one-2pi) (metis mult-minus-right of-int-of-nat)

qed

lemma cos-npi-int [simp]:
fixes n::int
shows cos (pi * of-int n) = (if even n then 1 else -1)
by (auto simp: algebra-simps cos-one-2pi-int elim!: evenE)

lemma sin-cos-sqrt: 0 ≤ sin x =⇒ sin x = sqrt (1 - (cos x ^ 2))
using sin-squared-eq real-sqrt-unique by fastforce

lemma sin-eq-0-pi: - pi < x =⇒ x < pi =⇒ sin x = 0 =⇒ x = 0
by (metis sin-gt-zero sin-minus minus-less-iff neg-0-less-iff-less not-less-iff-gr-or-eq)

lemma cos-treble-cos: cos (3 * x) = 4 * cos x ^ 3 - 3 * cos x
for x :: 'a::{real-normed-field,banach}

proof

have*: (sin x * (sin x * 3)) = 3 - (cos x * (cos x * 3))
by (simp add: mult.assoc [symmetric] sin-squared-eq [unfolded power2-eq-square])

have cos(3 * x) = cos(2 * x + x)
  by simp
also have \( 4 \cos x \cdot 3 - 3 \cos x \)

unfolding \( \cos \text{-add} \cos \text{-double} \sin \text{-double} \)
by \( \text{simp add: } * \text{ field-simps power2-eq-square power3-eq-cube} \)
finally show \( \text{thesis} \).

qed

lemma \( \text{cos-}45 \): \( \cos (\pi/4) = \sqrt{2} / 2 \)
proof –

let \( ?c = \cos (\pi/4) \)
let \( ?s = \sin (\pi/4) \)

have \( \text{nonneg: } 0 \leq ?c \)
by \( \text{simp add: cos-ge-zero} \)

have \( 0 = \cos (\pi/4 + \pi/4) \)
by \( \text{simp} \)
also have \( \cos (\pi/4 + \pi/4) = ?c^2 - ?s^2 \)
by \( \text{simp only: cos-add power2-eq-square} \)
also have \( \ldots = 2 * ?c^2 - 1 \)
by \( \text{simp add: sin-squared-eq} \)
finally have \( ?c^2 = (\sqrt{2} / 2)^2 \)
by \( \text{simp add: power-divide} \)
then show \( \text{thesis} \)
using \( \text{nonneg by (rule power2-eq-imp-eq)} \) simp

qed

lemma \( \text{cos-}30 \): \( \cos (\pi/6) = \sqrt{3}/2 \)
proof –

let \( ?c = \cos (\pi/6) \)
let \( ?s = \sin (\pi/6) \)

have \( \text{pos-c: } 0 < ?c \)
by \( \text{rule cos-gt-zero simp-all} \)

have \( 0 = \cos (\pi/6 + \pi/6 + \pi/6) \)
by \( \text{simp} \)
also have \( \ldots = (?c * ?c - ?s * ?s) * ?c - (?s * ?c + ?c * ?s) * ?s \)
by \( \text{simp only: cos-add sin-add} \)
also have \( \ldots = ?c * (?c^2 - 3 * ?s^2) \)
by \( \text{simp add: algebra-simps power2-eq-square} \)
finally have \( ?c^2 = (\sqrt{3}/2)^2 \)
using \( \text{pos-c by (simp add: sin-squared-eq power-divide)} \)
then show \( \text{thesis} \)
using \( \text{pos-c [THEN order-less-imp-le]} \)
by \( \text{rule power2-eq-imp-eq simp} \)

qed

lemma \( \text{sin-}45 \): \( \sin (\pi/4) = \sqrt{2} / 2 \)
by \( \text{simp add: sin-cos-eq cos-}45 \)

lemma \( \text{sin-}60 \): \( \sin (\pi/3) = \sqrt{3}/2 \)
by \( \text{simp add: sin-cos-eq cos-}30 \)
lemma cos-60: $\cos \left( \frac{\pi}{3} \right) = 1/2$
proof
  have $0 \leq \cos \left( \frac{\pi}{3} \right)$
    by (rule cos-ge-zero) (use pi-half-ge-zero in linarith+)
  then show ?thesis
    by (simp add: cos-squared-eq sin-60 power-divide power2-eq-imp-eq)
qed

lemma sin-30: $\sin \left( \frac{\pi}{6} \right) = 1/2$
  by (simp add: sin-cos-eq cos-60)

lemma cos-120: $\cos \left( 2 \ast \frac{\pi}{3} \right) = -1/2$
  and sin-120: $\sin \left( 2 \ast \frac{\pi}{3} \right) = \sqrt{3} / 2$
  using sin-double[of pi/3] cos-double[of pi/3]
  by (simp-all add: power2-eq-square sin-60 cos-60)

lemma cos-120': $\cos \left( \pi \ast \frac{2}{3} \right) = -1/2$
  using cos-120 by (subst mult.commute)

lemma sin-120': $\sin \left( \pi \ast \frac{2}{3} \right) = \sqrt{3} / 2$
  using sin-120 by (subst mult.commute)

lemma cos-integer-2pi: $n \in \mathbb{Z} \Rightarrow \cos(2 \ast \pi \ast n) = 1$
  by (metis Ints-cases cos-one-2pi-int mult.assoc mult.commute)

lemma sin-integer-2pi: $n \in \mathbb{Z} \Rightarrow \sin(2 \ast \pi \ast n) = 0$
  by (metis sin-two-pi Ints-mult mult.assoc mult.commute sin-times-pi-eq-0)

lemma cos-int-2pin [simp]: $\cos((2 \ast \pi) \ast \text{of-int } n) = 1$
  by (simp add: cos-one-2pi-int)

lemma sin-int-2pin [simp]: $\sin((2 \ast \pi) \ast \text{of-int } n) = 0$
  by (metis Ints-of-int sin-integer-2pi)

lemma sin-cos-eq-iff: $\sin y = \sin x \land \cos y = \cos x \leftrightarrow (\exists n::\text{int}. y = x + 2 \ast \pi \ast n)$ (is ?L=?R)
proof
  assume ?L
  then have $\cos (y-x) = 1$
    using cos-add[of y -x] by simp
  then show ?R
    by (metis cos-one-2pi-int add.commute diff-add-cancel mult.assoc mult.commute)
next
  assume ?R
  then show ?L
    by (auto simp: sin-add cos-add)
qed
lemma sincos-principal-value: \( \exists y. (-\pi < y \leq \pi) \land (\sin y = \sin x \land \cos y = \cos x) \)

proof

- define \( y \) where \( y \equiv \pi - (2 * \pi) \frac{(\pi - x)}{(2 * \pi)} \)

  have \(-\pi < y \leq \pi\)

  by (auto simp: field-simps frac-lt-1 y-def)

moreover

have \( \sin y = \sin x \)

by (simp-all add: y-def frac-def divide-simps sin-add cos-add mult-of-int-commute)

ultimately

show \(?thesis\) by metis
qed

113.15 Tangent

definition \( \tan :: 'a \Rightarrow 'a::\{real-normed-field,banach\} \)

where \( \tan = (\lambda x. \sin x / \cos x) \)

lemma tan-of-real: of-real (\( \tan x \)) = (\( \tan (\text{of-
real } x) \)) :: \( 'a::\{real-normed-field,banach\} \)

by (simp add: tan-def sin-of-real cos-of-real)

lemma tan-in-Reals [simp]: \( z \in 'a \Rightarrow \tan z \in 'a \)

for \( z :: 'a::\{real-normed-field,banach\} \)

by (simp add: tan-def)

lemma tan-zero [simp]: \( \tan 0 = 0 \)

by (simp add: tan-def)

lemma tan-pi [simp]: \( \tan \pi = 0 \)

by (simp add: tan-def)

lemma tan-npi [simp]: \( \tan (\text{real } n * \pi) = 0 \)

for \( n :: \text{nat} \)

by (simp add: tan-def)

lemma tan-pi-half [simp]: \( \tan (\pi / 2) = 0 \)

by (simp add: tan-def)

lemma tan-minus [simp]: \( \tan (- x) = - \tan x \)

by (simp add: tan-def)

lemma tan-periodic [simp]: \( \tan (x + 2 * \pi) = \tan x \)

by (simp add: tan-def)

lemma lemma-tan-add1: \( \cos x \neq 0 \Longrightarrow \cos y \neq 0 \Longrightarrow 1 - \tan x * \tan y = \cos (x + y) / (\cos x * \cos y) \)

by (simp add: tan-def cos-add field-simps)

lemma add-tan-eq: \( \cos x \neq 0 \Longrightarrow \cos y \neq 0 \Longrightarrow \tan x + \tan y = \sin(x + y) / (\cos x * \cos y) \)
THEORY "Transcendental"

\[ x \ast \cos y \]
for \( x \) :: 'a::{real-normed-field,banach}
by (simp add: tan-sin add field-simps)

\textbf{lemma} tan-eq-0-cos-sin: \( x = 0 \iff \cos x = 0 \lor \sin x = 0 \)
by (auto simp: tan-def)

Note: half of these zeros would normally be regarded as undefined cases.

\textbf{lemma} tan-eq-0-Ex:
assumes \( \tan x = 0 \)
obtains \( k \) :: int where \( x = (k/2) \ast \pi \)
using assms
by (metis cos-zero-iff-int mult.commute sin-zero-iff-int tan-eq-0-cos-sin times-divide-eq-left)

\textbf{lemma} tan-add:
\( \cos x \neq 0 \Longrightarrow \cos y \neq 0 \Longrightarrow \cos (x + y) \neq 0 \Longrightarrow \tan (x + y) = (\tan x + \tan y)/(1 - \tan x \ast \tan y) \)
for \( x \) :: 'a::{real-normed-field,banach}
by (simp add: add-tan-eq lemma-tan-add1 field-simps) (simp add: tan-def)

\textbf{lemma} tan-double:
\( \cos x \neq 0 \Longrightarrow \cos (2 \ast x) \neq 0 \Longrightarrow \tan (2 \ast x) = (2 \ast \tan x)/((1 - (\tan x)^2)) \)
for \( x \) :: 'a::{real-normed-field,banach}
using tan-add [of x x]
by (simp add: power2-eq-square)

\textbf{lemma} tan-gt-zero:
\( 0 < x \Longrightarrow x < \pi/2 \Longrightarrow 0 < \tan x \)
by (simp add: tan-def zero-less-divide_iff sin-gt-zero2 cos-gt-zero-pi)

\textbf{lemma} tan-less-zero:
assumes \( -\pi/2 < x \) and \( x < 0 \)
shows \( \tan x < 0 \)
proof -
  have \( 0 < \tan (-x) \)
  using assms
by (simp only: tan-gt-zero)
then show \( \thesis \) by simp
qed

\textbf{lemma} tan-half:
\( \tan x = \sin (2 \ast x) / (\cos (2 \ast x) + 1) \)
for \( x \) :: 'a::{real-normed-field,banach,field}
unfolding tan-def sin-double cos-double sin-squared-eq
by (simp add: power2-eq-square)

\textbf{lemma} tan-30:
\( \tan (\pi/6) = 1 / \sqrt{3} \)
unfolding tan-def
by (simp add: sin-30 cos-30)

\textbf{lemma} tan-45:
\( \tan (\pi/4) = 1 \)
unfolding tan-def
by (simp add: sin-45 cos-45)
THEORY "Transcendental"

lemma tan-60: tan (pi/3) = sqrt 3
  unfolding tan-def by (simp add: sin-60 cos-60)

lemma DERIV-tan [simp]: cos x ≠ 0 ⇒ DERIV tan x := inverse ((cos x)^2)
  for x :: 'a::{real-normed-field,banach}
  unfolding tan-def by (auto intro!: derivative-eq-intros, simp add: divide-inverse power2-eq-square)

declare DERIV-tan[THEN DERIV-chain2, derivative-intros]
and DERIV-tan[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]

lemmas has-derivative-tan[derivative-intros] = DERIV-tan[THEN DERIV-compose-FDERIV]

lemma isCont-tan: cos x ≠ 0 ⇒ isCont tan x
  for x :: 'a::{real-normed-field,banach}
  by (rule DERIV-tan[THEN DERIV-isCont])

lemma isCont-tan' [simp,continuous-intros]:
  fixes a :: 'a::{real-normed-field,banach} and f :: 'a ⇒ 'a
  shows isCont f a ⇒ cos (f a) ≠ 0 ⇒ isCont (λx. tan (f x)) a
  by (rule isCont-tendsto-compose [OF isCont-tan])

lemma tendsto-tan [tendsto-intros]:
  fixes f :: 'a ⇒ 'a::{real-normed-field,banach}
  shows (f −−→ a) F ⇒ cos a ≠ 0 ⇒ ((λx. tan (f x)) −−→ tan a) F
  by (rule isCont-tendsto-compose [OF isCont-tan])

lemma continuous-tan:
  fixes f :: 'a ⇒ 'a::{real-normed-field,banach}
  shows continuous F f ⇒ cos (f x) ≠ 0 ⇒ continuous (λx. tan (f x))
  unfolding continuous-def by (rule tendsto-tan)

lemma continuous-on-tan [continuous-intros]:
  fixes f :: 'a ⇒ 'a::{real-normed-field,banach}
  shows continuous-on s f ⇒ (∀x∈s. cos (f x) ≠ 0) ⇒ continuous-on s (λx. tan (f x))
  unfolding continuous-on-def by (auto intro: tendsto-tan)

lemma continuous-within-tan [continuous-intros]:
  fixes f :: 'a ⇒ 'a::{real-normed-field,banach}
  shows continuous (at x within s) f ⇒ cos (f x) ≠ 0 ⇒ continuous (at x within s) (λx. tan (f x))
  unfolding continuous-within by (rule tendsto-tan)

lemma LIM-cos-div-sin: (λx. cos(x)/sin(x)) −−→ 0 −−→ pi/2
  by (rule tendsto-cong-limit, (rule tendsto-intros)+, simp-all)
lemma lemma-tan-total:
assumes $0 < y$ shows $\exists x. \ 0 < x \land x < \pi/2 \land y < \tan x$
proof
  obtain $s$ where $0 < s$
  and $s$: $\forall x. [x \neq \pi/2; \| (x - \pi/2) \| < s] \implies \| \cos x / \sin x - 0 \| < \text{inverse } y$
  using $\text{LIM-D [OF LIM-cos-div-sin, of inverse } y\text{]}$ that assms by force
  obtain $e$ where $e: 0 < e < s < \pi/2$
  using $\langle 0 < s \rangle$ field-lbound-gt-zero pi-half-gt-zero by blast
  show $\?thesis$
  proof (intro exI conjI)
    have $0 < \sin e 0 < \cos e$
      using $e$ by (auto intro: cos-gt-zero sin-gt-zero2 simp: mult.commute)
    then
    show $y < \tan (\pi/2 - e)$
      using $s$ [of $\pi/2 - e$] e assms
      by (simp add: tan-def sin-diff cos-diff) (simp add: field-simps split: if-split-asm)
  qed (use $e$ in auto)
qed
lemma tan-total-pos:
assumes $0 \leq y$ shows $\exists x. \ 0 \leq x \land x < \pi/2 \land \tan x = y$
proof (cases $y = 0$)
  case True
  then show $\?thesis$
    using pi-half-gt-zero tan-zero by blast
next
  case False
  with assms have $y > 0$
    by linarith
  obtain $x$ where $x: 0 < x < \pi/2 \land y < \tan x$
    using lemma-tan-total $\langle 0 < y \rangle$ by blast
  have $\exists u \geq 0. \ u \leq x \land \tan u = y$
    proof (intro IVT allI implI)
      show isCont $\tan u$ if $0 \leq u \land u \leq x$ for $u$
        proof
          have $\cos u \neq 0$
            using antisym-conv2 cos-gt-zero that $x(2)$ by fastforce
          with assms show $\?thesis$
            by (auto intro!: DERIV-tan [THEN DERIV-isCont])
        qed
      qed (use assms x in auto)
      then show $\?thesis$
        using $x(2)$ by auto
    qed
major premise: $\exists x. -(\pi/2 < x < (\pi/2) \land \tan x = y$
proof (cases $0::\text{real}$ rule: le-cases)
case le

then show \(\exists\)thesis 
  by (meson less-le-trans minus-pi-half-less-zero tan-total-pos)
next
  case ge
  with tan-total-pos [of \(-\gamma\)] obtain \(x\) where \(0 \leq x < \pi/2\) tan \(x = -\gamma\)
  by force
then show \(\exists\)thesis 
  by (rule-tac \(x = -x\) in exI) auto
qed

proposition tan-total: \(\exists!\ x. -\pi/2 < x < \pi/2 \land\) tan \(x = y\)
proof
  have \(u = v\) if \(u: -\pi/2 < u < \pi/2\) and \(v: -\pi/2 < v < \pi/2\)
  and eq: tan \(u = tan v\) for \(u\) \(v\)
  proof (cases \(u\) \(v\) rule: linorder-cases)
  case less
  have \(\forall\ x. u \leq x \land x \leq v\) \(\implies\) isCont \(x\)
  by (metis cos-gt-zero-pi isCont-tan le-less-trans less-irrefl less-trans u v(1)
  then have continuous-on \(\{u..v\}\) tan 
  by (simp add: continuous-at-imp-continuous-on)
  moreover have \(\forall\ x. u < x \land x < v\) \(\implies\) tan differentiable \((at x)\)
  by (metis DERIV-tan cos-gt-zero-pi real-differentiable-def less-numeral-extra(3)
  order.strict-trans u(1) v(2))
  ultimately obtain \(z\) where \(u \leq z < v\) \(\implies\) DERIV tan \(z :> 0\)
  by (metis less Rolle eq)
  moreover have \(cos z \neq 0\)
  by (metis (no-types) \(u :< z\) \(z :< v\) cos-gt-zero-pi less-le-trans linorder-not-less
  not-less-iff-gr-or-eq u(1) v(2))
  ultimately show \(\exists\)thesis
  using DERIV-unique [OF - DERIV-tan] by fastforce
next
  case greater
  have \(\forall\ x. v \leq x \land x \leq u\) \(\implies\) isCont \(x\)
  by (metis cos-gt-zero-pi isCont-tan le-less-trans less-irrefl less-trans u(2)
  v(1))
  then have continuous-on \(\{v..u\}\) tan 
  by (simp add: continuous-at-imp-continuous-on)
  moreover have \(\forall\ x. v < x \land x < u\) \(\implies\) tan differentiable \((at x)\)
  by (metis DERIV-tan cos-gt-zero-pi real-differentiable-def less-numeral-extra(3)
  order.strict-trans u(2) v(1))
  ultimately obtain \(z\) where \(v \leq z < u\) \(\implies\) DERIV tan \(z :> 0\)
  by (metis greater Rolle eq)
  moreover have \(cos z \neq 0\)
  by (metis \(v :< z\) \(z :< u\) cos-gt-zero-pi less-eq-real-def less-le-trans order-less-irrefl u(2) v(1))
  ultimately show \(\exists\)thesis
  using DERIV-unique [OF - DERIV-tan] by fastforce
qed auto
then have \( \exists! x. -\frac{\pi}{2} < x < \frac{\pi}{2} \land \tan x = y \)
if \( x - \frac{\pi}{2} < x < \frac{\pi}{2} \tan x = y \) for \( x \)
using that by auto
then show \(?thesis \)
using lemma-tan-total1 [where \( y = y \)]
by auto
qed

lemma \texttt{tan-monotone'}:
assumes \( -\frac{\pi}{2} < y \) and \( y < x \) and \( x < \frac{\pi}{2} \)
shows \( y < x \) \( \iff \) \( \tan y < \tan x \)
proof
\( \begin{array}{l}
\text{assume } y < x \\
\text{then show } \tan y < \tan x \\
\quad \text{using } \texttt{tan-monotone} \texttt{ and } (-\frac{\pi}{2} < y \text{ and } x < \frac{\pi}{2}) \text{ by auto} \\
\end{array} \)
next
\( \begin{array}{l}
\text{assume } \tan y < \tan x \\
\text{show } y < x \\
\quad \text{proof } (\text{rule } \texttt{ccontr}) \\
\end{array} \)
assume \(\neg \text{thesis}\)
then have \(x \leq y\) by auto
then have \(\tan x \leq \tan y\)
proof (cases \(x = y\))
  case True
  then show \(\text{thesis}\) by auto
next
case False
then have \(x < y\) using \(x \leq y\) by auto
from \(\tan\text{-monotone}[\OF \langle -(\pi/2) < x\rangle \langle y < \pi/2\rangle]\) show \(\text{thesis}\)
  by auto
qed	hen then show \(\text{thesis}\) by auto
qed

lemma \(\tan\text{-inverse}\): \(1 / (\tan y) = \tan (\pi/2 - y)\)
unfolding \(\tan\text{-def} \sin\text{-cos-eq}[\of y] \cos\text{-sin-eq}[\of y]\) by auto

lemma \(\tan\text{-periodic-pi}[simp]\): \(\tan (x + \pi) = \tan x\)
by (simp add: \(\tan\text{-def}\))

proof (induct \(n\) arbitrary: \(x\))
case 0
then show \(\text{thesis}\) by simp
next
case (Suc \(n\))
  have \(\split\text{-pi-off}\): \(x + \text{real} \text{(Suc} \text{\(n\}) \ast \pi = (x + \text{real} \text{\(n\}) \ast \pi) + \pi\)
    unfolding Suc-eq-plus1 of-nat-add distrib-right by auto
  show \(\text{thesis}\)
    unfolding \(\split\text{-pi-off}\) using Suc by auto
qed

lemma \(\tan\text{-periodic-int}[simp]\): \(\tan (x + \text{of-int} \text{\(i\}) \ast \pi) = \tan x\)
proof (cases \(0 \leq i\))
case False
then have \(i\text{-nat}: \text{of-int} \text{\(i\}) = - \text{of-int} \text{\((nat (\neg i))\)}\) by auto
then show \(\text{thesis}\)
  by (smt (verit, best) mul-minus-left of-int-of-nat-eq tan-periodic-nat)
qed (use zero-le-imp-eq-int in fastforce)

lemma \(\tan\text{-periodic-n}[simp]\): \(\tan (x + \text{numeral} \text{\(n\}) \ast \pi) = \tan x\)
using \(\tan\text{-periodic-int}[of - numeral \text{\(n\}]\) by simp

lemma \(\tan\text{-minus-45}[simp]\): \(\tan \langle -(\pi/4)\rangle = -1\)
unfolding \(\tan\text{-def}\) by (simp add: sin-45 cos-45)
lemma tan-diff:
\[ \cos x \neq 0 \Rightarrow \cos y \neq 0 \Rightarrow \cos(x - y) \neq 0 \Rightarrow \tan(x - y) = (\tan x - \tan y)/(1 + \tan x \cdot \tan y) \]
for \( x :: 'a::{real-normed-field,banach} \)
using \( \text{tan-add [of } x - y \text{]} \) by simp

lemma tan-pos-pi2-le:
\[ 0 \leq x \Rightarrow x < \pi/2 \Rightarrow 0 \leq \tan x \]
using \( \text{less-eq-real-def tan-gt-zero} \) by auto

lemma cos-tan:
\[ |x| < \pi/2 \Rightarrow \cos x = 1/\sqrt{(1 + \tan x)^2} \]
using \( \text{cos-gt-zero [of } x \text{]} \) \( \text{cos-gt-zero [of } -x \text{]} \)
by (auto simp add: \text{field-split-simps tan-def real-sqrt-divide abs-if split: if-split-asm})

lemma sin-tan:
\[ |x| < \pi/2 \Rightarrow \sin x = \tan x/\sqrt{(1 + \tan x)^2} \]
using \( \text{cos-gt-zero [of } x \text{]} \) \( \text{cos-gt-zero [of } -x \text{]} \)
by (auto simp add: \text{field-split-simps tan-def real-sqrt-divide abs-if split: if-split-asm})

lemma tan-mono-le:
\[ -(\pi/2) < x \Rightarrow x < \pi/2 \Rightarrow -(\pi/2) < y \Rightarrow y < \pi/2 \Rightarrow \tan x < \tan y \]
\( \iff \)
\[ x < y \]
using \( \text{tan-monotone}' \) by blast

lemma tan-mono-le-eq:
\[ -(\pi/2) < x \Rightarrow x < \pi/2 \Rightarrow -(\pi/2) < y \Rightarrow y < \pi/2 \Rightarrow \tan x \leq \tan y \]
\( \iff \)
\[ x \leq y \]
by (meson \( \text{tan-mono-le not-le tan-monotone} \))

lemma tan-bound-pi2: \( |x| < \pi/4 \Rightarrow |\tan x| < 1 \)
using \( \text{tan-45 tan-monotone [of } x \pi/4 \text{]} \) \( \text{tan-monotone [of } -x \pi/4 \text{]} \)
by (auto simp: \( \text{abs-if split: if-split-asm} \))

lemma tan-cot: \( \tan((\pi/2) - x) = \text{inverse}(\tan x) \)
by (simp add: \( \text{tan-def sin-diff cos-diff} \))

113.16 Cotangent
definition cot :: 'a \Rightarrow 'a::{real-normed-field,banach}
where \( \text{cot} = (\lambda x. \cos x / \sin x) \)
lemma cot-of-real: of-real (cot x) = (cot (of-real x) :: 'a::{real-normed-field,banach})
  by (simp add: cot-def sin-of-real cos-of-real)

lemma cot-in-Reals [simp]: z ∈ ℝ ⇒ cot z ∈ ℝ
  for z :: 'a::{real-normed-field,banach}
  by (simp add: cot-def)

lemma cot-zero [simp]: cot 0 = 0
  by (simp add: cot-def)

lemma cot-pi [simp]: cot π = 0
  by (simp add: cot-def)

lemma cot-npi [simp]: cot (real n * π) = 0
  for n :: nat
  by (simp add: cot-def)

lemma cot-minus [simp]: cot (−x) = −cot x
  by (simp add: cot-def)

lemma cot-periodic [simp]: cot (x + 2 * π) = cot x
  by (simp add: cot-def)

lemma cot-altdef: cot x = inverse (tan x)
  by (simp add: cot-def tan-def)

lemma tan-altdef: tan x = inverse (cot x)
  by (simp add: cot-def tan-def)

lemma tan-cot': tan (π/2 − x) = cot x
  by (simp add: tan-cot cot-altdef)

lemma cot-gt-zero: 0 < x ⇒ x < π/2 ⇒ 0 < cot x
  by (simp add: cot-def zero-less-divide-iff sin-gt-zero2 cos-gt-zero-pi)

lemma cot-less-zero:
  assumes lb: − π/2 < x and x < 0
  shows cot x < 0
  by (smt (verit) assms cot-gt-zero cot-minus divide-minus-left)

lemma DERIV-cot [simp]: sin x ≠ 0 ⇒ DERIV cot x := −inverse ((sin x)^2)
  for x :: 'a::{real-normed-field,banach}
  unfolding cot-def using cos-squared-eq[of x]
  by (auto intro: derivative-eq-intros (simp add: divide-inverse power2_eq_square))

lemma isCont-cot: sin x ≠ 0 ⇒ isCont cot x
  for x :: 'a::{real-normed-field,banach}
  by (rule DERIV-cot [THEN DERIV-isCont])
lemma isCont-cot' [simp, continuous-intros]:
  \( \text{isCont } f \ a \rightarrow \sin (f \ a) \neq 0 \rightarrow \text{isCont } (\lambda x. \cot (f \ x)) \ a \)
for \( a :: 'a::\{\text{real-normed-field, banach}\} \) and \( f :: 'a \Rightarrow 'a \)
  by (rule isCont-o2 [OF isCont-cot])

lemma tendsto-cot [tendsto-intros]: \( (\rightarrow a) \ F \rightarrow \sin a \neq 0 \rightarrow ((\lambda x. \cot (f \ x)) \rightarrow \cot a) \ F \)
for \( f :: 'a \Rightarrow 'a::\{\text{real-normed-field, banach}\} \)
  by (rule isCont-tendsto-compose [OF isCont-cot])

lemma continuous-cot:
  \( \text{continuous } F \ f \Rightarrow \sin (f (\text{Lim } F (\lambda x. x))) \neq 0 \Rightarrow \text{continuous } F (\lambda x. \cot (f \ x)) \)
for \( f :: 'a \Rightarrow 'a::\{\text{real-normed-field, banach}\} \)
  unfolding continuous-def by (rule tendsto-cot)

lemma continuous-on-cot [continuous-intros]:
  fixes \( f :: 'a \Rightarrow 'a::\{\text{real-normed-field, banach}\} \)
  shows \( \text{continuous-on } s \ f \Rightarrow (\forall x \in s. \sin (f \ x) \neq 0) \Rightarrow \text{continuous-on } s (\lambda x. \cot (f \ x)) \)
  unfolding continuous-on-def by (auto intro: tendsto-cot)

lemma continuous-within-cot [continuous-intros]:
  fixes \( f :: 'a \Rightarrow 'a::\{\text{real-normed-field, banach}\} \)
  shows \( \text{continuous } (\text{at } x \text{ within } s) \ f \Rightarrow \sin (f \ x) \neq 0 \Rightarrow \text{continuous } (\text{at } x \text{ within } s) (\lambda x. \cot (f \ x)) \)
  unfolding continuous-within-def by (rule tendsto-cot)

113.17 Inverse Trigonometric Functions

definition arcsin :: real \Rightarrow real
  where \( \text{arcsin } y = (\text{THE } x. -(\pi/2) \leq x \land x \leq \pi/2 \land \sin x = y) \)

definition arccos :: real \Rightarrow real
  where \( \text{arccos } y = (\text{THE } x. 0 \leq x \land x \leq \pi \land \cos x = y) \)

definition arctan :: real \Rightarrow real
  where \( \text{arctan } y = (\text{THE } x. -(\pi/2) < x \land x < \pi/2 \land \tan x = y) \)

lemma arcsin: \(-1 \leq y \Rightarrow y \leq 1 \Rightarrow -(\pi/2) \leq \text{arcsin } y \land \text{arcsin } y \leq \pi/2 \land \sin (\text{arcsin } y) = y \)
  unfolding arcsin-def by (rule the1' [OF sin-total])

lemma arcsin-pi: \(-1 \leq y \Rightarrow y \leq 1 \Rightarrow -(\pi/2) \leq \text{arcsin } y \land \text{arcsin } y \leq \pi \land \sin (\text{arcsin } y) = y \)
  by (drule (1) arcsin) (force intro: order-trans)

lemma sin-arcsin [simp]: \(-1 \leq y \Rightarrow y \leq 1 \Rightarrow \sin (\text{arcsin } y) = y \)
  by (blast dest: arcsin)
lemma \textit{arcsin-bounded}: \(-1 \leq y \Rightarrow y \leq 1 \Rightarrow - (\pi/2) \leq \arcsin y \land \arcsin y \leq \pi/2 \)
by (blast dest: arcsin)

lemma \textit{arcsin-bound}: \(-1 \leq y \Rightarrow y \leq 1 \Rightarrow - (\pi/2) \leq \arcsin y \)
by (blast dest: arcsin)

lemma \textit{arcsin-ubound}: \(-1 \leq y \Rightarrow y \leq 1 \Rightarrow \arcsin y \leq \pi/2 \)
by (blast dest: arcsin)

lemma \textit{arcsin-lt-bounded}:
assumes \(-1 < y \land y < 1 \)
shows \(- (\pi/2) < \arcsin y \land \arcsin y < \pi/2 \)
proof
- have \(\arcsin y \neq \pi/2 \)
  by (metis arcsin assms not-less not-less-iff-gr-or-eq sin-pi-half)
moreover have \(\arcsin y \neq - \pi/2 \)
  by (metis arcsin assms minus-divide-left not-less not-less-iff-gr-or-eq sin_minus sin-pi-half)
ultimately show \(?thesis \)
  using arcsin-bounded [of y] assms by auto
qed

lemma \textit{arcsin-sin}:
\(- (\pi/2) \leq x \Rightarrow x \leq \pi/2 \Rightarrow \arcsin (\sin x) = x \)
unfolding arcsin-def
using \textit{the1-equality} [OF sin-total] by simp

lemma \textit{arcsin-unique}:
assumes \(-\pi/2 \leq x \land x \leq \pi/2 \land \sin x = y \)
shows \(\arcsin y = x \)
using arcsin-sin[of x] assms by force

lemma \textit{arcsin-0} [simp]; \(\arcsin 0 = 0 \)
using arcsin-sin [of 0] by simp

lemma \textit{arcsin-1} [simp]; \(\arcsin 1 = \pi/2 \)
using arcsin-sin [of \pi/2] by simp

lemma \textit{arcsin-minus-1} [simp]; \(\arcsin (-1) = - (\pi/2) \)
using arcsin-sin [of - \pi/2] by simp

lemma \textit{arcsin-minus}:
\(-1 \leq x \Rightarrow x \leq 1 \Rightarrow \arcsin (-x) = - \arcsin x \)
by (metis (no-types, opaque-lifting) arcsin arcsin-sin minus-minus neg-le-iff-le sin-minus)

lemma \textit{arcsin-one-half} [simp]; \(\arcsin (1/2) = \pi / 6 \)
and \(\arcsin-minus-one-half [simp]; \arcsin (- (1/2)) = -\pi / 6 \)
by (intro arcsin-unique; simp add: sin-30 field-simps)+
THEORY “Transcendental” 2445

lemma arcsin-one-over-sqrt-2: arcsin (1 / sqrt 2) = pi / 4
  by (rule arcsin-unique) (auto simp: sin-45 field-simps)

lemma arcsin-eq-iff: |x| ≤ 1 ⇒ |y| ≤ 1 ⇒ arcsin x = arcsin y ←→ x = y
  by (metis abs-le-iff arcsin minus-le-iff)

lemma cos-arcsin-nonzero: −1 < x ⇒ x < 1 ⇒ cos (arcsin x) ≠ 0
  using arcsin-lt-bounded cos-gt-zero-pi by force

lemma arccos-lbound: −1 ≤ y ⇒ y ≤ 1 ⇒ 0 ≤ arccos y ∧ arccos y ≤ pi
  by (blast dest: arccos)

lemma arccos-ubound: −1 ≤ y ⇒ y ≤ 1 ⇒ arccos y ≤ pi
  by (blast dest: arccos)

lemma arccos-lt-bounded:
  assumes −1 < y y < 1
  shows 0 < arccos y ∧ arccos y < pi
proof –
  have arccos y ≠ 0
    by (metis (no-types) arccos assms 1 assms 2 cos-zero less-eq-real-def less-irrefl)
  moreover have arccos y ≠ −pi
    by (metis arccos assms 1 assms 2 cos-minus cos-pi not-less not-less-iff-gr-or-eq)
  ultimately show ?thesis
    using arccos-lt-bounded [of y] assms
    by (metis arccos cos-pi not-less-not-less-iff-gr-or-eq)
qed

lemma arccos-cos: 0 ≤ x ⇒ x ≤ pi ⇒ arccos (cos x) = x
  by (auto simp: arccos-def intro!: the1-equality cos-total)

lemma arccos-cos2: x ≤ 0 ⇒ −pi ≤ x ⇒ arccos (cos x) = −x
  by (auto simp: arccos-def intro!: the1-equality cos-total)

lemma arccos-unique:
  assumes 0 ≤ x and x ≤ pi and cos x = y
  shows arccos y = x
  using arccos-cos assms by blast

lemma cos-arcsin:
assumes $-1 \leq x \leq 1$
shows $\cos (\arcsin x) = \sqrt{1 - x^2}$

proof (rule power2_eq_imp_eq)
  show $(\cos (\arcsin x))^2 = (\sqrt{1 - x^2})^2$
    by (simp add: square_le_1 assms cos_squared_eq)
  show $0 \leq \cos (\arcsin x)$
    using arcsin assms cos_ge_zero by blast
  show $0 \leq \sqrt{1 - x^2}$
    by (simp add: square_le_1 assms)
qed

lemma sin-arccos:
  assumes $-1 \leq x \leq 1$
  shows $\sin (\arccos x) = \sqrt{1 - x^2}$

