

Isabelle/HOL — Higher-Order Logic

January 18, 2026

Contents

1	Loading the code generator and related modules	33
2	The basis of Higher-Order Logic	34
2.1	Primitive logic	34
2.1.1	Core syntax	34
2.1.2	Defined connectives and quantifiers	35
2.1.3	Additional concrete syntax	35
2.1.4	Axioms and basic definitions	37
2.2	Fundamental rules	38
2.2.1	Equality	38
2.2.2	Congruence rules for application	39
2.2.3	Equality of booleans – iff	39
2.2.4	True (1)	39
2.2.5	Universal quantifier (1)	40
2.2.6	False	40
2.2.7	Negation	40
2.2.8	Implication	40
2.2.9	Disjunction (1)	41
2.2.10	Derivation of <i>iffI</i>	41
2.2.11	True (2)	41
2.2.12	Universal quantifier (2)	42
2.2.13	Existential quantifier	42
2.2.14	Conjunction	42
2.2.15	Disjunction (2)	42
2.2.16	Classical logic	42
2.2.17	Unique existence	43
2.2.18	Classical intro rules for disjunction and existential quantifiers	44
2.2.19	Intuitionistic Reasoning	45
2.2.20	Atomizing meta-level connectives	45
2.2.21	Atomizing elimination rules	46

2.3	Package setup	46
2.3.1	Sledgehammer setup	46
2.3.2	Classical Reasoner setup	46
2.3.3	THE: definite description operator	48
2.3.4	Simplifier	48
2.3.5	Generic cases and induction	56
2.3.6	Coherent logic	58
2.3.7	Reorienting equalities	58
2.4	Other simple lemmas and lemma duplicates	58
2.5	Basic ML bindings	58
3	<i>NO-MATCH</i> simproc	60
3.1	Code generator setup	60
3.1.1	Generic code generator preprocessor setup	60
3.1.2	Generic code generator foundation	61
3.1.3	Equality	62
3.1.4	Generic code generator target languages	62
3.1.5	Evaluation and normalization by evaluation	64
3.2	Counterexample Search Units	64
3.2.1	Quickcheck	64
3.2.2	Nitpick setup	64
3.3	Preprocessing for the predicate compiler	64
3.4	Legacy tactics and ML bindings	64
4	Abstract orderings	64
4.1	Abstract ordering	65
4.2	Syntactic orders	67
4.3	Quasi orders	68
4.4	Partial orders	69
4.5	Linear (total) orders	71
4.6	Reasoning tools setup	73
4.7	Bounded quantifiers	74
4.8	Transitivity reasoning	76
4.9	min and max – fundamental	80
4.10	(Unique) top and bottom elements	81
4.11	Dense orders	82
4.12	Wellorders	83
4.13	Order on <i>bool</i>	84
4.14	Order on $- \Rightarrow -$	85
4.15	Order on unary and binary predicates	87
4.16	Name duplicates	87

5	Groups, also combined with orderings	88
5.1	Dynamic facts	88
5.2	Abstract structures	89
5.3	Generic operations	90
5.4	Semigroups and Monoids	91
5.5	Groups	95
5.6	(Partially) Ordered Groups	98
5.7	Support for reasoning about signs	100
5.8	Canonically ordered monoids	108
5.9	Tools setup	110
6	Abstract lattices	111
6.1	Abstract semilattice	111
6.2	Syntactic infimum and supremum operations	113
6.3	Concrete lattices	113
6.3.1	Intro and elim rules	114
6.3.2	Equational laws	115
6.3.3	Strict order	117
6.4	Distributive lattices	118
6.5	Bounded lattices	118
6.6	<i>min/max</i> as special case of lattice	120
6.7	Uniqueness of inf and sup	121
6.8	Lattice on $- \Rightarrow -$	122
7	Boolean Algebras	123
7.1	Abstract boolean algebra	123
7.1.1	Complement	123
7.1.2	Conjunction	124
7.1.3	Disjunction	124
7.1.4	De Morgan's Laws	125
7.2	Symmetric Difference	125
7.3	Type classes	126
7.4	Lattice on <i>bool</i>	128
7.5	Lattice on unary and binary predicates	129
7.6	Simproc setup	130
8	Set theory for higher-order logic	131
8.1	Sets as predicates	131
8.2	Subsets and bounded quantifiers	134
8.3	Basic operations	138
8.3.1	Subsets	138
8.3.2	Equality	139
8.3.3	The empty set	140
8.3.4	The universal set – UNIV	140

8.3.5	The Powerset operator – <i>Pow</i>	141
8.3.6	Set complement	141
8.3.7	Binary intersection	142
8.3.8	Binary union	142
8.3.9	Set difference	143
8.3.10	Augmenting a set – <i>insert</i>	143
8.3.11	Singletons, using <i>insert</i>	144
8.3.12	Image of a set under a function	145
8.3.13	Some rules with <i>if</i>	148
8.4	Further operations and lemmas	149
8.4.1	The “proper subset” relation	149
8.4.2	Derived rules involving subsets.	150
8.4.3	Equalities involving union, intersection, inclusion, etc.	151
8.4.4	Monotonicity of various operations	161
8.4.5	Inverse image of a function	162
8.4.6	Singleton sets	164
8.4.7	Getting the contents of a singleton set	164
8.4.8	Monad operation	164
8.4.9	Operations for execution	165
9	HOL type definitions	168
10	Notions about functions	169
10.1	The Identity Function <i>id</i>	169
10.2	The Composition Operator $f \circ g$	170
10.3	The Forward Composition Operator <i>fcomp</i>	171
10.4	Mapping functions	171
10.5	Injectivity and Bijectivity	172
10.5.1	Inj/surj/bij of Algebraic Operations	180
10.6	Function Updating	182
10.7	<i>override-on</i>	183
10.8	Inversion of injective functions	184
10.9	Monotonicity	185
10.9.1	Specializations For <i>ord</i> Type Class And More	186
10.9.2	Least value operator	191
10.10	Setup	192
10.10.1	Proof tools	192
10.10.2	Functorial structure of types	192
11	Complete lattices	192
11.1	Syntactic infimum and supremum operations	192
11.2	Abstract complete lattices	194
11.3	Complete lattice on <i>bool</i>	202
11.4	Complete lattice on $- \Rightarrow -$	203

11.5	Complete lattice on unary and binary predicates	204
11.6	Complete lattice on - <i>set</i>	205
11.6.1	Inter	206
11.6.2	Intersections of families	207
11.6.3	Union	209
11.6.4	Unions of families	210
11.6.5	Distributive laws	213
11.7	Injections and bijections	214
11.7.1	Complement	215
11.7.2	Miniscoping and maxiscoping	215
12	Wrapping Existing Freely Generated Type's Constructors	217
13	Knaster-Tarski Fixpoint Theorem and inductive definitions	217
13.1	Least fixed points	218
13.2	General induction rules for least fixed points	218
13.3	Greatest fixed points	219
13.4	Coinduction rules for greatest fixed points	220
13.5	Even Stronger Coinduction Rule, by Martin Coen	221
13.6	Rules for fixed point calculus	221
13.7	Inductive predicates and sets	222
13.8	The Schroeder-Bernstein Theorem	222
13.9	Inductive datatypes and primitive recursion	223
14	Cartesian products	223
14.1	<i>bool</i> is a datatype	223
14.2	The <i>unit</i> type	224
14.3	The product type	227
14.3.1	Type definition	227
14.3.2	Tuple syntax	228
14.3.3	Code generator setup	229
14.3.4	Fundamental operations and properties	229
14.3.5	Derived operations	234
14.4	Code generator setup for paired and tripled bounded set comprehension	243
14.5	Simproc for rewriting a set comprehension into a pointfree expression	244
14.6	Lemmas about disjointness	244
14.7	Inductively defined sets	244
14.8	Legacy theorem bindings and duplicates	244

15 The Disjoint Sum of Two Types	244
15.1 Construction of the sum type and its basic abstract operations	244
15.2 Projections	247
15.3 The Disjoint Sum of Sets	247
16 Rings	248
16.1 Semirings and rings	248
16.2 Abstract divisibility	250
16.3 Towards integral domains	255
16.4 (Partial) Division	258
16.5 Quotient and remainder in integral domains	271
16.6 Interlude: basic tool support for algebraic and arithmetic calculations	272
16.7 Ordered semirings and rings	272
16.8 Dioids	288
17 Natural numbers	288
17.1 Type <i>ind</i>	288
17.2 Type <i>nat</i>	288
17.3 Arithmetic operators	291
17.3.1 Addition	292
17.3.2 Difference	293
17.3.3 Multiplication	293
17.4 Orders on <i>nat</i>	294
17.4.1 Operation definition	294
17.4.2 Introduction properties	295
17.4.3 Elimination properties	296
17.4.4 Inductive (?) properties	296
17.4.5 Monotonicity of Addition	300
17.4.6 <i>min</i> and <i>max</i>	301
17.4.7 Additional theorems about (\leq)	303
17.4.8 More results about difference	307
17.4.9 Monotonicity of multiplication	308
17.5 Natural operation of natural numbers on functions	310
17.6 Kleene iteration	312
17.7 Embedding of the naturals into any <i>semiring-1: of-nat</i>	313
17.8 The set of natural numbers	316
17.9 Further arithmetic facts concerning the natural numbers	317
17.9.1 Greatest operator	322
17.10 Monotonicity of <i>funpow</i>	322
17.11 Kleene's fixed point theorem for continuous functions	323
17.12 The divides relation on <i>nat</i>	323
17.13 Aliases	325
17.14 Size of a datatype value	326

17.15	Code module namespace	326
18	Fields	326
18.1	Division rings	326
18.2	Fields	331
18.3	Ordered fields	334
19	Relations – as sets of pairs, and binary predicates	346
19.1	Fundamental	347
19.1.1	Relations as sets of pairs	347
19.1.2	Conversions between set and predicate relations	347
19.2	Properties of relations	349
19.2.1	Reflexivity	349
19.2.2	Irreflexivity	351
19.2.3	Asymmetry	353
19.2.4	Symmetry	355
19.2.5	Antisymmetry	357
19.2.6	Transitivity	360
19.2.7	Totality	363
19.2.8	Left uniqueness	365
19.2.9	Right uniqueness	366
19.3	Relation operations	367
19.3.1	The identity relation	367
19.3.2	Diagonal: identity over a set	368
19.3.3	Composition	369
19.3.4	Converse	371
19.3.5	Domain, range and field	374
19.3.6	Image of a set under a relation	377
19.3.7	Inverse image	379
19.3.8	Powerset	380
20	Finite sets	380
20.1	Predicate for finite sets	380
20.1.1	Choice principles	381
20.1.2	Finite sets are the images of initial segments of natural numbers	381
20.2	Finiteness and common set operations	381
20.3	Further induction rules on finite sets	386
20.4	Class <i>finite</i>	387
20.5	A basic fold functional for finite sets	388
20.5.1	From <i>fold-graph</i> to <i>fold</i>	389
20.5.2	Liftings to <i>comp-fun-commute-on</i> etc.	392
20.5.3	<i>UNIV</i> as carrier set	392
20.5.4	Expressing set operations via <i>fold</i>	393

20.5.5	Expressing relation operations via <i>fold</i>	396
20.6	Locales as mini-packages for fold operations	397
20.6.1	The natural case	397
20.6.2	With idempotency	397
20.6.3	<i>UNIV</i> as the carrier set	398
20.7	Finite cardinality	399
20.7.1	Cardinality of image	404
20.7.2	Pigeonhole Principles	406
20.7.3	Cardinality of sums	406
20.8	Minimal and maximal elements of finite sets	406
20.8.1	Finite orders	407
20.8.2	Relating injectivity and surjectivity	408
20.9	Infinite Sets	408
20.10	The finite powerset operator	410
21	Reflexive and Transitive closure of a relation	410
21.1	Reflexive closure	412
21.2	Reflexive-transitive closure	413
21.3	Transitive closure	417
21.4	Symmetric closure	424
21.5	The power operation on relations	424
21.6	Bounded transitive closure	430
21.7	Acyclic relations	430
21.8	Setup of transitivity reasoner	431
22	Well-founded Recursion	431
22.1	Basic Definitions	432
22.2	Equivalence of Definitions	432
22.3	Induction Principles	432
22.4	Introduction Rules	433
22.5	Ordering Properties	433
22.6	Basic Results	434
22.6.1	Minimal-element characterization of well-foundedness	435
22.6.2	Finite characterization of well-foundedness	435
22.6.3	Antimonotonicity	435
22.6.4	Equivalence between <i>wfp-on</i> and <i>wfp</i>	436
22.6.5	Well-foundedness of transitive closure	436
22.6.6	Well-foundedness of image	438
22.7	Well-Foundedness Results for Unions	438
22.8	Well-Foundedness of Composition	439
22.9	Acyclic relations	439
22.9.1	Wellfoundedness of finite acyclic relations	440
22.10	<i>nat</i> is well-founded	440
22.11	Accessible Part	441

22.12	Tools for building wellfounded relations	442
22.12.1	Conversion to a known well-founded relation	443
22.12.2	Measure functions into <i>nat</i>	443
22.12.3	Lexicographic combinations	444
22.12.4	Bounded increase must terminate	446
22.13	Code Generation Setup	446
23	Well-Founded Recursion Combinator	447
23.0.1	Well-founded recursion via genuine fixpoints	448
23.1	Wellfoundedness of <i>samefst</i>	448
24	Orders as Relations	448
24.1	Orders on a set	448
24.2	Orders on the field	450
24.3	Relations given by a predicate and the field	451
24.4	Orders on a type	451
24.5	Order-like relations	452
24.5.1	Auxiliaries	452
24.5.2	The upper and lower bounds operators	452
24.6	Variations on Well-Founded Relations	455
24.6.1	Characterizations of well-foundedness	455
24.6.2	Characterizations of well-foundedness	455
25	Hilbert’s Epsilon-Operator and the Axiom of Choice	456
25.1	Hilbert’s epsilon	456
25.2	Hilbert’s Epsilon-operator	456
25.3	Axiom of Choice, Proved Using the Description Operator	457
25.4	Getting an element of a nonempty set	458
25.5	Function Inverse	458
25.6	Other Consequences of Hilbert’s Epsilon	463
25.7	An aside: bounded accessible part	464
25.8	More on injections, bijections, and inverses	464
25.9	Specification package – Hilbertized version	466
25.10	Complete Distributive Lattices – Properties depending on Hilbert Choice	466
26	Zorn’s Lemma and the Well-ordering Theorem	468
26.1	Zorn’s Lemma for the Subset Relation	468
26.1.1	Results that do not require an order	468
26.1.2	Hausdorff’s Maximum Principle	471
26.1.3	Results for the proper subset relation	471
26.1.4	Zorn’s lemma	472
26.2	Zorn’s Lemma for Partial Orders	472
26.3	Other variants of Zorn’s Lemma	473

26.4	The Well Ordering Theorem	474
27	Well-Order Relations as Needed by Bounded Natural Functors	476
27.1	Auxiliaries	476
27.2	Well-founded induction and recursion adapted to non-strict well-order relations	477
27.3	The notions of maximum, minimum, supremum, successor and order filter	478
27.3.1	Properties of max2	478
27.3.2	Existence and uniqueness for isMinim and well-definedness of minim	479
27.3.3	Properties of minim	479
27.3.4	Properties of successor	480
27.3.5	Properties of order filters	480
27.3.6	Other properties	481
28	Well-Order Embeddings as Needed by Bounded Natural Functors	481
28.1	Auxiliaries	482
28.2	(Well-order) embeddings, strict embeddings, isomorphisms and order-compatible functions	482
28.3	Given any two well-orders, one can be embedded in the other	484
28.4	Uniqueness of embeddings	485
28.5	More properties of embeddings, strict embeddings and isomorphisms	486
29	Constructions on Wellorders as Needed by Bounded Natural Functors	488
29.1	Restriction to a set	489
29.2	Order filters versus restrictions and embeddings	490
29.3	The strict inclusion on proper ofilters is well-founded	491
29.4	Ordering the well-orders by existence of embeddings	491
29.5	$< o$ is well-founded	496
29.6	Copy via direct images	497
29.7	Bounded square	498
30	Cardinal-Order Relations as Needed by Bounded Natural Functors	501
30.1	Cardinal orders	502
30.2	Cardinal of a set	503
30.3	Cardinals versus set operations on arbitrary sets	505
30.4	Cardinals versus set operations involving infinite sets	510
30.5	The cardinal ω and the finite cardinals	512

30.5.1	First as well-orders	512
30.5.2	Then as cardinals	513
30.6	The successor of a cardinal	514
30.7	Regular cardinals	517
30.8	Others	518
30.9	Regular vs. stable cardinals	518
31	Cardinal Arithmetic as Needed by Bounded Natural Func-	
	tors	520
31.1	Zero	520
31.2	(In)finite cardinals	521
31.3	Binary sum	522
31.4	One	523
31.5	Two	524
31.6	Family sum	524
31.7	Product	524
31.8	Exponentiation	526
32	Function Definition Base	530
33	Definition of Bounded Natural Functors	531
34	Composition of Bounded Natural Functors	536
35	Registration of Basic Types as Bounded Natural Functors	540
36	Shared Fixpoint Operations on Bounded Natural Functors	542
37	Least Fixpoint (Datatype) Operation on Bounded Natural	
	Functors	548
38	Equivalence Relations in Higher-Order Set Theory	551
38.1	Equivalence relations – set version	551
38.2	Equivalence classes	552
38.3	Quotients	553
38.4	Refinement of one equivalence relation WRT another	554
38.5	Defining unary operations upon equivalence classes	554
38.6	Defining binary operations upon equivalence classes	555
38.7	Quotients and finiteness	557
38.8	Kernel of a Function	557
38.9	Projection	557
38.10	Equivalence relations – predicate version	558
38.11	Equivalence closure	560

39 MESON Proof Method	564
39.1 Negation Normal Form	564
39.2 Pulling out the existential quantifiers	564
39.3 Lemmas for Forward Proof	565
39.4 Clausification helper	566
39.5 Skolemization helpers	567
39.6 Meson package	567
40 Automatic Theorem Provers (ATPs)	567
40.1 ATP problems and proofs	567
40.2 Higher-order reasoning helpers	567
40.3 Basic connection between ATPs and HOL	570
41 Metis Proof Method	570
41.1 Literal selection and lambda-lifting helpers	570
41.2 Metis package	571
42 Generic theorem transfer using relations	571
42.1 Relator for function space	571
42.2 Transfer method	572
42.3 Predicates on relations, i.e. “class constraints”	573
42.4 Properties of relators	576
42.5 Transfer rules	577
42.6 <i>of-bool</i> and <i>of-nat</i>	582
43 Lifting package	582
43.1 Function map	582
43.2 Quotient Predicate	583
43.3 Quotient composition	586
43.4 Respects predicate	586
43.5 Domains	591
43.6 ML setup	592
44 Definition of Quotient Types	592
44.1 Quotient Predicate	593
44.2 lemmas for regularisation of ball and bex	596
44.3 Bounded abstraction	597
44.4 <i>Bex1-rel</i> quantifier	598
44.5 Various respects and preserve lemmas	599
44.6 Quotient composition	601
44.7 Quotient3 to Quotient	601
44.8 ML setup	602
44.9 Methods / Interface	603
45 Lifting of BNFs	603

46 Binary Numerals	606
46.1 The <i>num</i> type	606
46.2 Numeral operations	607
46.3 Binary numerals	610
46.4 Class-specific numeral rules	611
46.4.1 Structures with addition: class <i>numeral</i>	611
46.4.2 Structures with negation: class <i>neg-numeral</i>	611
46.4.3 Structures with multiplication: class <i>semiring-numeral</i>	614
46.4.4 Structures with a zero: class <i>semiring-1</i>	615
46.4.5 Equality: class <i>semiring-char-0</i>	615
46.4.6 Comparisons: class <i>linordered-nonzero-semiring</i>	616
46.4.7 Multiplication and negation: class <i>ring-1</i>	618
46.4.8 Equality using <i>iszero</i> for rings with non-zero characteristic	619
46.4.9 Equality and negation: class <i>ring-char-0</i>	620
46.4.10 Structures with negation and order: class <i>linordered-idom</i>	621
46.4.11 Natural numbers	624
46.5 Particular lemmas concerning $\mathbb{Z}::'a$	627
46.6 Numeral equations as default simplification rules	627
46.6.1 Special Simplification for Constants	628
46.6.2 Optional Simplification Rules Involving Constants	629
46.7 Setting up simprocs	630
46.7.1 Simplification of arithmetic operations on integer constants	631
46.7.2 Simplification of arithmetic when nested to the right	632
46.8 Code module namespace	632
46.9 Printing of evaluated natural numbers as numerals	632
46.10 More on auxiliary conversion	633
47 Exponentiation	633
47.1 Powers for Arbitrary Monoids	633
47.2 Exponentiation on ordered types	640
47.3 Miscellaneous rules	647
47.4 Exponentiation for the Natural Numbers	648
47.4.1 Cardinality of the Powerset	649
47.5 Code generator tweak	649
48 Big sum and product over finite (non-empty) sets	649
48.1 Generic monoid operation over a set	649
48.1.1 Standard sum or product indexed by a finite set	650
48.1.2 HOL Light variant: sum/product indexed by the non-neutral subset	656
48.2 Generalized summation over a set	657
48.2.1 Properties in more restricted classes of structures	658

48.2.2	Cardinality as special case of <i>sum</i>	663
48.2.3	Cardinality of products	665
48.3	Generalized product over a set	665
48.3.1	Properties in more restricted classes of structures	666
49	Chain-complete partial orders and their fixpoints	671
49.1	Chains	671
49.2	Chain-complete partial orders	671
49.3	Transfinite iteration of a function	672
49.4	Fixpoint combinator	672
49.5	Fixpoint induction	673
50	Datatype option	674
50.0.1	Operations	676
50.1	Transfer rules for the Transfer package	681
50.1.1	Interaction with finite sets	681
50.1.2	Code generator setup	681
51	Partial Function Definitions	682
51.1	Axiomatic setup	683
51.2	Flat interpretation: tailrec and option	685
52	Reconstructing external resolution proofs for propositional logic	687
53	Function Definitions and Termination Proofs	687
53.1	Definitions with default value	688
53.2	Measure functions	689
53.3	Congruence rules	689
53.4	Simp rules for termination proofs	689
53.5	Decomposition	690
53.6	Reduction pairs	690
53.7	Concrete orders for SCNP termination proofs	690
53.8	Yet more induction principles on the natural numbers	692
53.9	Tool setup	692
54	The Integers as Equivalence Classes over Pairs of Natural Numbers	692
54.1	Definition of integers as a quotient type	692
54.2	Integers form a commutative ring	692
54.3	Integers are totally ordered	693
54.4	Ordering properties of arithmetic operations	694
54.5	Embedding of the Integers into any <i>ring-1</i> : <i>of-int</i>	695
54.6	Magnitude of an Integer, as a Natural Number: <i>nat</i>	700
54.7	Lemmas about the Function <i>of-nat</i> and Orderings	703

54.8	Cases and induction	704
54.8.1	Binary comparisons	705
54.8.2	Comparisons, for Ordered Rings	705
54.9	The Set of Integers	705
54.10	<i>sum</i> and <i>prod</i>	708
54.11	Setting up simplification procedures	708
54.12	More Inequality Reasoning	708
54.13	The functions <i>nat</i> and <i>int</i>	709
54.14	Induction principles for <i>int</i>	711
54.15	Intermediate value theorems	712
54.16	Products and 1, by T. M. Rasmussen	712
54.17	The divides relation	713
54.18	Powers with integer exponents	715
54.19	Finiteness of intervals	720
54.20	Configuration of the code generator	720
54.21	Duplicates	724
55	Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets	724
55.1	Generic lattice operations over a set	724
55.1.1	Without neutral element	724
55.1.2	With neutral element	726
55.2	Lattice operations on finite sets	728
55.3	Infimum and Supremum over non-empty sets	729
55.4	Minimum and Maximum over non-empty sets	730
55.5	An aside: code generation for <i>LEAST</i> and <i>GREATEST</i> . . .	736
55.6	Arg Min	737
55.7	Arg Max	739
56	Division in euclidean (semi)rings	740
56.1	Euclidean (semi)rings with explicit division and remainder . .	740
56.2	Euclidean (semi)rings with cancel rules	742
56.3	Uniquely determined division	747
56.4	Division on <i>nat</i>	748
56.5	Division on <i>int</i>	755
56.5.1	Basic instantiation	755
56.5.2	Algebraic foundations	756
56.5.3	Basic conversions	757
56.5.4	Euclidean division	758
56.5.5	Trivial reduction steps	759
56.5.6	More uniqueness rules	759
56.5.7	Laws for unary minus	760
56.5.8	Borders	760
56.5.9	Splitting Rules for div and mod	761