proof (rule power2_eq_imp_eq)
  show $(\sin (\arccos x))^2 = (\sqrt{1 - x^2})^2$
    by (simp add: square_le_1 assms sin_squared_eq)
  show $0 \leq \sin (\arccos x)$
    by (simp add: arccos_bounded assms sin_ge_zero)
  show $0 \leq \sqrt{1 - x^2}$
    by (simp add: square_le_1 assms)
qed

lemma arccos-0 [simp]: $\arccos 0 = \frac{\pi}{2}$
  using arccos-cos pi_half_ge_zero by fastforce

lemma arccos-1 [simp]: $\arccos 1 = 0$
  using arccos-cos by force

lemma arccos-minus-1 [simp]: $\arccos (-1) = \pi$
  by (metis arccos-cos arccos_minus order_refl pi_ge_zero)

lemma arccos-minus: $-1 \leq x \Rightarrow x \leq 1 \Rightarrow \arccos (-x) = \pi - \arccos x$
  by (smt (verit, ccfthr))

lemma arccos-one-half [simp]: $\arccos (1/2) = \frac{\pi}{3}$
  and arccos-minus-one-half [simp]: $\arccos (-1/2) = 2 * \pi / 3$
  by (intro arccos-unique; simp add: cos_60 cos_120)

lemma arccos-one-over-sqrt-2: $\arccos (1/\sqrt{2}) = \frac{\pi}{4}$
  by (rule arccos-unique) (auto simp: cos_45 field_simps)

corollary arccos-minus-abs:
  assumes $|x| \leq 1$
  shows $\arccos (-x) = \pi - \arccos x$
  using assms by (simp add: arccos_minus)

lemma sin-arccos-nonzero: $-1 < x \Rightarrow x < 1 \Rightarrow \sin (\arccos x) \neq 0$
  using arccos_lt_bounded sin_gt_zero by force
THEORY "Transcendental"

lemma arctan: \((\pi/2) < \arctan y \land \arctan y < \pi/2 \land \tan (\arctan y) = y\)
unfolding arctan-def by (rule theI'[OF tan-total])

lemma tan-arctan: \(\tan (\arctan y) = y\)
by (simp add: arctan)

lemma arctan-bounded: \(-(\pi/2) < \arctan y \land \arctan y < \pi/2\)
by (auto simp only: arctan)

lemma arctan-lbound: \(-(\pi/2) < \arctan y\)
by (simp add: arctan)

lemma arctan-ubound: \(\arctan y < \pi/2\)
by (auto simp only: arctan)

lemma arctan-unique: assumes \(-(\pi/2) < x\)
and \(x < \pi/2\)
and \(\tan x = y\)
shows \(\arctan y = x\)
using assms arctan[of y] tan-total[of y] by (fast elim: ex1E)

lemma arctan-tan: \(-\pi/2 < x \Rightarrow x < \pi/2 = \Rightarrow \arctan (\tan x) = x\)
by (rule arctan-unique simp-all)

lemma arctan-zero-zero [simp]: \(\arctan 0 = 0\)
by (rule arctan-unique simp-all)

lemma arctan-minus: \(\arctan (-x) = -\arctan x\)
using arctan[of x] by (auto simp: arctan-unique)

lemma cos-arctan-not-zero [simp]: \(\cos (\arctan x) \neq 0\)
by (intro less-imp-neq[symmetric] cos-gt-zero-pi arctan-lbound arctan-ubound)

lemma tan-eq-arctan-Ex: shows \(\tan x = y \iff (\exists k\cdot \int. x = \arctan y + k\pi \lor (x = \pi/2 + k\pi \land y=0))\)
proof
assume lhs: \(\tan x = y\)
obtain \(k\cdot \int. k\cdot (-\pi/2 < x - k\pi x - k\pi \leq \pi/2)\)
proof
define \(k\) where \(k = \text{ceiling}(x/\pi - 1/2)\)
show \(-\pi/2 < x - \text{real-of-int} k \cdot \pi\)
using ceiling-divide-lower[of \(
pi * 2 \cdot (x * 2 - \pi)\)] by (auto simp: k-def field-simps)
show \(x - k\pi \leq \pi/2\)
using ceiling-divide-upper[of \(
pi * 2 \cdot (x * 2 - \pi)\)] by (auto simp: k-def field-simps)
THEORY “Transcendental” 2448

qed
have \( x = \arctan y + \text{of-int } k \ast \pi \) when \( x \neq \pi/2 + k\pi \)
proof
  have \( \tan (x - k \ast \pi) = y \) using \( \text{lhs tan-periodic-int[of -} -k] \) by auto
  then have \( \arctan y = x - \text{real-of-int } k \ast \pi \)
  by (smt (verit) \( \text{arctan-tan lhs divide-minus-left k mult-minus-left of-int-minus tan-periodic-int that} \))
  then show \( \text{?thesis} \) by auto
qed
then show \( \exists k. \ x = \arctan y + \text{of-int } k \ast \pi \lor (x = \pi/2 + k\pi \land y=0) \)
  using \( \text{lhs k} \) by force
qed (auto simp: \( \text{arctan} \))

lemma arctan-tan-eq-abs-pi:
  assumes \( \cos \vartheta \neq 0 \)
  obtains \( k \) where \( \arctan (\tan \vartheta) = \vartheta - \text{of-int } k \ast \pi \)
  by (metis \( \text{add.commute assms cos-zero-iff-int2 eq-diff-eq tan-eq-arctan-Ex} \))

lemma tan-eq:
  assumes \( \tan x = \tan y \tan x \neq 0 \)
  obtains \( k :: \text{int} \) where \( x = y + k \ast \pi \)
proof
  obtain \( k0 \) where \( k0\): \( x = \arctan (\tan y) + \text{real-of-int } k0 \ast \pi \)
  using \( \text{assms tan-eq-arctan-Ex[of x tan y]} \) by auto
  obtain \( k1 \) where \( k1\): \( \arctan (\tan y) = y - \text{of-int } k1 \ast \pi \)
  using \( \text{arctan-tan-eq-abs-pi assms tan-eq-0-cos-sin} \) by auto
  have \( x = y + (k0-k1)\ast\pi \)
  using \( k0 k1 \) by (auto simp: \( \text{algebra-simps} \))
  with that show \( \text{?thesis} \)
  by blast
qed

lemma cos-arctan: \( \cos (\arctan x) = 1 / \sqrt{1 + x^2} \)
proof (rule power2-eq-imp-eq)
  have \( 0 < 1 + x^2 \) by (simp add: add-pos-nonneg)
  show \( 0 \leq 1 / \sqrt{1 + x^2} \) by simp
  show \( 0 \leq \cos (\arctan x) \)
  by (intro less-imp-le cos-gt-zero-pi arctan-lbound arctan-ubound)
  have \( (\cos (\arctan x))^2 \ast (1 + (\tan (\arctan x))^2) = 1 \)
  unfolding \( \text{tan-def simp add: distrib-left power-divide} \)
  then show \( (\cos (\arctan x))^2 = (1 / \sqrt{1 + x^2})^2 \)
  using \( 0 < 1 + x^2 \) by (simp add: \( \text{arctan power-divide eq-divide-eq} \))
qed

lemma sin-arctan: \( \sin (\arctan x) = x / \sqrt{1 + x^2} \)
using add-pos-nonneg [OF \( \text{zero-less-one zero-le-power2 \ of x} \)]
using tan-arctan [of x] unfolding \( \text{tan-def cos-arctan} \)
by (simp add: eq-divide-eq)
lemma \( \tan-sec \): \( \cos x \neq 0 \implies 1 + (\tan x)^2 = (\text{inverse} (\cos x))^2 \)
for \( x : \{ \text{real-normed-field}, \text{banach_field} \} \)
by \((\text{simp add: add-divide-eq-iff inverse-eq-divide power2-eq-square tan-def})\)

lemma \( \text{arctan-less-iff} \): \( \arctan x < \arctan y \iff x < y \)
by \((\text{metis tan-monotone' arctan-lbound arctan-ubound tan-arctan})\)

lemma \( \text{arctan-le-iff} \): \( \arctan x \leq \arctan y \iff x \leq y \)
by \((\text{simp only: not-less [symmetric] arctan-less-iff})\)

lemma \( \text{arctan-eq-iff} \): \( \arctan x = \arctan y \iff x = y \)
by \((\text{simp only: eq-iff [where 'a=real] arctan-le-iff})\)

lemma \( \text{zero-less-arctan-iff} \) \([\text{simp}]: 0 < \arctan x \iff 0 < x \)
using \( \text{arctan-less-iff [of 0 x]} \) by simp

lemma \( \text{arctan-less-zero-iff} \) \([\text{simp}]: \arctan x < 0 \iff x < 0 \)
using \( \text{arctan-less-iff [of x 0]} \) by simp

lemma \( \text{zero-le-arctan-iff} \) \([\text{simp}]: 0 \leq \arctan x \iff 0 \leq x \)
using \( \text{arctan-le-iff [of 0 x]} \) by simp

lemma \( \text{arctan-le-zero-iff} \) \([\text{simp}]: \arctan x \leq 0 \iff x \leq 0 \)
using \( \text{arctan-le-iff [of x 0]} \) by simp

lemma \( \text{arctan-eq-zero-iff} \) \([\text{simp}]: \arctan x = 0 \iff x = 0 \)
using \( \text{arctan-eq-iff [of x 0]} \) by simp

lemma \( \text{continuous-on-arcsin'} \): \( \text{continuous-on} \{-1..1\} \arcsin \)
proof –
have \( \text{continuous-on} (\sin' \{- \pi/2 .. \pi/2\}) \arcsin \)
by \((\text{rule continuous-on-inv}) (\text{auto intro: continuous-intros simp: arcsin-sin})\)
also have \( \sin' \{- \pi/2 .. \pi/2\} = \{-1 .. 1\} \)
proof safe
fix \( x :: \text{real} \)
assume \( x \in \{-1..1\} \)
then show \( x \in \sin' \{- \pi/2..\pi/2\} \)
using \( \arcsin-lbound \arcsin-ubound \)
by \((\text{intro image-eqI [where x=arcsin x]} \) \text{auto})\)
qed simp
finally show \(?thesis\).
qed

lemma \( \text{continuous-on-arcsin} \) \([\text{continuous-intros}]: \)
\( \text{continuous-on} f \implies (\forall x \in s. -1 \leq f x \land f x \leq 1) \implies \text{continuous-on} s (\lambda x. \arcsin (f x)) \)
using \( \text{continuous-on-compose[of s f, OF - continuous-on-subset[OF continuous-on-arcsin']]} \)
by \((\text{auto simp: comp-def subset-eq})\)
lemma isCont-arcsin: \(-1 < x \implies x < 1 \implies \text{isCont arcsin } x\)
using continuous-on-arcsin[TpHEN continuous-on-subset, of \([-1 <..< 1 \}]]
by (auto simp: continuous-on-eq-continuous-at subset-eq)

lemma continuous-on-arccos': continuous-on \([-1 .. 1\} arccos
proof –
  have continuous-on \((\cos' {0 .. pi}) \) arccos
    by (rule continuous-on-inv) (auto intro: continuous-intros simp: arccos-cos)
  also have \(\cos' {0 .. pi} = \{-1 .. 1\}\)
    proof safe
      fix x :: real
      assume x: x \in \{-1 .. 1\}
      then show x \in \cos' {0 .. pi}
        using arccos-lbound arccos-ubound
        by (intro image-eqI [where x=arccos x]) auto
    qed
  finally show \(?thesis\). qed

lemma continuous-on-arccos [continuous-intros]:
  continuous-on s f \implies (\forall x \in s. -1 \leq f x \land f x \leq 1) \implies \text{continuous-on } s \(\lambda x. \text{arccos } f x)\)
using continuous-on-compose[of s f, OF - continuous-on-subset[OF continuous-on-arccos']]
by (auto simp: comp-def subset-eq)

lemma isCont-arccos: \(-1 < x \implies x < 1 \implies \text{isCont } \text{arccos } x\)
using continuous-on-arccos[TpHEN continuous-on-subset, of \{-1 <..< 1 \}]]
by (auto simp: continuous-on-eq-continuous-at subset-eq)

lemma isCont-arctan: isCont arctan x
proof –
  obtain u where u: \(- (\pi/2 < u u < \text{arctan } x\)
    by (meson arctan arctan-less-iff linordered-field-no-lb)
  obtain v where v: \text{arctan } x < v v < \pi/2
    by (meson arctan-less-iff arctan-ubound linordered-field-no-ub)
  have isCont arctan \((\text{tan } \text{arctan } x)\)
    proof (rule isCont-inverse-function2 [of u arctan x v])
      show \(\forall z. [u \leq z; z \leq v] \implies \text{arctan } (\text{tan } z) = z\)
        using arctan-unqie u(1) v(2) by auto
      then show \(\forall z. [u \leq z; z \leq v] \implies \text{isCont tan } z\)
        by (metis arctan cos-gt-zero-pi isCont-tan less-irrefl)
    qed (use u v in auto)
  then show \(?thesis\). by (simp add: arctan)
qed

lemma tendsto-arctan [tendsto-intros]: \((\longrightarrow x) F \implies ((\lambda x. \text{arctan } f x) \longrightarrow)
arctan x) F
by (rule isCont-tendsto-compose [OF isCont-arctan])

lemma continuous-arctan [continuous-intros]: continuous F f \implies continuous F
(\lambda x. arctan (f x))
unfolding continuous-def by (rule tendsto-arctan)

lemma continuous-on-arctan [continuous-intros]:
continuous-on s f \implies continuous-on s (\lambda x. arctan (f x))
unfolding continuous-on-def by (auto intro: tendsto-arctan)

lemma DERIV-arcsin:
assumes -1 < x x < 1
shows DERIV arcsin x :\to inverse (sqrt (1 - x^2))
proof (rule DERIV-inverse-function)
show (sin has-real-derivative sqrt (1 - x^2)) (at (arcsin x))
  by (rule derivative-eq-intros | use assms cos-arcsin in force)+
show sqrt (1 - x^2) \neq 0
  using abs-square-eq-1 assms by force
qed (use assms isCont-arcsin in auto)

lemma DERIV-arccos:
assumes -1 < x x < 1
shows DERIV arccos x :\to inverse (-sqrt (1 - x^2))
proof (rule DERIV-inverse-function)
show (cos has-real-derivative -sqrt (1 - x^2)) (at (arccos x))
  by (rule derivative-eq-intros | use assms sin-arccos in force)+
show -sqrt (1 - x^2) \neq 0
  using abs-square-eq-1 assms by force
qed (use assms isCont-arccos in auto)

lemma DERIV-arctan:
DERIV arctan x :\to inverse (1 + x^2)
proof (rule DERIV-inverse-function)
have inverse ((cos (arctan x))^2) = 1 + x^2
  by (metis arctan cos-arctan-not-zero power-inverse tan-sec)
then show (tan has-real-derivative 1 + x^2) (at (arctan x))
  by (auto intro: derivative-eq-intros)
show \(\forall y. [x - 1 < y; y < x + 1] \imp tan (arctan y) = y\)
  using tan-arctan by blast
show 1 + x^2 \neq 0
  by (metis power-one sum-power2-eq-zero-iff zero-neq-one)
qed (use isCont-arctan in auto)

declar
DERIV-arcsin[THEN DERIV-chain2, derivative-intros]
DERIV-arcsin[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]
DERIV-arccos[THEN DERIV-chain2, derivative-intros]
DERIV-arccos[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]
 Theo-
lemma cos-multiple-reduce:
  \( \cos (x \cdot \text{numeral } n) = \cos (x \cdot \text{of-nat } (\text{pred-numeral } n)) \cdot \cos x - \sin (x \cdot \text{of-nat } (\text{pred-numeral } n)) \cdot \sin x \)
proof
  have \( \text{numeral } n = \text{of-nat } (\text{pred-numeral } n) + 1 \)
    by (metis \(\text{add.commute numeral-eq-Suc of-nat-Suc of-nat-numeral}\))
  also have \( \cos (x \cdot \ldots) = \cos (x \cdot \text{of-nat } (\text{pred-numeral } n)) + x \)
    unfolding \(\text{of-nat-Suc}\) by (simp add: \(\text{ring-distrib}\))
  finally show \(\text{thesis}\)
    by (simp add: \(\text{cos-add}\))
qed

lemma arccos-eq-pi-iff: \(x \in \{-1\ldots1\} \implies \arccos x = \pi \iff x = -1\)
proof
  have \(\text{arccos} - \text{pi-iff}\) by (metis \(\text{arccos - pi}\))
lemma arccos-eq-0-iff: \(x \in \{-1\ldots1\} \implies \arccos x = 0 \iff x = 1\)
proof
  have \(\text{arccos} - \text{0-iff}\) by (metis \(\text{arccos - 0}\))

113.18 Prove Totality of the Trigonometric Functions

lemma cos-arccos-abs: \(|y| \leq 1 \implies \cos (\arccos y) = y\)
proof
  have \(\text{abs-le-iff}\) by (simp add: \(\text{abs-le-iff}\))
lemma sin-arccos-abs: \(|y| \leq 1 \implies \sin (\arccos y) = \sqrt{(1 - y^2)}\)
proof
  have \(\text{sin-arccos abs-le-iff}\) by (simp add: \(\text{sin-arccos abs-le-iff}\))
lemma sin-mono-less-eq:
  \(- (\pi/2) \leq x \implies x \leq \pi/2 \implies - (\pi/2) \leq y \implies y \leq \pi/2 \implies \sin x < \sin y\)
proof
  have \(\text{not-less-iff-gr-or-eq sin-monotone-2pi}\) by (metis \(\text{not-less-iff-gr-or-eq sin-monotone-2pi}\))
lemma sin-mono-le-eq:
  \(- (\pi/2) \leq x \implies x \leq \pi/2 \implies - (\pi/2) \leq y \implies y \leq \pi/2 \implies \sin x \leq \sin y\)
proof
  have \(\text{leD le-less-linear sin-monotone-2pi sin-monotone-2pi-le}\) by (meson \(\text{leD le-less-linear sin-monotone-2pi sin-monotone-2pi-le}\))
lemma sin-inj-pi:
  \(- (\pi/2) \leq x \implies x \leq \pi/2 \implies - (\pi/2) \leq y \implies y \leq \pi/2 \implies \sin x = \sin y\)
THEORY “Transcendental”

\[ x = y \]
by (metis arcsin-sin)

lemma arcsin-le-iff:
  assumes \( x \geq -1 \leq 1 \) \( y \geq -\pi/2 \leq \pi/2 \)
  shows \( \text{arcsin } x \leq y \iff x \leq \sin y \)
proof
  have \( \text{arcsin } x \leq y \iff \sin (\text{arcsin } x) \leq \sin y \)
    using arcsin-bounded[of x] assms by (subst sin-mono-le-eq) auto
  also from assms have \( \sin (\text{arcsin } x) = x \) by simp
  finally show \( \text{thesis} \).
qed

lemma le-arcsin-iff:
  assumes \( x \geq -1 \leq 1 \) \( y \geq -\pi/2 \leq \pi/2 \)
  shows \( \text{arcsin } x \geq y \iff x \geq \sin y \)
proof
  have \( \text{arcsin } x \geq y \iff \sin (\text{arcsin } x) \geq \sin y \)
    using arcsin-bounded[of x] assms by (subst sin-mono-le-eq) auto
  also from assms have \( \sin (\text{arcsin } x) = x \) by simp
  finally show \( \text{thesis} \).
qed

lemma cos-mono-less-eq:
  \( 0 \leq x \Rightarrow x \leq \pi \Rightarrow 0 \leq y \Rightarrow y \leq \pi \Rightarrow \cos x < \cos y \iff y < x \)
by (meson cos-monotone-0-pi cos-monotone-0-pi-le leD le-less-linear)

lemma cos-mono-le-eq:
  \( 0 \leq x \Rightarrow x \leq \pi \Rightarrow 0 \leq y \Rightarrow y \leq \pi \Rightarrow \cos x \leq \cos y \iff y \leq x \)
by (metis arccos-cos cos-monotone-0-pi-le eq-iff linear)

lemma cos-inj-pi:
  \( 0 \leq x \Rightarrow x \leq \pi \Rightarrow 0 \leq y \Rightarrow y \leq \pi \Rightarrow \cos x = \cos y \Rightarrow x = y \)
by (metis arccos-cos)

lemma arccos-le-pi2: \[ [0 \leq y; y \leq 1] \Rightarrow \arccos y \leq \pi/2 \]
by (metis (mono_tags) arccos-0 arccos cos-le-one cos-monotone-0-pi-le
  cos-pi cos-pi-half pi-half-ge-zero antisym-conv less-eq-neg-nonneg linear
  minus-minus order.trans order-refl)

lemma sincos-total-pi-half:
  assumes \( 0 \leq x 0 \leq y x^2 + y^2 = 1 \)
  shows \( \exists t. 0 \leq t \land t \leq \pi/2 \land x = \cos t \land y = \sin t \)
proof
  have \( \exists t. x \leq 1 \)
    using assms by (metis le-add-same-cancel1 power2-le-imp-le power-one zero-le-power2)
  with assms have \( *: 0 \leq \arccos x \cos (\arccos x) = x \)
    by (auto simp: arccos)
  from assms have \( y = \sqrt{(1 - x^2)} \)
by (metis abs-of-nonneg add.commute add-diff-cancel real-sqrt-abs)
with x1 * assms arccos-le-pi2 [of x] show ?thesis
  by (rule_tac x=arccos x in exI) (auto simp: sin-arccos)
qed

lemma sincos-total-pi:
  assumes 0 ≤ y x2 + y2 = 1
  shows ∃ t. 0 ≤ t ∧ t ≤ pi ∧ x = cos t ∧ y = sin t
proof (cases rule: le-cases [of 0 x])
  case le
  from sincos-total-pi-half [OF le] show ?thesis
    by (metis assms le-add-same-cancel1 mult.commute mult-2-right order.trans)
next
case ge
  then have 0 ≤ −x
    by simp
  then obtain t where t: t≥0 t ≤ pi/2 −x = cos t y = sin t
    using sincos-total-pi-half assms
  by auto (metis 0 ≤ − x power2-minus)
  show ?thesis
    by (rule exI [where x = pi − t]) (use t in auto)
qed

lemma sincos-total-2pi-le:
  assumes x2 + y2 = 1
  shows ∃ t. 0 ≤ t ∧ t ≤ 2 * pi ∧ x = cos t ∧ y = sin t
proof (cases rule: le-cases [of 0 y])
  case le
  from sincos-total-pi [OF le] show ?thesis
    by (metis assms le-add-same-cancel1 mult.commute mult-2-right order.trans)
next
case ge
  then have 0 ≤ −y
    by simp
  then obtain t where t: t≥0 t ≤ pi x = cos t −y = sin t
    using sincos-total-pi-half assms
  by auto (metis 0 ≤ − y power2-minus)
  show ?thesis
    by (rule exI [where x = 2 * pi − t]) (use t in auto)
qed

lemma sincos-total-2pi:
  assumes x2 + y2 = 1
  obtains t where 0 ≤ t t < 2*pi x = cos t y = sin t
proof —
  from sincos-total-2pi-le [OF assms]
  obtain t where t: 0 ≤ t t ≤ 2*pi x = cos t y = sin t
    by blast
show \(?thesis
  by (cases t = 2 * pi) (use t that in \langle force+\rangle)
qed

lemma arcsin-less-mono: \(|x| \leq 1 \implies |y| \leq 1 \implies arcsin x < arcsin y \iff x < y
  by (rule trans [OF sin-mono-less-eq [symmetric]]) (use arcsin-ubound arcsin-lbound in auto)

lemma arcsin-le-mono: \(|x| \leq 1 \implies |y| \leq 1 \implies arcsin x \leq arcsin y \iff x \leq y
  using arcsin-less-mono not-le by blast

lemma arcsin-less-arcsin: \(-1 \leq x \implies x < y \implies y \leq 1 \implies arcsin x < arcsin y
  using arcsin-less-mono by auto

lemma arcsin-le-arcsin: \(-1 \leq x \implies x \leq y \implies y \leq 1 \implies arcsin x \leq arcsin y
  using arcsin-le-mono by auto

lemma arcsin-nonneg: \(x \in \{0..1\} \implies arcsin x \geq 0
  using arcsin-le-arcsin [of 0 x] by simp

lemma arccos-less-mono: \(|x| \leq 1 \implies |y| \leq 1 \implies arccos x < arccos y \iff y < x
  by (rule trans [OF cos-mono-less-eq [symmetric]]) (use arccos-ubound arccos-lbound in auto)

lemma arccos-le-mono: \(|x| \leq 1 \implies |y| \leq 1 \implies arccos x \leq arccos y \iff y \leq x
  using arccos-less-mono [of y x] by (simp add: not-le [symmetric])

lemma arccos-less-arccos: \(-1 \leq x \implies x < y \implies y \leq 1 \implies arccos y < arccos x
  using arccos-less-mono by auto

lemma arccos-le-arccos: \(-1 \leq x \implies x \leq y \implies y \leq 1 \implies arccos y \leq arccos x
  using arccos-le-mono by auto

lemma arccos-eq-iff: \(|x| \leq 1 \land |y| \leq 1 \implies arccos x = arccos y \iff x = y
  using cos-arccos-abs by fastforce

lemma arccos-cos-eq-abs:
  assumes \(|\vartheta| \leq pi
  shows arccos (cos \vartheta) = |\vartheta|
  unfolding arccos-def
proof (intro the-equality conjI; clarify?)
  show cos |\vartheta| = cos \vartheta
    by (simp add: abs-real-def)
  show x = |\vartheta| if cos x = cos \vartheta \and 0 \leq x \and x \leq pi for x
    by (simp add: \langle cos |\vartheta| = cos \vartheta, assms cos-inj-pi that\rangle
qed (use assms in auto)

lemma arccos-cos-eq-abs-2pi:
obtains $k$ where $\arccos (\cos \vartheta) = |\vartheta - \text{of-int } k \ast (2 \ast \pi)|$

proof –

define $k$ where $k \equiv \lfloor (\vartheta + \pi) / (2 \ast \pi) \rfloor$

have lepi: $|\vartheta - \text{of-int } k \ast (2 \ast \pi)| \leq \pi$

by (auto simp: k-def abs-if algebra-simps)

have $\arccos (\cos \vartheta) = \arccos (\cos (\vartheta - \text{of-int } k \ast (2 \ast \pi)))$

using $\cos\text{-int-2pin} \sin\text{-int-2pin}$ by (simp add: cos-diff mult.commute)

also have $\ldots = |\vartheta - \text{of-int } k \ast (2 \ast \pi)|$

using $\arccos\text{-eq-abs}$ lepi by blast

finally show $\text{thesis}$

using that by metis

qed

lemma $\arccos\text{-arctan}$:

assumes $-1 < x x < 1$

shows $\arccos x = \pi/2 - \arctan(x / \sqrt{1 - x^2})$

proof –

have $\arctan(x / \sqrt{1 - x^2}) - (\pi/2 - \arccos x) = 0$

proof (rule sin-inj-pi)

show $-\pi < \arctan (x / \sqrt{1 - x^2}) - (\pi/2 - \arccos x)$

using $\arctan\text{-bound}$ [of $x / \sqrt{1 - x^2}$] $\arccos\text{-bounded}$ [of $x$] assms

by (simp add: algebra-simps)

next

show $\arctan (x / \sqrt{1 - x^2}) - (\pi/2 - \arccos x) < \pi$

using $\arctan\text{-ubound}$ [of $x / \sqrt{1 - x^2}$] $\arccos\text{-bounded}$ [of $x$] assms

by (simp add: algebra-simps)

next

show $\sin (\arctan (x / \sqrt{1 - x^2}) - (\pi/2 - \arccos x)) = 0$

using assms

by (simp add: algebra-simps sin-diff cos-add sin-arccos sin-arctan cos-arctan

power2-eq-square square-eq-1-iff)

qed

then show $\text{thesis}$

by simp

qed

lemma $\arcsin\text{-plus-arccos}$:

assumes $-1 \leq x x \leq 1$

shows $\arcsin x + \arccos x = \pi/2$

proof –

have $\arcsin x = \pi/2 - \arccos x$

apply (rule sin-inj-pi)

using assms arcsin [OF assms] arccos [OF assms]

by (auto simp: algebra-simps sin-diff)

then show $\text{thesis}$

by (simp add: algebra-simps)

qed
lemma arcsin-arccos-eq: \(-1 \leq x \rightarrow x \leq 1 \rightarrow \arcsin x = \pi/2 - \arccos x\)
using arcsin-plus-arccos by force

lemma arccos-arcsin-eq: \(-1 \leq x \rightarrow x \leq 1 \rightarrow \arccos x = \pi/2 - \arcsin x\)
using arcsin-plus-arccos by force

lemma arcsin-arctan: \(-1 < x \rightarrow x < 1 \rightarrow \arcsin x = \arctan(x / \sqrt{1 - x^2})\)
by (simp add: arcsin-arctan arcsin-arccos-eq)

lemma arcsin-arccos-sqrt-pos: \(0 \leq x \rightarrow x \leq 1 \rightarrow \arcsin x = \arccos(\sqrt{1 - x^2})\)
by (smt (verit, del-insts) arccos-cos arcsin-0 arcsin-le-arcsin arcsin-pi cos-arcsin)

lemma arcsin-arccos-sqrt-neg: \(-1 \leq x \rightarrow x \leq 0 \rightarrow \arcsin x = -\arccos(\sqrt{1 - x^2})\)
using arcsin-arccos-sqrt-pos [of \(-x\)]
by (simp add: arcsin-minus)

lemma arccos-arcsin-sqrt-pos: \(0 \leq x \rightarrow x \leq 1 \rightarrow \arccos x = \arcsin(\sqrt{1 - x^2})\)
by (smt (verit, del-insts) arccos-lbound arccos-le-pi2 arcsin-sin sin-arccos)

lemma arccos-arcsin-sqrt-neg: \(-1 \leq x \rightarrow x \leq 0 \rightarrow \arccos x = \pi - \arcsin(\sqrt{1 - x^2})\)
using arccos-arcsin-sqrt-pos [of \(-x\)]
by (simp add: arccos-minus)

lemma cos-limit-1:
assumes \((\lambda j. \cos (\vartheta j)) \longrightarrow 1\)
shows \(\exists k. (\lambda j. \vartheta j - \text{of-int} (k j) \times (2 \times \pi)) \longrightarrow 0\)
proof -
  have \(\forall F j \in \text{sequentially}. \cos (\vartheta j) \in \{-1..1\}\)
  by auto
  then have \((\lambda j. \arccos (\cos (\vartheta j))) \longrightarrow \arccos 1\)
  using continuous-on-tendsto-compose [OF continuous-on-arccos assms] by auto
  moreover have \(\forall j. \arccos (\cos (\vartheta j)) = |\vartheta j - \text{of-int} (k j) \times (2 \times \pi)|\)
  using arccos-cos-eq-2pi by metis
  then have \(\forall k. \arccos (\cos (\vartheta j)) = |\vartheta j - \text{of-int} (k j) \times (2 \times \pi)|\)
  by metis
  ultimately have \(\exists k. (\lambda j. \vartheta j - \text{of-int} (k j) \times (2 \times \pi)) \longrightarrow 0\)
  by auto
  then show \(?thesis\)
  by (simp add: tendsto-rabs-zero-iff)
qed

lemma cos-diff-limit-1:
assumes \((\lambda j. \cos (\vartheta j - \Theta)) \longrightarrow 1\)
obtains k where \((\lambda j. \vartheta j - \text{of-int} (k j) \times (2 \times \pi)) \longrightarrow \Theta\)
proof -
obtain $k$ where $(\lambda_j (\vartheta_j - \Theta) - \text{of-int} (k_j) * (2 * \pi)) \longrightarrow 0$
using cos-limit-I [OF assms] by auto
then have $(\lambda_j \Theta + ((\vartheta_j - \Theta) - \text{of-int} (k_j) * (2 * \pi))) \longrightarrow \Theta + 0$
by (rule tendsto-add [OF tendsto-const])
with that show thesis
by auto
qed