56.5.10 Algebraic rewrites	761
56.5.11 Distributive laws for conversions.	762
56.5.12 Monotonicity in the First Argument (Dividend)	762
56.5.13 Monotonicity in the Second Argument (Divisor)	762
56.5.14 Quotients of Signs	763
56.5.15 Further properties	764
56.5.16 Computing <i>div</i> and <i>mod</i> by shifting	765
56.6 Code generation	765
57 Parity in rings and semirings	766
57.1 Ring structures with parity and <i>even/odd</i> predicates	766
57.2 Instance for <i>nat</i>	769
57.3 Parity and powers	771
57.4 Instance for <i>int</i>	773
57.5 Special case: euclidean rings structurally containing the nat- ural numbers	774
57.6 Generic symbolic computations	777
57.6.1 Computation by simplification	782
57.7 Computing congruences modulo 2^q	782
58 Combination and Cancellation Simprocs for Numeral Ex- pressions	783
59 Semiring normalization	785
60 Groebner bases	787
60.1 Groebner Bases	787
61 Set intervals	789
61.1 Various equivalences	791
61.2 Logical Equivalences for Set Inclusion and Equality	792
61.3 Two-sided intervals	793
61.3.1 Emptiness, singletons, subset	794
61.4 Infinite intervals	800
61.4.1 Intersection	801
61.5 Intervals of natural numbers	803
61.5.1 The Constant <i>lessThan</i>	803
61.5.2 The Constant <i>greaterThan</i>	803
61.5.3 The Constant <i>atLeast</i>	804
61.5.4 The Constant <i>atMost</i>	804
61.5.5 The Constant <i>atLeastLessThan</i>	804
61.5.6 The Constant <i>atLeastAtMost</i>	805
61.5.7 Intervals of nats with <i>Suc</i>	805
61.5.8 Intervals and numerals	806

61.5.9	Image	806
61.5.10	Finiteness	810
61.5.11	Proving Inclusions and Equalities between Unions . . .	811
61.5.12	Cardinality	811
61.6	Intervals of integers	813
61.6.1	Finiteness	813
61.6.2	Cardinality	813
61.7	Lemmas useful with the summation operator sum	814
61.7.1	Disjoint Unions	815
61.7.2	Disjoint Intersections	815
61.7.3	Some Differences	816
61.7.4	Some Subset Conditions	816
61.8	Generic big monoid operation over intervals	816
61.9	Summation indexed over intervals	819
61.9.1	Shifting bounds	823
61.9.2	Telescoping sums	824
61.9.3	The formula for geometric sums	825
61.9.4	Geometric progressions	826
61.9.5	The formulae for arithmetic sums	826
61.9.6	Division remainder	828
61.10	Products indexed over intervals	828
61.10.1	Telescoping products	829
61.11	Efficient folding over intervals	829
62	Decision Procedure for Presburger Arithmetic	830
62.1	The $-\infty$ and $+\infty$ Properties	830
62.2	The A and B sets	831
62.3	Cooper's Theorem $-\infty$ and $+\infty$ Version	833
62.3.1	First some trivial facts about periodic sets or predicates	833
62.3.2	The $-\infty$ Version	833
62.3.3	The $+\infty$ Version	833
62.4	Nice facts about division by $\not\sim :: 'a$	836
63	Bindings to Satisfiability Modulo Theories (SMT) solvers based on SMT-LIB 2	837
63.1	A skolemization tactic and proof method	837
63.2	Triggers for quantifier instantiation	837
63.3	Higher-order encoding	838
63.4	Normalization	838
63.5	Integer division and modulo for Z3	839
63.6	Extra theorems for veriT reconstruction	840
63.7	Setup	847
63.8	Configuration	848
63.9	General configuration options	848

63.10	Certificates	849
63.11	Tracing	849
63.12	Schematic rules for Z3 proof reconstruction	849
64	Sledgehammer: Isabelle–ATP Linkup	852
65	Setup for Lifting/Transfer for the set type	852
65.1	Relator and predicate properties	852
65.2	Quotient theorem for the Lifting package	854
65.3	Transfer rules for the Transfer package	854
65.3.1	Unconditional transfer rules	854
65.3.2	Rules requiring bi-unique, bi-total or right-total relations	855
66	The datatype of finite lists	858
66.1	Basic list processing functions	859
66.1.1	List comprehension	867
66.1.2	<code>[]</code> and <code>(#)</code>	868
66.1.3	<code>length</code>	868
66.1.4	<code>@ – append</code>	870
66.1.5	<code>map</code>	872
66.1.6	<code>rev</code>	874
66.1.7	<code>set</code>	876
66.1.8	<code>concat</code>	878
66.1.9	<code>filter</code>	879
66.1.10	List partitioning	881
66.1.11	<code>(!)</code>	882
66.1.12	<code>list-update</code>	883
66.1.13	<code>last</code> and <code>butlast</code>	885
66.1.14	<code>take</code> and <code>drop</code>	887
66.1.15	<code>takeWhile</code> and <code>dropWhile</code>	892
66.1.16	<code>zip</code>	895
66.1.17	<code>list-all2</code>	899
66.1.18	<code>List.product</code> and <code>product-lists</code>	901
66.1.19	<code>fold</code> with natural argument order	902
66.1.20	Fold variants: <code>foldr</code> and <code>foldl</code>	905
66.1.21	<code>upt</code>	906
66.1.22	<code>upto</code> : interval-list on <code>int</code>	909
66.1.23	<code>successively</code>	910
66.1.24	<code>distinct</code> and <code>remdups</code> and <code>remdups-adj</code>	911
66.2	<code>distinct-adj</code>	918
66.2.1	<code>insert</code>	919
66.2.2	<code>List.union</code>	919
66.2.3	<code>find</code>	919

66.2.4	<i>count-list</i>	920
66.2.5	<i>List.extract</i>	921
66.2.6	<i>remove1</i>	921
66.2.7	<i>removeAll</i>	922
66.2.8	<i>minus-list-mset</i>	924
66.2.9	<i>minus-list-set</i>	925
66.2.10	<i>inter-list-set</i>	926
66.2.11	<i>replicate</i>	926
66.2.12	<i>enumerate</i>	929
66.2.13	<i>rotate1</i> and <i>rotate</i>	930
66.2.14	<i>nths</i> — a generalization of (!) to sets	932
66.2.15	<i>subseqs</i> and <i>List.n-lists</i>	934
66.2.16	<i>splice</i>	934
66.2.17	<i>shuffles</i>	935
66.2.18	Transpose	936
66.2.19	<i>min</i> and <i>arg-min</i>	937
66.2.20	(In)finiteness	937
66.3	Sorting	939
66.3.1	<i>sorted-wrt</i>	939
66.3.2	<i>sorted</i>	940
66.3.3	Sorting functions	943
66.3.4	<i>transpose</i> on sorted lists	947
66.3.5	<i>sorted-key-list-of-set</i>	948
66.3.6	<i>lists</i> : the list-forming operator over sets	953
66.3.7	Inductive definition for membership	954
66.3.8	Lists as Cartesian products	955
66.3.9	Transitive Closure on Lists	955
66.4	Relations on Lists	956
66.4.1	Length Lexicographic Ordering	956
66.4.2	Lexicographic Ordering	958
66.4.3	Lexicographic combination of measure functions	963
66.4.4	Lifting Relations to Lists: one element	964
66.4.5	Lifting Relations to Lists: all elements	966
66.5	Size function	968
66.6	Monad operation	968
66.7	Code generation	969
66.7.1	Counterparts for set-related operations	969
66.7.2	Special implementations	975
66.7.3	Operations for optimization and efficiency	976
66.7.4	Implementation of sets by lists	976
66.7.5	Pretty lists	979
66.7.6	Use convenient predefined operations	979
66.8	Setup for Lifting/Transfer	980
66.8.1	Transfer rules for the Transfer package	980

66.9 Misc	984
67 Sum and product over lists	984
67.1 List summation	986
67.2 Horner sums	991
67.3 Further facts about <i>List.n-lists</i>	993
67.4 Tools setup	993
67.5 List product	993
68 Bit operations in suitable algebraic structures	995
68.1 Abstract bit structures	995
68.2 Bit operations	1000
68.3 Common algebraic structure	1014
68.4 Instance <i>int</i>	1016
68.5 Instance <i>nat</i>	1024
68.6 Symbolic computations on numeral expressions	1029
68.7 Symbolic computations for code generation	1038
68.8 More properties	1044
68.9 Bit concatenation	1044
68.10 Taking bits with sign propagation	1045
68.11 Key ideas of bit operations	1050
68.12 Lemma duplicates and other	1051
69 Numeric types for code generation onto target language numerals only	1054
69.1 Type of target language integers	1055
69.2 Code theorems for target language integers	1062
69.3 Serializer setup for target language integers	1071
69.4 Type of target language naturals	1077
69.5 Inductive representation of target language naturals	1082
69.6 Code refinement for target language naturals	1083
70 A HOL random engine	1086
70.1 Auxiliary functions	1086
70.2 Random seeds	1086
70.3 Base selectors	1087
70.4 <i>ML</i> interface	1088
71 Maps	1088
71.1 <i>empty</i>	1091
71.2 <i>map-upd</i>	1091
71.3 <i>map-of</i>	1091
71.4 <i>map-option</i> related	1093
71.5 <i>map-comp</i> related	1093

71.6	<code>++</code>	1094
71.7	<code>restrict-map</code>	1095
71.8	<code>map-upds</code>	1096
71.9	<code>dom</code>	1097
71.10	<code>ran</code>	1099
71.11	<code>graph</code>	1100
71.12	<code>map-le</code>	1101
71.13	Various	1102
72	Finite types as explicit enumerations	1103
72.1	Class <code>enum</code>	1103
72.2	Implementations using <code>enum</code>	1104
72.2.1	Unbounded operations and quantifiers	1104
72.2.2	An executable choice operator	1104
72.2.3	Equality and order on functions	1105
72.2.4	Operations on relations	1105
72.2.5	Bounded accessible part	1106
72.3	Default instances for <code>enum</code>	1107
72.4	Small finite types	1110
72.5	Closing up	1121
73	Character and string types	1121
73.1	Strings as list of bytes	1121
73.1.1	Bytes as datatype	1121
73.2	Strings as dedicated type for target language code generation	1127
73.2.1	Logical specification	1127
73.2.2	Syntactic representation	1128
73.2.3	Operations	1129
73.2.4	Executable conversions	1131
73.2.5	Technical code generation setup	1132
73.2.6	Code generation utility	1136
73.2.7	Finally	1136
74	Reflecting Pure types into HOL	1136
75	Predicates as enumerations	1137
75.1	The type of predicate enumerations (a monad)	1137
75.2	Emptiness check and definite choice	1141
75.3	Derived operations	1142
75.4	Implementation	1144

76 Lazy sequences	1149
76.1 Type of lazy sequences	1149
76.2 Code setup	1152
76.3 Generator Sequences	1153
76.3.1 General lazy sequence operation	1153
76.3.2 Small lazy typeclasses	1153
76.4 With Hit Bound Value	1154
77 Depth-Limited Sequences with failure element	1155
77.1 Depth-Limited Sequence	1155
77.2 Positive Depth-Limited Sequence	1157
77.3 Negative Depth-Limited Sequence	1158
77.4 Negation	1158
78 Term evaluation using the generic code generator	1159
78.1 Term representation	1159
78.1.1 Terms and class <i>term-of</i>	1159
78.1.2 Syntax	1160
78.2 Tools setup and evaluation	1160
78.3 Dedicated <i>term-of</i> instances	1161
78.4 Generic reification	1162
78.5 Diagnostic	1162
79 A simple counterexample generator performing random testing	1162
79.1 Catching Match exceptions	1162
79.2 The <i>random</i> class	1163
79.3 Fundamental and numeric types	1163
79.4 Complex generators	1166
79.5 Deriving random generators for datatypes	1168
79.6 Code setup	1168
80 The Random-Predicate Monad	1168
81 Various kind of sequences inside the random monad	1170
82 A simple counterexample generator performing exhaustive testing	1173
82.1 Basic operations for exhaustive generators	1173
82.2 Exhaustive generator type classes	1173
82.2.1 A smarter enumeration scheme for functions over finite datatypes	1179
82.3 Bounded universal quantifiers	1187
82.4 Fast exhaustive combinators	1187
82.5 Continuation passing style functions as plus monad	1187

82.6	Defining generators for any first-order data type	1189
82.7	Defining generators for abstract types	1189
83	A compiler for predicates defined by introduction rules	1190
83.1	Set membership as a generator predicate	1190
84	Counterexample generator performing narrowing-based testing	1191
84.1	Counterexample generator	1191
84.1.1	Code generation setup	1191
84.1.2	Narrowing's deep representation of types and terms	1192
84.1.3	From narrowing's deep representation of terms to <i>HOL.Code-Evaluation's</i> terms	1192
84.1.4	Auxiliary functions for Narrowing	1192
84.1.5	Narrowing's basic operations	1193
84.1.6	Narrowing generator type class	1194
84.1.7	class <i>is-testable</i>	1194
84.1.8	Defining a simple datatype to represent functions in an incomplete and redundant way	1194
84.1.9	Setting up the counterexample generator	1195
84.2	Narrowing for sets	1195
84.3	Narrowing for integers	1196
84.4	The <i>find-unused-assms</i> command	1197
84.5	Closing up	1197
85	Program extraction for HOL	1198
85.1	Setup	1198
85.2	Type of extracted program	1198
85.3	Realizability	1199
85.4	Computational content of basic inference rules	1201
86	Extensible records with structural subtyping	1206
86.1	Introduction	1206
86.2	Operators and lemmas for types isomorphic to tuples	1207
86.3	Logical infrastructure for records	1208
86.4	Concrete record syntax	1213
86.5	Record package	1214
87	Greatest common divisor and least common multiple	1214
87.1	Abstract bounded quasi semilattices as common foundation	1214
87.2	Abstract GCD and LCM	1216
87.3	An aside: GCD and LCM on finite sets for incomplete gcd rings	1226
87.4	Coprimality	1229

87.5	GCD and LCM for multiplicative normalisation functions . .	1232
87.6	GCD and LCM on <i>nat</i> and <i>int</i>	1233
87.7	Bezout's theorem	1240
87.8	LCM properties on <i>nat</i> and <i>int</i>	1241
87.9	The complete divisibility lattice on <i>nat</i> and <i>int</i>	1243
87.9.1	Setwise GCD and LCM for integers	1245
87.10	GCD and LCM on <i>integer</i>	1246
87.11	Characteristic of a semiring	1248
88	Nitpick: Yet Another Counterexample Generator for Isabelle/HOL	1250
89	Greatest Fixpoint (Codatatype) Operation on Bounded Natural Functors	1254
89.1	Equivalence relations, quotients, and Hilbert's choice	1259
90	Filters on predicates	1260
90.1	Filters	1260
90.1.1	Eventually	1260
90.2	Frequently as dual to eventually	1262
90.2.1	Finer-than relation	1265
90.2.2	Map function for filters	1268
90.2.3	Contravariant map function for filters	1269
90.2.4	Standard filters	1270
90.2.5	Order filters	1271
90.3	Sequentially	1273
90.4	Increasing finite subsets	1274
90.5	The cofinite filter	1275
90.5.1	Product of filters	1275
90.6	Limits	1278
90.7	Limits to <i>at-top</i> and <i>at-bot</i>	1281
90.8	Setup <i>'a filter</i> for lifting and transfer	1282
91	Conditionally-complete Lattices	1286
91.1	Preorders	1287
91.2	Lattices	1290
91.3	Conditionally complete lattices	1291
91.4	Complete lattices	1296
91.5	Instances	1297
92	Factorial Function, Rising Factorials	1301
92.1	Factorial Function	1302
92.2	Pochhammer's symbol: generalized rising factorial	1304
92.3	Misc	1307

93 Binomial Coefficients, Binomial Theorem, Inclusion-exclusion Principle	1308
93.1 Binomial coefficients	1308
93.2 The binomial theorem (courtesy of Tobias Nipkow):	1309
93.3 Generalized binomial coefficients	1311
93.4 Summation on the upper index	1315
93.5 More on Binomial Coefficients	1317
93.6 Inclusion-exclusion principle	1318
93.7 Versions for unrestrictedly additive functions	1319
93.8 General "Moebius inversion" inclusion-exclusion principle	1320
93.9 Executable code	1321
94 Main HOL	1321
94.1 Namespace cleanup	1321
94.2 Syntax cleanup	1322
94.3 Lattice syntax	1322
95 Archimedean Fields, Floor and Ceiling Functions	1323
95.1 Class of Archimedean fields	1323
95.2 Existence and uniqueness of floor function	1324
95.3 Floor function	1325
95.4 Ceiling function	1328
95.4.1 Ceiling with numerals.	1329
95.4.2 Addition and subtraction of integers.	1331
95.5 Negation	1332
95.6 Natural numbers	1332
95.7 Frac Function	1332
95.8 Fractional part arithmetic	1333
95.9 Rounding to the nearest integer	1334
96 Rational numbers	1336
96.1 Rational numbers as quotient	1336
96.1.1 Construction of the type of rational numbers	1336
96.1.2 Representation and basic operations	1336
96.1.3 Function <i>normalize</i>	1339
96.1.4 Various	1341
96.1.5 The ordered field of rational numbers	1341
96.1.6 Rationals are an Archimedean field	1343
96.2 Linear arithmetic setup	1343
96.3 Embedding from Rationals to other Fields	1344
96.4 The Set of Rational Numbers	1346
96.5 Implementation of rational numbers as pairs of integers	1348
96.6 Setup for Nitpick	1353
96.7 Float syntax	1353

96.8 Hiding implementation details	1353
97 Development of the Reals using Cauchy Sequences	1353
97.1 Preliminary lemmas	1354
97.2 Sequences that converge to zero	1354
97.3 Cauchy sequences	1355
97.4 Equivalence relation on Cauchy sequences	1356
97.5 The field of real numbers	1357
97.6 Positive reals	1358
97.7 Completeness	1360
97.8 Supremum of a set of reals	1361
97.9 Hiding implementation details	1361
97.10 Embedding numbers into the Reals	1361
97.11 Embedding the Naturals into the Reals	1362
97.12 The Archimedean Property of the Reals	1363
97.13 Rationals	1364
97.14 Density of the Rational Reals in the Reals	1364
97.15 Numerals and Arithmetic	1364
97.16 Simprules combining $x + y$ and 0	1364
97.17 Lemmas about powers	1365
97.18 Density of the Reals	1365
97.19 Archimedean properties and useful consequences	1366
97.20 Floor and Ceiling Functions from the Reals to the Integers	1367
97.21 Exponentiation with floor	1369
97.22 Implementation of rational real numbers	1370
97.23 Setup for Nitpick	1373
97.24 Setup for SMT	1373
97.25 Setup for Argo	1374
98 Topological Spaces	1374
98.1 Topological space	1374
98.2 Hausdorff and other separation properties	1377
98.3 Generators for topologies	1378
98.4 Order topologies	1379
98.5 Setup some topologies	1380
98.5.1 Boolean is an order topology	1380
98.5.2 Topological filters	1381
98.5.3 Tendsto	1386
98.5.4 Rules about <i>Lim</i>	1390
98.6 Limits on sequences	1391
98.7 Monotone sequences and subsequences	1392
98.7.1 Definition of subsequence.	1392
98.7.2 Subsequence (alternative definition, (e.g. Hoskins))	1394
98.7.3 Increasing and Decreasing Series	1397

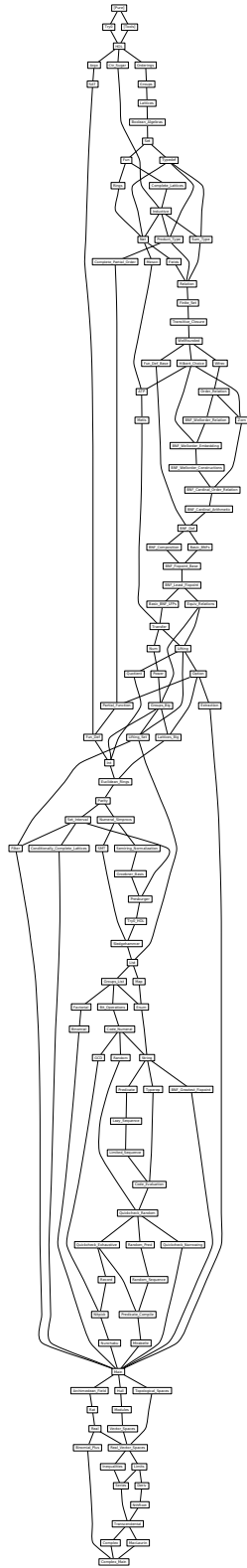
98.8	First countable topologies	1398
98.9	Function limit at a point	1399
98.9.1	Relation of <i>LIM</i> and <i>LIMSEQ</i>	1401
98.10	Continuity	1402
98.10.1	Continuity on a set	1402
98.10.2	Continuity at a point	1405
98.10.3	Open-cover compactness	1409
98.11	Finite intersection property	1410
98.12	Connectedness	1411
99	Linear Continuum Topologies	1414
99.1	Intermediate Value Theorem	1415
99.2	Uniform spaces	1417
99.2.1	Totally bounded sets	1418
99.2.2	Cauchy filter	1418
99.2.3	Uniformly continuous functions	1419
100	Product Topology	1420
100.1	Product is a topological space	1420
100.1.1	Continuity of operations	1421
100.1.2	Connectedness of products	1423
100.1.3	Separation axioms	1423
100.2A	A generic notion of the convex, affine, conic hull, or closed "hull".	1424
101	Modules	1425
101.1	Locale for additive functions	1425
102	Subspace	1427
103	Span: subspace generated by a set	1428
104	Dependent and independent sets	1431
105	Representation of a vector on a specific basis	1432
106	Vector Spaces	1437
107	Vector Spaces and Algebras over the Reals	1451
107.1	Real vector spaces	1451
107.2	Embedding of the Reals into any <i>real-algebra-1: of-real</i> . . .	1455
107.3	The Set of Real Numbers	1458
107.4	Ordered real vector spaces	1460
107.5	Real normed vector spaces	1463
107.6	Metric spaces	1470

107.7	Class instances for real numbers	1471
107.8	Extra type constraints	1472
107.9	Sign function	1473
107.10	Bounded Linear and Bilinear Operators	1474
107.11	Filters and Limits on Metric Space	1478
107.11.1	Limits of Sequences	1480
107.11.2	Limits of Functions	1480
107.12	Complete metric spaces	1481
107.13	Cauchy sequences	1481
107.13.1	Cauchy Sequences are Convergent	1483
107.14	The set of real numbers is a complete metric space	1483
108	Limits on Real Vector Spaces	1484
108.1	Filter going to infinity norm	1484
108.1.1	Boundedness	1485
108.1.2	Bounded Sequences	1486
108.1.3	A Few More Equivalence Theorems for Boundedness	1487
108.1.4	Upper Bounds and Lubs of Bounded Sequences	1487
108.1.5	Polynomial function extremal theorem, from HOL Light	1488
108.2	Convergence to Zero	1488
108.2.1	Distance and norms	1490
108.3	Topological Monoid	1491
108.3.1	Topological group	1492
108.3.2	Linear operators and multiplication	1494
108.3.3	Inverse and division	1500
108.4	Relate <i>at</i> , <i>at-left</i> and <i>at-right</i>	1505
108.5	Floor and Ceiling	1513
108.6	Limits of Sequences	1514
108.7	Convergence on sequences	1517
108.8	More about <i>filterlim</i> (thanks to Wenda Li)	1520
108.9	Power Sequences	1522
108.10	Limits of Functions	1523
108.11	Continuity	1524
108.12	Uniform Continuity	1526
108.13	Nested Intervals and Bisection – Needed for Compactness	1526
108.14	Boundedness of continuous functions	1527
109	Infinite Series	1529
109.1	Definition of infinite summability	1529
109.2	Infinite summability on topological monoids	1530
109.3	Infinite summability on ordered, topological monoids	1532
109.4	Infinite summability on topological monoids	1534
109.5	Infinite summability on real normed vector spaces	1535
109.6	Infinite summability on real normed algebras	1537

109.7	Infinite summability on real normed fields	1538
109.8	Telescoping	1539
109.9	Infinite summability on Banach spaces	1540
109.10	The Ratio Test	1541
109.11	Cauchy Product Formula	1542
109.12	Series on <i>reals</i>	1542
110	Differentiation	1544
110.1	Frechet derivative	1545
110.1.1	Limit transformation for derivatives	1548
110.2	Continuity	1549
110.3	Composition	1549
110.4	Uniqueness	1551
110.5	Differentiability predicate	1552
110.6	Vector derivative	1555
110.7	Derivatives	1557
110.8	Local extrema	1563
110.9	Rolle's Theorem	1565
110.10	Mean Value Theorem	1566
110.10.1	A function is constant if its derivative is 0 over an interval.	1566
110.10.2	A function with positive derivative is increasing	1567
110.11	Generalized Mean Value Theorem	1570
110.12	Hopitals rule	1570
111	Nth Roots of Real Numbers	1574
111.1	Existence of Nth Root	1574
111.2	Nth Root	1574
111.3	Square Root	1578
111.4	Square Root of Sum of Squares	1582
112	Power Series, Transcendental Functions etc.	1584
112.1	Properties of Power Series	1585
112.2	Alternating series test / Leibniz formula	1586
112.3	Term-by-Term Differentiability of Power Series	1587
112.4	The Derivative of a Power Series Has the Same Radius of Convergence	1588
112.5	Derivability of power series	1590
112.6	Exponential Function	1591
112.6.1	Properties of the Exponential Function	1592
112.6.2	Properties of the Exponential Function on Reals	1593
112.7	Natural Logarithm	1595
112.7.1	A couple of simple bounds	1604
112.8	The general logarithm	1604