113.19 Machin’s formula

lemma arctan-one: $\arctan 1 = \pi/4$
by (rule arctan-unique) (simp-all add: tan-45 m2pi-less-pi)

lemma tan-total-pi4:
assumes $|x| < 1$
shows $\exists z. - (\pi/4) < z < \pi/4 \land \tan z = x$
proof
show $-(\pi/4) < \arctan x \land \arctan x < \pi/4 \land \tan (\arctan x) = x$
unfolding arctan-one [symmetric] arctan-minus [symmetric]
unfolding arctan-less-iff
using assms by (auto simp: arctan)
qed

lemma arctan-add:
assumes $|x| \leq 1 \land |y| < 1$
shows $\arctan x + \arctan y = \arctan ((x + y) / (1 - x * y))$
proof (rule arctan-unique [symmetric])
have $-(\pi/4) \leq \arctan x - (\pi/4) < \arctan y$
unfolding arctan-one [symmetric] arctan-minus [symmetric]
unfolding arctan-le-iff arctan-less-iff
using assms by auto
from add-le-less-mono [OF this] show 1: $-(\pi/2) < \arctan x + \arctan y$
by simp
have $\arctan x \leq \pi/4 \land \arctan y < \pi/4$
unfolding arctan-one [symmetric]
unfolding arctan-le-iff arctan-less-iff
using assms by auto
from add-le-less-mono [OF this] show 2: $\arctan x + \arctan y < \pi/2$
by simp
show $\tan (\arctan x + \arctan y) = (x + y) / (1 - x * y)$
using cos-gt-zero-pi [OF 1 2] by (simp add: arctan tan-add)
qed

lemma arctan-double: $|x| < 1 \Longrightarrow 2 * \arctan x = \arctan ((2 * x) / (1 - x^2))$
by (metis arctan-add linear mult-2 not-less power2-eq-square)

theorem machin: $\pi/4 = 4 * \arctan (1/5) - \arctan (1/239)$
proof
have $|1/5| < (1 :: real)$
by auto

from arctan-add[OF less-imp-le[OF this] this] have 2 * arctan (1 / 5) = arctan (5 / 12)
  by auto

moreover
have |5 / 12| < (1 :: real)
  by auto

from arctan-add[OF less-imp-le[OF this] this] have 2 * arctan (5 / 12) = arctan (120 / 119)
  by auto

moreover
have |1| ≤ (1 :: real) and |1/239| < (1::real)
  by auto

from arctan-add[OF this] have arctan 1 + arctan (1/239) = arctan (120 / 119)
  by auto

ultimately have arctan 1 + arctan (1/239) = 4 * arctan (1 / 5)
  by auto

then show ?thesis
  unfolding arctan-one by algebra

qed

lemma machin-Euler: 5 * arctan (1 / 7) + 2 * arctan (3 / 79) = pi/4

proof -
  have 17: |1 / 7| < (1 :: real) by auto
    with arctan-double have 2 * arctan (1 / 7) = arctan (7 / 24)
      by simp (simp add: field-simps)

moreover
have |7 / 24| < (1 :: real) by auto
  with arctan-double have 2 * arctan (7 / 24) = arctan (336 / 527)
    by simp (simp add: field-simps)

moreover
have |336 / 527| < (1 :: real) by auto

from arctan-add[OF less-imp-le[OF 17] this] have arctan(1/7) + arctan (336 / 527) = arctan (2879 / 3353)
  by auto

ultimately have I: 5 * arctan (1 / 7) = arctan (2879 / 3353) by auto
  have 379: |3 / 79| < (1 :: real) by auto
    with arctan-double have II: 2 * arctan (3 / 79) = arctan (237 / 3116)
      by simp (simp add: field-simps)

    have *: |237 / 3116| < (1 :: real) by auto

    have |237 / 3116| < (1 :: real) by auto
    with I II show ?thesis by auto

qed
113.20   Introducing the inverse tangent power series

lemma monoseq-arctan-series:
fixes x :: real
assumes \(|x| \leq 1\)
shows monoseq \((\lambda n. 1/real(n * 2 + 1) * x^(n * 2 + 1))\)
(is monoseq ?a)
proof (cases \(x = 0\))
case True
then show ?thesis by (auto simp: monoseq-def)
next
case False
have norm \(x \leq 1\) and \(x \leq 1\) and \(-1 \leq x\)
  using assms by auto
show monoseq ?a
proof
  have mono: \(1/real(Suc(Suc(n * 2))) * x^Suc(n * 2) \leq 1/real(Suc(n * 2)) * x^Suc(n * 2)\)
    if \(0 \leq x\) and \(x \leq 1\) for \(n\) and \(x :: real\)
  proof (rule mult-mono)
    show \(1/real(Suc(Suc(n * 2))) \leq 1/real(Suc(n * 2))\)
      by (rule frac-le simp-all)
    show \(0 \leq 1/real(Suc(n * 2))\)
      by auto
    show \(x^Suc(Suc(n * 2)) \leq x^Suc(n * 2)\)
      by (rule power-decreasing) (simp-all add: \(0 \leq x\) \(x \leq 1\))
    show \(0 \leq x^Suc(Suc(n * 2))\)
      by (rule zero-le-power) (simp add: \(0 \leq x\))
  qed
  show ?thesis
proof (cases \(0 \leq x\))
  case True
  from mono[OF this \(x \leq 1\), THEN allI]
  show ?thesis
    unfolding Suc-eq-plus1[symmetric] by (rule mono-SucI2)
next
case False
then have \(0 \leq -x\) and \(-x \leq 1\)
  using \((-1 \leq x\) by auto
from mono[OF this]
have \(1/real(Suc(Suc(n * 2))) * x^Suc(Suc(n * 2)) \geq 1/real(Suc(n * 2)) * x^Suc(n * 2)\) for \(n\)
  using \(0 \leq -x\) by auto
then show ?thesis
  unfolding Suc-eq-plus1[symmetric] by (rule mono-SucI1[OF allI])
qed
qed

lemma zeroseq-arctan-series:
THEORY "Transcendental"

fixes x :: real
assumes |x| ≤ 1
shows (λn. 1 / real (n + 1) * x^((n + 1))) ----> 0
(is |a ----> 0)
proof (cases x = 0)
  case True
  then show ?thesis by simp
next
  case False
  have norm x ≤ 1 and x ≤ 1 and −1 ≤ x
  using assms by auto
  show ?a ----> 0
  proof (cases |x| < 1)
    case True
    then have norm x < 1 by auto
    from tendsto-mul[OF LIMSEQ-inverse-real-of-nat LIMSEQ-power-zero[OF (norm x < 1), THEN LIMSEQ-Suc]]
    have (λn. 1 / real (n + 1) * x^((n + 1))) ----> 0
    unfolding inverse-eq-divide Suc-eq-plus1 by simp
    then show ?thesis
      using pos2 by (rule LIMSEQ-linear)
  next
    case False
    then have x = −1 ∨ x = 1
    using |x| ≤ 1 by auto
    then have n-eq: ∃ n. x^((n + 1)) = x
    unfolding One-nat-def by auto
    from tendsto-mul[OF LIMSEQ-inverse-real-of-nat(THEN LIMSEQ-linear, OF pos2, unfolded inverse-eq-divide] tendsto-const[of x]]
    have ?thesis
      unfolding n-eq Suc-eq-plus1 by auto
  qed
qed

lemma summable-arctan-series:
fixes n :: nat
assumes |x| ≤ 1
shows summable (λ k. (−1) k * (1 / real (k*2+1) * x^((k*2+1))))
(is summable (?c x))
by (rule summable-Leibniz(1),
  rule zeroseq-arctan-series[OF assms],
  rule monoseq-arctan-series[OF assms])

lemma DERIV-arctan-series:
assumes |x| < 1
shows DERIV (λx'. ∑ k. (−1) k * (1 / real (k + 2 + 1) * x'^(k + 2 + 1))) x :=
  (∑ k. (−1) k * x^((k + 2)))
(is DERIV ?arctan -: > ?Int)
proof
let \( \mathcal{f} = \lambda n. \text{if even } n \text{ then } (-1)^{n \, \text{div} \, 2} \ast 1 / \text{real}(\text{Suc } n) \text{ else } 0 \)

have \( n\text{-even: even } n \implies 2 \ast (n \, \text{div} \, 2) = n \text{ for } n :: \text{nat} \)
  by presburger
then have \( \text{if-eq: } \text{if } n \ast \text{real}(\text{Suc } n) \ast x' \sim n = \)
  \( \text{if even } n \text{ then } (-1)^{(n \, \text{div} \, 2) \ast x' \sim (2 \ast (n \, \text{div} \, 2))} \text{ else } 0 \)
  for \( n \ast x' \)
  by auto

have \( \text{summable-Integral: summable } (\lambda n. (-1) \sim n \ast x' \sim (2 \ast n)) \text{ if } |x| < 1 \text{ for } x :: \text{real} \)

proof
  from that have \( x^2 < 1 \)
    by (simp add: abs-square-less-1)
  have \( \text{summable } (\lambda n. (-1) \sim n \ast (x^2) \sim n) \)
    by (rule summable-Leibniz(1))
    (auto intro!: LIMSEQ-realpow-zero monoseq-realpow \( x^2 < 1 \), order-less-imp-le[OF \( x^2 < 1 \)])
  then show \( \text{thesis} \)
    by (simp only: power-mult)
qed

have \( \text{sums-even: } (\text{sums } f = (\text{sums } (\lambda n. \text{if even } n \text{ then } f \,(n \, \text{div} \, 2) \text{ else } 0)) \text{ for } f :: \text{nat} ) \Rightarrow \text{real} \)

proof
  have \( f \text{ sums } x = (\lambda n. \text{if even } n \text{ then } f \,(n \, \text{div} \, 2) \text{ else } 0) \text{ sums } x \text{ for } x :: \text{real} \)
    by auto
  next
    assume \( (\lambda n. \text{if even } n \text{ then } f \,(n \, \text{div} \, 2) \text{ else } 0) \text{ sums } x \)
    from LIMSEQ-linear[OF this[simplified sums-def] pos2, simplified sum-split-even-odd[simplified mult.commute]]
    show \( f \text{ sums } x \)
      unfolding sums-def by auto
  qed
  then show \( \text{thesis} \)
    by auto
qed

have \( \text{Int-eq: } (\sum n. \text{if } n \ast \text{real}(\text{Suc } n) \ast x ^ \sim n) = ?\text{Int} \)
  unfolding \( \text{if-eq mult.commute[of - 2]} \)
  suminf-def sums-even[of \( \lambda n. (-1) \sim n \ast x' \sim (2 \ast n) \), symmetric]
  by auto

have \( \text{arctan-eq: } (\sum n. \text{if } n \ast x' \sim (\text{Suc } n)) = ?\text{arctan } x \text{ for } x \)

proof
have if-eq': \( \forall n. (if\ even\ n\ then\ (-1) \cdot (n \ div\ 2) \ast 1 / real\ (Suc\ n)\ else\ 0) \ast x \cdot Suc\ n =\)
(if even n then \((-1) \cdot (n \ div\ 2) \ast (1 / real\ (Suc\ (2 \ast (n \ div\ 2)))) \ast x \cdot Suc\ (2 \ast (n \ div\ 2)))\ else\ 0)
using n-even by auto
have idx-eq: \( \forall n. n \ast 2 + 1 = Suc\ (2 \ast n) \) 
by auto
then show ?thesis
unfolding if-eq' idx-eq suminf-def 
sums-even[of \( \lambda n. (-1) \cdot n \ast (1 / real\ (Suc\ (2 \ast n))) \ast x \cdot Suc\ (2 \ast n))\), symmetric by auto qed

have DERIV \( \langle \lambda x. \sum n. \ ?f n \ast x^n \rangle \cdot x \ast (\sum n. \ ?f n \ast real\ (Suc\ n) \ast x^n) \) 
proof (rule DERIV-power-series')
show \( x \in \{-1 <..< 1\} \) 
using \( |x| < 1 \) by auto
show summable \( \langle \lambda n. \ ?f n \ast x^n \rangle \) 
if \( x'-bounds: x' \in \{-1 <..< 1\} \) for \( x' :: real \) 
proof –
from that have \(|x'| < 1 \) by auto
then show ?thesis
using that summable-sums-if [OF sums-0 [of \( \lambda x. 0 \)] summable-sums
[OF summable-Integral]] 
by (auto simp add: if-distrib [of \( \lambda x. x \ast y \) for \( y \)] cong: if-cong)
qed
qed auto
then show ?thesis
by (simp only: Int-eq arctan-eq)
qed

lemma arctan-series:
assumes \(|x| \leq 1 \)
shows \( \arctan\ x = (\sum k. (-1) \cdot k \ast (1 / real\ (k \ast 2 + 1)) \ast x \cdot (k \ast 2 + 1)))\) 
is - = suminf \( \langle \lambda n. \ ?c x n \rangle \) 
proof –
let \( ?c' = \lambda x n. (-1) \cdot n \ast x^n \ast (n \ast 2) \)

have DERIV-arctan-suminf: DERIV \( \langle \lambda x. \ ?c x \rangle \cdot x \ast (\suminf \langle \ ?c' x \rangle) \) 
if \( \theta < r \) and \( r < 1 \) and \(|x| < r \) for \( r x :: real \)
proof (rule DERIV-arctan-series)
from that show \(|x| < 1 \)
using \( \theta < 1 \) and \(|x| < r \) by auto
qed

{ 
  fix \( x :: real \)
assumed } |x| \leq 1
\textbf{note} \textit{summable-Leibniz[OF \textit{zeroseq-arctan-series}[OF \textit{this}] \textit{monoseq-arctan-series}[OF \textit{this}]}}
\textbf{)} \textbf{note} \textit{arctan-series-borders = this}

\textbf{have} \textit{when-less-one: arctan} x = (\sum k. \ ?c x \ x k) \textit{if} |x| < 1 \textit{for} x :: \text{real}
\textbf{proof} –
\textbf{obtain} r \textbf{where} |x| < r \textbf{and} r < 1
\textbf{using} dense[\textit{OF} \ ?c x < 1] \textbf{by blast}
\textbf{then have} \theta < r \textbf{and} - r < x \textbf{and} x < r \textbf{by auto}

\textbf{have} \textit{suminf-eq-arctan-bounded: suminf} (?c x) - arctan x = \textit{suminf} (?c a) - arctan a
\textbf{if} -r < a \textbf{and} b < r \textbf{and} a < b \textbf{and} a \leq x \textbf{and} x \leq b \textbf{for} x \ b
\textbf{proof} –
\textbf{from} \textit{that} \textbf{have} |x| < r \textbf{by auto}
\textbf{show} \textit{suminf} (?c x) - arctan x = \textit{suminf} (?c a) - arctan a
\textbf{proof} (\textit{rule DERIV-isconst2}[\textit{of} a b])
\textbf{show} a < b \textbf{and} a < x \textbf{and} x < b
\textbf{using} a < b \textbf{a} \leq x \textbf{c} x \leq b \textbf{by auto}
\textbf{have} \forall x. - r < x \textbf{and} x < r \textbf{then have} DERIV (\lambda x. \textit{suminf} (?c x) - arctan x) x
:= 0
\textbf{proof} (\textit{rule allI}, \textit{rule impI})
\textbf{fix} x
\textbf{assume} -r < x \wedge x < r
\textbf{then have} |x| < r \textbf{by auto}
\textbf{with} \textit{r} \textit{<} 1 \textbf{have} |x| < 1 \textbf{by auto}
\textbf{have} |- (x^2)| < 1 \textbf{using} \textit{abs-square-less-1} [\textit{OF} \ ?c x < 1] \textbf{by auto}
\textbf{then have} (\lambda n. (- (x^2)) ^ n) \textit{sums} (1 / (1 - (- (x^2))))
\textbf{then have} (?c' x) \textit{sums} \textit{inverse} (1 / (1 - (- (x^2))))
\textbf{then unfolding} real-norm-def [\textit{symmetric}] \textbf{by} (\textit{rule geometric-sums})
\textbf{then have} (\textit{inverse} (?c' x) \textit{sums} (1 / (1 - (- (x^2)))))
\textbf{then unfolding} power-mult-distrib [\textit{symmetric}] \textit{power-mul}. \textit{commute}[\textit{of} - 2] \textbf{by auto}
\textbf{then have} \textit{suminf-c'-eq-geom: inverse} (1 + x^2) = \textit{suminf} (?) x
\textbf{using} \textit{sums-unique unfolding} inverse-eq-divide \textbf{by auto}
\textbf{have} DERIV (\lambda x. \textit{suminf} (?c x)) x := \textit{inverse} (1 + x^2)
\textbf{then unfolding} \textit{suminf-c'-eq-geom}
\textbf{by} (\textit{rule DERIV-arctan-suminf}[\textit{OF} \ 0 < r \ (r < 1) \ (|x| < r)])
\textbf{from} DERIV-diff \textit{[OF this DERIV-arctan]} \textbf{show} DERIV (\lambda x. \textit{suminf} (?c x) - arctan x) x := 0
\textbf{by auto}
\textbf{then have} DERIV-in-rball: \forall y. a \leq y \wedge y \leq b \textit{then have} DERIV (\lambda x. \textit{suminf} (?c x) - arctan x) y := 0
\textbf{using} (-r < a \ a < b \textit{by auto}
\textbf{then show} \forall y. [a < y \ w y < b] \textit{then have} DERIV (\lambda x. \textit{suminf} (?c x) - arctan x) y := 0
\textbf{using} \textit{abs} < r \textit{by auto}
\textbf{show} \textit{continuous-on} {\textit{a..b}} (\lambda x. \textit{suminf} (?c x) - arctan x)
using DERIV-in-rball DERIV-atLeastAtMost-imp-continuous-on by blast

qed

have suminf-arctan-zero: suminf (?c 0) − arctan 0 = 0

unfolding Suc-eq-plus1 [symmetric] power-Suc mult-zero-right arctan-zero-zero

by auto

have suminf (?c x) − arctan x = 0

proof (cases x = 0)
  case True
  then show ?thesis
    using suminf-arctan-zero by auto
  next
  case False
  then have 0 < |x| and − |x| < |x|
    by auto
  have suminf (?c (− |x|)) − arctan (− |x|) = suminf (?c 0) − arctan 0
    by (rule suminf-eq-arctan-bounded [where x1 = 0 and a1 = − |x| and b1 = |x|],
        symmetric)
    (simp-all only: |x| < r, (− |x|) < |x|, neg-less-iff-less)
    moreover
    have suminf (?c x) − arctan x = suminf (?c (− |x|)) − arctan (− |x|)
    by (rule suminf-eq-arctan-bounded [where x1 = x and a1 = − |x| and b1 = |x|])
    (simp-all only: |x| < r, (− |x|) < |x|, neg-less-iff-less)
    ultimately show ?thesis
    using suminf-arctan-zero by auto
  qed
  then show ?thesis by auto
  qed

show arctan x = suminf (λn. ?c x n)
proof (cases |x| < 1)
  case True
  then show ?thesis by (rule when-less-one)
  qed
next
  case False
  then have |x| = 1 using |x| ≤ 1 by auto
  let ?a = λx n. |x| / (n * 2 + 1) * x"(n * 2 + 1)|
  let ?diff = λx n. |arctan x − (∑ i<n. ?c x i)|
  have ?diff 1 n ≤ ?a 1 n for n :: nat
    proof −
      have 0 < (1 :: real) by auto
      moreover
      have ?diff x n ≤ ?a x n if 0 < x and x < 1 for x :: real
      proof −
        from that have |x| ≤ 1 and |x| < 1
        by auto
from \(0 < x\) have \(0 \leq 1 / \text{real } (n + 1)\) by auto

proof (cases even n)
  case True
  then have \(\sum_{i<n} (\frac{n}{n + 1}) \leq \sum_{i<n} (\frac{n}{n + 1})\) by (rule mult-pos-pos)
  then have \(\sum_{i<n} (\frac{n}{n + 1}) \leq \sum_{i<n+1} (\frac{n}{n + 1})\) by auto

next
  case False
  then have \(\sum_{i<n} (-1)^i \leq \sum_{i<n+1} (-1)^i\) by auto
  also have \(= \sum_{i<n} (-1)^i\) by auto
  also have \(= \sum_{i<n} -(-1)^i\) by auto

finally show \(\text{thesis}\).

qed
unfolding tendsto-rabs-zero-iff power-one divide-inverse One-nat-def
by (auto intro!: tendsto-mult LIMSEQ-linear LIMSEQ-inverse-real-of-nat simp
del: of-nat-Suc)

have \( \{\text{diff} 1 \longrightarrow 0 \) proof (rule LIMSEQ-I)

fix \( r :: \text{real} \)
assume \( 0 < r \)

obtain \( N :: \text{nat} \) where \( N-I: N \leq \text{n} \Rightarrow \{\text{a} 1 \ \text{n} < r \ \text{for} \ n \) using LIMSEQ-D[OF \( \{\text{a} 1 \ \text{n} \longrightarrow 0 \ \text{of} \infty \text{r} \) by auto

have \( \text{norm} (\text{a} 1 \ \text{n} \leq \text{n} \Longrightarrow \{\text{a} 1 \ \text{n} \leq \text{n} \ \text{for} \ n \) using \( \{\text{a} 1 \ \text{n} \leq \text{n} \ \text{for} \ n \) by auto

then show \( \exists N. \forall n \geq N. \text{norm} (\text{a} 1 \ \text{n} \leq \text{n} \Longrightarrow \{\text{a} 1 \ \text{n} \leq \text{n} \ \text{for} \ n \) by blast

qed

from this [unfolded tendsto-rabs-zero-iff, THEN tendsto-add [OF - tendsto-const],
of - arctan 1, THEN tendsto-minus]

have \( \{\text{c} 1 \) sums (arctan 1) unfolding sums-def by auto
then have arctan 1 = (∑ i. \( \text{c} 1 \ i \)) by (rule sums-unique)

show \( \{\text{thesis} \)

proof (cases \( x = 1 \) )

case True
then show \( \{\text{thesis} \) by (simp add: \( (\text{arctan} 1 = (∑ i. \text{c} 1 \ i)\))

next

case False
then have \( x = -1 \) using \( |x| = 1 \) by auto

have \( -(\text{pi}/2) < 0 \) using pi-gt-zero by auto

have \( -(\text{pi}/2) < 0 \) using pi-gt-zero by auto

have c-minus-minus: \( \text{c} (-1) i = - \text{c} 1 \ i \) for \( i \) by auto

have \( \text{arctan} (-1) = \text{arctan} (\text{tan} (-\text{pi}/4)) \)
unfolding tan-45 tan-minus ..
also have \( \ldots = - (\text{pi}/4) \)
by (rule arctan-tan) (auto simp: order-less-trans[OF \( (\text{pi}/2) < 0 \)
pis-gt-zero])
also have \( \ldots = - (\text{arctan} (\text{tan} \text{pi}/4)) \)
unfolding neg-equal-iff-equal
by (rule arctan-tan[symmetric]) (auto simp: order-less-trans[OF \( (\text{pi}/2) < 0 \)
pis-gt-zero])
also have \( \ldots = - (\text{arctan} 1) \)
unfolding tan-45 ..
also have \( \ldots = - (\sum i. \text{c} 1 \ i) \)
using \( (\text{arctan} 1 = (∑ i. \text{c} 1 \ i)\)) by auto
also have \( \ldots = (\sum i. \text{c} (-1) i) \)
using suminf-minus[OF sums-summable[OF \( \{\text{c} 1 \) sums (arctan 1)\)]]
unfolding c-minus-minus by auto
finally show \( \{\text{thesis} \) using \( \{x = -1 \) by auto

qed
lemma arctan-half: arctan x = 2 * arctan (x / (1 + sqrt(1 + x^2)))
for x :: real
proof -
  obtain y where low: -(pi/2) < y and high: y < pi/2 and y-eq: tan y = x
    using tan-total by blast
  then have low2: -(pi/2) < y / 2 and high2: y / 2 < pi/2
    by auto
  have 0 < cos y by (rule cos-gt-zero-pi[OF low high])
  then have cos y ≠ 0 and cos-sqrt: sqrt ((cos y)^2) = cos y
    by auto
  have 1 + (tan y)^2 = 1 / (cos y)^2
    unfolding tan-def power-divide ..
  also have ... = (cos y)^2 / (cos y)^2 + (sin y)^2 / (cos y)^2
    using ‹cos y ≠ 0› by auto
  also have ... = 1 / (cos y)^2
    unfolding add-divide-distrib[symmetric] sin-cos-squared-add2 ..
  finally have 1 + (tan y)^2 = 1 / (cos y)^2 .
  have sin y / (cos y + 1) = tan y / ((cos y + 1) / cos y)
    unfolding tan-def using ‹cos y ≠ 0› by (simp add: field-simps)
  also have ... = tan y / (1 + 1 / cos y)
    using ‹cos y ≠ 0› unfolding add-divide-distrib by auto
  also have ... = tan y / (1 + sqrt ((cos y)^2))
    unfolding cos-sqrt ..
  also have ... = tan y / (1 + sqrt (1 / (cos y)^2))
    unfolding real-sqrt-divide by auto
  finally have eq: sin y / (cos y + 1) = tan y / (1 + sqrt(1 + (tan y)^2))
    unfolding ‹1 + (tan y)^2 = 1 / (cos y)^2› .
  have arctan x = y
    using arctan-tan low high y-eq by auto
  also have ... = 2 * (arctan (tan (y/2)))
    using arctan-tan[OF low2 high2] by auto
  also have ... = 2 * (arctan (sin y / (cos y + 1)))
    unfolding tan-half by auto
  finally show ?thesis
    unfolding eq ‹tan y = x› .
qed

lemma arctan-monotone: x < y ⇒ arctan x < arctan y
by (simp only: arctan-less-iff)

lemma arctan-monotone’: x ≤ y ⇒ arctan x ≤ arctan y
by (simp only: arctan-le-iff)
lemma arctan-inverse:
  assumes \( x \neq 0 \)
  shows \( \arctan \left( \frac{1}{x} \right) = sgn x \cdot \frac{\pi}{2} - \arctan x \)
proof (rule arctan-unique)
  have \( \frac{\pi}{2} < sgn x \cdot \frac{\pi}{2} - \arctan x \)
    using arctan-bounded \([\text{of } x]\) by linarith
  show \( sgn x \cdot \frac{\pi}{2} - \arctan x < \pi/2 \)
    using arctan-bounded \([\text{of } -x]\) assms by (auto simp: sgn-real-def arctan algebra-simps)
  show \( sgn x \cdot \frac{\pi}{2} - \arctan x < \frac{\pi}{2} \)
    using arctan-bounded \([\text{of } -x]\) assms by (auto simp: algebra-simps sgn-real-def arctan-minus)
  show \( \tan \left( sgn x \cdot \frac{\pi}{2} - \arctan x \right) = \frac{1}{x} \)
    unfolding tan-inverse \([\text{of } \arctan x]\)
    by (simp add: tan-def cos-arctan sin-arctan sin-diff cos-diff)
qed

theorem pi-series: \( \pi/4 = \sum k \cdot (-1)^k \cdot \frac{1}{2k+1} \)
(is - = ?SUM)
proof --
  have \( \pi/4 = \arctan 1 \)
    using arctan-one by auto
  also have \ldots = ?SUM
    using arctan-series \([\text{of } 1]\) by auto
  finally show ?thesis by auto
qed

113.21 Existence of Polar Coordinates

lemma cos-x-y-le-one: \( x/\sqrt{x^2+y^2} \leq 1 \)
by (rule power2-le-imp-le \([\text{OF - zero-le-one}]\)
  (simp add: power-divide divide-le-eq not-sum-power2-lt-zero)

lemma polar-Ex: \( \exists a::real. \exists x. x = r \cdot \cos a \land y = r \cdot \sin a \)
proof --
  have polar-ex1: \( \exists r a. x = r \cdot \cos a \land y = r \cdot \sin a \mbox{ if } 0 < y \mbox{ for } y \)
    proof --
      have \( x = \sqrt{x^2+y^2} \cdot \cos (\arccos (x/\sqrt{x^2+y^2})) \)
        by (simp add: cos-arccos-abs \([\text{OF cos-x-y-le-one}]\))
      moreover have \( y = \sqrt{x^2+y^2} \cdot \sin (\arccos (x/\sqrt{x^2+y^2})) \)
        using that
        by (simp add: sin-arccos-abs \([\text{OF cos-x-y-le-one}]\)
           power-divide right-diff-distrib flip: real-sqrt-mult)
      ultimately show ?thesis
        by blast
    qed
  show ?thesis
    proof (cases 0::real y rule: linorder-cases)
      case less
    qed
then show ?thesis
  by (rule polar-ex1)
next
case equal
then show ?thesis
  by (force simp: intro!: cos-zero sin-zero)
next
case greater
with polar-ex1 [where y=−y] show ?thesis
  by auto (metis cos-minus minus-minus minus-mult-right sin-minus)
qed

113.22 Basics about polynomial functions: products, extremal
behaviour and root counts

lemma polynomial-product-nat:
  fixes x :: nat
  assumes m: !i. i > m ⇒ int (a i) = 0
  and n: !j. j > n ⇒ int (b j) = 0
  shows (∑ i≤m. (a i) * x^i) * (∑ j≤n. (b j) * x^j) =
    (∑ r≤m + n. (∑ k≤r. (a k) * (b (r - k))) * x^r)
  using polynomial-product [of m a n b x] assms
  by (simp only: of-nat-mult [symmetric] of-nat-power [symmetric]
    of-nat-eq-iff Int.int-sum [symmetric])

lemma polyfun-diff:
  fixes x :: 'a::idom
  assumes l ≤ n
  shows (∑ i≤n. a i * x^i) − (∑ i≤n. a i * y^i) =
    (x − y) * (∑ j<n. a i * (y^j * (i − n) − 1)) * x^j)
  proof –
  have h: bij-betw (λ(i,j). (j,i)) ((SIGMA i : atMost n. lessThan i)) (SIGMA j : lessThan n. {Suc j..n})
    by (auto simp: bij-betw-def inj-on-def)
  have (∑ i≤n. a i * x^i) − (∑ i≤n. a i * y^i) = (∑ i≤n. a i * (x^i − y^i))
    by (simp add: right-diff-distrib sum-subtractf)
  also have ... = (∑ i≤n. a i * (x − y) * (b j * x^j))
    by (simp add: power-diff-sumr2 mult.assoc)
  also have ... = (∑ i≤n. ∑ j<i. a i * (x − y) * (y^j * Suc j * x^j))
    by (simp add: sum-distrib-left)
  also have ... = (SIGMA i : atMost n. lessThan i). a i * (x − y) *
    (y^j * Suc j * x^j))
    by (simp add: sum Sigma)
  also have ... = (SIGMA j : lessThan n. {Suc j..n}). a i * (x − y) *
    (y^j * Suc j * x^j))
    by (auto simp: Sum.reindex-bij_betw [OF h, symmetric] intro: sum.cong-simp)
  also have ... = (SIGMA j : lessThan n. {Suc j..n}). a i * (x − y) *
    (y^j * Suc j * x^j))
    by (simp add: sum Sigma)
also have \( \ldots = (x - y) \ast (\sum j < n. (\sum i = \text{Suc } j \ldots n. a \ast y^i (i - j - 1)) \ast x^j) \)

by \((\text{simp add: sum-distrib-left mult-ac})\)

finally show \(?\text{thesis}\).

qed

lemma \texttt{polyfun-linear-factor}:

fixes \(x::'a::\text{idom}\)

assumes \(1 \leq n\)

shows \((\sum i \leq n. a \ast x^i) \ast (\sum i \leq n. a \ast y^i) = (x - y) \ast ((\sum j < n. \sum k < n - j. a(j + k + 1) \ast y^k \ast x^j))\)

proof

have \((\sum i = \text{Suc } j \ldots n. a \ast y^i (i - j - 1)) = (\sum k < n - j. a(j + k + 1) \ast y^k)\)

if \(j < n\) for \(j::\text{n}\)

proof

have \(\forall k. k < n - j \Rightarrow k \in (\lambda x.i - \text{Suc } j \cdot \{\text{Suc } j \ldots n\})\)

by \((\text{rule-tac } x = k + \text{Suc } j \text{ in image-eql, auto})\)

then have \(h: \text{bij-betw } (\lambda x.i - (j + 1)) \{\text{Suc } j \ldots n\} (\text{lessThan } (n - j))\)

by \((\text{auto simp: bij-betw-def inj-on-def})\)

then show \(?\text{thesis}\)

by \((\text{auto simp: sum.reindex-bij-betw OF h, symmetric} \text{ intro: sum.cong-simp})\)

qed

then show \(?\text{thesis}\)

by \((\text{simp add: polyfun-diff OF \text{assms} sum-distrib-right})\)

qed

lemma \texttt{polyfun-linear-factor-root}:

fixes \(a::'a::\text{idom}\)

shows \(\exists b. \forall z. (\sum i \leq n. c(i) \ast z^i) = (z - a) \ast (\sum i < n. b(i) \ast z^i) + (\sum i \leq n. c(i) \ast a^i)\)

proof \((\text{cases } n = 0)\)

case \text{True} then show \(?\text{thesis}\)

by simp

next

case \text{False} have \((\exists b. \forall z. (\sum i \leq n. c(i) \ast z^i) = (z - a) \ast (\sum i < n. b(i) \ast z^i) + (\sum i \leq n. c(i) \ast a^i)) \leftarrow (\exists b. \forall z. (\sum i \leq n. c(i) \ast z^i) - (\sum i \leq n. c(i) \ast a^i) = (z - a) \ast (\sum i < n. b(i) \ast z^i)))\)

by \((\text{simp add: algebra-simps})\)

also have \(\ldots \leftarrow (\exists b. \forall z. (z - a) \ast (\sum i < n. (\sum i = \text{Suc } j \ldots n. c \ast a^i (i - \text{Suc } j)) \ast z^j) = (z - a) \ast (\sum i < n. b(i) \ast z^i)))\)

using \texttt{False} by \((\text{simp add: polyfun-diff})\)

also have \(\ldots = \text{True} \text{ by auto}\)

finally show \(?\text{thesis}\)

by simp

qed

lemma \texttt{polyfun-linear-factor-root}:
fixes $a :: 'a::idom$
assumes $(\sum i \leq n. c(i) * a^i) = 0$
optains $b$ where $\forall z. (\sum i \leq n. c i * z^i) = (z - a) * (\sum i < n. b i * z^i)$
using polyfun-linear-factor [of $c n a$] assms by auto

lemma isCont-polynom: isCont $(\lambda w. \sum i \leq n. c i * w^i)$ $a$
for $c :: nat \Rightarrow 'a::real-normed-div-algebra$
by simp

lemma zero-polynom-imp-zero-coeffs:
fixes $c :: nat \Rightarrow 'a::\{ab-semigroup-mult, real-normed-div-algebra\}$
assumes $\forall w. (\sum i \leq n. c i * w^i) = 0 \quad k \leq n$
shows $c k = 0$
using assms
proof (induction $n$ arbitrary: $c k$)
  case 0
  then show ?case by simp
next
case (Suc $n$ $c$ $k$)
  have [simp]: $c 0 = 0$ using Suc.prems(1) [of 0]
    by simp
  have $(\sum i \leq Suc n. c i * w^i) = w * (\sum i \leq n. c (Suc i) * w^i)$ for $w$
    proof
      have $(\sum i \leq Suc n. c i * w^i) = (\sum i \leq n. c (Suc i) * w ^ Suc i)$
        unfolding Set-Interval.sum.atMost-Suc-shift
      by simp
      also have $\ldots = w * (\sum i \leq n. c (Suc i) * w^i)$
        by (simp add: sum-distrib-left ac-simps)
      finally show $?thesis$.
    qed
  then have $w: \forall w. w \neq 0 \implies (\sum i \leq n. c (Suc i) * w^i) = 0$
    using Suc by auto
  then have $(\lambda h. \sum i \leq n. c (Suc i) * h^i) \rightarrow 0$
    by (simp cong: LIM-cong) — the case $w = 0$ by continuity
  then have $(\sum i \leq n. c (Suc i) * 0^i) = 0$
    using isCont-polynom [of 0 \lambda. c (Suc i) n] LIM-unique
    by (force simp: Limits.isCont-iff)
  then have $\forall w. (\sum i \leq n. c (Suc i) * w^i) = 0$
    using w by metis
  then have $\forall i. i \leq n \implies c (Suc i) = 0$
    using Suc.IH [of \lambda. c (Suc i)] by blast
  then show $?case$ using $\forall k \leq Suc n$
    by (cases $k$) auto
qed

lemma polyfun-rootbound:
fixes \(c :: \text{nat} \Rightarrow \{\text{idom}, \text{real-normed-div-algebra}\}\)
assumes \(c \neq 0\) \(k \leq n\)
shows finite \(\{z. (\sum i \leq n. c(i) \ast z^i) = 0\}\) \(\land\) card \(\{z. (\sum i \leq n. c(i) \ast z^i) = 0\}\) \(\leq n\)
using assms
proof (induction \(n\) arbitrary: \(c\) \(k\))
case 0
then show \(?case\)
by simp
next
case (Suc \(m\) \(c\) \(k\))
let \(?succase\) = \(?case\)
show \(?case\)
proof (cases \(z. (\sum i \leq \text{Suc} \ m. c(i) \ast z^i) = 0\) = \(\{\}\))
case True
then show \(?succase\)
by simp
next
case False
then obtain \(z0\)
where \(z0\): \((\sum i \leq \text{Suc} \ m. c(i) \ast z0^i) = 0\)
by blast
then obtain \(b\)
where \(b\): \((\sum i \leq \text{Suc} \ m. c(i) \ast w^i) = (w - z0) \ast (\sum i \leq m. b(i) \ast w^i)\)
using polyfun-linear-factor-root \([\text{OF} \ z0, \text{unfolded lessThan-Suc-atMost}]\)
by blast
then have \(eq\): \(\{z. (\sum i \leq \text{Suc} \ m. c(i) \ast z0^i) = 0\} = \text{insert} \ z0 \ \{z. (\sum i \leq m. b(i) \ast z0^i) = 0\}\)
by auto
have \(\neg (\forall k \leq m. b(k) = 0)\)
proof
assume \([\text{simpl}]\): \(\forall k \leq m. b(k) = 0\)
then have \((\sum i \leq m. b(i) \ast w^i) = 0\)
by simp
then have \((\sum i \leq \text{Suc} \ m. c(i) \ast w^i) = 0\)
using \(b\) by simp
then have \((\sum i \leq \text{Suc} \ m. c(i) \ast w^i) = 0\)
using zero-polynom-imp-zero-coeffs by blast
then show False using Suc.prems by blast
qed
then obtain \(k'\) where \(bk'\): \(b(k') \neq 0\) \(k' \leq m\)
by blast
show \(?succase\)
using Suc.IH \([\text{of} \ b \ k']\) \(bk'\)
by (simp add: eq card-insert-if del: sum.atMost-Suc)
qed

lemma
fixes \(c :: \text{nat} \Rightarrow \{\text{idom}, \text{real-normed-div-algebra}\}\)
assumes \( c k \neq 0 \) \( k \leq n \)
shows polyfun-roots-finite: finite \{ \( z. (\sum i \leq n. c(i) \ast z^i) = 0 \) \}
and polyfun-roots-card: card \{ z. (\sum i \leq n. c(i) \ast z^i) = 0 \} \leq n
using polyfun-rootbound assms by auto

lemma polyfun-finite-roots:
fixes \( c : \text{nat} \Rightarrow 'a : \{\text{idom},\text{real-normed-div-algebra}\} \)
shows finite \{ \( x. (\sum i \leq n. c(i) \ast x^i) = 0 \) \} ←→ (\exists i \leq n. c i \neq 0)
(is ?lhs = ?rhs)
proof
assume ?lhs
moreover have ¬ finite \{ \( x. (\sum i \leq n. c(i) \ast x^i) = 0 \) \} if \( \forall i \leq n. c i = 0 \)
proof –
from that have \( \bigwedge x. (\sum i \leq n. c i \ast x^i) = 0 \)
by simp
then show ?thesis
using ex-new-if-finite [OF infinite-UNIV-char-0 [where \( 'a = 'a \)]
by auto
qed
ultimately show ?rhs by metis
next
assume ?rhs
with polyfun-rootbound show ?lhs by blast
qed

lemma polyfun-eq-0: \( \forall x. (\sum i \leq n. c i \ast x^i) = 0 \) ←→ \( \forall i \leq n. c i = 0 \)
for \( c : \text{nat} \Rightarrow 'a : \{\text{idom},\text{real-normed-div-algebra}\} \)
using zero-polynom-imp-zero-coeffs by auto

lemma polyfun-eq-coeffs: \( \forall x. (\sum i \leq n. c i \ast x^i) = (\sum i \leq n. d i \ast x^i) \) ←→ \( \forall i \leq n. c i = d i \)
for \( c : \text{nat} \Rightarrow 'a : \{\text{idom},\text{real-normed-div-algebra}\} \)
proof –
have \( \forall x. (\sum i \leq n. c i \ast x^i) = (\sum i \leq n. d i \ast x^i) \) ←→ \( \forall x. (\sum i \leq n. (c i - d i) \ast x^i) = 0 \)
by (simp add: left-diff-distrib Groups-Big.sum-subtractf)
also have \( \ldots \) ←→ \( \forall i \leq n. c i - d i = 0 \)
by (rule polyfun-eq-0)
finally show ?thesis
by simp
qed

lemma polyfun-eq-const:
fixes \( c : \text{nat} \Rightarrow 'a : \{\text{idom},\text{real-normed-div-algebra}\} \)
shows \( \forall x. (\sum i \leq n. c i \ast x^i) = k \) ←→ \( c 0 = k \wedge (\forall i \in \{1..n\}. c i = 0) \)
(is ?lhs = ?rhs)
proof –
have \( \ast : \forall x. (\sum i \leq n. \text{if } i=0 \text{ then } k \text{ else } 0) \ast x^i = k \)
by (induct \( n \)) auto

proof

assume \( \_lhs \)

with \( \ast \) have \((\forall \ i\leq\ n.\ c\ i\ =\ (if\ i=0\ then\ k\ else\ 0))\)

by (simp add: polyfun-eq-coeffs [symmetric])

then show \(?rhs\) by simp

next

assume \( \_rhs \)

then show \(?lhs\) by (induct \( n \)) auto

qed

qed

lemma root-polyfun:

fixes \( z :: 'a::idom \)

assumes \( 1 \leq n \)

shows \( z^n = a \iff (\sum i\leq n.\ (if\ i=0\ then\ -a\ else\ if\ i=n\ then\ 1\ else\ 0) \ast z^i) = 0 \)

using assms by (cases \( n \)) (simp-all add: sum.atLeast-Suc-atMost atLeast0AtMost [symmetric])

lemma

assumes SORT-CONSTRAINT('a::{idom,real-normed-div-algebra})

and \( 1 \leq n \)

shows finite-roots-unity: finite \( \{z::'a.\ z^n = 1\} \)

and card-roots-unity: card \( \{z::'a.\ z^n = 1\} \leq n \)

using polyfun-rootbound [of \( \lambda i.\ if\ i=0\ then\ -1\ else\ if\ i=n\ then\ 1\ else\ 0\ n\ n\] assms(2)

by (auto simp: root-polyfun [OF assms(2)])

113.23 Hyperbolic functions

definition sinh :: 'a::{banach, real-normed-algebra-1} ⇒ 'a where
sinh \( x \) = \((exp\ x - exp\ (-x)) / R\ 2\)

definition cosh :: 'a::{banach, real-normed-algebra-1} ⇒ 'a where
cosh \( x \) = \((exp\ x + exp\ (-x)) / R\ 2\)

definition tanh :: 'a::{banach, real-normed-field} ⇒ 'a where
tanh \( x \) = sinh \( x \) / cosh \( x \)

definition arsinh :: 'a::{banach, real-normed-algebra-1, ln} ⇒ 'a where
arsinh \( x \) = ln \((x + (x^2 + 1) \ pow of-reald (1/2))\)

definition arcosh :: 'a::{banach, real-normed-algebra-1, ln} ⇒ 'a where
arcosh \( x \) = ln \((x + (x^2 - 1) \ pow of-reald (1/2))\)

definition artanh :: 'a::{banach, real-normed-field, ln} ⇒ 'a where
artanh \( x \) = ln \(((1 + x) / (1 - x)) / 2\)
lemma arsinh-0 [simp]: \( \text{arsinh } 0 = 0 \)
by (simp add: arsinh-def)

lemma arcosh-1 [simp]: \( \text{arcosh } 1 = 0 \)
by (simp add: arcosh-def)

lemma artanh-0 [simp]: \( \text{artanh } 0 = 0 \)
by (simp add: artanh-def)

lemma tanh-altdef:
\[ \text{tanh } x = \frac{e^x - e^{(-x)}}{e^x + e^{(-x)}} \]
proof –
have tanh x = (2 * R sinh x) / (2 * R cosh x)
  by (simp add: tanh-def scaleR-conv-of-real)
also have 2 * R sinh x = exp x - exp (-x)
  by (simp add: sinh-def)
also have 2 * R cosh x = exp x + exp (-x)
  by (simp add: cosh-def)
finally show ?thesis .
qed

lemma tanh-real-altdef: \( \text{tanh } (x :: real) = \frac{1 - e^{-2*x}}{1 + e^{-2*x}} \)
proof –
have \[ e^{2*x} = e^x \cdot e^x \cdot e^{(x \cdot 2)} = e^x \cdot e^x \]
  by (subst exp-add [symmetric]; simp)+
have tanh x = (2 * exp (-x) * sinh x) / (2 * exp (-x) * cosh x)
  by (simp add: tanh-def)
also have 2 * exp (-x) * sinh x = 1 - exp (-2*x)
  by (simp add: exp-minus field-simps sinh-def)
also have 2 * exp (-x) * cosh x = 1 + exp (-2*x)
  by (simp add: exp-minus field-simps cosh-def)
finally show ?thesis .
qed

lemma sinh-converges: \( \lambda n. \text{if even } n \text{ then } 0 \text{ else } x ^ n / R \text{ fact } n \) sums sinh x
proof –
have \( \lambda n. (x ^ n / R \text{ fact } n - (-x) ^ n / R \text{ fact } n) / R \text{ 2} \) sums sinh x
  unfolding sinh-def by (intro sums-scaleR-right sums-diff exp-converges)
also have \( \lambda n. (x ^ n / R \text{ fact } n - (-x) ^ n / R \text{ fact } n) / R \text{ 2} \) =
  \( \lambda n. \text{if even } n \text{ then } 0 \text{ else } x ^ n / R \text{ fact } n \) by auto
finally show ?thesis .
qed

lemma cosh-converges: \( \lambda n. \text{if even } n \text{ then } x ^ n / R \text{ fact } n \text{ else } 0 \) sums cosh x
proof –
have \( \lambda n. (x ^ n / R \text{ fact } n + (-x) ^ n / R \text{ fact } n) / R \text{ 2} \) sums cosh x
unfolding cosh-def by (intro sums-scaleR-right sums-add exp-converges)
also have \((\lambda n. (x ^ n / R \text{ fact } n + (-x) ^ n / R \text{ fact } n)) / R 2) =
(\lambda n. if even n then x ^ n / R \text{ fact } n else 0) \text{ by auto}

finally show \(?thesis .

qed

lemma sinh-0 [simp]: sinh 0 = 0
  by (simp add: sinh-def)

lemma cosh-0 [simp]: cosh 0 = 1
proof –
  have cosh 0 = (1/2) * R (1 + 1) \text{ by (simp add: cosh-def)}
  also have \(
\ldots = 1 \)
  \text{ by (rule scaleR-half-double)}
  finally show \(?thesis .

qed

lemma tanh-0 [simp]: tanh 0 = 0
  by (simp add: tanh-def)

lemma sinh-minus [simp]: sinh (-x) = -sinh x
  by (simp add: sinh-def algebra-simps)

lemma cosh-minus [simp]: cosh (-x) = cosh x
  by (simp add: cosh-def algebra-simps)

lemma tanh-minus [simp]: tanh (-x) = -tanh x
  by (simp add: tanh-def)

lemma sinh-ln-real: \( x > 0 \implies \sinh (ln x :: real) = (x - inverse x) / 2 \)
  by (simp add: sinh-def exp-minus)

lemma cosh-ln-real: \( x > 0 \implies \cosh (ln x :: real) = (x + inverse x) / 2 \)
  by (simp add: cosh-def exp-minus)

lemma tanh-ln-real:
  \( \tanh (ln x :: real) = (x ^ 2 - 1) / (x ^ 2 + 1) \) \text{ if } x > 0
proof –
  from that have \((x * 2 - inverse x * 2) * (x ^ 2 + 1) =
  (x ^ 2 - 1) * (2 * x + 2 * inverse x) \)
  \text{ by (simp add: field-simps power2-eq-square)}
moreover have \( x ^ 2 + 1 > 0 \)
  \text{ using that by (simp add: ac-simps add-pos-nonneg)}
moreover have \( 2 * x + 2 * inverse x > 0 \)
  \text{ using that by (simp add: add-pos-pos)}
ultimately have \((x * 2 - inverse x * 2) /
(2 * x + 2 * inverse x) =
(x ^ 2 - 1) / (x ^ 2 + 1) \)
  \text{ by (simp add: frac-eq-eq)}
with that show \(?thesis
proof

lemma has-field-derivative-scaleR-right [derivative-intros]:
(\f\ has-field-derivative D) F \Longrightarrow ((\lambda x. c *R f x) has-field-derivative (c *R D)) F
unfolding has-field-derivative-def
using has-derivative-scaleR-right[of f \lambda x. D * x F c]
by (simp add: mult-scaleR-left [symmetric] del: mult-scaleR-left)

lemma has-field-derivative-sinh [THEN DERIV-chain2, derivative-intros]:
(sinh has-field-derivative cosh x) (at (x :: 'a :: {banach, real-normed-field}))
unfolding sinh-def cosh-def by (auto intro!: derivative-eq-intros)

lemma has-field-derivative-cosh [THEN DERIV-chain2, derivative-intros]:
(cosh has-field-derivative sinh x) (at (x :: 'a :: {banach, real-normed-field}))
unfolding sinh-def cosh-def by (auto intro!: derivative-eq-intros)

lemma has-field-derivative-tanh [THEN DERIV-chain2, derivative-intros]:
cosh x \neq 0 \Longrightarrow (\tanh has-field-derivativie 1 = - \tanh x \cdot 2)
(at (x :: 'a :: {banach, real-normed-field}))
unfolding tanh-def by (auto intro!: derivative-eq-intros simp: power2-eq-square field-split-simps)

lemma has-derivative-sinh [derivative-intros]:
fixes g :: 'a \Rightarrow ('a :: {banach, real-normed-field})
assumes (g has-derivative (\lambda x. Db * x)) (at x within s)
shows ((\lambda x. sinh (g x)) has-derivative (\lambda y. (cosh (g x) * Db) * y)) (at x within s)
proof
  have ((\lambda x. - g x) has-derivative (\lambda y. -(Db * y))) (at x within s)
    using assms by (intro derivative-intros)
  also have (\lambda y. -(Db * y)) = (\lambda x. -(Db) * x) by (simp add: fun-eq-iff)
  finally have ((\lambda x. sinh (g x)) has-derivative
    (\lambda y. (exp (g x) * Db * y - exp (-g x) * (-(Db) * y)) / R 2)) (at x within s)
    unfolding sinh-def by (intro derivative-intros assms)
  also have (\lambda y. (exp (g x) * Db * y - exp (-g x) * (-(Db) * y)) / R 2) = (\lambda y.
    (cosh (g x) * Db) * y)
    by (simp add: fun-eq-iff cosh-def algebra-simps)
  finally show ?thesis.
qed

lemma has-derivative-cosh [derivative-intros]:
fixes g :: 'a \Rightarrow ('a :: {banach, real-normed-field})
assumes (g has-derivative (\lambda y. Db * y)) (at x within s)
shows ((\lambda x. cosh (g x)) has-derivative (\lambda y. (sinh (g x) * Db) * y)) (at x within s)
proof
  have ((\lambda x. - g x) has-derivative (\lambda y. -(Db * y))) (at x within s)
    using assms by (intro derivative-intros)
also have \((\lambda y. - (D_b \ast y)) = (\lambda y. (-D_b) \ast y)\) by (simp add: fun-eq-iff)
finally have \(((\lambda x. \cosh (g \cdot x)) \ast \text{has-derivative} (\lambda y. \exp (g \cdot x) \ast (\exp (-g \cdot x) \ast (\exp (-D_b \ast y) \ast R) / 2)) \at x \in s)\)
unfolding \(\cosh \text{-def}\) by (intro derivative-intros assms)
also have \((\lambda y. \exp (g \cdot x) \ast (\exp (-g \cdot x) \ast (\exp (-D_b \ast y) \ast R) / 2) = (\lambda y. (\sinh (g \cdot x) \ast D_b) \ast y)\)
by (simp add: fun-eq-iff \(\sinh\) \text{-def} algebra-simps)
finally show \(?thesis\).
qed

lemma \(\sinh\text{-plus-cosh}: \sinh x + \cosh x = \exp x\)
proof –
have \(\sinh x + \cosh x = (1/2) \ast_R (\exp x + \exp x)\)
by (simp add: \(\sinh\) \text{-def} \(\cosh\) \text{-def} algebra-simps)
also have \(\ldots = \exp x\) by (rule scaleR-half-double)
finally show \(?thesis\).
qed

lemma \(\cosh\text{-plus-sinh}: \cosh x + \sinh x = \exp x\)
by (subst add.commute) (rule \(\sinh\text{-plus-cosh}\))

lemma \(\cosh\text{-minus-sinh}: \cosh x - \sinh x = \exp (-x)\)
proof –
have \(\cosh x - \sinh x = (1/2) \ast_R (\exp (-x) + \exp (-x))\)
by (simp add: \(\sinh\) \text{-def} \(\cosh\) \text{-def} algebra-simps)
also have \(\ldots = \exp (-x)\) by (rule scaleR-half-double)
finally show \(?thesis\).
qed

lemma \(\sinh\text{-minus-cosh}: \sinh x - \cosh x = - \exp (-x)\)
using \(\cosh\text{-minus-sinh}[\text{of} x]\) by (simp add: algebra-simps)

context
  fixes \(x::'a::\{\text{real-normed-field, banach}\}\)
begin

lemma \(\sinh\text{-zero-iff}: \sinh x = 0 \iff \exp x \in \{1, -1\}\)
by (auto simp: \(\sinh\) \text{-def} field-simps exp-minus power2-eq-square square-eq-1-iff)

lemma \(\cosh\text{-zero-iff}: \cosh x = 0 \iff \exp x ^ 2 = -1\)
by (auto simp: \(\cosh\) \text{-def} exp-minus field-simps power2-eq-square eq-neg-iff-add-eq-0)

lemma \(\cosh\text{-square-eq}: \cosh x ^ 2 = \sinh x ^ 2 + 1\)
by (simp add: \(\cosh\) \text{-def} \(\sinh\) \text{-def} algebra-simps power2-eq-square exp-add [symmetric]
scaleR-conv-of-real)

lemma \(\sinh\text{-square-eq}: \sinh x ^ 2 = \cosh x ^ 2 - 1\)
by (simp add: \(\cosh\) \text{-square-eq}
lemma hyperbolic-pythagoras: \( \cosh x \sim 2 \sim \sinh x \sim 2 = 1 \)

by (simp add: cosh-square-eq)

lemma sinh-add: \( \sinh (x + y) = \sinh x \times \cosh y + \cosh x \times \sinh y \)

by (simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric])

lemma sinh-diff: \( \sinh (x - y) = \sinh x \times \cosh y - \cosh x \times \sinh y \)

by (simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric])

lemma cosh-add: \( \cosh (x + y) = \cosh x \times \cosh y + \sinh x \times \sinh y \)

by (simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric])

lemma cosh-diff: \( \cosh (x - y) = \cosh x \times \cosh y - \sinh x \times \sinh y \)

by (simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric])

lemma tanh-add: \( \tanh (x + y) = \frac{\tanh x \times \cosh y + \cosh x \times \sinh y}{1 + \tanh x \times \tanh y} \)

if \( \cosh x \neq 0 \) \( \cosh y \neq 0 \)

proof –

have \( (\cosh x \times \cosh y + \sinh x \times \sinh y) \times (1 + \sinh x \times \sinh y / (\cosh x \times \cosh y)) = \)

\( (\cosh x \times \cosh y + \sinh x \times \sinh y) \times ((\sinh x \times \cosh y + \sinh y \times \cosh x) / (\cosh y \times \cosh x)) \)

using that by (simp add: field-split-simps)

also have \( (\sinh x \times \cosh y + \sinh y \times \cosh x) / \cosh y \times \cosh x = \sinh x / \cosh x + \sinh y / \cosh y \)

using that by (simp add: field-split-simps)

finally have \( (\sinh x \times \cosh y + \cosh x \times \sinh y) \times (1 + \sinh x \times \sinh y / \cosh x \times \cosh y) = \)

\( (\sinh x \times \cosh y + \sinh y \times \cosh x) / \cosh y \times \cosh x + \sinh y \times \cosh x \times \sinh y) \)

by simp

then show \(?thesis \)

using that by (auto simp add: tanh-def sinh-add cosh-add eq-divide-eq)

(simp-all add: field-split-simps)

qed

lemma sinh-double: \( \sinh (2 \times x) = 2 \times \sinh x \times \cosh x \)

using sinh-add[of x] by simp

lemma cosh-double: \( \cosh (2 \times x) = \cosh x \sim 2 + \sinh x \sim 2 \)

using cosh-add[of x] by (simp add: power2-ec-square)

end

lemma sinh-field-def: \( \sinh z = (\exp z - \exp (-z)) / (2 :: 'a :: \{banach, real-normed-field\}) \)

by (simp add: sinh-def scaleR-conv-of-real)

lemma cosh-field-def: \( \cosh z = (\exp z + \exp (-z)) / (2 :: 'a :: \{banach, real-normed-field\}) \)
by \( \text{simp add: cosh-def scaleR-conv-of-real} \)

113.23.1 More specific properties of the real functions

\textbf{lemma plus-inverse-ge-2:}
\begin{itemize}
  \item fixes \( x :: \text{real} \)
  \item assumes \( x > 0 \)
  \item shows \( x + \text{inverse } x \geq 2 \)
\end{itemize}
\textbf{proof –}
\begin{itemize}
  \item have \( 0 \leq (x - 1)^2 \) by simp
  \item also have \( \ldots = x^2 - 2x + 1 \) by (simp add: power2-eq-square algebra-simps)
  \item finally show \( \text{thesis using assms by (simp add: field-sims power2-eq-square)} \)
\end{itemize}
\textbf{qed}

\textbf{lemma sinh-real-nonneg-iff [simp]:}
\( \text{sinh } (x :: \text{real}) \geq 0 \iff x \geq 0 \)
by (simp add: sinh-def)

\textbf{lemma sinh-real-pos-iff [simp]:}
\( \text{sinh } (x :: \text{real}) > 0 \iff x > 0 \)
by (simp add: sinh-def)

\textbf{lemma sinh-real-nonpos-iff [simp]:}
\( \text{sinh } (x :: \text{real}) \leq 0 \iff x \leq 0 \)
by (simp add: sinh-def)

\textbf{lemma sinh-real-neg-iff [simp]:}
\( \text{sinh } (x :: \text{real}) < 0 \iff x < 0 \)
by (simp add: sinh-def)

\textbf{lemma cosh-real-ge-1:}
\( \text{cosh } (x :: \text{real}) \geq 1 \)
\textbf{using plus-inverse-ge-2[of exp x] by (simp add: cosh-def exp-minus)}

\textbf{lemma cosh-real-pos [simp]:}
\( \text{cosh } (x :: \text{real}) > 0 \)
\textbf{using cosh-real-ge-1[of x] by simp}

\textbf{lemma cosh-real-nonneg [simp]:}
\( \text{cosh } (x :: \text{real}) \geq 0 \)
\textbf{using cosh-real-ge-1[of x] by simp}

\textbf{lemma cosh-real-nonzero [simp]:}
\( \text{cosh } (x :: \text{real}) \neq 0 \)
\textbf{using cosh-real-ge-1[of x] by simp}

\textbf{lemma arsinh-real-def:}
\( \text{arsinh } (x :: \text{real}) = \ln (x + \sqrt{x^2 + 1}) \)
\textbf{by (simp add: arsinh-def powr-half-sqrt)}

\textbf{lemma arcosh-real-def:}
\( x \geq 1 \implies \text{arcosh } (x :: \text{real}) = \ln (x + \sqrt{x^2 - 1}) \)
\textbf{by (simp add: arcosh-def powr-half-sqrt)}

\textbf{lemma arsinh-real-aux:}
\( 0 < x + \sqrt{x^2 + 1 :: \text{real}} \)
\textbf{proof (cases \( x < 0 \))}
\begin{itemize}
  \item case \text{True}
  \item have \( (x) \cdot 2 = x \cdot 2 \) by simp
  \item also have \( x \cdot 2 < x \cdot 2 + 1 \) by simp
\end{itemize}
finally have \( \sqrt{(-x)^2} < \sqrt{x^2 + 1} \)
by (rule real-sqrt-less-mono)
thus \( \thesis \) using True by simp
qed (auto simp: add-nonneg-pos)

lemma arsinh-minus-real [simp]: \( \text{arsinh} (-x::\text{real}) = -\text{arsinh} x \)
proof –
  have \( \text{arsinh} (-x) = \ln (\sqrt{x^2 + 1} - x) \)
    by (simp add: arsinh-real-def)
  also have \( \sqrt{x^2 + 1} - x = \inverse (\sqrt{x^2 + 1} + x) \)
    using arsinh-real-aux[of x] by (simp add: field-split-simps algebra-simps power2-eq-square)
  also have \( \ln \ldots = -\text{arsinh} x \)
    using arsinh-real-aux[of x]
    by (simp add: field-simps arsinh-real-def ln-inverse)
finally show \( \thesis \).
qed

lemma artanh-minus-real [simp]:
assumes abs x < 1
shows \( \text{artanh} (-x::\text{real}) = -\text{artanh} x \)
using assms by (simp add: artanh-def ln-div field-simps)

lemma sinh-less-cosh-real: \( \text{sinh} (x :: \text{real}) < \text{cosh} x \)
by (simp add: sinh-def cosh-def)

lemma sinh-le-cosh-real: \( \text{sinh} (x :: \text{real}) \leq \text{cosh} x \)
by (simp add: sinh-def cosh-def)

lemma tanh-real-lt-1: \( \text{tanh} (x :: \text{real}) < 1 \)
by (simp add: tanh-def sinh-less-cosh-real)

lemma tanh-real-gt-neg1: \( \text{tanh} (x :: \text{real}) > -1 \)
proof –
  have \( -\cosh x < \sinh x \) by (simp add: sinh-def cosh-def field-split-simps)
  thus \( \thesis \) by (simp add: field-simps)
qed

lemma tanh-real-bounds: \( \text{tanh} (x :: \text{real}) \in \{-1<..<1\} \)
using tanh-real-lt-1 tanh-real-gt-neg1 by simp

context
fixes x :: real
begin

lemma arsinh-sinh-real: \( \text{arsinh} (\text{sinh} x) = x \)
by (simp add: arsinh-real-def powr-def sinh-square-eq sinh-plus-cosh)

lemma arcosh-cosh-real: \( x \geq 0 \implies \text{arcosh} (\text{cosh} x) = x \)
by (simp add: arcosh-real-def powr-def cosh-square-eq cosh-real-ge-1 cosh-plus-sinh)
lemma artanh-tanh-real: artanh (tanh x) = x
proof
  have artanh (tanh x) = ln (cosh x * (cosh x + sinh x) / (cosh x * (cosh x - sinh x))) / 2
    by (simp add: artanh-def tanh-def field-split-simps)
  also have cosh x * (cosh x + sinh x) / (cosh x * (cosh x - sinh x)) =
    (cosh x + sinh x) / (cosh x - sinh x) by simp
  also have ... = (exp x)^2
    by (simp add: cosh-plus-sinh cosh-minus-sinh exp-minus field-simps power2-eq-square)
  also have ln ((exp x)^2) / 2 = x by (simp add: ln-realpow)
  finally show ?thesis.
qed
lemma sinh-real-zero-iff [simp]: sinh x = 0 <-> x = 0
  by (metis arsinh-0 arsinh-sinh-real sinh-0)
lemma cosh-real-one-iff [simp]: cosh x = 1 <-> x = 0
  by (smt (verit, best) Transcendental.arcosh-cosh-real cosh-0 cosh-minus)
lemma tanh-real-nonneg-iff [simp]: tanh x >= 0 <-> x >= 0
  by (simp add: tanh-def field-simps)
lemma tanh-real-pos-iff [simp]: tanh x > 0 <-> x > 0
  by (simp add: tanh-def field-simps)
lemma tanh-real-nonpos-iff [simp]: tanh x <= 0 <-> x <= 0
  by (simp add: tanh-def field-simps)
lemma tanh-real-neg-iff [simp]: tanh x < 0 <-> x < 0
  by (simp add: tanh-def field-simps)
lemma tanh-real-zero-iff [simp]: tanh x = 0 <-> x = 0
  by (simp add: tanh-def field-simps)
end
lemma sinh-real-strict-mono: strict-mono (sinh :: real => real)
  by (rule pos-deriv-imp-strict-mono derivative-intros)+ auto
lemma cosh-real-strict-mono:
  assumes 0 <= x and x < (y::real)
  shows cosh x < cosh y
proof
  from assms have \exists z>x. z < y \land cosh y - cosh x = (y - x) * sinh z
    by (intro MVT2) (auto dest: connectedD-interval intro: derivative-eq-intros)
  then obtain z where z > x z < y cosh y - cosh x = (y - x) * sinh z by blast
  note \langle cosh y - cosh x = (y - x) * sinh z\rangle
  also from \langle z > x\rangle and assms have (y - x) * sinh z > 0 by (intro mult-pos-pos)
auto

finally show \( \cosh x < \cosh y \) by simp

qed

lemma \( \tanh\)-real-strict-mono: strict-mono \((\tanh :: \mathbb{R} \Rightarrow \mathbb{R})\)

proof

  have *: \( \tanh x ^ 2 < 1 \) for \( x :: \mathbb{R} \)
    using \( \tanh\)-real-bounds[of \( x \)] by (simp add: abs-square-less-1 abs-if)
  show ?thesis
    by (rule pos-deriv-imp-strict-mono) (insert *, auto intro!: derivative-intros)

qed

lemma \( \sinh\)-real-abs [simp]: \( \sinh (abs x :: \mathbb{R}) = abs (\sinh x) \)
by (simp add: abs-if)

lemma \( \cosh\)-real-abs [simp]: \( \cosh (abs x :: \mathbb{R}) = \cosh x \)
by (simp add: abs-if)

lemma \( \tanh\)-real-abs [simp]: \( \tanh (abs x :: \mathbb{R}) = abs (\tanh x) \)
by (auto simp: abs-if)

lemma \( \sinh\)-real-eq-iff [simp]: \( \sinh x = \sinh y \) \( \iff \) \( x = (y :: \mathbb{R}) \)
using \( \sinh\)-real-strict-mono by (simp add: strict-mono-eq)

lemma \( \tanh\)-real-eq-iff [simp]: \( \tanh x = \tanh y \) \( \iff \) \( x = (y :: \mathbb{R}) \)
using \( \tanh\)-real-strict-mono by (simp add: strict-mono-eq)

lemma \( \cosh\)-real-eq-iff [simp]: \( \cosh x = \cosh y \) \( \iff \) \( abs x = abs (y :: \mathbb{R}) \)
proof

  have \( \cosh x = \cosh y \) \( \iff \) \( x = y \) if \( x \geq 0 \) \( y \geq 0 \) for \( x y :: \mathbb{R} \)
    using \( \cosh\)-real-strict-mono[of \( x \) \( y \)] \( \cosh\)-real-strict-mono[of \( y \) \( x \)]
    that
    by (cases \( x \) \( y \) rule: linorder-cases) auto
  from this[of abs \( x \) abs \( y \)] show ?thesis by simp

qed

lemma \( \sinh\)-real-le-iff [simp]: \( \sinh x \leq \sinh y \) \( \iff \) \( x \leq (y :: \mathbb{R}) \)
using \( \sinh\)-real-strict-mono by (simp add: strict-mono-less-eq)

lemma \( \cosh\)-real-nonneg-le-iff: \( x \geq 0 \) \( \Longrightarrow \) \( y \geq 0 \) \( \Longrightarrow \) \( \cosh x \leq \cosh y \) \( \iff \) \( x \leq (y :: \mathbb{R}) \)
using \( \cosh\)-real-strict-mono[of \( x \) \( y \)] \( \cosh\)-real-strict-mono[of \( y \) \( x \)]
by (cases \( x \) \( y \) rule: linorder-cases) auto

lemma \( \cosh\)-real-nonpos-le-iff: \( x \leq 0 \) \( \Longrightarrow \) \( y \leq 0 \) \( \Longrightarrow \) \( \cosh x \leq \cosh y \) \( \iff \) \( x \geq (y :: \mathbb{R}) \)
using \( \cosh\)-real-nonneg-le-iff[of \(-x\) \(-y\)] by simp

lemma \( \tanh\)-real-le-iff [simp]: \( \tanh x \leq \tanh y \) \( \iff \) \( x \leq (y :: \mathbb{R}) \)
using \( \tanh\)-real-strict-mono by (simp add: strict-mono-less-eq)
lemma sinh-real-less-iff [simp]: \( \sinh x < \sinh y \iff x < (y::real) \)
using sinh-real-strict-mono by (simp add: strict-mono-less)

lemma cosh-real-nonneg-less-iff: \( x \geq 0 \implies y \geq 0 \implies \cosh x < \cosh y \iff x < (y::real) \)
using cosh-real-strict-mono[of x y] cosh-real-strict-mono[of y x]
by (cases x y rule: linorder-cases) auto

lemma cosh-real-nonneg-less-iff: \( x \leq 0 \implies y \leq 0 \implies \cosh x < \cosh y \iff x > (y::real) \)
using cosh-real-strict-mono[of -x -y] by simp

lemma tanh-real-less-iff [simp]: \( \tanh x < \tanh y \iff x < (y::real) \)
using tanh-real-strict-mono by (simp add: strict-mono-less)

113.23.2 Limits

lemma sinh-real-at-top: filterlim (\( \sinh :: \text{real} \Rightarrow \text{real} \)) at-top at-top
proof
have \( *: ((\lambda x. -exp (-x)) \longrightarrow (-0::real)) \text{ at-top} \)
by (intro tendsto-minus filterlim-compose[OF exp-at-bot filterlim-uminus-at-bot-at-top])
have filterlim (\( \lambda x. (1/2) * (-exp (-x) + exp x) :: \text{real} \)) at-top at-top
by (rule filterlim-tendsto-pos-mult-at-top[OF - -
    [filterlim-tendsto-add-at-top[OF OF *]] tendsto-const)+ (auto simp: exp-at-top)
also have \( (\lambda x. (1/2) * (-exp (-x) + exp x) :: \text{real}) = \sinh \)
by (simp add: fan-eq-iff sinh-def)
finally show \( ?\text{thesis} \).

qed

lemma sinh-real-at-bot: filterlim (\( \sinh :: \text{real} \Rightarrow \text{real} \)) at-bot at-bot
proof
have filterlim (\( \lambda x. -\sinh x :: \text{real} \)) at-bot at-top
by (simp add: filterlim-uminus-at-top [symmetric] sinh-real-at-top)
also have \( (\lambda x. -\sinh x :: \text{real}) = (\lambda x. \sinh (-x)) \) by simp
finally show \( ?\text{thesis} \) by (subst filterlim-at-bot-mirror)

qed

lemma cosh-real-at-top: filterlim (\( \cosh :: \text{real} \Rightarrow \text{real} \)) at-top at-top
proof
have \( *: ((\lambda x. exp (-x)) \longrightarrow (0::real)) \text{ at-top} \)
by (intro filterlim-compose[OF exp-at-bot filterlim-uminus-at-bot-at-top])
have filterlim (\( \lambda x. (1/2) * (exp (-x) + exp x) :: \text{real} \)) at-top at-top
by (rule filterlim-tendsto-pos-mult-at-top[OF - -
    [filterlim-tendsto-add-at-top[OF OF *]] tendsto-const]+ (auto simp: exp-at-top)
also have \( (\lambda x. (1/2) * (exp (-x) + exp x) :: \text{real}) = \cosh \)
by \(\text{simp add: fun-eq-iff cosh-def}\)
finally show ?thesis .
qed

lemma cosh-real-at-bot: \(\text{filterlim (cosh :: real \Rightarrow real) at-top at-bot}\)
proof –
  have \(\text{filterlim (\lambda x. \cosh (-x) :: real) at-top at-top}\)
  by (simp add: cosh-real-at-top)
  thus ?thesis by (subst \text{filterlim-at-bot-mirror})
qed

lemma tanh-real-at-top: \((\tanh \longrightarrow (1::real)) at-top\)
proof –
  have \(\((\lambda x::real. (1 - \exp (-2 * \, x)) / (1 + \exp (-2 * x))) \longrightarrow (1 - 0) / (1 + 0))\) at-top
  by (intro \text{tendsto-intros} \text{filterlim-compose}[\text{OF exp-at-bot}]
      \text{filterlim-tendsto-neg-mult-at-bot}[\text{OF tendsto-const}] \text{filterlim-ident}) auto
  also have \((\lambda x::real. (1 - \exp (-2 * \, x)) / (1 + \exp (-2 * x))) = \tanh\)
  by (rule ext) (simp add: \text{tanh-real-altdef})
  finally show ?thesis by simp
qed

lemma tanh-real-at-bot: \((\tanh \longrightarrow (-1::real)) at-bot\)
proof –
  have \(\((\lambda x::real. -\tanh x) \longrightarrow -1) at-top\)
  by (intro \text{tendsto-minus} \text{tanh-real-at-top})
  also have \((\lambda x. -\tanh x :: real) = (\lambda x. \tanh (-x))\) by simp
  finally show ?thesis by (subst \text{filterlim-at-bot-mirror})
qed