112.9	Sine and Cosine	1619
112.10	Properties of Sine and Cosine	1622
112.11	Deriving the Addition Formulas	1622
112.12	The Constant Pi	1625
112.13	More Corollaries about Sine and Cosine	1633
112.14	Tangent	1635
112.15	Cotangent	1639
112.16	Inverse Trigonometric Functions	1641
112.17	Prove Totality of the Trigonometric Functions	1647
112.18	Machin's formula	1650
112.19	Introducing the inverse tangent power series	1651
112.20	Existence of Polar Coordinates	1652
112.21	Basics about polynomial functions: products, extremal behaviour and root counts	1652
112.22	Hyperbolic functions	1654
112.22.1	More specific properties of the real functions	1658
112.22.2	Limits	1661
112.22.3	Properties of the inverse hyperbolic functions	1661
112.23	Improvements for root and power literals	1667
113	Complex Numbers: Rectangular and Polar Representation	1668
113.1	Addition and Subtraction	1668
113.2	Multiplication and Division	1669
113.3	Scalar Multiplication	1670
113.4	Numerals, Arithmetic, and Embedding from \mathbb{R}	1670
113.5	The Complex Number i	1672
113.6	Vector Norm	1673
113.7	Absolute value	1675
113.8	Completeness of the Complexes	1676
113.9	Complex Conjugation	1677
113.10	Basic Lemmas	1680
113.11	Polar Form for Complex Numbers	1683
113.11.1	$\cos \theta + i \sin \theta$	1683
113.11.2	$(\cos \theta + i \sin \theta)$	1684
113.11.3	Complex exponential	1685
113.11.4	Complex argument	1687
113.12	Complex n -th roots	1688
113.13	Square root of complex numbers	1688
114	MacLaurin and Taylor Series	1691
114.1	MacLaurin's Theorem with Lagrange Form of Remainder	1691
114.2	More Convenient "Bidirectional" Version.	1693
114.3	Version for Exponential Function	1694
114.4	Version for Sine Function	1694

114.5Maclaurin Expansion for Cosine Function	1695
115Taylor series	1696
116More facts about binomial coefficients	1696
116.1More facts about binomial coefficients	1697
116.2Results about binomials and integers, thanks to Alexander Maletzky	1698
116.3Sums	1699
117Comprehensive Complex Theory	1700



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

   $\langle ML \rangle$ 

  code-datatype TYPE('a::{})

  definition holds :: prop where
    holds  $\equiv ((\lambda x::prop. x) \equiv (\lambda x. x))$ 

  lemma holds: PROP holds
     $\langle proof \rangle$ 

  code-datatype holds

  lemma implies-code [code]:
    (PROP P  $\implies$  PROP holds)  $\equiv$  PROP holds
    (PROP holds  $\implies$  PROP P)  $\equiv$  PROP P
     $\langle proof \rangle$ 

   $\langle ML \rangle$ 

  hide-const (open) holds

end

theory Try0
  imports Pure
  keywords try0 :: diag
begin

   $\langle ML \rangle$ 

end

```

2 The basis of Higher-Order Logic

```

theory HOL
imports Pure Try0 Tools.Code-Generator
keywords
  try solve-direct quickcheck print-coercions print-claset
  print-induct-rules :: diag and
  quickcheck-params :: thy-decl
abbrevs ?< =  $\exists_{\leq 1}$ 
begin

```

⟨ML⟩

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 $(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)$ 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

```

⟨ML⟩
default-sort type
⟨ML⟩

```

```

axiomatization where fun-arity: OFCLASS('a  $\Rightarrow$  'b, type-class)
instance fun :: (type, type) type ⟨proof⟩

```

```

axiomatization where itself-arity: OFCLASS('a itself, type-class)
instance itself :: (type) type ⟨proof⟩

```

```

typedecl bool

```

```

judgment Trueprop :: bool  $\Rightarrow$  prop (⟨⟨notation=judgment⟩-⟩ 5)

```

```

axiomatization implies :: [bool, bool]  $\Rightarrow$  bool (infixr <math>\longrightarrow> 25)
  and eq :: ['a, 'a]  $\Rightarrow$  bool
  and The :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a

```

```

notation (input)
  eq (infixl <math>\Leftarrow> 50)
notation (output)
  eq (infix <math>\Rightarrow> 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 *bool*) \Rightarrow *bool* (**binder** $\langle \forall \rangle$ 10)

where *All* *P* $\equiv (P = (\lambda x. \text{True}))$

definition *Ex* :: (*a* \Rightarrow *bool*) \Rightarrow *bool* (**binder** $\langle \exists \rangle$ 10)

where *Ex* *P* $\equiv \forall Q. (\forall x. P\ x \longrightarrow Q) \longrightarrow Q$

definition *False* :: *bool*

where *False* $\equiv (\forall P. P)$

definition *Not* :: *bool* \Rightarrow *bool* ($\langle (\langle \text{open-block notation} = \langle \text{prefix } \neg \rangle \neg) \rangle$ [40] 40)

where *not-def*: $\neg P \equiv P \longrightarrow \text{False}$

definition *conj* :: [*bool*, *bool*] \Rightarrow *bool* (**infixr** $\langle \wedge \rangle$ 35)

where *and-def*: $P \wedge Q \equiv \forall R. (P \longrightarrow Q \longrightarrow R) \longrightarrow R$

definition *disj* :: [*bool*, *bool*] \Rightarrow *bool* (**infixr** $\langle \vee \rangle$ 30)

where *or-def*: $P \vee Q \equiv \forall R. (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$

definition *Uniq* :: (*a* \Rightarrow *bool*) \Rightarrow *bool*

where *Uniq* *P* $\equiv (\forall x\ y. P\ x \longrightarrow P\ y \longrightarrow y = x)$

definition *Ex1* :: (*a* \Rightarrow *bool*) \Rightarrow *bool*

where *Ex1* *P* $\equiv \exists x. P\ x \wedge (\forall y. P\ y \longrightarrow y = x)$

2.1.3 Additional concrete syntax

syntax (*ASCII*) *-Uniq* :: *pttrn* \Rightarrow *bool* \Rightarrow *bool* ($\langle (\langle \text{indent}=4 \text{ notation} = \langle \text{binder } ?<\rangle ?< \text{ -./ -} \rangle) \rangle$ [0, 10] 10)

syntax *-Uniq* :: *pttrn* \Rightarrow *bool* \Rightarrow *bool* ($\langle (\langle \text{indent}=2 \text{ notation} = \langle \text{binder } \exists_{\leq 1} \rangle \exists_{\leq 1} \text{ -./ -} \rangle) \rangle$ [0, 10] 10)

syntax-consts *-Uniq* \equiv *Uniq*

translations $\exists_{\leq 1} x. P \equiv \text{CONST } \text{Uniq } (\lambda x. P)$

$\langle ML \rangle$

syntax (*ASCII*)

-Ex1 :: *pttrn* \Rightarrow *bool* \Rightarrow *bool* ($\langle (\langle \text{indent}=3 \text{ notation} = \langle \text{binder } EX! \rangle EX! \text{ -./ -} \rangle) \rangle$ [0, 10] 10)

syntax (*input*)

-*Ex1* :: *pttrn* \Rightarrow *bool* \Rightarrow *bool* ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } ?! \rangle \rangle ?! \text{ -./ -} \rangle [0, 10] 10$)

syntax -*Ex1* :: *pttrn* \Rightarrow *bool* \Rightarrow *bool* ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \exists ! \rangle \rangle \exists ! \text{ -./ -} \rangle [0, 10] 10$)

syntax-consts -*Ex1* \equiv *Ex1*

translations $\exists !x. P \equiv \text{CONST } Ex1 (\lambda x. P)$

$\langle ML \rangle$

syntax

-*Not-Ex* :: *idts* \Rightarrow *bool* \Rightarrow *bool* ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \# \rangle \rangle \# \text{ -./ -} \rangle [0, 10] 10$)

-*Not-Ex1* :: *pttrn* \Rightarrow *bool* \Rightarrow *bool* ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \# ! \rangle \rangle \# ! \text{ -./ -} \rangle [0, 10] 10$)

syntax-consts

-*Not-Ex* \equiv *Ex* **and**

-*Not-Ex1* \equiv *Ex1*

translations

$\# x. P \equiv \neg (\exists x. P)$

$\# !x. P \equiv \neg (\exists !x. P)$

abbreviation *not-equal* :: [*a*, *a*] \Rightarrow *bool* (**infix** $\langle \neq \rangle$ 50)

where $x \neq y \equiv \neg (x = y)$

notation (*ASCII*)

Not ($\langle \langle \text{open-block notation}=\langle \text{prefix } \sim \rangle \rangle \sim \text{ -} \rangle [40] 40$) **and**

conj (**infix** $\langle \& \rangle$ 35) **and**

disj (**infix** $\langle | \rangle$ 30) **and**

implies (**infix** $\langle \longrightarrow \rangle$ 25) **and**

not-equal (**infix** $\langle \sim = \rangle$ 50)

abbreviation (*iff*)

iff :: [*bool*, *bool*] \Rightarrow *bool* (**infix** $\langle \longleftrightarrow \rangle$ 25)

where $A \longleftrightarrow B \equiv A = B$

syntax -*The* :: [*pttrn*, *bool*] \Rightarrow *a* ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } THE \rangle \rangle THE \text{ -./ -} \rangle [0, 10] 10$)

syntax-consts -*The* \equiv *The*

translations *THE* *x. P* \equiv *CONST The* ($\lambda x. P$)

$\langle ML \rangle$

nonterminal *case-syn* **and** *cases-syn*

syntax

-*case-syntax* :: [*a*, *cases-syn*] \Rightarrow *b* ($\langle \langle \text{notation}=\langle \text{mixfix case expression} \rangle \rangle \text{case -}$

$of / - \rangle 10)$
 $-case1 :: ['a, 'b] \Rightarrow case-syn$
 $(\langle \langle indent=2 notation=\langle mixfix case clause \rangle \rangle (\langle open-block notation=\langle pattern case \rangle \rangle -) \rangle$
 $\Rightarrow / - \rangle 10)$
 $:: case-syn \Rightarrow cases-syn (\langle - \rangle)$
 $-case2 :: [case-syn, cases-syn] \Rightarrow cases-syn (\langle - / \mid - \rangle)$
syntax (ASCII)
 $-case1 :: ['a, 'b] \Rightarrow case-syn$
 $(\langle \langle indent=2 notation=\langle mixfix case clause \rangle \rangle (\langle open-block notation=\langle pattern case \rangle \rangle -) \rangle$
 $\Rightarrow / - \rangle 10)$

notation (ASCII)
 $All \text{ (binder } \langle ALL \rangle 10) \text{ and}$
 $Ex \text{ (binder } \langle EX \rangle 10)$

notation (input)
 $All \text{ (binder } \langle ! \rangle 10) \text{ and}$
 $Ex \text{ (binder } \langle ? \rangle 10)$

2.1.4 Axioms and basic definitions

axiomatization where

$refl: t = (t :: 'a) \text{ and}$
 $subst: s = t \Longrightarrow P s \Longrightarrow P t \text{ and}$
 $ext: (\bigwedge x :: 'a. (f x :: 'b) = g x) \Longrightarrow (\lambda x. f x) = (\lambda x. g x)$
 — 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

$impI: (P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q \text{ and}$
 $mp: \llbracket P \longrightarrow Q; P \rrbracket \Longrightarrow Q \text{ and}$

$True-or-False: (P = True) \vee (P = False)$

definition $If :: bool \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a (\langle \langle notation=\langle mixfix if expression \rangle \rangle if (-) /$
 $then (-) / else (-) \rangle [0, 0, 10] 10)$
where $If P x y \equiv (THE z :: 'a. (P = True \longrightarrow z = x) \wedge (P = False \longrightarrow z = y))$

definition $Let :: 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b$
where $Let s f \equiv f s$

nonterminal $letbinds$ **and** $letbind$

open-bundle $let-syntax$
begin

syntax

$-bind :: [pttrn, 'a] \Rightarrow letbind$ $(\langle (\langle indent=2 \text{ notation}=\langle mixfix \text{ let binding} \rangle - = / - \rangle \rangle 10)$
 $:: letbind \Rightarrow letbinds$ $(\langle - \rangle)$
 $-binds :: [letbind, letbinds] \Rightarrow letbinds$ $(\langle -; / - \rangle)$
 $-Let :: [letbinds, 'a] \Rightarrow 'a$ $(\langle (\langle notation=\langle mixfix \text{ let expression} \rangle \rangle let (-) / in (-) \rangle [0, 10] 10)$

syntax-consts

$-bind -binds -Let \Rightarrow Let$

translations

$-Let (-binds b bs) e \Rightarrow -Let b (-Let bs e)$
 $let x = a in e \Rightarrow CONST Let a (\lambda x. e)$

end

axiomatization $undefined :: 'a$

class $default = fixes default :: 'a$

2.2 Fundamental rules**2.2.1 Equality**

lemma $sym: s = t \Longrightarrow t = s$
 $\langle proof \rangle$

lemma $ssubst: t = s \Longrightarrow P s \Longrightarrow P t$
 $\langle proof \rangle$

lemma $trans: \llbracket r = s; s = t \rrbracket \Longrightarrow r = t$
 $\langle proof \rangle$

lemma $trans\text{-}sym [Pure.elim?]: r = s \Longrightarrow t = s \Longrightarrow r = t$
 $\langle proof \rangle$

lemma $meta\text{-}eq\text{-}to\text{-}obj\text{-}eq:$

assumes $A \equiv B$

shows $A = B$

$\langle proof \rangle$

Useful with *erule* for proving equalities from known equalities.

lemma $box\text{-}equals: \llbracket a = b; a = c; b = d \rrbracket \Longrightarrow c = d$
 $\langle proof \rangle$

For calculational reasoning:

lemma $forw\text{-}subst: a = b \Longrightarrow P b \Longrightarrow P a$
 $\langle proof \rangle$

lemma $back\text{-}subst: P a \Longrightarrow a = b \Longrightarrow P b$
 $\langle proof \rangle$

2.2.2 Congruence rules for application

Similar to *AP-THM* in Gordon’s HOL.

lemma *fun-cong*: $(f :: 'a \Rightarrow 'b) = g \implies f\ x = g\ x$
 $\langle proof \rangle$

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

lemma *arg-cong*: $x = y \implies f\ x = f\ y$
 $\langle proof \rangle$

lemma *arg-cong2*: $\llbracket a = b; c = d \rrbracket \implies f\ a\ c = f\ b\ d$
 $\langle proof \rangle$

lemma *arg-cong3*: $\llbracket x = x'; y = y'; z = z' \rrbracket \implies f\ x\ y\ z = f\ x'\ y'\ z'$
 $\langle proof \rangle$

lemma *arg-cong4*: $\llbracket w = w'; x = x'; y = y'; z = z' \rrbracket \implies f\ w\ x\ y\ z = f\ w'\ x'\ y'\ z'$
 $\langle proof \rangle$

lemma *cong*: $\llbracket f = g; (x :: 'a) = y \rrbracket \implies f\ x = g\ y$
 $\langle proof \rangle$

$\langle ML \rangle$

2.2.3 Equality of booleans – iff

lemma *iffD2*: $\llbracket P = Q; Q \rrbracket \implies P$
 $\langle proof \rangle$

lemma *rev-iffD2*: $\llbracket Q; P = Q \rrbracket \implies P$
 $\langle proof \rangle$

lemma *iffD1*: $Q = P \implies Q \implies P$
 $\langle proof \rangle$

lemma *rev-iffD1*: $Q \implies Q = P \implies P$
 $\langle proof \rangle$

lemma *iffE*:
 assumes *major*: $P = Q$
 and *minor*: $\llbracket P \longrightarrow Q; Q \longrightarrow P \rrbracket \implies R$
 shows R
 $\langle proof \rangle$

2.2.4 True (1)

lemma *TrueI*: *True*
 $\langle proof \rangle$

lemma *eqTrueE*: $P = \text{True} \implies P$
 $\langle \text{proof} \rangle$

2.2.5 Universal quantifier (1)

lemma *spec*: $\forall x::'a. P\ x \implies P\ x$
 $\langle \text{proof} \rangle$

lemma *allE*:
assumes *major*: $\forall x. P\ x$ **and** *minor*: $P\ x \implies R$
shows R
 $\langle \text{proof} \rangle$

lemma *all-dupE*:
assumes *major*: $\forall x. P\ x$ **and** *minor*: $\llbracket P\ x; \forall x. P\ x \rrbracket \implies R$
shows R
 $\langle \text{proof} \rangle$

2.2.6 False

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

lemma *FalseE*: $\text{False} \implies P$
 $\langle \text{proof} \rangle$

lemma *False-neq-True*: $\text{False} = \text{True} \implies P$
 $\langle \text{proof} \rangle$

2.2.7 Negation

lemma *notI*:
assumes $P \implies \text{False}$
shows $\neg P$
 $\langle \text{proof} \rangle$

lemma *False-not-True*: $\text{False} \neq \text{True}$
 $\langle \text{proof} \rangle$

lemma *True-not-False*: $\text{True} \neq \text{False}$
 $\langle \text{proof} \rangle$

lemma *notE*: $\llbracket \neg P; P \rrbracket \implies R$
 $\langle \text{proof} \rangle$

2.2.8 Implication

lemma *impE*:
assumes $P \longrightarrow Q$ $P\ Q \implies R$
shows R

$\langle proof \rangle$

Reduces Q to $P \longrightarrow Q$, allowing substitution in P .

lemma *rev-mp*: $\llbracket P; P \longrightarrow Q \rrbracket \Longrightarrow Q$
 $\langle proof \rangle$

lemma *contrapos-nn*:
assumes *major*: $\neg Q$
and *minor*: $P \Longrightarrow Q$
shows $\neg P$
 $\langle proof \rangle$

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

lemma *contrapos-pn*:
assumes *major*: Q
and *minor*: $P \Longrightarrow \neg Q$
shows $\neg P$
 $\langle proof \rangle$

lemma *not-sym*: $t \neq s \Longrightarrow s \neq t$
 $\langle proof \rangle$

lemma *eq-neq-eq-imp-neq*: $\llbracket x = a; a \neq b; b = y \rrbracket \Longrightarrow x \neq y$
 $\langle proof \rangle$

2.2.9 Disjunction (1)

lemma *disjE*:
assumes *major*: $P \vee Q$
and *minorP*: $P \Longrightarrow R$
and *minorQ*: $Q \Longrightarrow R$
shows R
 $\langle proof \rangle$

2.2.10 Derivation of *iffI*

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

lemma *iffI*:
assumes $P \Longrightarrow Q$ **and** $Q \Longrightarrow P$
shows $P = Q$
 $\langle proof \rangle$

2.2.11 True (2)

lemma *eqTrueI*: $P \Longrightarrow P = True$
 $\langle proof \rangle$

2.2.12 Universal quantifier (2)

lemma *allI*:
 assumes $\bigwedge x::'a. P\ x$
 shows $\forall x. P\ x$
 $\langle proof \rangle$

2.2.13 Existential quantifier

lemma *exI*: $P\ x \implies \exists x::'a. P\ x$
 $\langle proof \rangle$

lemma *exE*:
 assumes *major*: $\exists x::'a. P\ x$
 and *minor*: $\bigwedge x. P\ x \implies Q$
 shows Q
 $\langle proof \rangle$

2.2.14 Conjunction

lemma *conjI*: $\llbracket P; Q \rrbracket \implies P \wedge Q$
 $\langle proof \rangle$

lemma *conjunct1*: $\llbracket P \wedge Q \rrbracket \implies P$
 $\langle proof \rangle$

lemma *conjunct2*: $\llbracket P \wedge Q \rrbracket \implies Q$
 $\langle proof \rangle$

lemma *conjE*:
 assumes *major*: $P \wedge Q$
 and *minor*: $\llbracket P; Q \rrbracket \implies R$
 shows R
 $\langle proof \rangle$

lemma *context-conjI*:
 assumes $P \implies Q$
 shows $P \wedge Q$
 $\langle proof \rangle$

2.2.15 Disjunction (2)

lemma *disjI1*: $P \implies P \vee Q$
 $\langle proof \rangle$

lemma *disjI2*: $Q \implies P \vee Q$
 $\langle proof \rangle$

2.2.16 Classical logic

lemma *classical*:

assumes $\neg P \implies P$
shows P
 $\langle proof \rangle$

lemmas *ccontr* = *FalseE* [*THEN classical*]

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

lemma *rev-notE*:
assumes *premp*: P
and *premnnot*: $\neg R \implies \neg P$
shows R
 $\langle proof \rangle$

Double negation law.

lemma *notnotD*: $\neg\neg P \implies P$
 $\langle proof \rangle$

lemma *contrapos-pp*:
assumes *p1*: Q
and *p2*: $\neg P \implies \neg Q$
shows P
 $\langle proof \rangle$

2.2.17 Unique existence

lemma *Uniq-I* [*intro?*]:
assumes $\bigwedge x y. \llbracket P\ x; P\ y \rrbracket \implies y = x$
shows *Uniq* P
 $\langle proof \rangle$

lemma *Uniq-D* [*dest?*]: $\llbracket \text{Uniq } P; P\ a; P\ b \rrbracket \implies a=b$
 $\langle proof \rangle$

lemma *ex1I*:
assumes $P\ a \bigwedge x. P\ x \implies x = a$
shows $\exists!x. P\ x$
 $\langle proof \rangle$

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

lemma *ex-ex1I*:
assumes *ex-prem*: $\exists x. P\ x$
and *eq*: $\bigwedge x y. \llbracket P\ x; P\ y \rrbracket \implies x = y$
shows $\exists!x. P\ x$
 $\langle proof \rangle$

lemma *ex1E*:
assumes *major*: $\exists!x. P\ x$ **and** *minor*: $\bigwedge x. \llbracket P\ x; \forall y. P\ y \longrightarrow y = x \rrbracket \implies R$
shows R

$\langle proof \rangle$

lemma *ex1-implies-ex*: $\exists!x. P\ x \implies \exists x. P\ x$
 $\langle proof \rangle$

2.2.18 Classical intro rules for disjunction and existential quantifiers

lemma *disjCI*:
assumes $\neg Q \implies P$
shows $P \vee Q$
 $\langle proof \rangle$

lemma *excluded-middle*: $\neg P \vee P$
 $\langle proof \rangle$

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 $P \implies Q \ \neg P \implies Q$
shows Q
 $\langle proof \rangle$

Classical implies (\longrightarrow) elimination.

lemma *impCE*:
assumes *major*: $P \longrightarrow Q$
and *minor*: $\neg P \implies R \ Q \implies R$
shows R
 $\langle proof \rangle$

This version of \longrightarrow 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 *impCE'*:
assumes *major*: $P \longrightarrow Q$
and *minor*: $Q \implies R \ \neg P \implies R$
shows R
 $\langle proof \rangle$

The analogous introduction rule for conjunction, above, is even constructive

lemma *context-disjE*:
assumes *major*: $P \vee Q$ **and** *minor*: $P \implies R \ \neg P \implies Q \implies R$
shows R
 $\langle proof \rangle$

Classical \longleftrightarrow elimination.

lemma *iffCE*:
assumes *major*: $P = Q$

and *minor*: $\llbracket P; Q \rrbracket \Longrightarrow R \llbracket \neg P; \neg Q \rrbracket \Longrightarrow R$
 shows R
 $\langle proof \rangle$

lemma *exCI*:
 assumes $\forall x. \neg P x \Longrightarrow P a$
 shows $\exists x. P x$
 $\langle proof \rangle$

2.2.19 Intuitionistic Reasoning

lemma *impE'*:
 assumes $1: P \longrightarrow Q$
 and $2: Q \Longrightarrow R$
 and $3: P \longrightarrow Q \Longrightarrow P$
 shows R
 $\langle proof \rangle$

lemma *allE'*:
 assumes $1: \forall x. P x$
 and $2: P x \Longrightarrow \forall x. P x \Longrightarrow Q$
 shows Q
 $\langle proof \rangle$

lemma *notE'*:
 assumes $1: \neg P$
 and $2: \neg P \Longrightarrow P$
 shows R
 $\langle proof \rangle$

lemma *TrueE*: $True \Longrightarrow P \Longrightarrow P$ $\langle proof \rangle$

lemma *notFalseE*: $\neg False \Longrightarrow P \Longrightarrow P$ $\langle proof \rangle$

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

lemmas $[trans] = trans$
 and $[sym] = sym\ not-sym$
 and $[Pure.elim?] = iffD1\ iffD2\ impE$

2.2.20 Atomizing meta-level connectives

axiomatization where

eq-reflection: $x = y \Longrightarrow x \equiv y$ — admissible axiom

lemma *atomize-all* $[atomize]$: $(\bigwedge x. P x) \equiv Trueprop (\forall x. P x)$
 $\langle proof \rangle$

lemma *atomize-imp* [*atomize*]: $(A \Longrightarrow B) \equiv \text{Trueprop } (A \longrightarrow B)$
 $\langle \text{proof} \rangle$

lemma *atomize-not*: $(A \Longrightarrow \text{False}) \equiv \text{Trueprop } (\neg A)$
 $\langle \text{proof} \rangle$

lemma *atomize-eq* [*atomize*, *code*]: $(x \equiv y) \equiv \text{Trueprop } (x = y)$
 $\langle \text{proof} \rangle$

lemma *atomize-conj* [*atomize*]: $(A \&\&\& B) \equiv \text{Trueprop } (A \wedge B)$
 $\langle \text{proof} \rangle$