113.23.3 Properties of the inverse hyperbolic functions

lemma isCont-sinh: \(\text{isCont sinh (x :: 'a :: \{real-normed-field, banach\})}\)
unfolding sinh-def [abs-def] by (auto intro!: \text{continuous-intros})

lemma isCont-cosh: \(\text{isCont cosh (x :: 'a :: \{real-normed-field, banach\})}\)
unfolding cosh-def [abs-def] by (auto intro!: \text{continuous-intros})

lemma isCont-tanh: \(\cosh x \neq 0 \Rightarrow \text{isCont tanh (x :: 'a :: \{real-normed-field, banach\})}\)
unfolding tanh-def [abs-def] by (auto intro!: \text{continuous-intros} isCont-divide \text{isCont-sinh} isCont-cosh)

lemma continuous-on-sinh [\text{continuous-intros}]:
  fixes f :: - \Rightarrow 'a::\{real-normed-field,banach\}
  assumes continuous-on A f
  shows continuous-on A (\lambda x. \sinh (f x))
  unfolding sinh-def using assms by (intro continuous-intros)
THEORY "Transcendental" 2488

lemma continuous-on-cosh [continuous-intros]:
  fixes f :: -, a :: {real-normed-field, banach}
  assumes continuous-on A f
  shows continuous-on A (λx. cosh (f x))
  unfolding cosh-def using assms by (intro continuous-intros)

lemma continuous-sinh [continuous-intros]:
  fixes f :: -, a :: {real-normed-field, banach}
  assumes continuous F f
  shows continuous F (λx. sinh (f x))
  unfolding sinh-def using assms by (intro continuous-intros)

lemma continuous-cosh [continuous-intros]:
  fixes f :: -, a :: {real-normed-field, banach}
  assumes continuous F f
  shows continuous F (λx. cosh (f x))
  unfolding cosh-def using assms by (intro continuous-intros)

lemma continuous-at-within-tanh [continuous-intros]:
  fixes f :: - ⇒' a :: {real-normed-field, banach}
  assumes continuous (at x within A) f cosh (f x) ≠ 0
  shows continuous (at x within A) (λx. tanh (f x))
  unfolding tanh-def using assms by (intro continuous-intros continuous-divide) auto

lemma continuous-tanh [continuous-intros]:
  fixes f :: - ⇒' a :: {real-normed-field, banach}
  assumes continuous F f cosh (Lim F (λx. x)) ≠ 0
  shows continuous F (λx. tanh (f x))
  unfolding tanh-def using assms by (intro continuous-intros continuous-divide) auto

lemma tendsto-sinh [tendsto-intros]:
  fixes f :: - ⇒' a :: {real-normed-field, banach}
  shows (f −→ a) F ⇒ (λx. sinh (f x)) −→ sinh a) F
  by (rule isCont-tendsto-compose [OF isCont-sinh])

lemma tendsto-cosh [tendsto-intros]:
  fixes f :: - ⇒' a :: {real-normed-field, banach}
  shows (f −→ a) F ⇒ (λx. cosh (f x)) −→ cosh a) F
  by (rule isCont-tendsto-compose [OF isCont-cosh])

lemma tendsto-tanh [tendsto-intros]: 
fixes $f :: \Rightarrow a::\{\text{real-normed-field,banach}\}$
shows $(f \longrightarrow a) F \implies \cosh a \neq 0 \implies ((\lambda x. \tanh (f \cdot x)) \longrightarrow \tanh a) F$
by (rule isCont-tendsto-compose [OF isCont-tanh])

lemma arsinh-real-has-field-derivative [derivative-intros]:
fixes $x :: \text{real}$
shows $(\text{arsinh has-field-derivative } (1 / (\sqrt{(x^2 + 1)}))) (at \ x \ \text{within} \ A)$
proof -
have pos: $1 + x^2 > 0$ by (intro add-pos-nonneg) auto
from pos arsinh-real-aux [of $x$] show ?thesis unfolding arsinh-def [abs-def]
by (auto intro!: derivative-eq-intros simp: powr-minus powr-half-sqrt field-split-simps)
qed

lemma arcosh-real-has-field-derivative [derivative-intros]:
fixes $x :: \text{real}$
assumes $x > 1$
shows $(\text{arcosh has-field-derivative } (1 / (\sqrt{(x^2 - 1)}))) (at \ x \ \text{within} \ A)$
proof -
from assms have $x + \sqrt{(x^2 - 1)} > 0$ by (simp add: add-pos-pos)
thus ?thesis using assms unfolding arcosh-def [abs-def]
by (auto intro!: derivative-eq-intros simp: powr-minus powr-half-sqrt field-split-simps power2-eq-1-iff)
qed

lemma artanh-real-has-field-derivative [derivative-intros]:
$(\text{artanh has-field-derivative } (1 / (1 - x))) (at \ x \ \text{within} \ A) \text{ if } |x| < 1 \ \text{for} \ x :: \text{real}$
proof -
from that have $-1 < x < 1$ by linarith+
hence $(\text{artanh has-field-derivative } (4 - 4 \cdot x) / ((1 + x) \cdot (1 - x) \cdot (1 - x) \cdot 4)) (at \ x \ \text{within} \ A)$ unfolding artanh-def [abs-def]
by (auto intro!: derivative-eq-intros simp: powr-minus powr-half-sqrt)
also have $(4 - 4 \cdot x) / ((1 + x) \cdot (1 - x) \cdot (1 - x) \cdot 4) = 1 / ((1 + x) \cdot (1 - x))$
using $(-1 < x) \cdot (x < 1)$ by (simp add: frac-eq-eq)
also have $(1 + x) \cdot (1 - x) = 1 - x^2$
by (simp add: algebra-simps power2-eq-square)
finally show ?thesis.
qed

lemma cosh-double-cosh:
$cosh (2 \cdot x :: a :: \{\text{banach, real-normed-field}\}) = 2 \cdot (cosh x)^2 - 1$
using cosh-double[of $x$] by (simp add: sinh-square-eq)

lemma sinh-multiple-reduce:
$\sinh (x \cdot \text{numeral n} :: a :: \{\text{real-normed-field, banach}\}) =$
$\sinh x \cdot \cosh (x \cdot \text{of-nat (pred-numeral n)) + \cosh x \cdot \sinh (x \cdot \text{of-nat}$
THEORY "Transcendental"

(proof)

have numeral n = of-nat (pred-numeral n) + (1 :: 'a)
  by (metis add.commute numeral-eq-Suc of-nat-Suc of-nat-numeral)
also have sinh (x * ... = sinh (x * of-nat (pred-numeral n) + x)
  unfolding of-nat-Suc by (simp add: ring-distrib)
finally show ?thesis
  by (simp add: sinh-add)

lemma cosh-multiple-reduce:
cosh (x * numeral n :: 'a :: {real-normed-field, banach}) =
cosh (x * of-nat (pred-numeral n)) * cosh x + sinh (x * of-nat (pred-numeral n)) * sinh x
(proof)

have numeral n = of-nat (pred-numeral n) + (1 :: 'a)
  by (metis add.commute numeral-eq-Suc of-nat-Suc of-nat-numeral)
also have cosh (x * ...) = cosh (x * of-nat (pred-numeral n) + x)
  unfolding of-nat-Suc by (simp add: ring-distrib)
finally show ?thesis
  by (simp add: cosh-add)

lemma cosh-arcosh-real [simp]:
  assumes x ≥ (1 :: real)
  shows cosh (arcosh x) = x
(proof)

have eventually (λ::real. cosh t ≥ x) at-top
  using cosh-real-at-top by (simp add: filterlim-at-top)
then obtain t where t ≥ 1 cosh t ≥ x
  by (metis eventually-at-top-linorder linorder-not-le order-le-less)
moreover have isCont cosh (y :: real) for y
  by (intro continuous-intros)
ultimately obtain y where y ≥ 0 x = cosh y
  using IVT[of cosh 0 x t] assms by auto
thus ?thesis
  by (simp add: arcosh-cosh-real)

lemma arcosh-eq-0-iff-real [simp]: x ≥ 1 ⟷ arcosh x = 0 ↔ x = (1 :: real)
  using cosh-arcosh-real by fastforce

lemma arcosh-nonneg-real [simp]:
  assumes x ≥ 1
  shows arcosh (x :: real) ≥ 0
(proof)

have 1 + 0 ≤ x + (x^2 - 1) powr (1 / 2)
  using assms by (intro add-mono) auto
thus ?thesis unfolding arcosh-def by simp
lemma arcosh-real-strict-mono:
  fixes x y :: real
  assumes 1 ≤ x x < y
  shows arcosh x < arcosh y
proof –
  have cosh (arcosh x) < cosh (arcosh y)
    by (subst (1 2) cosh-arcosh-real (use assms in auto)
  thus ?thesis
    using assms by (subst (asm) cosh-real-nonneg-less-iff) auto
qed

lemma arcosh-less-iff-real [simp]:
  fixes x y :: real
  assumes 1 ≤ x 1 ≤ y
  shows arcosh x < arcosh y ←→ x < y
  using arcosh-real-strict-mono[of x y] arcosh-real-strict-mono[of y x] assms
  by (cases x y rule: linorder-cases) auto

lemma arcosh-real-gt-1-iff [simp]: x ≥ 1 =⇒ arcosh x > 0 ←→ x ≠ (1 :: real)
  using arcosh-less-iff-real [of 1 x] by (auto simp del: arcosh-less-iff-real)

lemma sinh-arcosh-real:
proof –
  have eventually (∀t::real. sinh t ≥ x) at-top
    using sinh-real-at-top by (simp add: filterlim-at-top)
  then obtain t where sinh t ≥ x
    by (metis eventually-at-top-linorder linorder-not-le order-le-less)
  moreover have eventually (∀t::real. sinh t ≤ x) at-bot
    using sinh-real-at-bot by (simp add: filterlim-at-bot)
  then obtain t' where t' ≤ t sinh t' ≤ x
    by (metis eventually-at-bot-linorder nle-le)
  moreover have isCont sinh (y :: real) for y
    by (intro continuous-intros)
  ultimately obtain y where x = sinh y
    using IVT[of sinh t' x t] by auto
  thus ?thesis
    by (simp add: arsinh-sinh-real)
qed

lemma arsinh-real-strict-mono:
  fixes x y :: real
  assumes x < y
  shows arsinh x < arsinh y
proof
  have \( \sinh (\arsinh x) < \sinh (\arsinh y) \)
    by (subst (1 2) \( \sinh-\arsinh\)-real) (use assms in auto)
  thus \(?thesis\)
    using assms by (subst (asm) \( \sinh\)-real-less-iff) auto
qed

lemma \( \arsinh\)-less-iff-real [simp]:
  fixes \( x \) \( y \): real
  shows \( \arsinh x < \arsinh y \longleftrightarrow x < y \)
    using \( \arsinh\)-real-strict-mono[of \( x \) \( y \)] \( \arsinh\)-real-strict-mono[of \( y \) \( x \)]
    by (cases \( x \) \( y \) rule: linorder-cases) auto

lemma \( \arsinh\)-real-eq-0-iff [simp]:
  \( \arsinh x = 0 \longleftrightarrow x = (0 :: \text{real}) \)
    by (metis \( \arsinh\)-0 \( \sinh\)-\( \arsinh\)-real)

lemma \( \arsinh\)-real-pos-iff [simp]:
  \( \arsinh x > 0 \longleftrightarrow x > (0 :: \text{real}) \)
    using \( \arsinh\)-less-iff-real[of \( 0 \) \( x \)] by simp del: \( \arsinh\)-less-iff-real

lemma \( \arsinh\)-real-neg-iff [simp]:
  \( \arsinh x < 0 \longleftrightarrow x < (0 :: \text{real}) \)
    using \( \arsinh\)-less-iff-real[of \( x \) \( 0 \)] by simp del: \( \arsinh\)-less-iff-real

lemma \( \cosh\)-\( \arsinh\)-real:
  \( \cosh (\arsinh x) = \sqrt{(x^2 + 1)} \)
    by (rule sym, rule real-sqrt-unique) (auto simp: \( \cosh\)-square-eq)

lemma continuous-on-\( \arsinh\) [continuous-intros]:
  assumes \( A \subseteq \{1..\} \)
  shows continuous-on \( A \) \( \arsinh \) \( \Rightarrow \) real
    by (rule DERIV-continuous-on derivative-intros)+

lemma continuous-on-arcosh [continuous-intros]:
  assumes \( A \subseteq \{-1..<1\} \)
  shows continuous-on \( A \) \( \arcosh \) \( \Rightarrow \) real
proof
  have \( \text{pos}: x + \sqrt{x^2 - 2 - 1} > 0 \) if \( x \geq 1 \) for \( x \)
    using that by (intro add-pos-nonneg) auto
  show \(?thesis\)
    unfolding arcosh-def [abs-def]
    by (intro continuous-on-subset [OF - assms] continuous-on-ln continuous-on-add
      continuous-on-id continuous-on-powr')
      (auto dest: pos simp: powr-half-sqrt intro!: continuous-intros)
qed

lemma continuous-on-artanh [continuous-intros]:
  assumes \( A \subseteq \{-1<..<1\} \)
  shows continuous-on \( A \) \( \artanh \) \( \Rightarrow \) real
    unfolding artanh-def [abs-def]
    by (intro continuous-on-subset [OF - assms]) (auto intro!: continuous-intros)

lemma continuous-on-\( \arsinh\)' [continuous-intros]:
fixes $f : \mathbb{R} \Rightarrow \mathbb{R}$
assumes continuous-on $A \ f$
shows continuous-on $A \ (\lambda x. \text{arsinh} \ (f \ x))$
by (rule continuous-on-compose2 [OF continuous-on-arsinh assms]) auto

lemma continuous-on-arcosh' [continuous-intros]:
fixes $f : \mathbb{R} \Rightarrow \mathbb{R}$
assumes continuous-on $A \ f \ \forall x. \ x \in A \Rightarrow f \ x \geq 1$
shows continuous-on $A \ (\lambda x. \text{arcosh} \ (f \ x))$
by (rule continuous-on-compose2 [OF continuous-on-arcosh assms (1) order_refl])
(use assms (2) in auto)

lemma isCont-arcosh [continuous-intros]:
assumes $x > 1$
shows isCont $\text{arcosh} \ (x :: \mathbb{R})$
proof
  have continuous-on $\{1::\mathbb{R}<\ldots\} \ \text{arcosh}$
    by (rule continuous-on-arcosh) auto
  with assms show \ ?thesis by (auto simp: continuous-on-eq-continuous-at)
qed

lemma isCont-artanh [continuous-intros]:
assumes $x > -1 \ x < 1$
shows isCont $\text{artanh} \ (x :: \mathbb{R})$
proof
  have continuous-on $\{-1<..<(1::\mathbb{R})\} \ \text{artanh}$
    by (rule continuous-on-artanh) auto
  with assms show \ ?thesis by (auto simp: continuous-on-eq-continuous-at)
qed

lemma tendsto-arcosh-strong [tendsto-intros]:
fixes $f : - \Rightarrow \mathbb{R}$
assumes $(f \longrightarrow a) \ F \ a \geq 1 \ \text{eventually} \ (\lambda x. \ f \ x \geq 1) \ F$
shows $((\lambda x. \text{arcosh} \ (f \ x)) \longrightarrow \text{arcosh} \ a) \ F$
for $f :: - \Rightarrow \mathbb{R}$
by (rule isCont-tendsto-compose [OF isCont-arcosh])
by (rule continuous-on-tendsto-compose[OF continuous-on-arcosh[OF order.refl]])
(use assms in auto)

lemma tendsto-arcosh:
fixes f :: - ⇒ real
assumes (f −−−→ a) F a > 1
shows ((λx. arcosh (f x)) −−−→ arcosh a) F
by (rule isCont-tendsto-compose [OF isCont-arcosh]) (use assms in auto)

lemma tendsto-arcosh-at-left-1: (arcosh −−−→ 0) (at-right (1::real))
proof –
  have (arcosh −−−→ arcosh 1) (at-right (1::real))
  by (rule tendsto-arcosh-strong) (auto simp: eventually_at intro: exI[of - 1])
thus ?thesis by simp
qed

lemma tendsto-artanh [tendsto-intros]:
fixes f :: 'a ⇒ real
assumes (f −−−→ a) F a > −1 a < 1
shows ((λx. artanh (f x)) −−−→ artanh a) F
by (rule isCont-tendsto-compose [OF isCont-artanh]) (use assms in auto)

lemma continuous-arsinh [continuous-intros]:
continuous F f ⇒ continuous F (λx. arsinh (f x :: real))
unfolding continuous-def by (rule tendsto-arsinh)

lemma continuous-arcosh-strong [continuous-intros]:
assumes continuous F f eventually (λx. f x ≥ 1) F
shows continuous F (λx. arcosh (f x :: real))
proof (cases F = bot)
case False
  show ?thesis
  unfolding continuous-def
  proof (intro tendsto-arcosh-strong)
    show 1 ≤ f (Lim F (λx. x))
      using assms False unfolding continuous-def by (rule tendsto-lowerbound)
  qed (insert assms, auto simp: continuous-def)
qed auto

lemma continuous-arcosh:
continuous F f ⇒ f (Lim F (λx. x)) > 1 ⇒ continuous F (λx. arcosh (f x :: real))
unfolding continuous-def by (rule tendsto-arcosh) auto

lemma continuous-artanh [continuous-intros]:
continuous F f ⇒ f (Lim F (λx. x)) ∈ {−1..<1} ⇒ continuous F (λx. artanh (f x :: real))
unfolding continuous-def by (rule tendsto-artanh) auto
lemma arsinh-real-at-top:
filterlim (arsinh :: real ⇒ real) at-top at-top

proof (subst filterlim-cong [OF refl refl])
  show filterlim (λx. ln (x + sqrt (1 + x^2))) at-top at-top
    by (intro filterlim-compose [OF ln-at-top filterlim-at-top-add-at-top] filterlim-ident
      filterlim-compose [OF sqrt-at-top] filterlim-tendsto-add-at-top [OF tendsto-const])
  qed (auto intro !: eventually-mono [OF eventually-ge-at-top [of 1]] simp: arsinh-real-def add-ac)

lemma arsinh-real-at-bot:
filterlim (arsinh :: real ⇒ real) at-bot at-bot

proof
  have filterlim (λx::real. − arsinh x) at-bot at-top
    by (rule arsinh-real-at-top)
  also have (λx::real. − arsinh x) = (λx. arsinh (−x)) by simp
  finally show ?thesis
    by (rule filterlim-at-bot-mirror)
  qed

lemma arcosh-real-at-top:
filterlim (arcosh :: real ⇒ real) at-top at-top

proof (subst filterlim-cong [OF refl refl])
  show filterlim (λx. ln (x + sqrt (−1 + x^2))) at-top at-top
    by (rule LIM-at-top-divide)
  also have (λx::real. (1 / 2) * ln (1 + x) / (1 − x)) = arcanh
    by (simp add: arcanh-def [abs-def])
  finally show ?thesis .
  qed

lemma arcosh-real-at-right-1:
filterlim (arcosh :: real ⇒ real) at-bot (at-right (−1))
proof

have \(?thesis \leftarrow \rightarrow \) filterlim (\(\lambda x::real. \arctanh x\)) at-top (at-right \((-1)\))
  by (simp add: filterlim-uminus-at-bot)
also have \(\ldots \leftarrow \rightarrow \) filterlim (\(\lambda x::real. \arctanh (-x)\)) at-top (at-right \((-1)\))
  by (intro filterlim-cong refl eventually-mono[OF eventually-at-right-real[of \(-1\) \(\rightarrow \) eventually-at-right-real[of \(-1\)]]) auto
also have \(\ldots \leftarrow \rightarrow \) filterlim (\(\arctanh :: real \Rightarrow real\)) at-top (at-left \(1\))
  by (simp add: filterlim-at-left-to-right)
also have \(\ldots \) by (rule arctanh-real-at-left-1)
finally show ?thesis .
qed

113.24 Simprocs for root and power literals

lemma numeral-powr-numeral-real [simp]:
  \(\text{numeral } m \text{ powr numeral } n = (\text{numeral } m \ ^{\text{numeral } n} :: \text{real})\)
  by (simp add: powr-numeral)

context
begin

private lemma sqrt-numeral-simproc-aux:
  assumes \(m \ast m \equiv n\)
  shows \(\sqrt{\text{numeral } n :: \text{real}} \equiv \text{numeral } m\)
proof

  have \(\text{numeral } n \equiv \text{numeral } m \ast (\text{numeral } m :: \text{real})\) by (simp add: assms [symmetric])
  moreover have \(\sqrt{\ldots} \equiv \text{numeral } m\) by (subst real-sqrt-abs2 simp)
  ultimately show \(\sqrt{\text{numeral } n :: \text{real}} \equiv \text{numeral } m\) by simp
qed

private lemma root-numeral-simproc-aux:
  assumes \(\text{Num. } pow m n \equiv x\)
  shows \(\sqrt[n]{\text{numeral } n :: \text{real}} \equiv \text{numeral } m\)
  by (subst assms [symmetric], subst numeral-pow, subst real-root-pos2) simp-all

private lemma powr-numeral-simproc-aux:
  assumes \(\text{Num.pow y n }\equiv x\)
  shows \(\text{numeral } x \text{ powr } (m / \text{numeral } n :: \text{real}) \equiv \text{numeral } y \text{ powr } m\)
  by (subst assms [symmetric], subst numeral-powr, subst powr-numeral [symmetric])
    (simp, subst powr-powr, simp-all)

private lemma numeral-powr-inverse-eq:
  \(\text{numeral } x \text{ powr } (\text{inverse } (\text{numeral } n)) = \text{numeral } x \text{ powr } (1 / \text{numeral } n :: \text{real})\)
  by simp

ML
signature ROOT-NUMERAL-SIMPROC = sig

val sqrt : int option -> int -> int option
val sqrt' : int option -> int -> int option
val nth-root : int option -> int -> int -> int option
val nth-root' : int option -> int -> int -> int option
val sqrt-proc : Simplifier.proc
val root-proc : int * int -> Simplifier.proc
val powr-proc : int * int -> Simplifier.proc

end

structure Root-Numeral-Simproc : ROOT-NUMERAL-SIMPROC = struct

fun iterate NONE p f x =
  let
    fun go x = if p x then x else go (f x)
  in
    SOME (go x)
  end
  | iterate (SOME threshold) p f x =
    let
      fun go (threshold, x) =
        if p x then SOME x else if threshold = 0 then NONE else go (threshold - 1, f x)
    in
      go (threshold, x)
    end

fun nth-root - 1 x = SOME x
  | nth-root - - 0 = SOME 0
  | nth-root - - 1 = SOME 1
  | nth-root threshold n x =
    let
      fun newton-step y = ((n - 1) * y + x div Integer.pow (n - 1) y) div n
      fun is-root y = Integer.pow n y <= x andalso x < Integer.pow n (y + 1)
    in
      if x < n then
        SOME 1
      else if x < Integer.pow n 2 then
        SOME 1
      else
        let
          val y = Real.floor (Math.pow (Real.fromInt x, Real.fromInt 1 / Real.fromInt n))
        in
          if is-root y then
            SOME y
else
  iterate threshold is-root newton-step ((x + n - 1) div n)
end

fun nth-root' - 1 x = SOME x
| nth-root' - 0 = SOME 0
| nth-root' - 1 = SOME 1
| nth-root' threshold n x = if x < n then NONE else if x < Integer.pow n 2 then
  NONE else
  case nth-root threshold n x of
    NONE => NONE
  | SOME y => if Integer.pow n y = x then SOME y else NONE

fun sqrt - 0 = SOME 0
| sqrt - 1 = SOME 1
| sqrt threshold n x = let
  fun aux (a, b) = if n >= b * b then aux (b, b * b) else (a, b)
  in
  case sqrt threshold (n div aux (1, 2)) of
    NONE => NONE
  | SOME m => SOME (Thm.instantiate' [] (map (SOME o Thm.term_of ctxt o HLogic.mk-numeral) [m, n]))
  @ (thm sqrt-numeral-simproc-aux)
end
THEORY "Transcendental"

handle TERM - => NONE

fun root-proc (threshold1, threshold2) ctxt ct = let val [n, x] = ct |> Thm.term-of |> strip-comb |> snd |> map (dest-comb #> snd #> HOLogic.dest-numeral)
  in if n > threshold1 orelse x > threshold2 then NONE else
    case nth-root' (SOME 100) n x of
      NONE => NONE
    | SOME m =>
      SOME (Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt o HOLogic.mk-numeral) [m, n, x]))
        @{thm root-numeral-simproc-aux}
end handle TERM - => NONE |
  Match => NONE
end

fun powr-proc (threshold1, threshold2) ctxt ct = let val eq-thm = Conv.try-conv (Conv.rewr-conv @{thm numeral-powr-inverse-eq}) ct
  val ct = Thm.dest-equals-rhs (Thm.cprop-of eq-thm)
  val (_, [x, t]) = strip-comb (Thm.term-of ct)
  val (_, [m, n]) = strip-comb t
  val [x, n] = map (dest-comb #> snd #> HOLogic.dest-numeral) [x, n]
  in if n > threshold1 orelse x > threshold2 then NONE else
    case nth-root' (SOME 100) n x of
      NONE => NONE
    | SOME y =>
      let val [y, n, x] = map HOLogic.mk-numeral [y, n, x]
        val thm = Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt) [y, n, x, m])
          @{thm powr-numeral-simproc-aux}
        in SOME (@{thm transitive} OF [eq-thm, thm])
      end
    end handle TERM - => NONE |
  Match => NONE
end

end
114 Complex Numbers: Rectangular and Polar Representations

theory Complex
imports Transcendental Real-Vector-Spaces
begin

We use the codatatype command to define the type of complex numbers. This allows us to use primcorec to define complex functions by defining their real and imaginary result separately.

codatatype complex = Complex (Re: real) (Im: real)

lemma complex-surj: Complex (Re z) (Im z) = z
  by (rule complex.collapse)

lemma complex-eqI [intro?]: Re x = Re y \implies Im x = Im y \implies x = y
  by (rule complex.expand) simp

lemma complex-eq-iff: x = y \iff Re x = Re y \land Im x = Im y
  by (auto intro: complex.expand)
114.1 Addition and Subtraction

instantiation complex :: ab-group-add
begin

primcorec zero-complex
where
  Re 0 = 0
  | Im 0 = 0

primcorec plus-complex
where
  Re (x + y) = Re x + Re y
  | Im (x + y) = Im x + Im y

primcorec uminus-complex
where
  Re (−x) = −Re x
  | Im (−x) = −Im x

primcorec minus-complex
where
  Re (x − y) = Re x − Re y
  | Im (x − y) = Im x − Im y

instance
  by standard (simp-all add: complex-eq-iff)
end

114.2 Multiplication and Division

instantiation complex :: field
begin

primcorec one-complex
where
  Re 1 = 1
  | Im 1 = 0

primcorec times-complex
where
  Re (x ∗ y) = Re x ∗ Re y − Im x ∗ Im y
  | Im (x ∗ y) = Re x ∗ Im y + Im x ∗ Re y

primcorec inverse-complex
where
  Re (inverse x) = Re x / ((Re x)² + (Im x)²)
  | Im (inverse x) = −Im x / ((Re x)² + (Im x)²)
**THEORY** “Complex”

**definition** $x \div y = x \ast \text{inverse } y$ for $x, y :: \text{complex}$

**instance**

by standard

(simp-all add: complex-eq-iff divide-complex-def distrib-left distrib-right right-diff-distrib left-diff-distrib power2-eq-square add-divide-distrib [symmetric])

end

**lemma** $\text{Re-divide}: \text{Re} \left(\frac{x}{y}\right) = \frac{(\text{Re } x \ast \text{Re } y + \text{Im } x \ast \text{Im } y)}{((\text{Re } y)^2 + (\text{Im } y)^2)}$

by (simp add: divide-complex-def add-divide-distrib)

**lemma** $\text{Im-divide}: \text{Im} \left(\frac{x}{y}\right) = \frac{(-\text{Re } x \ast \text{Im } y + \text{Im } x \ast \text{Re } y)}{((\text{Re } y)^2 + (\text{Im } y)^2)}$

by (simp add: divide-complex-def diff-divide-distrib)

**lemma** $\text{Complex-divide}$:

$(x / y) = \text{Complex} \left(\frac{((\text{Re } x \ast \text{Re } y + \text{Im } x \ast \text{Im } y)}{((\text{Re } y)^2 + (\text{Im } y)^2)}\right)$

by (metis Im-divide Re-divide complex-surj)

**lemma** $\text{Re-power2}: \text{Re} \left(x^2\right) = (\text{Re } x)^2 - (\text{Im } x)^2$

by (simp add: power2-eq-square)

**lemma** $\text{Im-power2}: \text{Im} \left(x^2\right) = 2 \ast \text{Re } x \ast \text{Im } x$

by (simp add: power2-eq-square)

**lemma** $\text{Re-power-real} [\text{simp}]: \text{Im } x = 0 \Longrightarrow \text{Re} \left(x^n\right) = \text{Re } x^n$

by (induct n) simp-all

**lemma** $\text{Im-power-real} [\text{simp}]: \text{Im } x = 0 \Longrightarrow \text{Im} \left(x^n\right) = 0$

by (induct n) simp-all

### 114.3 Scalar Multiplication

**instantiation** complex :: real-field

begin

**primcorec** scaleR-complex

where

$\text{Re} \left(\text{scaleR } r \ x\right) = r \ast \text{Re } x$

$\text{Im} \left(\text{scaleR } r \ x\right) = r \ast \text{Im } x$

**instance**

**proof**

fix $a, b :: \text{real}$ and $x, y :: \text{complex}$

show $\text{scaleR } a \ (x + y) = \text{scaleR } a \ x + \text{scaleR } a \ y$
by (simp add: complex-eq-iff distrib-left)

show scaleR (a + b) x = scaleR a x + scaleR b x
  by (simp add: complex-eq-iff distrib-right)

show scaleR a (scaleR b x) = scaleR (a * b) x
  by (simp add: complex-eq-iff mult.assoc)

show scaleR 1 x = x
  by (simp add: complex-eq-iff)

show scaleR a x * y = scaleR a (x * y)
  by (simp add: complex-eq-iff algebra-simps)

show x * scaleR a y = scaleR a (x * y)
  by (simp add: complex-eq-iff algebra-simps)

qed

114.4 Numerals, Arithmetic, and Embedding from R

declare [[coercion of-real :: real ⇒ complex]]
declare [[coercion of-rat :: rat ⇒ complex]]
declare [[coercion of-int :: int ⇒ complex]]
declare [[coercion of-nat :: nat ⇒ complex]]

abbreviation complex-of-nat::nat ⇒ complex
  where complex-of-nat ≡ of-nat

abbreviation complex-of-int::int ⇒ complex
  where complex-of-int ≡ of-int

abbreviation complex-of-rat::rat ⇒ complex
  where complex-of-rat ≡ of-rat

abbreviation complex-of-real :: real ⇒ complex
  where complex-of-real ≡ of-real

lemma complex-Re-of-nat [simp]: Re (of-nat n) = of-nat n
  by (induct n) simp-all

lemma complex-Im-of-nat [simp]: Im (of-nat n) = 0
  by (induct n) simp-all

lemma complex-Re-of-int [simp]: Re (of-int z) = of-int z
  by (cases z rule: int-diff-cases) simp

lemma complex-Im-of-int [simp]: Im (of-int z) = 0
  by (cases z rule: int-diff-cases) simp

lemma complex-Re-numeral [simp]: Re (numeral v) = numeral v
  using complex-Re-of-int [of numeral v] by simp
lemma complex-Im-numeral [simp]: \( \operatorname{Im} (\text{numeral } v) = 0 \)
using complex-Im-of-int [of numeral v] by simp

lemma Re-complex-of-real [simp]: \( \operatorname{Re} (\operatorname{complex-of-real } z) = z \)
by (simp add: af-real-def)

lemma Im-complex-of-real [simp]: \( \operatorname{Im} (\operatorname{complex-of-real } z) = 0 \)
by (simp add: of-real-def)

lemma Re-divide-numeral [simp]: \( \operatorname{Re} (z / \text{numeral } w) = \frac{\operatorname{Re } z}{\text{numeral } w} \)
by (simp add: Re-divide sqr-conv-mult)

lemma Im-divide-numeral [simp]: \( \operatorname{Im} (z / \text{numeral } w) = \frac{\operatorname{Im } z}{\text{numeral } w} \)
by (simp add: Im-divide sqr-conv-mult)

lemma Re-divide-of-nat [simp]: \( \operatorname{Re} (z / \operatorname{of-nat } n) = \frac{\operatorname{Re } z}{\operatorname{of-nat } n} \)
by (cases n) (simp-all add: Re-divide field-split-simps power2-eq-square del: of-nat-Suc)

lemma Im-divide-of-nat [simp]: \( \operatorname{Im} (z / \operatorname{of-nat } n) = \frac{\operatorname{Im } z}{\operatorname{of-nat } n} \)
by (cases n) (simp-all add: Im-divide field-split-simps power2-eq-square del: of-nat-Suc)

lemma Re-inverse [simp]: \( r \in \mathbb{R} \implies \operatorname{Re} (\operatorname{inverse } r) = \frac{1}{\operatorname{Re } r} \)
by (metis Re-complex-of-real Reals-cases of-real-inverse)

lemma Im-inverse [simp]: \( r \in \mathbb{R} \implies \operatorname{Im} (\operatorname{inverse } r) = 0 \)
by (metis Im-complex-of-real Reals-cases of-real-inverse)

lemma of-real-Re [simp]: \( z \in \mathbb{R} \implies \operatorname{of-real } (\operatorname{Re } z) = z \)
by (auto simp: Reals-def)

lemma complex-Re-fact [simp]: \( \operatorname{Re} (\text{fact } n) = \text{fact } n \)
proof
  have \( \text{fact } n :: \text{complex} = \operatorname{of-real } (\text{fact } n) \)
  by simp
  also have \( \operatorname{Re} \ldots = \text{fact } n \)
  by (subst Re-complex-of-real) simp-all
  finally show \( \text{thesis} \).
qed

lemma surj-Re: \( \text{surj } \operatorname{Re} \)
by (metis Re-complex-of-real surj-def)

lemma surj-Im: \( \text{surj } \operatorname{Im} \)
by (metis complex.sel(2) surj-def)

lemma complex-Im-fact [simp]: \( \operatorname{Im} (\text{fact } n) = 0 \)
by (metis complex-Im-of-nat of-nat-fact)

lemma Re-prod-Reals: \( \forall x. x \in A \implies f x \in \mathbb{R} \)
\( \implies \operatorname{Re} (\prod f A) = \prod (\lambda x.) \).
THEORY “Complex” 2505

Re \((f\ x)\) \(A\)

proof (induction \(A\) rule: infinite-finite-induct)
  case (insert \(x\) \(A\))
  hence \(Re\ \prod f\ (insert\ x\ A) = Re\ \prod f\ A - Im\ \prod f\ 0\)
    by simp
  also from insert.prems have \(f\ x\in\mathbb{R}\) by simp
  hence \(Im\ \prod f\ A = 0\) by (auto elim!: Reals-cases)
  also have \(Re\ \prod f\ A = (\prod x\in A. Re\ (f\ x))\)
    by (intro insert.IH insert.prems) auto
  finally show \(?case\ using\ insert.hyps\ by\ simp\)
  qed auto

114.5 The Complex Number \(i\)

primcorec imaginary-unit :: complex \((i)\)
  where
  \(Re\ i = 0\)
  | \(Im\ i = 1\)

lemma Complex-eq: Complex \(a\ b = a + i \ast b\)
  by (simp add: complex-eq-iff)

lemma complex-eq: \(a = Re\ a + i \ast Im\ a\)
  by (simp add: complex-eq-iff)

lemma fun-complex-eq: \(f = (\lambda x. Re\ (f\ x) + i \ast Im\ (f\ x))\)
  by (simp add: fun-eq-iff complex-eq)

lemma i-squared [simp]: \(i \ast i = -1\)
  by (simp add: complex-eq-iff)

lemma power2-i [simp]: \(i^2 = -1\)
  by (simp add: power2-eq-square)

lemma inverse-i [simp]: \(inverse\ i = -i\)
  by (rule inverse-unique) simp

lemma divide-i [simp]: \(x / i = -i \ast x\)
  by (simp add: divide-complex-def)

lemma complex-i-mult-minus [simp]: \(i \ast (i \ast x) = -x\)
  by (simp add: mult.assoc [symmetric])

lemma complex-i-not-zero [simp]: \(i \neq 0\)
  by (simp add: complex-eq-iff)

lemma complex-i-not-one [simp]: \(i \neq 1\)
  by (simp add: complex-eq-iff)
lemma \( \text{complex-i-not-numeral} \) [simp]: \( \text{i} \neq \text{numeral w} \)
by (simp add: complex-eq-iff)

lemma \( \text{complex-i-not-neg-numeral} \) [simp]: \( \text{i} \neq -\text{numeral w} \)
by (simp add: complex-eq-iff)

lemma \( \text{complex-split-polar} \)
\[ \exists r \ a. \ z = \text{complex-of-real r} \ast (\cos a + \text{i} \ast \sin a) \]
by (simp add: complex-eq-iff polar-Ex)

lemma \( \text{i-even-power} \) [simp]: \( \text{i} \ast (n \ast 2) = (-1) \ast n \)
by (metis mult.commute power2-i power-mult)

lemma \( \text{i-even-power'} \) [simp]: even \( n \) \( \Rightarrow \) \( \text{i} \ast n = (-1) \ast (n \div 2) \)
by (metis dvd-mult-div-cancel power2-i power-mult)

lemma \( \text{Re-i-times} \) [simp]: \( \text{Re} (\text{i} \ast z) = -\text{Im} z \)
by simp

lemma \( \text{Im-i-times} \) [simp]: \( \text{Im} (\text{i} \ast z) = \text{Re} z \)
by simp

lemma \( \text{i-times-eq-iff} \)
\[ \text{i} \ast w = z \iff w = - (\text{i} \ast z) \]
by auto

lemma \( \text{divide-numeral-i} \) [simp]: \( z / (\text{numeral n} \ast \text{i}) = - (\text{i} \ast z) / \text{numeral n} \)
by (metis divide-divide-eq-left divide-i mult-commute mult-minus-right)

lemma \( \text{imaginary-eq-real-iff} \) [simp]:
\begin{align*}
\text{assumes} & \quad y \in \text{Reals} \\
\text{shows} & \quad \text{i} \ast y = x \iff x=0 \land y=0
\end{align*}
by (metis Im-complex-of-real Im-i-times assms mult-zero-right of-real-0 of-real-Re)

lemma \( \text{real-eq-imaginary-iff} \) [simp]:
\begin{align*}
\text{assumes} & \quad y \in \text{Reals} \\
\text{shows} & \quad x = \text{i} \ast y \iff x=0 \land y=0 \\
\text{using} & \quad \text{assms imaginary-eq-real-iff by fastforce}
\end{align*}