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*]: $(\bigwedge x. P x \Longrightarrow Q) \equiv ((\exists x. P x) \Longrightarrow Q)$
 $\langle \text{proof} \rangle$

lemma *atomize-conjL*[*atomize-elim*]: $(A \Longrightarrow B \Longrightarrow C) \equiv (A \wedge B \Longrightarrow C)$
 $\langle \text{proof} \rangle$

lemma *atomize-disjL*[*atomize-elim*]: $((A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C) \equiv ((A \vee B \Longrightarrow C) \Longrightarrow C)$
 $\langle \text{proof} \rangle$

lemma *atomize-elimL*[*atomize-elim*]: $(\bigwedge B. (A \Longrightarrow B) \Longrightarrow B) \equiv \text{Trueprop } A$ $\langle \text{proof} \rangle$

2.3 Package setup

$\langle \text{ML} \rangle$

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

lemma *imp-elim*: $P \longrightarrow Q \Longrightarrow (\neg R \Longrightarrow P) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
 $\langle \text{proof} \rangle$

lemma *swap*: $\neg P \Longrightarrow (\neg R \Longrightarrow P) \Longrightarrow R$
 $\langle \text{proof} \rangle$

lemma *thin-refl*: $\llbracket x = x; \text{PROP } W \rrbracket \Longrightarrow \text{PROP } W$ $\langle \text{proof} \rangle$

$\langle ML \rangle$

```

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]

```

$\langle ML \rangle$

```

lemma contrapos-np:  $\neg Q \implies (\neg P \implies Q) \implies P$ 
   $\langle proof \rangle$ 

```

```

declare ex-ex1I [rule del, intro! 2]
  and ex1I [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:  $\bigwedge x. \llbracket P\ x; \forall y\ y'. P\ y \wedge P\ y' \longrightarrow y = y' \rrbracket \implies R$ 
  shows R
   $\langle proof \rangle$ 

```

And again using Uniq

```

lemma alt-ex1E':
  assumes  $\exists! x. P\ x \wedge x. \llbracket P\ x; \exists_{\leq 1} x. P\ x \rrbracket \implies R$ 
  shows R
   $\langle proof \rangle$ 

```

lemma *ex1-iff-ex-Uniq*: $(\exists!x. P\ x) \longleftrightarrow (\exists x. P\ x) \wedge (\exists_{\leq 1}x. P\ x)$
 $\langle proof \rangle$

$\langle ML \rangle$

2.3.3 THE: definite description operator

lemma *the-equality* [intro]:
 assumes $P\ a$
 and $\bigwedge x. P\ x \implies x = a$
 shows $(THE\ x. P\ x) = a$
 $\langle proof \rangle$

lemma *theI*:
 assumes $P\ a$
 and $\bigwedge x. P\ x \implies x = a$
 shows $P\ (THE\ x. P\ x)$
 $\langle proof \rangle$

lemma *theI'*: $\exists!x. P\ x \implies P\ (THE\ x. P\ x)$
 $\langle proof \rangle$

Easier to apply than *theI*: only one occurrence of P .

lemma *theI2*:
 assumes $P\ a \wedge x. P\ x \implies x = a \wedge x. P\ x \implies Q\ x$
 shows $Q\ (THE\ x. P\ x)$
 $\langle proof \rangle$

lemma *theI12*:
 assumes $\exists!x. P\ x \wedge x. P\ x \implies Q\ x$
 shows $Q\ (THE\ x. P\ x)$
 $\langle proof \rangle$

lemma *the1-equality* [elim?]: $\llbracket \exists!x. P\ x; P\ a \rrbracket \implies (THE\ x. P\ x) = a$
 $\langle proof \rangle$

lemma *the1-equality'*: $\llbracket \exists_{\leq 1}x. P\ x; P\ a \rrbracket \implies (THE\ x. P\ x) = a$
 $\langle proof \rangle$

lemma *the-sym-eq-trivial*: $(THE\ y. x = y) = x$
 $\langle proof \rangle$

2.3.4 Simplifier

lemma *eta-contract-eq*: $(\lambda s. f\ s) = f\ \langle proof \rangle$

lemma *subst-all*:
 $\langle (\bigwedge x. x = a \implies PROP\ P\ x) \equiv PROP\ P\ a \rangle$

$\langle (\bigwedge x. a = x \implies PROP P x) \equiv PROP P a \rangle$
 $\langle proof \rangle$

lemma *simp-thms*:

shows *not-not*: $(\neg \neg P) = P$

and *Not-eq-iff*: $((\neg P) = (\neg Q)) = (P = Q)$

and

$(P \neq Q) = (P = (\neg Q))$

$(P \vee \neg P) = True \quad (\neg P \vee P) = True$

$(x = x) = True$

and *not-True-eq-False* [code]: $(\neg True) = False$

and *not-False-eq-True* [code]: $(\neg False) = True$

and

$(\neg P) \neq P \quad P \neq (\neg P)$

$(True = P) = P$

and *eq-True*: $(P = True) = P$

and $(False = P) = (\neg P)$

and *eq-False*: $(P = False) = (\neg P)$

and

$(True \longrightarrow P) = P \quad (False \longrightarrow P) = True$

$(P \longrightarrow True) = True \quad (P \longrightarrow P) = True$

$(P \longrightarrow False) = (\neg P) \quad (P \longrightarrow \neg P) = (\neg P)$

$(P \wedge True) = P \quad (True \wedge P) = P$

$(P \wedge False) = False \quad (False \wedge P) = False$

$(P \wedge P) = P \quad (P \wedge (P \wedge Q)) = (P \wedge Q)$

$(P \wedge \neg P) = False \quad (\neg P \wedge P) = False$

$(P \vee True) = True \quad (True \vee P) = True$

$(P \vee False) = P \quad (False \vee P) = P$

$(P \vee P) = P \quad (P \vee (P \vee Q)) = (P \vee Q)$ **and**

$(\forall x. P) = P \quad (\exists x. P) = P \quad \exists x. x = t \quad \exists x. t = x$

and

$\bigwedge P. (\exists x. x = t \wedge P x) = P t$

$\bigwedge P. (\exists x. t = x \wedge P x) = P t$

$\bigwedge P. (\forall x. x = t \longrightarrow P x) = P t$

$\bigwedge P. (\forall x. t = x \longrightarrow P x) = P t$

$(\forall x. x \neq t) = False \quad (\forall x. t \neq x) = False$

$\langle proof \rangle$

lemma *disj-absorb*: $A \vee A \longleftrightarrow A$

$\langle proof \rangle$

lemma *disj-left-absorb*: $A \vee (A \vee B) \longleftrightarrow A \vee B$

$\langle proof \rangle$

lemma *conj-absorb*: $A \wedge A \longleftrightarrow A$

$\langle proof \rangle$

lemma *conj-left-absorb*: $A \wedge (A \wedge B) \longleftrightarrow A \wedge B$

$\langle proof \rangle$

lemma *eq-ac*:

shows *eq-commute*: $a = b \longleftrightarrow b = a$

and *iff-left-commute*: $(P \longleftrightarrow (Q \longleftrightarrow R)) \longleftrightarrow (Q \longleftrightarrow (P \longleftrightarrow R))$

and *iff-assoc*: $((P \longleftrightarrow Q) \longleftrightarrow R) \longleftrightarrow (P \longleftrightarrow (Q \longleftrightarrow R))$

<proof>

lemma *neq-commute*: $a \neq b \longleftrightarrow b \neq a$ *<proof>*

lemma *conj-comms*:

shows *conj-commute*: $P \wedge Q \longleftrightarrow Q \wedge P$

and *conj-left-commute*: $P \wedge (Q \wedge R) \longleftrightarrow Q \wedge (P \wedge R)$ *<proof>*

lemma *conj-assoc*: $(P \wedge Q) \wedge R \longleftrightarrow P \wedge (Q \wedge R)$ *<proof>*

lemmas *conj-ac* = *conj-commute conj-left-commute conj-assoc*

lemma *disj-comms*:

shows *disj-commute*: $P \vee Q \longleftrightarrow Q \vee P$

and *disj-left-commute*: $P \vee (Q \vee R) \longleftrightarrow Q \vee (P \vee R)$ *<proof>*

lemma *disj-assoc*: $(P \vee Q) \vee R \longleftrightarrow P \vee (Q \vee R)$ *<proof>*

lemmas *disj-ac* = *disj-commute disj-left-commute disj-assoc*

lemma *conj-disj-distribL*: $P \wedge (Q \vee R) \longleftrightarrow P \wedge Q \vee P \wedge R$ *<proof>*

lemma *conj-disj-distribR*: $(P \vee Q) \wedge R \longleftrightarrow P \wedge R \vee Q \wedge R$ *<proof>*

lemma *disj-conj-distribL*: $P \vee (Q \wedge R) \longleftrightarrow (P \vee Q) \wedge (P \vee R)$ *<proof>*

lemma *disj-conj-distribR*: $(P \wedge Q) \vee R \longleftrightarrow (P \vee R) \wedge (Q \vee R)$ *<proof>*

lemma *imp-conjR*: $(P \longrightarrow (Q \wedge R)) = ((P \longrightarrow Q) \wedge (P \longrightarrow R))$ *<proof>*

lemma *imp-conjL*: $((P \wedge Q) \longrightarrow R) = (P \longrightarrow (Q \longrightarrow R))$ *<proof>*

lemma *imp-disjL*: $((P \vee Q) \longrightarrow R) = ((P \longrightarrow R) \wedge (Q \longrightarrow R))$ *<proof>*

These two are specialized, but *imp-disj-not1* is useful in *Auth/Yahalom*.

lemma *imp-disj-not1*: $(P \longrightarrow Q \vee R) \longleftrightarrow (\neg Q \longrightarrow P \longrightarrow R)$ *<proof>*

lemma *imp-disj-not2*: $(P \longrightarrow Q \vee R) \longleftrightarrow (\neg R \longrightarrow P \longrightarrow Q)$ *<proof>*

lemma *imp-disj1*: $((P \longrightarrow Q) \vee R) \longleftrightarrow (P \longrightarrow Q \vee R)$ *<proof>*

lemma *imp-disj2*: $(Q \vee (P \longrightarrow R)) \longleftrightarrow (P \longrightarrow Q \vee R)$ *<proof>*

lemma *imp-cong*: $(P = P') \implies (P' \implies (Q = Q')) \implies ((P \longrightarrow Q) \longleftrightarrow (P' \longrightarrow Q'))$

<proof>

lemma *de-Morgan-disj*: $\neg (P \vee Q) \longleftrightarrow \neg P \wedge \neg Q$ *<proof>*

lemma *de-Morgan-conj*: $\neg (P \wedge Q) \longleftrightarrow \neg P \vee \neg Q$ *<proof>*

lemma *not-imp*: $\neg (P \longrightarrow Q) \longleftrightarrow P \wedge \neg Q$ *<proof>*

lemma *not-iff*: $P \neq Q \longleftrightarrow (P \longleftrightarrow \neg Q)$ *<proof>*

lemma *disj-not1*: $\neg P \vee Q \longleftrightarrow (P \longrightarrow Q)$ *<proof>*

lemma *disj-not2*: $P \vee \neg Q \longleftrightarrow (Q \longrightarrow P) \langle proof \rangle$

lemma *imp-conv-disj*: $(P \longrightarrow Q) \longleftrightarrow (\neg P) \vee Q \langle proof \rangle$

lemma *disj-imp*: $P \vee Q \longleftrightarrow \neg P \longrightarrow Q \langle proof \rangle$

lemma *iff-conv-conj-imp*: $(P \longleftrightarrow Q) \longleftrightarrow (P \longrightarrow Q) \wedge (Q \longrightarrow P) \langle proof \rangle$

lemma *cases-simp*: $(P \longrightarrow Q) \wedge (\neg P \longrightarrow Q) \longleftrightarrow Q$

— Avoids duplication of subgoals after *if-split*, when the true and false

— cases boil down to the same thing.