114.6 Vector Norm

instantiation \( \text{complex} :: \text{real-normed-field} \)
begin

definition \( \text{norm z} = \sqrt{((\text{Re} z)^2 + (\text{Im} z)^2)} \)

abbreviation \( \text{cmod :: complex} \Rightarrow \text{real} \)
\[ \text{where} \quad \text{cmod} \equiv \text{norm} \]

definition \( \text{complex-sgn-def} \): \( \text{sgn x} = x / \text{Reals} \text{cmod x} \)
definition dist-complex-def: dist x y = cmod (x - y)

definition uniformity-complex-def [code def]:
  (uniformity :: (complex × complex) filter) = (INF e∈{0 <..}. principal {((x, y). dist x y < e)})

definition open-complex-def [code def]:
  open (U :: complex set) ⇔ (∀ x∈U. eventually (λ(x', y). x' = x → y ∈ U) uniformity)

instance
proof
  fix r :: real and x y :: complex and S :: complex set
  show (norm x = 0) = (x = 0)
    by (simp add: norm-complex-def complex-eq-iff)
  show norm (x + y) ≤ norm x + norm y
    by (simp add: norm-complex-def complex-eq-iff real-sqrt-sum-squares-triangle-ineq)
  show norm (scaleR r x) = |r| * norm x
    by (simp add: norm-complex-def complex-eq-iff power-mult-distrib distrib-left [symmetric] real-sqrt-mult)
  show norm (x * y) = norm x * norm y
    by (simp add: norm-complex-def complex-eq-iff real-sqrt-mult [symmetric] power2-eq-square algebra-simps)
  qed (rule complex-sgn-def dist-complex-def open-complex-def uniformity-complex-def)

end

declare uniformity-Abort[where ′a = complex, code]

lemma norm-ii [simp]: norm i = 1
  by (simp add: norm-complex-def)

lemma cmod-unit-one: cmod (cos a + i * sin a) = 1
  by (simp add: norm-complex-def)

lemma cmod-complex-polar: cmod (r * (cos a + i * sin a)) = |r|
  by (simp add: norm-mult cmod-unit-one)

lemma complex-Re-le-cmod: Re x ≤ cmod x
  unfolding norm-complex-def by (rule real-sqrt-sum-squares-ge1)

lemma complex-mod-minus-le-complex-mod: ¬ cmod x ≤ cmod x
  by (rule order-trans [OF - norm-ge-zero] simp)

lemma complex-mod-triangle-ineq2: cmod (b + a) - cmod b ≤ cmod a
  by (rule ord-le-eq-trans [OF norm-triangle-ineq2]) simp
lemma abs-Re-le-cmod: \(|Re\ z| \leq cm\ o\ d\ x\)
by (simp add: norm-complex-def)

lemma abs-Im-le-cmod: \(|Im\ x| \leq cm\ o\ d\ x\)
by (simp add: norm-complex-def)

lemma cmod-le: \(cm\ o\ d\ z \leq |Re\ z| + |Im\ z|\)
using norm-complex-def sqrt-sum-squares-le-sum-abs by presburger

lemma cmod-eq-Re: \(Im\ z = 0 = \Rightarrow cm\ o\ d\ z = |Re\ z|\)
by (simp add: norm-complex-def)

lemma cmod-eq-Im: \(Re\ z = 0 = \Rightarrow cm\ o\ d\ z = |Im\ z|\)
by (simp add: norm-complex-def)

lemma cmod-power2: \((cm\ o\ d\ z)^2 = (Re\ z)^2 + (Im\ z)^2\)
by (simp add: norm-complex-def)

lemma cmod-plus-Re-le-0-iff: \(cm\ o\ d\ z + Re\ z \leq 0 \iff Re\ z = - cm\ o\ d\ z\)
using abs-Re-le-cmod[of z] by auto

lemma cmod-Re-le-iff: \(Im\ x = Im\ y = \Rightarrow cm\ o\ d\ x \leq cm\ o\ d\ y \iff |Re\ x| \leq |Re\ y|\)
by (metis add.commute add-le-cancel-left norm-complex-def real-sqrt-abs real-sqrt-le-iff)

lemma cmod-Im-le-iff: \(Re\ x = Re\ y = \Rightarrow cm\ o\ d\ x \leq cm\ o\ d\ y \iff |Im\ x| \leq |Im\ y|\)
by (metis add-le-cancel-left norm-complex-def real-sqrt-abs real-sqrt-le-iff)

lemma Im-eq-0: \(|Re\ z| = cm\ o\ d\ z \Rightarrow Im\ z = 0\)
by (subst (asm) power-eq-iff-eq-base[symmetric, where n=2]) (auto simp add: norm-complex-def)

lemma abs-sqrt-wlog: \(\forall x. x \geq 0 \Rightarrow P x (x^2)) \Rightarrow P |x| (x^2)\)
for x::'a::linordered-idom
by (metis abs-ge-zero power2-abs)

lemma complex-ineq-le-norm: \(|Re\ z| + |Im\ z| \leq \sqrt{2} * norm\ z\)
unfolding norm-complex-def
apply (rule abs-sqrt-wlog [where x=Re z])
apply (rule abs-sqrt-wlog [where x=Im z])
apply (rule power2-le-imp-le)
apply (simp-all add: power2-sum add.commute sum-squares-bound real-sqrt-mult [symmetric])
done

lemma complex-unit-circle: \(z \neq 0 \Rightarrow (Re\ z / cm\ o\ d\ z)^2 + (Im\ z / cm\ o\ d\ z)^2 = 1\)
by (simp add: norm-complex-def complex-eq-iff power2-eq-square add-divide-distrib [symmetric])

Properties of complex signum.
lemma sgn-eq: \( \text{sgn } z = \frac{z}{\text{cmod } z} \) 
by (simp add: sgn-div-norm divide-inverse scaleR-conv-of-real mult.commute)

lemma Re-sgn [simp]: \( \text{Re}(\text{sgn } z) = \frac{\text{Re}(z)}{\text{cmod } z} \) 
by (simp add: complex-sgn-def divide-inverse)

lemma Im-sgn [simp]: \( \text{Im}(\text{sgn } z) = \frac{\text{Im}(z)}{\text{cmod } z} \) 
by (simp add: complex-sgn-def divide-inverse)

114.7 Absolute value

instantiation complex :: field-\text{abs-sgn} 
begin

definition abs-complex :: complex \Rightarrow complex 
where abs-complex = of-real \circ \text{norm}

instance 
proof qed (auto simp add: abs-complex-def complex-sgn-def norm-divide norm-mult scaleR-conv-of-real field-simps)
end

114.8 Completeness of the Complexes

lemma bounded-linear-Re: bounded-linear Re 
by (rule bounded-linear-intro [where \( K=1 \)]) (simp-all add: norm-complex-def)

lemma bounded-linear-Im: bounded-linear Im 
by (rule bounded-linear-intro [where \( K=1 \)]) (simp-all add: norm-complex-def)

lemmas Cauchy-Re = bounded-linear.Cauchy [OF bounded-linear-Re]
lemmas Cauchy-Im = bounded-linear.Cauchy [OF bounded-linear-Im]
lemmas tendsto-Re [tendsto-intros] = bounded-linear.tendsto [OF bounded-linear-Re]
lemmas tendsto-Im [tendsto-intros] = bounded-linear.tendsto [OF bounded-linear-Im]
lemmas isCont-Re [simp] = bounded-linear.isCont [OF bounded-linear-Re]
lemmas isCont-Im [simp] = bounded-linear.isCont [OF bounded-linear-Im]
lemmas continuous-Re [simp] = bounded-linear.continuous [OF bounded-linear-Re]
lemmas continuous-Im [simp] = bounded-linear.continuous [OF bounded-linear-Im]
lemmas continuous-on-Re [continuous-intros] = bounded-linear.continuous-on[OF bounded-linear-Re]
lemmas continuous-on-Im [continuous-intros] = bounded-linear.continuous-on[OF bounded-linear-Im]
lemmas has-derivative-Re [derivative-intros] = bounded-linear.has-derivative[OF bounded-linear-Re]
lemmas has-derivative-Im [derivative-intros] = bounded-linear.has-derivative[OF bounded-linear-Im]
lemmas sums-Re = bounded-linear.sums [OF bounded-linear-Re]
lemmas sums-Im = bounded-linear.sums [OF bounded-linear-Im]
lemmas Re-suminf = bounded-linear.suminf[OF bounded-linear-Re]
lemmas Im-suminf = bounded-linear.suminf[OF bounded-linear-Im]
lemma continuous-on-Complex [continuous-intros]:
  continuous-on A f \implies \text{continuous-on} A g \implies \text{continuous-on} A (\lambda x. \text{Complex} (f x) (g x))
 unfolding Complex-eq by (intro continuous-intros)

lemma tendsto-Complex [tendsto-intros]:
  \((f \longrightarrow a) \Rightarrow (g \longrightarrow b) \Rightarrow ((\lambda x. \text{Complex} (f x) (g x)) \longrightarrow \text{Complex} a b) F\)
 unfolding Complex-eq by (auto intro: tendsto-intros)

lemma tendsto-complex-iff:
  \((f \longrightarrow x) F \iff ((\lambda x. \text{Re} (f x)) \longrightarrow \text{Re} x) F \land ((\lambda x. \text{Im} (f x)) \longrightarrow \text{Im} x) F\)
 proof safe
  assume \((\lambda x. \text{Re} (f x)) \longrightarrow \text{Re} x) F \land ((\lambda x. \text{Im} (f x)) \longrightarrow \text{Im} x) F\)
 from tendsto-Complex[OF this] show \((f \longrightarrow x) F\)
 unfolding complex-collapse .
qed (auto intro: tendsto-intros)

lemma continuous-complex-iff:
  \(\text{continuous} F f \iff \text{continuous} F (\lambda x. \text{Re} (f x)) \land \text{continuous} F (\lambda x. \text{Im} (f x))\)
 by (simp only: continuous-def tendsto-complex-iff)

lemma continuous-on-of-real-o-iff [simp]:
  \(\text{continuous-on} S (\lambda x. \text{complex-of-real} (g x)) = \text{continuous-on} S g\)
 using continuous-on-Re continuous-on-of-real by fastforce

lemma continuous-on-of-real-id [simp]:
  \(\text{continuous-on} S (\text{af-real} :: \text{real} \Rightarrow 'a::\text{real-normed-algebra-1})\)
 by (rule continuous-on-of-real [OF continuous-on-id])

lemma has-vector-derivative-complex-iff:
  \((f \text{ has-vector-derivative} (\lambda x. \text{Re} (f x)) \text{ has-field-derivative} (\text{Re} x)) F \land \\
 (\lambda x. \text{Im} (f x)) \text{ has-field-derivative} (\text{Im} x) F\)
 by (simp add: has-vector-derivative-def has-field-derivative-def \\
 tendsto-complex-iff algebra-simps bounded-linear-scaleR-left bounded-linear-mult-right)

lemma has-field-derivative-Re[derivative-intros]:
  \((f \text{ has-vector-derivative} D) F \implies ((\lambda x. \text{Re} (f x)) \text{ has-field-derivative} (\text{Re} D)) F\)
 unfolding has-vector-derivative-complex-iff by safe

lemma has-field-derivative-Im[derivative-intros]:
  \((f \text{ has-vector-derivative} D) F \implies ((\lambda x. \text{Im} (f x)) \text{ has-field-derivative} (\text{Im} D)) F\)
 unfolding has-vector-derivative-complex-iff by safe

instance complex :: banach
proof
  fix X :: nat \Rightarrow complex
assume \(X: \text{Cauchy} \ X\)
then have \(\lambda n. \text{Complex} (\text{Re} (X n)) (\text{Im} (X n)) \rightarrow\)
\(\text{Complex} (\lim (\lambda n. \text{Re} (X n))) (\lim (\lambda n. \text{Im} (X n)))\)
by (intro tendsto-Complex convergent-LIMSEQ-iff THEN iffD1)
Cauchy-convergent-iff[THEN iffD1] Cauchy-Re Cauchy-Im
then show convergent \(X\)
unfolding complex.collapse by (rule convergentI)
qed

declare DERIV-power[where 'a=complex, unfolded of-nat-def[symmetric], derivative-intros]

114.9 Complex Conjugation

primcorec cnj :: complex \Rightarrow complex
where
\(\text{Re} (\text{cnj} z) = \text{Re} z\)
| \(\text{Im} (\text{cnj} z) = -\text{Im} z\)

lemma complex-cnj-cancel-iff [simp]: \(\text{cnj} x = \text{cnj} y \iff x = y\)
by (simp add: complex-eq-iff)

lemma complex-cnj-cnj [simp]: \(\text{cnj} (\text{cnj} z) = z\)
by (simp add: complex-eq-iff)

lemma in-image-cnj-iff: \(z \in \text{cnj ' A} \iff \text{cnj} z \in A\)
by (metis complex-cnj-cnj image-iff)

lemma image-cnj-conv-vimage-cnj: \(\text{cnj ' A = cnj - ' A}\)
using in-image-cnj-iff by blast

lemma complex-cnj-zero [simp]: \(\text{cnj} 0 = 0\)
by (simp add: complex-eq-iff)

lemma complex-cnj-zero-iff [iff]: \(\text{cnj} z = 0 \iff z = 0\)
by (simp add: complex-eq-iff)

lemma complex-cnj-one-iff [simp]: \(\text{cnj} z = 1 \iff z = 1\)
by (simp add: complex-eq-iff)

lemma complex-cnj-add [simp]: \(\text{cnj} (x + y) = \text{cnj} x + \text{cnj} y\)
by (simp add: complex-eq-iff)

lemma cnj-sum [simp]: \(\text{cnj} (\text{sum} f s) = \bigsum x \in s. \text{cnj} (f x)\)
by (induct s rule: infinite-finite-induct) auto

lemma complex-cnj-diff [simp]: \(\text{cnj} (x - y) = \text{cnj} x - \text{cnj} y\)
by (simp add: complex-eq-iff)
lemma complex-cnj-minus [simp]: \( \text{cnj} (-x) = -\text{cnj} x \)
  by (simp add: complex-eq-iff)

lemma complex-cnj-one [simp]: \( \text{cnj} 1 = 1 \)
  by (simp add: complex-eq-iff)

lemma complex-cnj-mult [simp]: \( \text{cnj} (x \ast y) = \text{cnj} x \ast \text{cnj} y \)
  by (simp add: complex-eq-iff)

lemma cnj-prod [simp]: \( \text{cnj} \prod f s = (\prod_{x \in s} \text{cnj} (f x)) \)
  by (induct s rule: infinite-finite-induct) auto

lemma complex-cnj-inverse [simp]: \( \text{cnj} (\text{inverse} x) = \text{inverse} (\text{cnj} x) \)
  by (simp add: complex-eq-iff)

lemma complex-cnj-divide [simp]: \( \text{cnj} (x / y) = \text{cnj} x / \text{cnj} y \)
  by (simp add: divide-complex-def)

lemma complex-cnj-power [simp]: \( \text{cnj} (x \sim n) = \text{cnj} x \sim n \)
  by (induct n) simp-all

lemma complex-cnj-of-nat [simp]: \( \text{cnj} (\text{of-nat} n) = \text{of-nat} n \)
  by (simp add: complex-eq-iff)

lemma complex-cnj-of-int [simp]: \( \text{cnj} (\text{of-int} z) = \text{of-int} z \)
  by (simp add: complex-eq-iff)

lemma complex-cnj-numeral [simp]: \( \text{cnj} (\text{numeral} w) = \text{numeral} w \)
  by (simp add: complex-eq-iff)

lemma complex-cnj-neg-numeral [simp]: \( \text{cnj} (-\text{numeral} w) = -\text{numeral} w \)
  by (simp add: complex-eq-iff)

lemma complex-cnj-scaleR [simp]: \( \text{cnj} (\text{scaleR} r x) = \text{scaleR} r (\text{cnj} x) \)
  by (simp add: complex-eq-iff)

lemma complex-mod-cnj [simp]: \( \text{cmod} (\text{cnj} z) = \text{cmod} z \)
  by (simp add: norm-complex-def)

lemma complex-cnj-complex-of-real [simp]: \( \text{cnj} (\text{of-real} x) = \text{of-real} x \)
  by (simp add: complex-eq-iff)

lemma complex-cnj-i [simp]: \( \text{cnj} i = -i \)
  by (simp add: complex-eq-iff)

lemma complex-add-cnj: \( z + \text{cnj} z = \text{complex-of-real} (2 \ast \text{Re} z) \)
  by (simp add: complex-eq-iff)

lemma complex-diff-cnj: \( z - \text{cnj} z = \text{complex-of-real} (2 \ast \text{Im} z) \ast i \)
by (simp add: complex-eq-iff)

lemma Ints-cnj [intro]: \( x \in \mathbb{Z} \implies \text{cnj} \, x \in \mathbb{Z} \)
  by (auto elim!: Ints-cases)

lemma cnj-in-Ints-iff [simp]: \( \text{cnj} \, x \in \mathbb{Z} \iff x \in \mathbb{Z} \)
  using Ints-cnj[of x] Ints-cnj[of cnj x] by auto

lemma complex-mult-cnj: \( z \ast \text{cnj} \, z = \text{complex-of-real} \((\text{Re} \, z)^2 + (\text{Im} \, z)^2)\)
  by (simp add: complex-eq-iff power2-eq-square)

lemma cnj-add-mult-eq-Re: \( z \ast \text{cnj} \, w + \text{cnj} \, z \ast w = 2 \ast \text{Re} \,(z \ast \text{cnj} \, w) \)
  by (rule complex-eqI) auto

lemma complex-mod-mult-cnj: \( \text{cmod} \,(z \ast \text{cnj} \, z) = (\text{cmod} \, z)^2 \)
  by (simp add: norm-mult power2-eq-square)

lemma complex-mod-sqrt-Re-mult-cnj: \( \text{cmod} \, z = \sqrt{\text{Re} \,(z \ast \text{cnj} \, z)} \)
  by (simp add: norm-complex-def power2-eq-square)

lemma complex-In-mult-cnj-zero [simp]: \( \text{Im} \,(z \ast \text{cnj} \, z) = 0 \)
  by simp

lemma complex-cnj-fact [simp]: \( \text{cnj} \,(\text{fact} \, n) = \text{fact} \, n \)
  by (subst of-nat-fact [symmetric], subst complex-cnj-of-nat) simp

lemma complex-cnj-pochhammer [simp]: \( \text{cnj} \,(\text{pochhammer} \, z \, n) = \text{pochhammer} \,(\text{cnj} \, z) \, n \)
  by (induct n arbitrary: z) (simp-all add: pochhammer-rec)

lemma bounded-linear-cnj: bounded-linear \( \text{cnj} \)
  using complex-cnj-add complex-cnj-scaleR by (rule bounded-linear-intro [where \( K=1 \)]) simp

lemma linear-cnj: linear \( \text{cnj} \)
  using bounded-linear.linear[OF bounded-linear-cnj]

lemmas tendsto-cnj [tendsto-intros] = bounded-linear.tendsto [OF bounded-linear-cnj]
  and isCont-cnj [simp] = bounded-linear.isCont [OF bounded-linear-cnj]
  and continuous-cnj [simp, continuous-intros] = bounded-linear.continuous [OF bounded-linear-cnj]
  and continuous-on-cnj [simp, continuous-intros] = bounded-linear.continuous-on [OF bounded-linear-cnj]
  and has-derivative-cnj [simp, derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-cnj]

lemma lim-cnj: \( ((\lambda x. \text{cnj}(f \, x)) \to \text{cnj} \, l) \to ((f \to l) \to F) \)
lemma sums-cnj: \((\lambda x. \text{cnj}(f x)) \text{ sums cnj } l) \leftrightarrow (f \text{ sums } l)
by (simp add: sums-def lim-cnj cnj-sum [symmetric] del: cnj-sum)

lemma differentiable-cnj-iff:
\((\lambda z. \text{cnj}(f z)) \text{ differentiable at } x \text{ within } A \leftrightarrow f \text{ differentiable at } x \text{ within } A\)
proof
assume \((\lambda z. \text{cnj}(f z)) \text{ differentiable at } x \text{ within } A\)
then obtain \(D\) where \(((\lambda z. \text{cnj}(f z)) \text{ has-derivative } D) \text{ at } x \text{ within } A\)
  by (auto simp: differentiable-def)
from has-derivative-cnj[OF this] show \(f \text{ differentiable at } x \text{ within } A\)
  by (auto simp: differentiable-def)
next
assume \(f \text{ differentiable at } x \text{ within } A\)
then obtain \(D\) where \((f \text{ has-derivative } D) \text{ at } x \text{ within } A\)
  by (auto simp: differentiable-def)
from has-derivative-cnj[OF this] show \(((\lambda z. \text{cnj}(f z)) \text{ differentiable at } x \text{ within } A)\)
  by (auto simp: differentiable-def)
qed

lemma has-vector-derivative-cnj [derivative-intros]:
assumes \((f \text{ has-vector-derivative } f') \text{ at } z \text{ within } A\)
shows \(((\lambda z. \text{cnj}(f z)) \text{ has-vector-derivative } \text{cnj } f') \text{ at } z \text{ within } A)\)
using assms by (auto simp: has-vector-derivative-complex-iff intro: derivative-intros)

lemma has-field-derivative-cnj-cnj:
assumes \((f \text{ has-field-derivative } F) \text{ at } (\text{cnj } z)\)
shows \(((\text{cnj} \circ f \circ \text{cnj}) \text{ has-field-derivative } \text{cnj } F) \text{ at } z)\)
proof
  have \(\text{cnj} - 0 \rightarrow \text{cnj } 0\)
    by (subst lim-cnj auto)
  also have \(\text{cnj } 0 = 0\)
    by simp
  finally have \(*: \text{filterlim } \text{cnj} \text{ at } 0 \text{ at } 0)\)
    by (auto simp: filterlim-at eventually-at-filter)
  have \((\lambda h. (f \circ (\text{cnj } z + \text{cnj } h) - f \circ (\text{cnj } z)) \circ \text{cnj } h) - 0 \rightarrow F)\)
    by (rule filterlim-compose[OF *]) (use assms in auto simp: DERIV-def)
  thus \(?thesis\)
    by (subst (asm) lim-cnj [symmetric]) (simp add: DERIV-def)
qed

114.10 Basic Lemmas

lemma complex-of-real-code[code-unfold]: \(\text{of-real } = (\lambda x. \text{Complex } x \ 0)\)
by (intro ext, auto simp: complex-eq-iff)

lemma complex-eq-0: \(z = 0 \leftrightarrow (\text{Re } z)^2 + (\text{Im } z)^2 = 0\)
by (metis zero-complex.sel complex-eqI sum-power2-eq-zero-iff)
lemma complex-neq-0: \( z \neq 0 \iff (Re\, z)^2 + (Im\, z)^2 > 0 \)
  by (metis complex-eq-0 less-numeral-extra(3) sum-power2-gt-zero-iff)

lemma complex-norm-square: of-real \((\text{norm } z)^2\) = \(z \ast \text{cnj } z\)
  by (cases z)
  (auto simp: complex-eq-iff norm-complex-def power2-eq-square[ symmetric] of-real-power[ symmetric]
   simp del: of-real-power)

lemma complex-div-cnj: \(a / b = (a \ast \text{cnj } b) / (\text{norm } b)^2\)
  using complex-norm-square by auto

lemma Re-complex-div-eq-0:
  \(Re\, (a / b) = 0 \iff Re\, (a \ast \text{cnj } b) = 0\)
  by (auto simp add: Re-divide)

lemma Im-complex-div-eq-0:
  \(Im\, (a / b) = 0 \iff Im\, (a \ast \text{cnj } b) = 0\)
  by (auto simp add: Im-divide)

lemma complex-div-gt-0:
  \((Re\, (a / b) > 0 \iff Re\, (a \ast \text{cnj } b) > 0) \land (Im\, (a / b) > 0 \iff Im\, (a \ast \text{cnj } b) > 0)\)
  proof (cases \(b = 0\))
    case True
    then show \(?thesis\) by auto
  next
    case False
    then have \(0 < (Re\, b)^2 + (Im\, b)^2\)
      by (simp add: complex-eq-iff sum-power2-gt-zero-iff)
    then show \(?thesis\)
      by (simp add: Re-divide Im-divide zero-less-divide-iff)
  qed

lemma Re-complex-div-gt-0:
  \(Re\, (a / b) > 0 \iff Re\, (a \ast \text{cnj } b) > 0\)
  and \(Im\, (a / b) > 0 \iff Im\, (a \ast \text{cnj } b) > 0\)
  using complex-div-gt-0 by auto

lemma Re-complex-div-ge-0:
  \(Re\, (a / b) \geq 0 \iff Re\, (a \ast \text{cnj } b) \geq 0\)
  by (metis le-less Re-complex-div-eq-0 Re-complex-div-gt-0)

lemma Im-complex-div-ge-0:
  \(Im\, (a / b) \geq 0 \iff Im\, (a \ast \text{cnj } b) \geq 0\)
  by (metis Im-complex-div-eq-0 Im-complex-div-gt-0 le-less)

lemma Re-complex-div-le-0:
  \(Re\, (a / b) \leq 0 \iff Re\, (a \ast \text{cnj } b) \leq 0\)
  by (metis not-le Re-complex-div-gt-0)

lemma Im-complex-div-le-0:
  \(Im\, (a / b) \leq 0 \iff Im\, (a \ast \text{cnj } b) \leq 0\)
  by (metis Im-complex-div-eq-0 Im-complex-div-gt-0 less-asym neq-iff)

lemma Re-complex-div-lt-0:
  \(Re\, (a / b) < 0 \iff Re\, (a \ast \text{cnj } b) < 0\)
  by (metis less-asym neq-iff Re-complex-div-eq-0 Re-complex-div-gt-0)

lemma Im-complex-div-lt-0:
  \(Im\, (a / b) < 0 \iff Im\, (a \ast \text{cnj } b) < 0\)
  by (metis Im-complex-div-eq-0 Im-complex-div-gt-0 less-asym neq-iff)
lemma \textit{Im-complex-div-le-0}: \(\text{Im} (a / b) \leq 0 \iff \text{Im} (a \ast \text{cnj } b) \leq 0\)
by (metis \textit{Im-complex-div-gt-0} not-le)

lemma \textit{Re-divide-of-real [simp]}: \(\text{Re} (z / \text{of-real } r) = \text{Re } z / r\)
by (simp add: \textit{Re-divide} power2-eq-square)

lemma \textit{Im-divide-of-real [simp]}: \(\text{Im} (z / \text{of-real } r) = \text{Im } z / \text{Re } r\)
by (simp add: \textit{Im-divide} power2-eq-square)

lemma \textit{Re-divide-Reals [simp]}: \(r \in \mathbb{R} \Rightarrow \text{Re } (z / r) = \text{Re } z / \text{Re } r\)
by (metis \textit{Re-divide-of-real} of-real-Re)

lemma \textit{Im-divide-Reals [simp]}: \(r \in \mathbb{R} \Rightarrow \text{Im } (z / r) = \text{Im } z / \text{Re } r\)
by (metis \textit{Im-divide-of-real} of-real-Re)

lemma \textit{Re-sum [simp]}: \(\text{Re } (\sum f s) = (\sum x \in s. \text{Re } (f x))\)
by (induct s rule: infinite-finite-induct) auto

lemma \textit{Im-sum [simp]}: \(\text{Im } (\sum f s) = (\sum x \in s. \text{Im } (f x))\)
by (induct s rule: infinite-finite-induct) auto

lemma \textit{Rats-complex-of-real-iff [iff]}: \(\text{complex-of-real } x \in \mathbb{Q} \iff x \in \mathbb{Q}\)
proof
  have \(\forall a \ b. \ [0 < b; x = \text{complex-of-int } a / \text{complex-of-int } b] \Rightarrow x \in \mathbb{Q}\)
  by (metis \textit{Rats-divide Rats-of-int} Re-complex-of-real Re-divide-of-real of-real-of-int-eq)
  then show \(?thesis\)
  by (auto simp: elim!: Rats-cases')
qed

lemma \textit{sum-Re-le-cmod}: \((\sum i \in I. \text{Re } (z i)) \leq \text{cmod } (\sum i \in I. z i)\)
by (metis Re-sum complex-Re-le-cmod)

lemma \textit{sum-Im-le-cmod}: \((\sum i \in I. \text{Im } (z i)) \leq \text{cmod } (\sum i \in I. z i)\)
by (smt (verit, best) Im-sum abs-Im-le-cmod sum cong)

lemma \textit{sums-complex-iff}: \(f \text{ sums } x \iff (\lambda x. \text{Re } (f x)) \text{ sums } \text{Re } x \land ((\lambda x. \text{Im } (f x)) \text{ sums } \text{Im } x)\)
unfolding \textit{sums-def} \textit{tendsto-complex-iff} \textit{Im-sum} \textit{Re-sum} ..

lemma \textit{summable-complex-iff}: \(\text{summable } f \iff \text{summable } (\lambda x. \text{Re } (f x)) \land \text{summable } (\lambda x. \text{Im } (f x))\)
unfolding \textit{summable-def} \textit{sums-complex-iff} [abs-def] by (metis complex.sel)

lemma \textit{summable-complex-of-real [simp]}: \(\text{summable } (\lambda n. \text{complex-of-real } (f n)) \iff \text{summable } f\)
unfolding \textit{summable-def} \textit{sums-complex-iff} \textit{by simp}

lemma \textit{summable-Re}: \(\text{summable } f \Rightarrow \text{summable } (\lambda x. \text{Re } (f x))\)
unfolding summable-complex-iff by blast

lemma summable-Im: summable f \implies \text{summable } (\lambda x. \text{Im } (f x))

unfolding summable-complex-iff by blast

lemma complex-is-Nat-iff: \( z \in \mathbb{N} \iff \text{Im } z = 0 \land (\exists i. \text{Re } z = \text{of-nat } i) \)
by (auto simp: Nats-def complex-eq-iff)

lemma complex-is-Int-iff: \( z \in \mathbb{Z} \iff \text{Im } z = 0 \land (\exists i. \text{Re } z = \text{of-int } i) \)
by (auto simp: Ints-def complex-eq-iff)

lemma complex-is-Real-iff: \( z \in \mathbb{R} \iff \text{Im } z = 0 \)
by (auto simp: Reals-def complex-eq-iff)

lemma Reals-cnj-iff: \( z \in \mathbb{R} \iff \text{cnj } z = z \)
by (auto simp: complex-is-Real-iff complex-eq-iff)

lemma in-Reals-norm: \( z \in \mathbb{R} \implies \text{norm } z = |\text{Re } z| \)
by (simp add: complex-is-Real-iff norm-complex-def)

lemma Re-Reals-divide: \( r \in \mathbb{R} \implies \text{Re } (r / z) = \text{Re } r \cdot \text{Re } z / (\text{norm } z)^2 \)
by (simp add: Re-divide complex-is-Real-iff cmod-power2)

lemma Im-Reals-divide: \( r \in \mathbb{R} \implies \text{Im } (r / z) = -\text{Re } r \cdot \text{Im } z / (\text{norm } z)^2 \)
by (simp add: Im-divide complex-is-Real-iff cmod-power2)

lemma series-comparison-complex:
fixes f:: nat \Rightarrow 'a::banach
assumes sg: \text{summable } g
and \( n. \ g n \in \mathbb{R} \land n. \text{Re } (g n) \geq 0 \)
and \( n. \ g n \in \mathbb{N} \implies \text{norm } (f n) \leq \text{norm } (g n) \)
shows \text{summable } f
proof
have g: \( \land n. \text{cmod } (g n) = \text{Re } (g n) \)
using assms by (metis abs-of-nonneg in-Reals-norm)
show \text{thesis}
by (metis fg g sg summable-comparison-test summable-complex-iff)
qed

114.11 Polar Form for Complex Numbers

lemma complex-unimodular-polar:
assumes \( \text{norm } z = 1 \)
obtains t where \( 0 \leq t < 2 \cdot \pi \implies \text{Complex } (\cos t) (\sin t) \)
by (metis cmod-power2 one-power2 complex-surj sincos-total-2pi [of \text{Re } z \text{ Im } z] assms)

114.11.1 \( \cos \theta + i \sin \theta \)
primcorec cis :: real \Rightarrow complex
where
\[ \text{Re} (\text{cis} \ a) = \cos a \]
\[ \text{Im} (\text{cis} \ a) = \sin a \]

lemma cis-zero [simp]: \( \text{cis} \ 0 = 1 \)
by (simp add: complex-eq-iff)

lemma norm-cis [simp]: \( \text{norm} (\text{cis} \ a) = 1 \)
by (simp add: norm-complex-def)

lemma sgn-cis [simp]: \( \text{sgn} (\text{cis} \ a) = \text{cis} a \)
by (simp add: sgn-div-norm)

lemma cis-2pi [simp]: \( \text{cis} (2 \pi) = 1 \)
by (simp add: cis ctr complex-eq-iff)

lemma cis-neq-zero [simp]: \( \text{cis} a \neq 0 \)
by (metis norm-cis norm-zero zero-neq-one)

lemma cis-cnj: \( \text{cnj} (\text{cis} \ t) = \text{cis} (-t) \)
by (simp add: complex-eq-iff)

lemma cis-mult: \( \text{cis} a \ast \text{cis} b = \text{cis} (a + b) \)
by (simp add: complex-eq-iff cos-add sin-add)

lemma DeMoivre: \( (\text{cis} a) ^ n = \text{cis} (\text{real} n \ast a) \)
by (induct n) (simp-all add: algebra-simps cis-mult)

lemma cis-inverse [simp]: \( \text{inverse} (\text{cis} a) = \text{cis} (-a) \)
by (simp add: complex-eq-iff)

lemma cis-divide: \( \text{cis} a / \text{cis} b = \text{cis} (a - b) \)
by (simp add: divide-complex-def cis-mult)

lemma divide-conv-cnj: \( \text{norm} z = 1 \implies x / z = x \ast \text{cnj} z \)
by (metis complex-div-cnj die-by-1 mult-1 of-real-1 power2-eq-square)

lemma i-not-in-Reals [simp, intro]: \( i \notin \mathbb{R} \)
by (auto simp: complex-is-Real-iff)

lemma cos-n-Re-cis-pow-n: \( \cos (\text{real} n \ast a) = \text{Re} (\text{cis} a ^ n) \)
by (auto simp add: DeMoivre)

lemma sin-n-Im-cis-pow-n: \( \sin (\text{real} n \ast a) = \text{Im} (\text{cis} a ^ n) \)
by (auto simp add: DeMoivre)

lemma cis-pi [simp]: \( \text{cis} \pi = -1 \)
by (simp add: complex-eq-iff)
THEORY “Complex” 2519

lemma cis-pi-half[simp]: \( \text{cis} (\pi / 2) = i \)
by (simp add: cis.ctr complex-eq-iff)

lemma cis-minus-pi-half[simp]: \( \text{cis} (- (\pi / 2)) = -i \)
by (simp add: cis.ctr complex-eq-iff)

lemma cis-multiple-2pi[simp]: \( \forall n \in \mathbb{Z} \ . \text{cis} (2 \pi \ast n) = 1 \)
by (auto elim!: Ints-cases simp: cis.ctr one-complex.ctr)

lemma minus-cis: \( -\text{cis} x = \text{cis} (x + \pi) \)
by (simp flip: cis-mult)

lemma minus-cis': \( -\text{cis} x = \text{cis} (x - \pi) \)
by (simp flip: cis-divide)

114.11.2 \( r (\cos \theta + i \sin \theta) \)

definition rcis :: real \Rightarrow real \Rightarrow complex
where rcis \( r \ a = \text{complex-of-real} \ r \ast \text{cis} \ a \)

lemma Re-rcis [simp]: \( \text{Re} (\text{rcis} r a) = r \ast \cos a \)
by (simp add: rcis-def)

lemma Im-rcis [simp]: \( \text{Im} (\text{rcis} r a) = r \ast \sin a \)
by (simp add: rcis-def)

lemma rcis-Ex: \( \exists r a. \ z = \text{rcis} r a \)
by (simp add: complex-eq-iff polar-Ex)

lemma complex-mod-rcis [simp]: \( \text{cmod} (\text{rcis} r a) = |r| \)
by (simp add: rcis-def norm-mult)

lemma cis-rcis-eq: \( \text{cis} a = \text{rcis} 1 a \)
by (simp add: rcis-def)

lemma rcis-mult: \( \text{rcis} r1 a \ast \text{rcis} r2 b = \text{rcis} (r1 \ast r2) \ast (a + b) \)
by (simp add: rcis-def cis-mult)

lemma rcis-zero-mod [simp]: \( \text{rcis} 0 a = 0 \)
by (simp add: rcis-def)

lemma rcis-zero-arg [simp]: \( \text{rcis} r 0 = \text{complex-of-real} r \)
by (simp add: rcis-def)

lemma rcis-eg-zero-iff [simp]: \( \text{rcis} r a = 0 \iff r = 0 \)
by (simp add: rcis-def)

lemma DeMoivre2: \( (\text{rcis} r a) \ast n = \text{rcis} (r \ast n) \ast (\text{real} n \ast a) \)
by (simp add: rcis-def power-mult-distrib DeMoivre)
lemma \texttt{rcis-inverse}: \( \text{inverse(rcis r a)} = \text{rcis} \left( \frac{1}{r} \right) (-a) \)
\text{by (simp add: divide-inverse rcis-def)}

lemma \texttt{rcis-divide}: \( \text{rcis r1 a / rcis r2 b} = \text{rcis} \left( \frac{r1}{r2} \right) (a - b) \)
\text{by (simp add: rcis-def cis-divide [symmetric])}

114.11.3 Complex exponential

lemma \texttt{exp-Reals-eq}:
\text{assumes } z \in \mathbb{R}
\text{shows } \text{exp z} = \text{of-real} \left( \text{exp} (\text{Re z}) \right)
\text{using assms by (auto elim!: Reals-cases simp: exp-of-real)}

lemma \texttt{cis-conv-exp}: \( \text{cis b} = \text{exp} (i \ast b) \)
\text{proof}
\begin{itemize}
\item have \( (i \ast \text{complex-of-real b}) \sim n / \mathbb{R} \) fact \( n = \text{of-real} (\text{cos-coeff n} \ast b^n) + i \ast \text{af-real} (\text{sin-coeff n} \ast b^n) \)
\text{for } n :: \text{nat}
\item proof
\begin{itemize}
\item have \( i \sim n = \text{fact } n \ast_R (\text{cos-coeff n} + i \ast \text{sin-coeff n}) \)
\text{by (induct n)}
\begin{itemize}
\item (simp-all add: sin-coeff-Suc cos-coeff-Suc complex-eq-iff Re-divide Im-divide field-simps
\item power2-eq-square add-nonneg-eq-0-iff)
\end{itemize}
\item then show \( \text{?thesis} \)
\text{by (simp add: field-simps)}
\end{itemize}
\end{itemize}
\text{qed}

then show \( \text{?thesis} \)
\text{using \( \text{sin-converges [of b]} \) cos-converges [of b]}
\text{by (auto simp add: Complex-eq cis.ctr exp-def simp del: of-real-mult intro!: sums-unique sums-add sums-mutild sums-of-real)}
\text{qed}

lemma \texttt{exp-eq-polar}: \( \text{exp z} = \text{exp} (\text{Re z}) \ast \text{cis} (\text{Im z}) \)
\text{unfolding cis-conv-exp exp-of-real [symmetric] mult-exp-exp}
\text{by (cases z) (simp add: Complex-eq)}

lemma \texttt{Re-exp}: \( \text{Re} \ (\text{exp z}) = \text{exp} (\text{Re z}) \ast \cos (\text{Im z}) \)
\text{unfolding exp-eq-polar by simp}

lemma \texttt{Im-exp}: \( \text{Im} \ (\text{exp z}) = \text{exp} (\text{Re z}) \ast \sin (\text{Im z}) \)
\text{unfolding exp-eq-polar by simp}

lemma \texttt{norm-cos-sin} [simp]: \( \text{norm} (\text{Complex} (\cos t) (\sin t)) = 1 \)
\text{by (simp add: norm-complex-def)}

lemma \texttt{norm-exp-eq-Re} [simp]: \( \text{norm} (\text{exp z}) = \text{exp} (\text{Re z}) \)
\text{by (simp add: cis.code cmod-complex-polar exp-eq-polar Complex-eq)}
lemma complex-exp-exists: \( \exists \ a \ r. \ z = \text{complex-of-real} \ r * \text{exp} \ a \)
using cis-conv-exp \( \text{rcis-Ex} \ \text{rcis-def} \) by force

lemma exp-pi-i [simp]: \( \text{exp} (\text{of-real} \ \pi * \text{i}) = -1 \)
by (metis cis-conv-exp cis-pi mult.commute)

lemma exp-pi-i' [simp]: \( \text{exp} (\text{i} * (\text{of-real} \ \pi)) = -1 \)
using cis-conv-exp cis-pi by auto

lemma exp-two-pi-i [simp]: \( \text{exp} (2 * \text{of-real} \ \pi * \text{i}) = 1 \)
by (simp add: exp-eq-polar complex-eq-iff)

lemma exp-two-pi-i' [simp]: \( \text{exp} (\text{i} * (2 * (\text{of-real} \ \pi))) = 1 \)
by (metis exp-two-pi-i mult.commute)

lemma continuous-on-cis [continuous-intros]:
\( \text{continuous-on} \ A \ f \Rightarrow \text{continuous-on} \ A (\lambda x. \text{cis} (f x)) \)
by (auto simp: cis-conv-exp intro!: continuous-intros)

lemma tendsto-exp-0-Re-at-bot: \( (\text{exp} \ \longrightarrow 0) \) (filtercomap Re at-bot)
proof –
  have \( (\lambda z. \text{cmod} (\text{exp} \ z)) \longrightarrow 0 \) (filtercomap Re at-bot)
  by (auto intro!: filterlim-filtercomapI exp-at-bot)
  thus ?thesis
  using tendsto-norm-zero-iff by blast
qed

lemma filterlim-exp-at-infinity-Re-at-top: filterlim \( \text{exp} \ at\text{-infinity} \) (filtercomap Re at-top)
proof –
  have filterlim \( (\lambda z. \text{norm} (\text{exp} \ z)) \) at-top (filtercomap Re at-top)
  by (auto intro!: filterlim-filtercomapI exp-at-top)
  thus ?thesis
  using filterlim-norm-at-top-imp-at-infinity by blast
qed

114.11.4 Complex argument

definition Arg :: complex \Rightarrow real
where Arg \( z = (\text{if} \ z = 0 \ \text{then} \ 0 \ \text{else} \ (\text{SOME} \ a. \ \text{sgn} \ z = \text{cis} \ a \wedge -\pi < a \wedge a \leq \pi)) \)

lemma Arg-zero: Arg \( 0 = 0 \)
by (simp add: Arg-def)

lemma cis-Arg-unique:
assumes sgn \( z = \text{cis} \ x \) \text{ and } -\pi < x \text{ and } x \leq \pi
shows Arg \( z = x \)
proof
  from assms have \(z \neq 0\) by auto
  have \((\text{SOME} \ a. \ \text{sgn} \ z = \text{cis} \ a \land -\pi < a \land a \leq \pi) = x\)
  proof
    fix \(a\)
    define \(d\) where \(d = a - x\)
    assume \(a. \ \text{sgn} \ z = \text{cis} \ a \land -\pi < a \land a \leq \pi\)
    unfolding \(d\)-def by simp
    moreover
    from \(a\) assms have \((- (2*\pi)) < d \land d < 2*\pi\)
    proof
      unfolding \(d\)-def by simp
      moreover from \(a\) assms have \(\cos a = \cos x \land \sin a = \sin x\)
      by (simp-all add: complex-eq-iff)
      ultimately have \(\cos d = 1\)
      by (simp add: \(d\)-def cos-diff)
      moreover from \(\cos d\) have \(\sin d = 0\)
      by (rule cos-one-sin-zero)
      ultimately have \(d = 0\)
      by (auto simp: \(d\)-def by simp)
    qed
  qed (simp add: assms del: Re-sgn Im-sgn)
  with \(\langle z \neq 0 \rangle\) show \(\text{Arg} z = x\)
  by (simp add: Arg-def)
qed

lemma \(\text{Arg-correct}\):
  assumes \(z \neq 0\)
  shows \(\text{sgn} \ z = \text{cis} (\text{Arg} z) \land -\pi < \text{Arg} z \land \text{Arg} z \leq \pi\)
proof
  \(r\ a\ where\ z.\ z = \text{rcis} r a\)
  using rcis-Ex by fast
  with assms have \(r \neq 0\) by auto
  define \(b\) where \(b = (if \ 0 < r \ then \ a \ else \ a + \pi)\)
  have \(b.\ \text{sgn} \ z = \text{cis} b\)
  using \(r \neq 0\) by (simp add: \(b\)-def rcis-def of-real-def sgn-scaleR sgn-if complex-eq-iff)
  have \(\text{cis-2pi-nat}:\ \text{cis} (2 * \pi * \text{real-of-nat} \ n) = 1\ for \ n\)
  by (induct \(n\)) (simp-all add: distrib-left cis-mult [symmetric] complex-eq-iff)
  have \(\text{cis-2pi-int}:\ \text{cis} (2 * \pi * \text{real-of-int} \ x) = 1\ for \ x\)
  by (cases \(x\) rule: int-diff-cases)
  (simp add: right-diff-distrib cis-divide [symmetric] cis-2pi-nat)
  define \(c\) where \(c = b - 2 * \pi * \text{of-int} \ [(b - \pi) / (2 * \pi)]\)
  have \(\text{sgn} \ z = \text{cis} c\)
  by (simp add: \(b\)-def cis-divide [symmetric] cis-2pi-int)
  moreover have \(- \pi < c \land c \leq \pi\)
  using ceiling-correct [of \(b - \pi\) / \(2*\pi\)]
  by (simp add: \(c\)-def less-diff-diff-eq divide-le-eq algebra-simps del: le-of-int-ceiling)
  ultimately show \(\exists a. \ \text{sgn} \ z = \text{cis} a \land -\pi < a \land a \leq \pi\)
  by fast
theory Complex

lemma Arg-bounded: \(-\pi < \text{Arg } z \leq \pi\)
  by (cases \(z = 0\)) (simp-all add: Arg-zero Arg-correct)

lemma cis-Arg: \(z \neq 0 \implies \text{cis } (\text{Arg } z) = \text{sgn } z\)
  by (simp add: Arg-correct)

lemma rcis-cmod-Arg: \(\text{rcis } (\text{cmod } z) (\text{Arg } z) = z\)
  by (cases \(z = 0\)) (simp-all add: rcis-def cis-Arg sgn-div-norm of-real-def)

lemma rcis-cnj: shows \(\text{cnj } a = \text{rcis } (\text{cmod } a) (\text{Arg } a)\)
  by (metis cis-cnj complex-cnj-complex-of-real complex-cnj-mult rcis-cmod-Arg rcis-def)

lemma cos-Arg-i-mult-zero [simp]: \(y \neq 0 \implies \text{Re } y = 0 \implies \cos (\text{Arg } y) = 0\)
  using cis-Arg [of \(y\)] by (simp add: complex-eq-iff)

lemma Arg-ii [simp]: \(\text{Arg } i = \pi/2\)
  by (rule cis-Arg-unique; simp add: sgn-eq)

lemma Arg-minus-ii [simp]: \(\text{Arg } (-i) = -\pi/2\)
  proof (rule cis-Arg-unique)
    show \(\text{sgn } (-i) = \text{cis } (-\pi/2)\)
      by (simp add: sgn-eq)
    show \(-\pi/2 \leq \pi\)
      using pi-not-less-zero by linarith
  qed auto

lemma cos-Arg: \(z \neq 0 \implies \cos (\text{Arg } z) = \text{Re } z / \text{norm } z\)
  by (metis Re-sgn cis.sel(1) cis-Arg)

lemma sin-Arg: \(z \neq 0 \implies \sin (\text{Arg } z) = \text{Im } z / \text{norm } z\)
  by (metis Im-sgn cis.sel(2) cis-Arg)

subsection{Complex \(n\)-th roots}

lemma bij-betw-roots-unity:
  assumes \(n > 0\)
  shows \(\text{bij-betw } (\lambda k. \text{cis } (2 * \pi * \text{real } k / \text{real } n)) \{..<n\} \{z. z \sim n = 1\}\)
  (is bij-betw \(?f\)) unfolding bij-betw-def
  proof (intro conjI)
    show \(\text{inj-on } ?f \{..<n\}\)
      unfolding inj-on-def
    proof (safe, goal-cases)
      case (\(1 k\ l\))
      hence \(k < n l < n\) by simp-all
      from \(1\) have \(1 = ?f k / (?f l)\) by simp
also have \( \ldots = \cis (2 \cdot \pi \cdot (\real k - \real l) / n) \)
using assms by (simp add: field-simps cis-divide)
finally have \( \cos (2 \cdot \pi \cdot (\real k - \real l) / n) = 1 \)
by (simp add: complex-eq-iff)
then obtain \( m : \int \) where \( 2 \cdot \pi \cdot (\real k - \real l) / \real n = \real_of_int m \)
* \( 2 \cdot \pi \)
by (subst (asm) cos-one-2pi-int blast)
hence \( \real_of_int (\int k - \int l) = \real_of_int (m \cdot \int n) \)
using assms by (simp add: nonzero-divide-eq-eq)
also note \( \real_of_int (\int k - \int l) = \real_of_int (m \cdot \int n) \)
unfolding of-int-diff of-int-mult using assms by (simp add: nonzero-divide-eq-eq)
also have \( \ldots < \int n \)
using kl by linarith
finally have \( m = 0 \)
using assms by simp
qed

have subset: \( \?f' \{..<n\} \subseteq \{z. z \sim n = 1\} \)
proof safe
fix \( k : \nat \)
have \( \cis (2 \cdot \pi \cdot \real k / \real n) \sim n = \cis (2 \cdot \pi) \sim k \)
using assms by (simp add: DeMoivre mult-ac)
also have \( \cis (2 \cdot \pi) = 1 \) by (simp add: complex-eq-iff)
finally show \( \?f k \sim n = 1 \) by simp
qed

have \( n = \card \{..<n\} \)
also from assms have \( \ldots \leq \card \{z::\complex. z \sim n = 1\} \)
by (intro card-inj-on-le[OF inj]) (auto simp: finite-roots-unity)
finally have \( \card: \card \{z::\complex. z \sim n = 1\} = n \)
using assms by (intro antisym card-roots-unity) auto

have \( \card: \card \{z::\complex. z \sim n = 1\} = n \)
using card inj by (subst card-image) auto
with subset and assms show \( \?f' \{..<n\} = \{z::\complex. z \sim n = 1\} \)
by (intro card-subset-eq finite-roots-unity) auto
qed

lemma card-roots-unity-eq:
\( \assumes n > 0 \)
\( \shows \card \{z::\complex. z \sim n = 1\} = n \)
\( \using bij-betw-same-card [OF bij-betw-roots-unity [OF assms]] \) by simp

lemma bij-betw-nth-root-unity:
\( \fixes c :: \complex \and n :: \nat \)
\( \assumes c : c \neq 0 \and n : n > 0 \)
\( \defines c' \equiv \root n (\norm c) \cdot \cis (\Arg c / n) \)
\( \shows bij-betw (\lambda z. c' * z) \{z. z \sim n = 1\} \{z. z \sim n = c\} \)
proof –
have $c' \overset{n}{=} \text{of-real} (\text{root } n (\text{norm } c) \overset{n}{=} \text{norm } c) * \text{cis} (\text{Arg } c)$

unfolding of-real-power using $n$ by (simp add: $c'$-def power-mult-distrib DeMoivre)

also from $n$ have $\text{root } n (\text{norm } c) \overset{n}{=} \text{norm } c$ by simp
also from $c$ have $\text{of-real} \ldots * \text{cis} (\text{Arg } c) = c$ by (simp add: cis-Arg Complex.sgn-eq)
finally have $\text{simp}: c' \overset{n}{=} c$

show $\text{thesis}$ unfolding bij-betw-def inj-on-def
proof safe
  fix $z :: \text{complex}$ assume $z \overset{n}{=} 1$
  hence $(c' * z) \overset{n}{=} c' \overset{n}{=} 1$ by (simp add: power-mult-distrib)
also have $c' \overset{n}{=} \text{of-real} (\text{root } n (\text{norm } c) \overset{n}{=} \text{cis} (\text{Arg } c)$
unfolding of-real-power using $n$ by (simp add: $c'$-def power-mult-distrib DeMoivre)
also from $n$ have $\text{root } n (\text{norm } c) \overset{n}{=} \text{norm } c$ by simp
also from $c$ have $\ldots * \text{cis} (\text{Arg } c) = c$ by (simp add: cis-Arg Complex.sgn-eq)
finally show $(c' * z) \overset{n}{=} c$

next
  fix $z$ assume $z = z \overset{n}{=}$
  define $z'$ where $z' = z / c'$
  from $c$ and $n$ have $c' \neq 0$ by (auto simp: $c'$-def)
  with $n c$ have $z = c' * z'$ and $z' \overset{n}{=} 1$
    by (auto simp: $z'$-def power-divide $z$)
  thus $z \in (\lambda z. c' * z) ^{\{z. z \overset{n}{=} 1\}}$ by blast
  qed (insert $c n$, auto simp: $c'$-def)

qed

lemma finite-nth-roots [intro]:
  assumes $n > 0$
  shows finite $\{z :: \text{complex}. z \overset{n}{=} c\}$
proof (cases $c = 0$)
  case True
  with assms have $\{z :: \text{complex}. z \overset{n}{=} c\} = \emptyset$ by auto
  thus $\text{thesis}$ by simp
next
  case False
  from assms have finite $\{z :: \text{complex}. z \overset{n}{=} 1\}$ by (intro finite-roots-unity)
  simp-all
  also have $\text{thesis} \iff \text{thesis}$
    by (rule bij-betw-finite, rule bij-betw-nth-root-unity) fact+
  finally show $\text{thesis}$.
qed

lemma card-nth-roots:
  assumes $c \neq 0 n > 0$
  shows $\text{card} \{z :: \text{complex}. z \overset{n}{=} c\} = n$
proof
  have $\text{card} \{z. z \overset{n}{=} c\} = \text{card} \{z :: \text{complex}. z \overset{n}{=} 1\}$
by (rule sym, rule bij-betw-same-card, rule bij-betw-nth-root-unity) fact+
also have \ldots = n by (rule card-roots-unity-eq) fact+
finally show \text{?thesis}.
qed

\text{lemma \textit{sum-roots-unity}}: \text{assumes } n > 1 \text{ shows } \sum \{ z :: \text{complex}. \ z ^ n = 1 \} = 0 \text{ proof --}
\text{define } \omega \text{ where } \omega = \text{cis} \ (2 \ast \pi / \text{real } n) \text{ have } [\text{simp}]: \omega \neq 1 \text{ proof }
\text{assume } \omega = 1 \text{ with } \text{assms obtain } k :: \text{int} \text{ where } 2 \ast \pi / \text{real } n = 2 \ast \pi \ast \text{of-int } k \text{ by } (\text{auto simp: } \omega - \text{def complex-eq-iff cos-one-2pi-int})
\text{with } \text{assms have real } n \ast \text{of-int } k = 1 \text{ by } (\text{simp add: field-simps})
\text{also have real } n \ast \text{of-int } k = \text{of-int } (\text{int } n \ast k) \text{ by simp}
\text{also have } 1 = (\text{of-int } 1 :: \text{real}) \text{ by simp}
\text{also note } \text{of-int-eq-iff}
\text{finally show False using } \text{assms by } (\text{auto simp: } \text{zmult-eq-1-iff})
qed

have (\sum z \mid z ^ n = 1. \ z :: \text{complex}) = (\sum k<\text{n}. \ \text{cis} \ (2 \ast \pi \ast \text{real } k / \text{real } n)) \text{ using } \text{assms by } (\text{intro sum.reindex-bij-betw [symmetric] bij-betw-roots-unity})
\text{auto}
\text{also have } \ldots = (\sum k<\text{n}. \ \omega ^ k)
\text{ by } (\text{intro sum.cong refl}) (\text{auto simp: } \omega - \text{def DeMoivre mult-ac})
\text{also have } \ldots = (\omega ^ n - 1) / (\omega - 1)
\text{ by } (\text{subst geometric-sum}) \text{ auto}
\text{also have } \omega ^ n - 1 = \text{cis} \ (2 \ast \pi) - 1 \text{ using } \text{assms by } (\text{auto simp: } \omega - \text{def DeMoivre})
\text{also have } \ldots = 0 \text{ by } (\text{simp add: complex-eq-iff})
\text{finally show } \text{?thesis by simp}
qed

\text{lemma \textit{sum-nth-roots}}: \text{assumes } n > 1 \text{ shows } \sum \{ z :: \text{complex}. \ z ^ n = c \} = 0 \text{ proof (cases } c = 0)\text{ case True with } \text{assms have } \{ z :: \text{complex}. \ z ^ n = c \} = \{ 0 \} \text{ by auto}
\text{also have } \sum \ldots = 0 \text{ by simp}
\text{finally show } \text{?thesis .}
\text{next}
\text{case False define } c' \text{ where } c' = \text{root } n \ (\text{norm } c) \ast \text{cis } (\text{Arg } c / n) \text{ from False and } \text{assms have } (\sum \{ z. \ z ^ n = c \}) = (\sum z \mid z ^ n = 1. \ c' \ast z)
\text{ by } (\text{subst sum.reindex-bij-betw [OF bij-betw-nth-root-unity, symmetric]}) (\text{auto simp: sum-distrib-left finite-roots-unity c'-def})
also from assms have \ldots = 0
  by (simp add: sum-distrib-left [symmetric] sum-roots-unity)
finally show ?thesis.
qed

114.13 Square root of complex numbers

primcorec csqrt :: complex ⇒ complex
where
  Re (csqrt z) = sqrt ((cmod z + Re z) / 2)
| Im (csqrt z) = (if Im z = 0 then 1 else sgn (Im z)) * sqrt ((cmod z - Re z) / 2)

lemma csqrt-of-real-nonneg [simp]: Im x = 0 ⇒ Re x ≥ 0 ⇒ csqrt x = sqrt (Re x)
  by (simp add: complex-eq-iff norm-complex-def)

lemma csqrt-of-real-nonpos [simp]: Im x = 0 ⇒ Re x ≤ 0 ⇒ csqrt x = i * sqrt (Re x)
  by (simp add: complex-eq-iff norm-complex-def)

lemma of-real-sqrt: x ≥ 0 ⇒ of-real (sqrt x) = csqrt (of-real x)
  by (simp add: complex-eq-iff norm-complex-def)

lemma csqrt-0 [simp]: csqrt 0 = 0
  by simp

lemma csqrt-1 [simp]: csqrt 1 = 1
  by simp

lemma csqrt-ii [simp]: csqrt i = (1 + i) / sqrt 2
  by (simp add: complex-eq-iff Re-divide Im-divide real-sqrt-divide real-div-sqrt)

lemma power2-csqrt [simp,algebra]: (csqrt z)^2 = z
proof (cases Im z = 0)
  case True
  then show ?thesis
    using real-sqrt-pow2[of Re z] real-sqrt-pow2[of - Re z]
    by (cases 0::real Re z rule: linorder-cases)
      (simp-all add: complex-eq-iff Re-power2 Im-power2 power2-eq-square cmod-eq-Re)
next
  case False
  moreover have cmod z * cmod z - Re z * Re z = Im z * Im z
    by (simp add: norm-complex-def power2-eq-square)
  moreover have \|Re z\| ≤ cmod z
    by (simp add: norm-complex-def)
  ultimately show ?thesis
    by (simp add: Re-power2 Im-power2 complex-eq-iff real-sgn-eq field-simps real-sqrt-mult [symmetric] real-sqrt-divide)
lemma csqrt-power-even:
  assumes even n
  shows \( \sqrt[n]{z} \) = \( z^{n \div 2} \)
  by (metis assms dvd-mult-div-cancel power2-csqrt power-mult)

lemma norm-csqrt [simp]: \( |\sqrt{z}| = \sqrt{|z|} \)
  by (metis abs-of-nonneg norm-ge-zero norm-mul power2-csqrt power2-eq-square real-sqrt-abs)

lemma csqrt-eq-0 [simp]: \( \sqrt{z} = 0 \) \(\iff\) \( z = 0 \)
  by auto

lemma csqrt-eq-1 [simp]: \( \sqrt{z} = 1 \) \(\iff\) \( z = 1 \)
  by auto

lemma csqrt-principal: \( \Re (\sqrt{z}) > 0 \) \lor \( \Re (\sqrt{z}) = 0 \) \land \( 0 \leq \Im (\sqrt{z}) \)
  by (auto simp add: not-less cmod-plus-Re-le-0-iff Im-eq-0)

lemma Re-csqrt: \( \Re (\sqrt{z}) \leq 0 \)
  by (metis csqrt-principal le-less)

lemma csqrt-square:
  assumes 0 < Re b \lor (Re b = 0 \land 0 \leq Im b)
  shows \( \sqrt{b^2} = b \)
  proof
  have csqrt (\( b^2 \)) = b \lor \sqrt{b^2} = -b
    by (simp add: power2-eq-iff[symmetric])
  moreover have \( \sqrt{b^2} \neq -b \lor b = 0 \)
    using csqrt-principal[of \( b^2 \)] assms
    by (intro disjCI notI) (auto simp: complex-eq-iff)
  ultimately show \( \sqrt{b^2} = b \)
    by auto
  qed

lemma csqrt-unique: \( w^2 = z \) \(\implies\) \( 0 < \Re w \lor \Re w = 0 \land 0 \leq \Im w \) \(\implies\) \( \sqrt{z} = w \)
  by (auto simp: csqrt-square)

lemma csqrt-minus [simp]:
  assumes \( \Im x < 0 \lor (\Im x = 0 \land 0 \leq \Re x) \)
  shows \( -x = i * \sqrt{x} \)
  proof
  have \( \sqrt{(-x)} = i * \sqrt{x} \)
    proof (rule csqrt-square)
    have \( \Im (\sqrt{x}) \leq 0 \)
      using assms by (auto simp: cmod-eq-0 Re-le-0 iff field-simps complex-Re-le-cmod)
THEORY “Complex”

then show \( 0 < \text{Re} (i \ast \text{csqrt} \ x) \lor \text{Re} (i \ast \text{csqrt} \ x) = 0 \land 0 \leq \text{Im} (i \ast \text{csqrt} \ x) \)
by (auto simp add: Re-csqrt simp del: csqrt.simps)
qed
also have \((i \ast \text{csqrt} \ x)^2 = -x\)
by (simp add: power-mult-distrib)
finally show \(?thesis\).
qed

Legacy theorem names

lemmas cmod-def = norm-complex-def

lemma legacy-Complex-simps:
shows Complex-eq-0: Complex \(a \ b = 0 \iff a = 0 \land b = 0\)
and complex-add: Complex \(a \ b + \text{Complex} \ c \ d = \text{Complex} \ (a \ c) \ (b \ d)\)
and complex-minus: \((-\text{Complex} \ a \ b) = \text{Complex} \ (-a) \ (-b)\)
and complex-diff: Complex \(a \ b - \text{Complex} \ c \ d = \text{Complex} \ (a - c) \ (b - d)\)
and Complex-eq-I: Complex \(a \ b = 1 \iff a = 1 \land b = 0\)
and Complex-eq-neg-I: Complex \(a \ b = -1 \iff a = -1 \land b = 0\)
and complex-mult: \(\text{Complex} \ a \ b \ast \text{Complex} \ c \ d = \text{Complex} \ (a \ c - b \ d) \ (a \ d + b \ c)\)
and complex-inverse: inverse \((\text{Complex} \ a \ b) = \text{Complex} \ (a / (a^2 + b^2)) \ (-b / (a^2 + b^2))\)
and Complex-eq-numeral: Complex \(a \ b = \text{numeral} \ w \iff a = \text{numeral} \ w \land b = 0\)
and Complex-eq-neg-numeral: Complex \(a \ b = -\text{numeral} \ w \iff a = -\text{numeral} \ w \land b = 0\)
and complex-scaleR: scaleR \(r \ (\text{Complex} \ a \ b) = \text{Complex} \ (r \ast a) \ (r \ast b)\)
and Complex-eq-i: Complex \(x \ y = i \iff x = 0 \land y = 1\)
and i-mult-Complex: \(i \ast \text{Complex} \ a \ b = \text{Complex} \ (-b) \ a\)
and Complex-mult-i: Complex \(a \ b \ast i = \text{Complex} \ (-b) \ a\)
and i-complex-of-real: \(i \ast \text{complex-of-real} \ r = \text{Complex} \ 0 \ r\)
and complex-of-real-i: \(\text{complex-of-real} \ r \ast i = \text{Complex} \ 0 \ r\)
and Complex-add-complex-of-real: Complex \(x \ y + \text{complex-of-real} \ r = \text{Complex} \ (x+r) \ y\)
and complex-of-real-add-Complex: \(\text{complex-of-real} \ r + \text{Complex} \ x \ y = \text{Complex} \ (r+x) \ y\)
and Complex-mult-complex-of-real: Complex \(x \ y \ast \text{complex-of-real} \ r = \text{Complex} \ (x*r) \ (y*r)\)
and complex-of-real-mult-Complex: \(\text{complex-of-real} \ r \ast \text{Complex} \ x \ y = \text{Complex} \ (r*x) \ (r*y)\)
and complex-eq-cancel-iff2: \((\text{Complex} \ x \ y = \text{complex-of-real} \ xa) = (x = xa \land y = 0)\)
and complex-cnj: \(\text{cnj} \ (\text{Complex} \ a \ b) = \text{Complex} \ a \ (-b)\)
and Complex-sum': \(\text{sum} \ (\lambda x. \ \text{Complex} \ (f x) \ 0) \ s = \text{Complex} \ (\text{sum} \ f \ s) \ 0\)
and Complex-sum: \(\text{sum} \ (\lambda x. \ \text{Complex} \ (f x) \ 0) \ s = \text{sum} \ (\lambda x. \ \text{Complex} \ (f x) \ 0) \ s\)
and complex-of-real-def: \(\text{complex-of-real} \ r = \text{Complex} \ r \ 0\)
and complex-norm: \(\text{cmod} \ (\text{Complex} \ x \ y) = \text{sqrt} \ (x^2 + y^2)\)
by (simp-all add: norm-complex-def field-simps complex-eq-iff Re-divide Im-divide)
lemma Complex-in-Reals: Complex x 0 ∈ Ry (metis Reals-of-real complex-of-real-def)

Express a complex number as a linear combination of two others, not collinear with the origin

lemma complex-axes:
assumes Im (y/x) ≠ 0
obtains a b where z = of-real a * x + of-real b * y

proof –
define dd where dd ≡ Re y * Im x - Im y * Re x
define a where a = (Im z * Re y - Re z * Im y) / dd
define b where b = (Re z * Im x - Im z * Re x) / dd
have dd ≠ 0
  using assms by (auto simp: dd-def Im-complex-div-eq-0)
have a * Re x + b * Re y = Re z
  using (dd ≠ 0)
  apply (simp add: a-def b-def field-simps)
  by (metis dd-def diff-add-cancel distrib-right mult.assoc mult.commute)
moreover have a * Im x + b * Im y = Im z
  using (dd ≠ 0)
  apply (simp add: a-def b-def field-simps)
  by (metis (no-types) dd-def diff-add-cancel distrib-right mult.assoc mult.commute)
ultimately have z = of-real a * x + of-real b * y
  by (simp add: complex-eqI)
then show ?thesis using that by simp
qed

end

115 MacLaurin and Taylor Series

theory MacLaurin
imports Transcendental
begin

115.1 Maclaurin’s Theorem with Lagrange Form of Remainder

This is a very long, messy proof even now that it’s been broken down into lemmas.

lemma Maclaurin-lemma:
0 < h ⊨
  ∃B::real. f h = (∑m<n. (j m / (fact m)) * (hˆm)) + (B * ((hˆn) / (fact n)))
by (rule extI[where x = (f h - (∑m<n. (j m / (fact m)) * hˆm)) * (fact n) / (hˆn)]) simp

lemma eq-diff-eq’: x = y - z ⟷ y = x + z
for x y z :: real
by arith

lemma fact-diff-Suc: \( n < \text{Suc} \ m \implies \text{fact} (\text{Suc} m - n) = (\text{Suc} m - n) * \text{fact} (m - n) \)
by (subst fact-reduce) auto

lemma Maclaurin-lemma2:
fixes \( B \)
assumes DERIV: \( \forall \, m \, t. \, m < n \land 0 \leq t \land t \leq h \implies \text{DERIV} (\text{diff} \, m) \, t \implies \text{diff} (\text{Suc} m) \, t \)
and INIT: \( n = \text{Suc} \, k \)
defines \( \text{difg} \equiv \)
(\( \lambda m \, t.\, \text{real} \, m \, t - ((\sum_{p<n-m} \text{difg} \, (m + p) \, 0 / \text{fact} \, p * \text{t} \, ^p) + B \, * \, (\text{t} \, ^{\text{n} - \text{Suc} \, m} / \text{fact} \, (n - m)))) \))
shows \( \forall \, m \, t. \, m < n \land 0 \leq t \land t \leq h \implies \text{DERIV} (\text{difg} \, m) \, t \implies \text{difg} (\text{Suc} \, m) \, t \)
proof (rule allI simpI+)
fix \( m \, t \)
assume INIT2: \( m < n \land 0 \leq t \land t \leq h \)
have DERIV: \( (\text{difg} \, m) \, t \implies \text{diff} (\text{Suc} \, m) \, t - ((\sum_{x<n-m} \text{real} \, x \, * \, \text{t} \, ^{x} - (x - \text{Suc} \, 0) \, * \, \text{diff} \, (m + x) \, 0 / \text{fact} \, x) + \text{real} \, (n - m) \, * \, \text{t} \, ^{\text{n} - \text{Suc} \, m} / \text{fact} \, (n - m)) \)
by (auto simp: difg-def intro: derivative-eq-intros DERIV[rule-format, OF INIT2])
moreover
from INIT2 have intvl: \( \{..<n-m\} = \text{insert} \, 0 \, (\text{Suc} \, ^+ \{..<n-Suc \, m\}) \)
and \( 0 < n - m \)
unfolding atLeast0LessThan[symmetric] by auto
have \( \sum_{x<n-m} \text{real} \, x \, * \, \text{t} \, ^{x} - (x - \text{Suc} \, 0) \, * \, \text{diff} \, (m + x) \, 0 / \text{fact} \, x = (\sum_{x<n-Suc \, m} \text{real} \, (\text{Suc} \, m) \, * \, \text{t} \, ^{x} \, * \, \text{diff} \, (\text{Suc} \, m + x) \, 0 / \text{fact} \, (\text{Suc} \, x)) \)
unfolding intvl by (subst sum.insert) (auto simp: sum.addinit)
moreover
have \( \text{fact-neq-0}: \forall x. \, (\text{fact} \, x) + \text{real} \, x \, * \, (\text{fact} \, x) \neq 0 \)
by (metis add-pos-pos fact-gt-zero less-add-cancel1 less-add-cancel2
less-numeral-extra3 mult-less-0-iff of-nat-less-0-iff)
have \( \forall x. \, (\text{Suc} \, m) \, * \, \text{t} \, ^{x} \, * \, \text{diff} \, (\text{Suc} \, m + x) \, 0 / \text{fact} \, (\text{Suc} \, x) = \text{diff} \, (\text{Suc} \, m + x) \, 0 \, * \, \text{t} \, ^{x} / \text{fact} \, x \)
by (rule nonzero-divide-eq-eq[THEN iffD2]) auto
moreover
have \( \sum_{x<n-m} \text{real} \, x \, * \, \text{t} \, ^{x} \, * \, \text{diff} \, (\text{Suc} \, m) \, 0 \, / \text{fact} \, (\text{Suc} \, m) / \text{fact} \, (n - \text{Suc} \, m)) \)
using \( 0 < n - m \) by (simp add: field-split-simps fact-gt-zero
ultimately show DERIV: \( \text{difg} \, m \, t \implies \text{difg} \, (\text{Suc} \, m) \, t \)
unfolding difg-def by (simp add: mult.commute)
qed

lemma Maclaurin:
assumes \( h: \, 0 < h \)
and \( n \leq 0 < n \)

and \( \text{diff-0: } \text{diff} 0 = f \)

and \( \text{diff-Suc: } \forall m. \text{diff} m < n \land 0 \leq t \land t \leq h \rightarrow \text{DERIV} (\text{diff} m) t \Rightarrow \text{diff} (\text{Suc} m) t \)

shows

\[ \exists t. \text{real} \cdot 0 < t \land t < h \land f \cdot h = \sum (\lambda n. (\text{diff} m 0 / \text{fact} m) * h ^ m) \{..<n\} + (\text{diff} n t / \text{fact} n) * h ^ m \]

proof

from \( n \) obtain \( m \) where \( m = \text{Suc} m \)

by (cases \( n \)) (simp add: \( n \))

from \( m \) have \( m < n \) by simp

obtain \( B \) where \( f \cdot h = (\sum m < n. \text{diff} m 0 / \text{fact} m * h ^ m) + B * (h ^ m) + \)

using Maclaurin-lemma [OF \( h \)] ..

define \( g \) where \( [\text{abs-def}]: g \cdot t = \)

\[ f \cdot t - (\sum (\lambda p. (\text{diff} m 0 / \text{fact} m) * h ^ m) \{..<n\} + B * (h ^ m) \}

have \( g' \cdot g = 0 \)

by (simp-all add: \( m \cdot f \cdot h \cdot g \cdot \text{def} \cdot \text{lessThan-Suc-0} \cdot \text{image-iff} \cdot \text{diff-0} \cdot \text{sum} \cdot \text{reindex})

define \( \text{difg} \) where \( [\text{abs-def}]: \text{difg} m \cdot t = \)

\[ \text{diff} m \cdot t - (\sum (\lambda p. (\text{diff} m 0 / \text{fact} m) * (h ^ m)) \{..<n-m\} +

B * ((h ^ m) / \text{fact} (n - m))) \text{for} m \cdot t \]

have \( \text{difg-0: } \text{difg} 0 = g \)

by (simp add: \( \text{difg-def} \cdot \text{g-def} \cdot \text{diff-0})

have \( \text{difg-Suc: } \forall m. \text{difg} m \cdot t \Rightarrow \text{difg} (\text{Suc} m) \cdot t \)

using \( \text{diff-Suc m unfolding} \) \( \text{difg-def} \cdot \text{abs-def} \) by (rule Maclaurin-lemma2)

have \( \text{difg-eq-0: } \forall m < n. \text{difg} m 0 = 0 \)

by (auto simp: \( \text{difg-def} \cdot \text{Suc-diff-le} \cdot \text{lessThan-Suc-eq-insert-0} \cdot \text{image-iff} \cdot \text{sum} \cdot \text{reindex})

have \( \text{isCont-difg: } \forall m. \text{isCont} (\text{difg} m) \leq x \Rightarrow x \leq h \Rightarrow \text{isCont} (\text{difg} m) \)

by (rule DERIV-isCont [OF \( \text{difg-Suc [rule-format]} \)]) simp

have \( \text{differentiable-difg: } \forall m. \text{diff} m 0 \leq x \Rightarrow x \leq h \Rightarrow \text{difg} m \text{ differentiable (at} x \)

using \( \text{difg-Suc real-differentiable-def} \) by auto

have \( \text{difg-Suc-eq-0: } \)

\[ \bigvee m t. \text{diff} m 0 \leq t \Rightarrow t \leq h \Rightarrow \text{DERIV} (\text{difg} m) t \Rightarrow 0 \Rightarrow \text{difg} \text{ Suc} m t \leq 0 \]

by (rule DERIV-anique [OF \( \text{difg-Suc [rule-format]} \)]) simp

have \( \exists t. 0 < t \land t < h \land \text{DERIV} (\text{difg} m) t \Rightarrow 0 \)

using \( m < n \)

proof (induct \( m \))

case 0

show \( \exists case \)

proof (rule Rolle)

show \( 0 < h \) by fact
show \textit{difg} 0 0 = \textit{difg} 0 h
   by (simp add: \textit{difg}-0 g2)
show continuous-on \{0..h\} (\textit{difg} 0)
   by (simp add: continuous-at-imp-continuous-on isCont-difg n)
qed (simp add: differentiable-difg n)

next

  case (Suc \textit{m}')
  then obtain \textit{t} where
    \begin{align*}
    \textit{t} &< 0 \\
    \textit{t} &< \textit{h} \land \textit{DERIV} (\textit{difg} \text{ \textit{Suc m}'}) \text{ \textit{t}} :> 0
    \end{align*}
  by force
  have \exists t'. 0 < t' \land t' < \textit{t} \land \textit{DERIV} (\textit{difg} (\text{ \textit{Suc m}'}) ) t' :> 0
  proof (rule Rolle)
    show 0 < t by fact
    show \textit{difg} (\text{ \textit{Suc m}'}) 0 = \textit{difg} (\text{ \textit{Suc m}'}) t
      using t < Suc \textit{m}' < n by (simp add: difg-Suc-eq-0 difg-eq-0)
    have \(\forall x. 0 \leq x \land x \leq \textit{t} \implies \text{isCont} (\textit{difg} (\text{ \textit{Suc m}'}) ) x\)
      using t < h < Suc \textit{m}' < n by (simp add: isCont-difg)
    then show continuous-on \{0..t\} (\textit{difg} (\text{ \textit{Suc m}'}) )
      by (simp add: continuous-at-imp-continuous-on)
    qed (use \textit{t} < \textit{h} in (simp add: differentiable-difg))
  with \textit{t} < \textit{h} show \text{\text{case}}
    by auto
  qed

then obtain \textit{t} where 0 < \textit{t} < \textit{h} \textit{difg} (\text{ \textit{Suc m}}) \textit{t} = 0
  using \langle \textit{m} < \textit{n} \rangle \text{\textit{difg-Suc-eq-0}} by force
show \text{\text{thesis}}
proof (intro exI conjI)
  show 0 < \textit{t} < \textit{h} by fact+
  show \exists h = (\sum m < n. \textit{diff m 0} \ast \langle \text{\textit{fact m}} \rangle \ast \textit{h} ^ \langle \text{\textit{m}} \rangle) + \textit{diff n t} \ast \langle \text{\textit{fact n}} \rangle \ast \textit{h} ^ \langle \text{\textit{n}} \rangle
    using \langle \textit{difg} (\text{\textit{Suc m}}) \textit{t} = 0 \rangle by (simp add: m f-h difg-def)
  qed
qed

lemma Maclaurin2:

  fixes \textit{n} :: nat
  and \textit{h} :: real

  assumes INIT1: \textit{0} < \textit{h}
  and INIT2: \textit{diff 0} = \textit{f}
  and \textit{DERIV}: \forall \textit{m} \textit{t}. \textit{m} < \textit{n} \land 0 \leq \textit{t} \land \textit{t} \leq \textit{h} \implies \textit{DERIV} (\textit{diff m}) \textit{t} :> \textit{diff}
  \textit{(Suc m)} \textit{t}
  shows \exists \textit{t}. 0 < \textit{t} \land \textit{t} \leq \textit{h} \land \textit{f h} = (\sum \textit{m < n}. \textit{diff m 0} \ast \langle \textit{fact m} \rangle \ast \textit{h} ^ \langle \textit{m} \rangle) + \textit{diff n t} \ast \langle \textit{fact n} \rangle \ast \textit{h} ^ \langle \textit{n} \rangle
proof (cases \textit{n})
  case 0
  with INIT1 INIT2 show \text{\text{thesis}} by fastforce
next
  case Suc
  then have \textit{n} > 0 by simp
  from Maclaurin \text{[OF INIT1 this INIT2 DERIV]}
  show \text{\text{thesis}} by fastforce
qed

lemma Maclaurin-minus:
  fixes n :: nat and h :: real
  assumes h < 0 0 < n diff 0 = f
  and DERIV: \forall m t. m < n \land h \leq t \land t \leq 0 \rightarrow DERIV (diff m) t := diff (Suc m) t
  shows \exists t. h < t \land t < 0 \land f h = (\sum m<n. diff m 0 / fact m * h ^ m) + diff n t / fact n * h ^ n
proof –

Transform ABL’ into derivative-intros format.

note DERIV' = DERIV-chain[OF - DERIV [rule-format], THEN DERIV-cong]
let ?sum = \lambda t. (\sum m<n. (- 1) ^ m * diff m (- 0) / (fact m) * (- h) ^ m) + (- 1) ^ n * diff n (- t) / (fact n) * (- h) ^ n
from assms have \exists t>0. t < - h \land f (- (- h)) = ?sum t
  by (intro Maclaurin) (auto intro: derivative-eq-intros DERIV')
then obtain t where 0 < t t < - h f (- (- h)) = ?sum t
  by blast
moreover have (- 1) ^ n * diff n (- t) * (- h) ^ n / fact n = diff n (- t) * h ^ n / fact n
  by (auto simp: power-mult-distrib[symmetric])
moreover have (\sum m<n. (- 1) ^ m * diff m 0 * (- h) ^ m / fact m) = (\sum m<n. diff m 0 * h ^ m / fact m)
  by (auto intro: sum.cong simp add: power-mult-distrib[symmetric])
ultimately have h < - t \land - t < 0 \land f h = (\sum m<n. diff m 0 / (fact m) * h ^ m) + diff n (- t) / (fact n) * h ^ n
  by auto
then show ?thesis ..
qed

115.2 More Convenient "Bidirectional" Version.

lemma Maclaurin-bi-le:
  fixes n :: nat and x :: real
  assumes diff 0 = f
  and DERIV : \forall m t. m < n \land |t| \leq |x| \rightarrow DERIV (diff m) t := diff (Suc m) t
  shows \exists t. |t| \leq |x| \land f x = (\sum m<n. diff m 0 / (fact m) * x ^ m) + diff n t / (fact n) * x ^ n
  (is \exists t. - \land f x = ?f x t)
proof (cases n = 0)
case True
  with (diff 0 = f) show ?thesis by force
next
case False
  show ?thesis
proof (cases rule: linorder-cases)
assume \( x = 0 \)
with \( n \neq 0 \), \( \langle \text{diff } 0 = f \rangle \) DERIV have \( |0| \leq |x| \wedge f x = ?f x 0 \)
by auto
then show \(?thesis ..\)
next
assume \( x < 0 \)
with \( n \neq 0 \), DERIV have \( \exists t > x \cdot t < 0 \wedge \text{diff } 0 x = ?f x t \)
by (intro Maclaurin-minus) auto
then obtain \( t \) where \( x < t t < 0 \)
\( \text{diff } 0 x = (\sum m < n. \text{diff } m 0 / \text{fact } m * x ^ m) + \text{diff } n t / \text{fact } n * x ^ n \)
by blast
with \( x < 0 \), \( \langle \text{diff } 0 = f \rangle \) show \(?thesis by force\)
next
assume \( x > 0 \)
with \( n \neq 0 \), \( \langle \text{diff } 0 = f \rangle \) DERIV have \( \exists t > 0 \cdot t < x \wedge \text{diff } 0 x = ?f x t \)
by (intro Maclaurin) auto
then obtain \( t \) where \( 0 < t t < x \)
\( \text{diff } 0 x = (\sum m < n. \text{diff } m 0 / \text{fact } m * x ^ m) + \text{diff } n t / \text{fact } n * x ^ n \)
by blast
with \( x > 0 \), \( \langle \text{diff } 0 = f \rangle \) have \( |t| \leq |x| \wedge f x = ?f x t \) by simp
then show \(?thesis ..\)
qed
\( ^m \) = \( \text{diff} \ 0 \ 0 \)
for \( x :: \text{real} \) and \( n :: \text{nat} \)
by \( \text{simp} \)

lemma \( \text{Maclaurin-all-le}: \)
fixes \( x :: \text{real} \) and \( n :: \text{nat} \)
assumes \( \text{INIT}: \ \text{diff} \ 0 = f \)
and \( \text{DERIV}: \ \forall m x. \ \text{DERIV} (\text{diff} \ m) \ x :: \text{diff} (\text{Suc} \ m) \ x \)
shows \( \exists t. |t| \leq |x| \land f x = (\sum m<n. (\text{diff} \ m \ 0 / \ \text{fact} \ m) \ast x ^ m) + (\text{diff} \ n \ t / \ \text{fact} \ n) \ast x ^ n \) \( |t| - \land f x = ?f x t \)
proof (cases \( n = 0 \))
case True
with \( \text{INIT} \) show \( ?\text{thesis} \) by \( \text{force} \)
next
case False
show \( ?\text{thesis} \)
using \( \text{DERIV} \ \text{INIT} \ \text{Maclaurin-bi-le} \) by \( \text{auto} \)
qed

lemma \( \text{Maclaurin-all-le-objl}: \)
\( \text{diff} \ 0 = f \land (\forall m x. \ \text{DERIV} (\text{diff} \ m) \ x :: \text{diff} (\text{Suc} \ m) \ x \) \( \rightarrow \)
\( (\exists t::\text{real}. |t| \leq |x| \land f x = (\sum m<n. (\text{diff} \ m \ 0 / \ \text{fact} \ m) \ast x ^ m) + (\text{diff} \ n \ t / \ \text{fact} \ n) \ast x ^ n) \)
for \( x :: \text{real} \) and \( n :: \text{nat} \)
by \( \text{blast intro: \text{Maclaurin-all-le}} \)

115.3 Version for Exponential Function

lemma \( \text{Maclaurin-exp-lt}: \)
fixes \( x :: \text{real} \) and \( n :: \text{nat} \)
shows \( x \neq 0 \Longrightarrow n > 0 \Longrightarrow \)
\( (\exists t. \ 0 < |t| \land |t| < |x| \land \exp x = (\sum m<n. (x ^ m) / \text{fact} m) + (\exp t / \text{fact} n) \ast x ^ n) \)
using \( \text{Maclaurin-all-lt} \ [\text{where} \ \text{diff} = \lambda n. \ \exp \ \text{and} \ f = \exp \ \text{and} \ x = x \ \text{and} \ n = n] \) by \( \text{auto} \)

lemma \( \text{Maclaurin-exp-le}: \)
fixes \( x :: \text{real} \) and \( n :: \text{nat} \)
shows \( \exists t. |t| \leq |x| \land \exp x = (\sum m<n. (x ^ m) / \text{fact} m) + (\exp t / \text{fact} n) \ast x ^ n \)
using \( \text{Maclaurin-all-le-objl} \ [\text{where} \ \text{diff} = \lambda n. \ \exp \ \text{and} \ f = \exp \ \text{and} \ x = x \ \text{and} \ n = n] \) by \( \text{auto} \)

corollary \( \exp-lower-Taylor-quadratic: \ 0 \leq x \Longrightarrow 1 + x + x^2 \ / \ 2 \leq \exp x \)
for \( x :: \text{real} \)
using \( \text{Maclaurin-exp-le} \ [\text{of} \ x \ 3] \) by \( \text{(auto simp: numeral-3-eq-3 power2-eq-square)} \)
corollary \textit{ln-2-less-1}: \ln 2 < (1::real)
by (smt (verit) \textit{ln-eq-minus-one \ ln-le-minus-one})

115.4 Version for Sine Function

lemma \textit{mod-exhaust-less-4}: \( m \mod 4 = 0 \lor m \mod 4 = 1 \lor m \mod 4 = 2 \lor m \mod 4 = 3 \)
for \( m :: \text{nat} \)
by \textit{auto}

It is unclear why so many variant results are needed.

lemma \textit{sin-expansion-lemma}: \( \sin (x + \text{real} (\text{Suc } m) \ast \text{pi} / 2) = \cos (x + \text{real } m \ast \text{pi} / 2) \)
by (\textit{auto simp: cos-add sin-add add-divide-distrib distrib-right})

lemma \textit{Maclaurin-sin-expansion2}: 
\( \exists t. \ |t| \leq |x| \land 
\sin x = (\sum m < n. \sin-coeff m \ast x ^ m) + (\sin (t + 1/2 \ast \text{real } n \ast \text{pi}) / \text{fact } n) \ast x ^ n \)
proof (cases \( n = 0 \lor x = 0 \))
case False
let \( ?\text{diff} = \lambda n x. \sin (x + 1/2 \ast \text{real } n \ast \text{pi}) \)

have \( \exists t. 0 < |t| \land |t| < |x| \land \sin x = \sum m < n. (?\text{diff } m / \text{fact } m) \ast x ^ m + (?\text{diff } n / \text{fact } n) \ast x ^ n \)
proof (rule \textit{Maclaurin-all-lt})
show \( \forall m x. ((\lambda t. \sin (t + 1/2 \ast \text{real } m \ast \text{pi})) \text{ has-real-derivative} \sin (x + 1/2 \ast \text{real } (\text{Suc } m) \ast \text{pi}) \text{ at } x) \)
by (rule \textit{allI \ derivative-eq-intros} \ use \textit{sin-expansion-lemma} in force)+
qed (use False in auto)
then show \( ?\text{thesis} \)
apply (rule \textit{ex-forward}, \textit{simp})
apply (rule \textit{sum.cong[OF refl]})
apply (\textit{auto simp: sin-coeff-def sin-zero-iff elim: oddE \ simp del: of-nat-Suc})
done
qed \textit{auto}

lemma \textit{Maclaurin-sin-expansion}: 
\( \exists t. \sin x = (\sum m < n. \sin-coeff m \ast x ^ m) + (\sin (t + 1/2 \ast \text{real } n \ast \text{pi}) / \text{fact } n) \ast x ^ n \)
using \textit{Maclaurin-sin-expansion2} [of \( x \) \( n \)] by \textit{blast}

lemma \textit{Maclaurin-sin-expansion3}: 
\textit{assumes} \( n > 0 \ x > 0 \)
\textit{shows} \( \exists t. 0 < t \land t < x \land \sin x = (\sum m < n. \sin-coeff m \ast x ^ m) + (\sin (t + 1/2 \ast \text{real } n \ast \text{pi}) / \text{fact } n) \ast x ^ n \)
proof –
let \( ?\text{diff} = \lambda n x. \sin (x + 1/2 \ast \text{real } n \ast \text{pi}) \)
have \( \exists t. 0 < t \wedge t < x \wedge \sin x = (\sum_{m<n} \diff m 0 / (\text{fact } m) \times x ^ m) + \diff n t / \text{fact } n \times x ^ n \)

proof (rule Maclaurin)
  show \( \forall m t. m < n \wedge 0 \leq t \wedge t \leq x \rightarrow ((\lambda u. \sin (u + 1/2 \times \text{real } m \times \pi)) \text{ has-real-derivative} \sin (t + 1/2 \times \text{real } (\text{Suc } m) \times \pi)) \text{ (at t)} \)
    apply (simp add: sin-expansion-lemma del: of-nat-Suc)
    apply (force intro!: derivative-eq-intros)
  done
qed (use assms in auto)

then show \( \exists t. 0 < t \wedge t \leq x \wedge \sin x = (\sum_{m<n} \diff m 0 / (\text{fact } m) \times x ^ m) + \diff n t / \text{fact } n \times x ^ n \)

proof (rule Maclaurin2)
  show \( \forall m t. m < n \wedge 0 \leq t \wedge t \leq x \rightarrow ((\lambda u. \sin (u + 1/2 \times \text{real } m \times \pi)) \text{ has-real-derivative} \sin (t + 1/2 \times \text{real } (\text{Suc } m) \times \pi)) \text{ (at t)} \)
    apply (simp add: sin-expansion-lemma del: of-nat-Suc)
    apply (force intro!: derivative-eq-intros)
  done
qed

lemma Maclaurin-sin-expansion4:
  assumes \( 0 < x \)
  shows \( \exists t. 0 < t \wedge t \leq x \wedge \sin x = (\sum_{m<n} \diff m 0 / (\text{fact } m) \times x ^ m) + \diff n t / \text{fact } n \times x ^ n \)

proof (rule ex-forward, simp)
  show \( \forall m t. m < n \wedge 0 \leq t \wedge t \leq x \rightarrow ((\lambda u. \sin (u + 1/2 \times \text{real } m \times \pi)) \text{ has-real-derivative} \sin (t + 1/2 \times \text{real } (\text{Suc } m) \times \pi)) \text{ (at t)} \)
    apply (simp add: sin-expansion-lemma del: of-nat-Suc)
    apply (force intro!: derivative-eq-intros)
  done
qed (use assms in auto)

115.5 Maclaurin Expansion for Cosine Function

lemma sumr-cos-zero-one [simp]: \( (\sum_{m<Suc n} \cos-coeff m \times 0 ^ m) = 1 \)
  by (induct n) auto

lemma cos-expansion-lemma: \( \cos (x + \text{real } (\text{Suc } m) \times \pi / 2) = - \sin (x + \text{real } m \times \pi / 2) \)
  by (auto simp: cos-add sin-add distrib-right add-divide-distrib)
lemma Maclaurin-cos-expansion:
\[ \exists t. \text{real. } |t| \leq |x| \wedge \cos x = (\sum_{m < n} \text{cos-coeff } m \cdot x^{-m}) + (\cos t + 1/2 \cdot \text{real } n \cdot \pi ) / \text{fact } n \cdot x^{-n} \]

proof (cases \( n = 0 \lor x = 0 \))
  case False
  let \( \text{?diff} = \lambda n x. \cos (x + 1/2 \cdot \text{real } n \cdot \pi ) \)
  have \( \exists t. 0 < |t| \wedge |t| < |x| \wedge \cos x = \\
    (\sum_{m < n} \text{?diff } m \cdot (\text{fact } m) \cdot x^{-m}) + (\text{?diff } n \cdot t \cdot (\text{fact } n) \cdot x^{-n} \)
  proof (rule Maclaurin-all-lt)
    show \( \forall m x. ((\lambda t. \cos (t + 1/2 \cdot \text{real } m \cdot \pi )) \text{ has-real-derivative} \\
    \cos (x + 1/2 \cdot \text{real } (\text{Suc } m) \cdot \pi )) \text{ (at } x) \)
    using cos-expansion-lemma
    by (intro allI derivative-eq-intros | simp)+
  qed (use False in auto)
  then show ?thesis
  apply (rule ex-forward, simp)
  apply (rule sum.cong[OF refl])
  apply (auto simp: cos-coeff-def cos-zero-iff elim: evenE simp del: of-nat-Suc)
  done
qed

lemma Maclaurin-cos-expansion2:
assumes \( n > 0 \text{ and } x > 0 \)
shows \( \exists t. 0 < t \wedge t < x \wedge \cos x = \\
(\sum_{m < n} \text{cos-coeff } m \cdot x^{-m}) + (\cos (t + 1/2 \cdot \text{real } n \cdot \pi ) / \text{fact } n \cdot x^{-n}) \)

proof –
  let \( \text{?diff} = \lambda n x. \cos (x + 1/2 \cdot \text{real } n \cdot \pi ) \)
  have \( \exists t. 0 < t \wedge t < x \wedge \cos x = (\sum_{m < n} \text{?diff } m \cdot (\text{fact } m) \cdot x^{-m}) + \\
\text{?diff } n \cdot t \cdot (\text{fact } n) \cdot x^{-n} \)
  proof (rule Maclaurin)
    show \( \forall m t. m < n \wedge 0 \leq t \wedge t \leq x \longrightarrow \\
    ((\lambda u. \cos (u + 1/2 \cdot \text{real } m \cdot \pi )) \text{ has-real-derivative} \\
    \cos (t + 1/2 \cdot \text{real } (\text{Suc } m) \cdot \pi )) \text{ (at } t) \)
    by (simp add: cos-expansion-lemma del: of-nat-Suc)
  qed (use assms in auto)
  then show ?thesis
  apply (rule ex-forward, simp)
  apply (rule sum.cong[OF refl])
  apply (auto simp: cos-coeff-def cos-zero-iff elim: evenE)
  done
qed

lemma Maclaurin-minus-cos-expansion:
assumes \( n > 0 \text{ and } x < 0 \)
shows \( \exists t. x < t \wedge t < 0 \wedge \cos x = (\sum_{m < n} \text{cos-coeff } m \cdot x^{-m}) + (\cos (t + 1/2 \cdot \text{real } n \cdot \pi ) / \\
\text{fact } n \cdot x^{-n}) \)
proof –
let \( \text{?diff} = \lambda n x. \cos (x + 1/2 * \text{real} n * \pi) \)
have \( \exists t. x < t \land t < 0 \land \cos x = (\sum m<n. \text{?diff} m 0 / (\text{fact} m) * x ^ m) + \)
\( \text{?diff} n t / \text{fact} n * x ^ n \)
proof (rule Maclaurin-minus)
show \( \forall m t. m < n \land x < t \land t < 0 \rightarrow \)
(\( (\lambda u. \cos (u + 1/2 * \text{real} m * \pi)) \)) has-real-derivative
\( \cos (t + 1/2 * \text{real} (\text{Suc} m) * \pi) \)) (at t)
by (simp add: cos-expansion-lemma del: of-nat-Suc)
qed (use assms in auto)
then show \( \) \(\) \(\) \(\) \(\)
apply (rule ex-forward, simp)
apply (rule sum.cong[OF refl])
apply (auto simp: cos-coeff-def cos-zero-iff elim: evenE)
done
qed

lemma sin-bound-lemma: \( x = y \Rightarrow |u| \leq v \Rightarrow |(x + u) - y| \leq v \)
for \( x y u v :: \text{real} \)
by auto

lemma Maclaurin-sin-bound: \( |\sin x - (\sum m<n. \sin-coeff m * x ^ m)| \leq \frac{1}{\text{fact} n} * |x| ^ n \)
proof –
have est: \( x \leq 1 \Rightarrow 0 \leq y \Rightarrow x * y \leq 1 * y \) for \( x y :: \text{real} \)
by (rule mult-right-mono) simp-all
let \( \text{?diff} = \lambda(n::\text{nat}) (x::\text{real}). \)
if \( n \) mod 4 = 0 then \( \sin x \)
else if \( n \) mod 4 = 1 then \( \cos x \)
else if \( n \) mod 4 = 2 then \( -\sin x \)
else \( \cos x \)
have diff-0: \( \text{?diff} 0 = \sin \) by simp
have DERIV (\( \text{?diff} m \) x) \( \) \( \text{for} \) \( m \) \( \) \( \) \( \) \( \) \( \)
using mod-exhaust-less-4 [of m]
by (auto simp: mod-Suc intro: derivative-eq-intros)
then have DERIV-diff: \( \forall m x. \) DERIV (\( \text{?diff} m \) x) \( \) \( \text{for} \) \( m \)
by blast
from Maclaurin-all-le [OF diff-0 DERIV-diff]
obtain \( t \) \( \) \( \) \( \) \( \) \( \) \( \) \( \)
where \( t1: |t| \leq |x| \)
and \( t2: \sin x = (\sum m<n. \text{?diff} m 0 / (\text{fact} m) * x ^ m) + \text{?diff} n t / (\text{fact} n) \)
* \( x ^ n \)
by fast
have diff-m-0: \( \text{?diff} m 0 = (if even m then 0 else (1) ^ ((m - \text{Suc} 0) \text{ div} 2)) \)
for \( m \)
using mod-exhaust-less-4 [of m]
by (auto simp: minus-one-power-iff even-even-mod-4-iff [of m] dest: even-mod-4-div-2 odd-mod-4-div-2)

show ?thesis
  apply (subst t2)
  apply (rule sin-bound-lemma)
  apply (rule sum.cong[OF refl])
  unfolding sin-coeff-def
  apply (subst diff-m-0, simp)
  using est
  apply (auto intro: mult-right-mono[where b=1, simplified] mult-right-mono simp: ac-simps divide-inverse power-abs [symmetric] abs-mult)
  done
qed

116 Taylor series

We use MacLaurin and the translation of the expansion point $c$ to $0$ to prove Taylor’s theorem.

lemma Taylor-up:
  assumes INIT: $n > 0 \text{ diff } 0 = f$
  and DERIV: $\forall m \ t. \ m < n \land a \leq t \land t \leq b \rightarrow \text{DERIV (diff } m) \ t :> (\text{diff (Suc } m) \ t)$
  and INTERV: $a \leq c \land c < b$
  shows $\exists t :: \text{real}. \ c < t \land t < b$\n  $f \ b = (\sum_{m<n} \text{ (diff } m \ c / \text{fact } m) * (b - c)^m) + (\text{diff } n \ t / \text{fact } n) * (b - c)^n$

proof
  from INTERV have $0 < b - c$ by arith
  moreover from INIT have $n > 0 \ (\lambda m \ x. \ \text{diff } m \ (x + c)) \ 0 = (\lambda x. \ f \ (x + c))$
    by auto
  moreover have $\forall m \ t. \ m < n \land 0 \leq t \land t \leq b - c \rightarrow \text{DERIV (\lambda x. \ \text{diff } m \ (x + c))} \ t :> \text{diff (Suc } m) \ (t + c)$
    proof (intro strip)
      fix $m \ t$
      assume $m < n \land 0 \leq t \land t \leq b - c$
      with DERIV and INTERV have DERIV (diff $m$) (t + c) :> (diff (Suc $m$) (t + c))
        by auto
      moreover from DERIV-ident and DERIV-const have DERIV $\ (\lambda x. \ x + c) \ t :> 1 + 0$
        by (rule DERIV-add)
      ultimately have DERIV $\ (\lambda x. \ \text{diff } m \ (x + c)) \ t :> \text{diff (Suc } m) \ (t + c) * (1 + 0)$
        by (rule DERIV-chain:2)
      then show DERIV $\ (\lambda x. \ \text{diff } m \ (x + c)) \ t :> \text{diff (Suc } m) \ (t + c)$
        by simp
    qed

qed
ultimately obtain $x$ where
\[
0 < x \land x < b - c \land \\
f (b - c + c) = \\
\left( \sum_{m<n} \text{diff } m (0 + c) \right) / \text{fact } m \times (b - c)^m + \text{diff } n (x + c) / \text{fact } n \\
\]
by (rule Maclaurin [THEN exE])
then show \textit{thesis}
by (smt (verit) sum.cong)
qed

lemma \textit{Taylor-down}:
fixes $a :: \text{real}$ and $n :: \text{nat}$
assumes INIT: $n > 0 \land \text{diff } 0 = f$
and DERIV: $(\forall m \ t. \ m < n \land a \leq t \land t \leq b \longrightarrow \text{DERIV (diff } m \ t :> \text{diff (Suc } m \ t))$
and INTERV: $a < c \land c < b$
shows $\exists t. \ a \leq t \land t \leq c \land \\
f a = \left( \sum_{m<n} \left( \text{diff } m c / \text{fact } m \right) \right) \times (a - c)^m + \text{diff } n t / \text{fact } n \times (a - c)^n$
proof –
from INTERV have $a - c < 0$ by arith
moreover from INIT have $n > 0 \land (\lambda x. \ \text{diff } m (x + c)) 0 = (\lambda x. \ f (x + c))$
by auto
moreover
have $\forall m \ t. \ m < n \land a - c \leq t \land t \leq 0 \longrightarrow \text{DERIV (\lambda x. \ f (x + c)) } t :> \text{diff (Suc } m \ t + c)$
proof (rule allI impI)+
fix $m \ t$
assume $m < n \land a - c \leq t \land t \leq 0$

with DERIV and INTERV have \text{DERIV (diff } m \ t + c) :> \text{diff (Suc } m \ t + c)$
by auto

moreover from DERIV-ident and DERIV-const have \text{DERIV (\lambda x. \ f (x + c)) } t :> 1 + 0
by (rule DERIV-add)
ultimately show \text{DERIV (\lambda x. \ f (x + c)) } t :> \text{diff (Suc } m \ t + c)
using \text{DERIV-chain2 DERIV-shift by blast}
qed

ultimately obtain $x$ where
\[
a - c < x \land x < 0 \land \\
f (a - c + c) = \\
\left( \sum_{m<n} \text{diff } m (0 + c) \right) / \text{fact } m \times (a - c)^m + \text{diff } n (x + c) / \text{fact } n \\
\]
by (rule Maclaurin-minus [THEN exE])
then have $a < x + c \land x + c < c \land \\
f a = \left( \sum_{m<n} \left( \text{diff } m c / \text{fact } m \right) \right) \times (a - c)^m + \text{diff } n (x + c) / \text{fact } n \times (a - c)^n$
by fastforce
then show \textit{thesis} by fastforce
theorem Taylor:
    fixes a :: real and n :: nat
    assumes INIT: n > 0 diff 0 = f
    and DERIV: \( \forall m. m < n \land a \leq t \land t \leq b \rightarrow \text{DERIV} (\text{diff } m) t \rightarrow \text{diff} \) 
    (Suc m) t
    and INTERV: a \leq c \land c \leq b \land a \leq x \land x \leq b \land x \neq c
    shows \( \exists t. \) 
    \( (\text{if } x < c \text{ then } x < t \land t < c \text{ else } c < t \land t < x) \land 
    f x = (\sum_{m<n.} (\text{diff } m c / \text{fact } m) \ast (x - c)^m) + (\text{diff } n t / \text{fact } n) \ast (x - c)^n \) 
    proof (cases x < c)
    case True
    note INIT
    moreover have \( \forall m t. m < n \land x \leq t \land t \leq b \rightarrow \text{DERIV} (\text{diff } m) t \rightarrow \text{diff} \) 
    (Suc m) t
    using DERIV and INTERV by fastforce
    ultimately show ?thesis
    using True INTERV Taylor-down by simp
    next
    case False
    note INIT
    moreover have \( \forall m t. m < n \land a \leq t \land t \leq x \rightarrow \text{DERIV} (\text{diff } m) t \rightarrow \text{diff} \) 
    (Suc m) t
    using DERIV and INTERV by fastforce
    ultimately show ?thesis
    using Taylor-up INTERV False by simp
qed

117 Comprehensive Complex Theory

theory Complex-Main
imports
  Complex
  MacLaurin
begin

end

References

REFERENCES