$\langle proof \rangle$

lemma *not-all*: $\neg (\forall x. P x) \longleftrightarrow (\exists x. \neg P x) \langle proof \rangle$

lemma *imp-all*: $((\forall x. P x) \longrightarrow Q) \longleftrightarrow (\exists x. P x \longrightarrow Q) \langle proof \rangle$

lemma *not-ex*: $\neg (\exists x. P x) \longleftrightarrow (\forall x. \neg P x) \langle proof \rangle$

lemma *imp-ex*: $((\exists x. P x) \longrightarrow Q) \longleftrightarrow (\forall x. P x \longrightarrow Q) \langle proof \rangle$

lemma *all-not-ex*: $(\forall x. P x) \longleftrightarrow \neg (\exists x. \neg P x) \langle proof \rangle$

declare *All-def* [no-atp]

lemma *ex-disj-distrib*: $(\exists x. P x \vee Q x) \longleftrightarrow (\exists x. P x) \vee (\exists x. Q x) \langle proof \rangle$

lemma *all-conj-distrib*: $(\forall x. P x \wedge Q x) \longleftrightarrow (\forall x. P x) \wedge (\forall x. Q x) \langle proof \rangle$

lemma *all-imp-conj-distrib*: $(\forall x. P x \longrightarrow Q x \wedge R x) \longleftrightarrow (\forall x. P x \longrightarrow Q x) \wedge (\forall x. P x \longrightarrow R x)$

$\langle proof \rangle$

The \wedge congruence rule: not included by default! May slow rewrite proofs down by as much as 50%

lemma *conj-cong*: $P = P' \Longrightarrow (P' \Longrightarrow Q = Q') \Longrightarrow (P \wedge Q) = (P' \wedge Q')$
 $\langle proof \rangle$

lemma *rev-conj-cong*: $Q = Q' \Longrightarrow (Q' \Longrightarrow P = P') \Longrightarrow (P \wedge Q) = (P' \wedge Q')$
 $\langle proof \rangle$

The $|$ congruence rule: not included by default!

lemma *disj-cong*: $P = P' \Longrightarrow (\neg P' \Longrightarrow Q = Q') \Longrightarrow (P \vee Q) = (P' \vee Q')$
 $\langle proof \rangle$

if-then-else rules

lemma *if-True* [code]: $(\text{if True then } x \text{ else } y) = x$
 $\langle proof \rangle$

lemma *if-False* [code]: $(\text{if False then } x \text{ else } y) = y$
 $\langle proof \rangle$

lemma *if-P*: $P \Longrightarrow (\text{if } P \text{ then } x \text{ else } y) = x$
 $\langle proof \rangle$

lemma *if-not-P*: $\neg P \implies (\text{if } P \text{ then } x \text{ else } y) = y$
 ⟨proof⟩

lemma *if-split*: $P (\text{if } Q \text{ then } x \text{ else } y) = ((Q \longrightarrow P x) \wedge (\neg Q \longrightarrow P y))$
 ⟨proof⟩

lemma *if-split-asm*: $P (\text{if } Q \text{ then } x \text{ else } y) = (\neg ((Q \wedge \neg P x) \vee (\neg Q \wedge \neg P y)))$
 ⟨proof⟩

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

lemma *if-cancel*: $(\text{if } c \text{ then } x \text{ else } x) = x$
 ⟨proof⟩

lemma *if-eq-cancel*: $(\text{if } x = y \text{ then } y \text{ else } x) = x$
 ⟨proof⟩

lemma *if-bool-eq-conj*: $(\text{if } P \text{ then } Q \text{ else } R) = ((P \longrightarrow Q) \wedge (\neg P \longrightarrow R))$
 — This form is useful for expanding *ifs* on the RIGHT of the \implies symbol.
 ⟨proof⟩

lemma *if-bool-eq-disj*: $(\text{if } P \text{ then } Q \text{ else } R) = ((P \wedge Q) \vee (\neg P \wedge R))$
 — And this form is useful for expanding *ifs* on the LEFT.
 ⟨proof⟩

lemma *Eq-TrueI*: $P \implies P \equiv \text{True}$ ⟨proof⟩

lemma *Eq-FalseI*: $\neg P \implies P \equiv \text{False}$ ⟨proof⟩

let rules for *simproc*

lemma *Let-folded*: $f x \equiv g x \implies \text{Let } x f \equiv \text{Let } x g$
 ⟨proof⟩

lemma *Let-unfold*: $f x \equiv g \implies \text{Let } x f \equiv g$
 ⟨proof⟩

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* :: $\text{prop} \Rightarrow \text{prop} \Rightarrow \text{prop}$ (**infixr** $\langle =\text{simp}= \rangle$ 1)
 where *simp-implies* $\equiv (\implies)$

lemma *simp-impliesI*:
 assumes *PQ*: $(\text{PROP } P \implies \text{PROP } Q)$
 shows $\text{PROP } P =\text{simp}= \text{PROP } Q$
 ⟨proof⟩

lemma *simp-impliesE*:
 assumes *PQ*: $\text{PROP } P =\text{simp}= \text{PROP } Q$
 and *P*: $\text{PROP } P$

and $QR: PROP\ Q \Longrightarrow PROP\ R$
shows $PROP\ R$
 $\langle proof \rangle$

lemma *simp-implies-cong*:
assumes $PP': PROP\ P \equiv PROP\ P'$
and $P'QQ': PROP\ P' \Longrightarrow (PROP\ Q \equiv PROP\ Q')$
shows $(PROP\ P =_{simp} PROP\ Q) \equiv (PROP\ P' =_{simp} PROP\ Q')$
 $\langle proof \rangle$

lemma *uncurry*:
assumes $P \longrightarrow Q \longrightarrow R$
shows $P \wedge Q \longrightarrow R$
 $\langle proof \rangle$

lemma *iff-allI*:
assumes $\bigwedge x. P\ x = Q\ x$
shows $(\forall x. P\ x) = (\forall x. Q\ x)$
 $\langle proof \rangle$

lemma *iff-exI*:
assumes $\bigwedge x. P\ x = Q\ x$
shows $(\exists x. P\ x) = (\exists x. Q\ x)$
 $\langle proof \rangle$

lemma *all-comm*: $(\forall x\ y. P\ x\ y) = (\forall y\ x. P\ x\ y)$
 $\langle proof \rangle$

lemma *ex-comm*: $(\exists x\ y. P\ x\ y) = (\exists y\ x. P\ x\ y)$
 $\langle proof \rangle$

$\langle ML \rangle$

Simproc for proving $(y = x) \equiv False$ from premise $\neg (x = y)$:

$\langle ML \rangle$

lemma *True-implies-equals*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
 $\langle proof \rangle$

lemma *implies-True-equals*: $(PROP\ P \Longrightarrow True) \equiv Trueprop\ True$
 $\langle proof \rangle$

lemma *False-implies-equals*: $(False \Longrightarrow P) \equiv Trueprop\ True$
 $\langle proof \rangle$

lemma *implies-False-swap*:
 $(False \Longrightarrow PROP\ P \Longrightarrow PROP\ Q) \equiv (PROP\ P \Longrightarrow False \Longrightarrow PROP\ Q)$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *ex-simps*:

$$\begin{aligned} \bigwedge P Q. (\exists x. P x \wedge Q) &= ((\exists x. P x) \wedge Q) \\ \bigwedge P Q. (\exists x. P \wedge Q x) &= (P \wedge (\exists x. Q x)) \\ \bigwedge P Q. (\exists x. P x \vee Q) &= ((\exists x. P x) \vee Q) \\ \bigwedge P Q. (\exists x. P \vee Q x) &= (P \vee (\exists x. Q x)) \\ \bigwedge P Q. (\exists x. P x \longrightarrow Q) &= ((\forall x. P x) \longrightarrow Q) \\ \bigwedge P Q. (\exists x. P \longrightarrow Q x) &= (P \longrightarrow (\exists x. Q x)) \end{aligned}$$

— Miniscoping: pushing in existential quantifiers.
 $\langle proof \rangle$

lemma *all-simps*:

$$\begin{aligned} \bigwedge P Q. (\forall x. P x \wedge Q) &= ((\forall x. P x) \wedge Q) \\ \bigwedge P Q. (\forall x. P \wedge Q x) &= (P \wedge (\forall x. Q x)) \\ \bigwedge P Q. (\forall x. P x \vee Q) &= ((\forall x. P x) \vee Q) \\ \bigwedge P Q. (\forall x. P \vee Q x) &= (P \vee (\forall x. Q x)) \\ \bigwedge P Q. (\forall x. P x \longrightarrow Q) &= ((\exists x. P x) \longrightarrow Q) \\ \bigwedge P Q. (\forall x. P \longrightarrow Q x) &= (P \longrightarrow (\forall x. Q x)) \end{aligned}$$

— Miniscoping: pushing in universal quantifiers.
 $\langle proof \rangle$

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 $(P \vee Q \longrightarrow R) = ((P \longrightarrow R) \wedge (Q \longrightarrow R))$ 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 $[cong] = imp-cong \text{ simp-implies-cong}$
lemmas $[split] = if-split$

$\langle ML \rangle$

Simplifies x assuming c and y assuming $\neg c$.

lemma *if-cong*:
assumes $b = c$
and $c \implies x = u$
and $\neg c \implies y = v$
shows $(if\ b\ then\ x\ else\ y) = (if\ c\ then\ u\ else\ v)$
 $\langle proof \rangle$

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

lemma *if-weak-cong* $[cong]$:
assumes $b = c$
shows $(if\ b\ then\ x\ else\ y) = (if\ c\ then\ x\ else\ y)$
 $\langle proof \rangle$

Prevents simplification of t : much faster

lemma *let-weak-cong*:
assumes $a = b$
shows $(let\ x = a\ in\ t\ x) = (let\ x = b\ in\ t\ x)$
 $\langle proof \rangle$

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

lemma *eq-cong2*:
assumes $u = u'$
shows $(t \equiv u) \equiv (t \equiv u')$
 $\langle proof \rangle$

lemma *if-distrib*: $f\ (if\ c\ then\ x\ else\ y) = (if\ c\ then\ f\ x\ else\ f\ y)$
 $\langle proof \rangle$

lemma *if-distribR*: $(if\ b\ then\ f\ else\ g)\ x = (if\ b\ then\ f\ x\ else\ g\ x)$
 $\langle proof \rangle$

lemma *all-if-distrib*: $(\forall x. if\ x = a\ then\ P\ x\ else\ Q\ x) \longleftrightarrow P\ a \wedge (\forall x. x \neq a \longrightarrow Q\ x)$
 $\langle proof \rangle$

lemma *ex-if-distrib*: $(\exists x. \text{if } x = a \text{ then } P \ x \text{ else } Q \ x) \longleftrightarrow P \ a \vee (\exists x. x \neq a \wedge Q \ x)$
 $\langle \text{proof} \rangle$

lemma *if-if-eq-conj*: $(\text{if } P \text{ then if } Q \text{ then } x \text{ else } y \text{ else } y) = (\text{if } P \wedge Q \text{ then } x \text{ else } y)$
 $\langle \text{proof} \rangle$

As a simplification rule, it replaces all function equalities by first-order equalities.

lemma *fun-eq-iff*: $f = g \longleftrightarrow (\forall x. f \ x = g \ x)$
 $\langle \text{proof} \rangle$

2.3.5 Generic cases and induction

Rule projections:

$\langle ML \rangle$

context
begin

qualified definition *induct-forall* $P \equiv \forall x. P \ x$

qualified definition *induct-implies* $A \ B \equiv A \longrightarrow B$

qualified definition *induct-equal* $x \ y \equiv x = y$

qualified definition *induct-conj* $A \ B \equiv A \wedge B$

qualified definition *induct-true* $\equiv \text{True}$

qualified definition *induct-false* $\equiv \text{False}$

lemma *induct-forall-eq*: $(\bigwedge x. P \ x) \equiv \text{Trueprop } (\text{induct-forall } (\lambda x. P \ x))$
 $\langle \text{proof} \rangle$

lemma *induct-implies-eq*: $(A \implies B) \equiv \text{Trueprop } (\text{induct-implies } A \ B)$
 $\langle \text{proof} \rangle$

lemma *induct-equal-eq*: $(x \equiv y) \equiv \text{Trueprop } (\text{induct-equal } x \ y)$
 $\langle \text{proof} \rangle$

lemma *induct-conj-eq*: $(A \ \&\&\& \ B) \equiv \text{Trueprop } (\text{induct-conj } A \ B)$
 $\langle \text{proof} \rangle$

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*: $\text{induct-forall } (\lambda x. \text{induct-conj } (A \ x) (B \ x)) =$

induct-conj (induct-forall A) (induct-forall B)
<proof>

lemma *induct-implies-conj*: *induct-implies C (induct-conj A B) =*
induct-conj (induct-implies C A) (induct-implies C B)
<proof>

lemma *induct-conj-curry*: *(induct-conj A B \implies PROP C) \equiv (A \implies B \implies PROP C)*
<proof>

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

lemma *induct-trueI*: *induct-true*
<proof>

Method setup.

<ML>

Pre-simplification of induction and cases rules

lemma [*induct-simp*]: *($\bigwedge x. \text{induct-equal } x \ t \implies \text{PROP } P \ x$) \equiv PROP P t*
<proof>

lemma [*induct-simp*]: *($\bigwedge x. \text{induct-equal } t \ x \implies \text{PROP } P \ x$) \equiv PROP P t*
<proof>

lemma [*induct-simp*]: *(induct-false \implies P) \equiv Trueprop induct-true*
<proof>

lemma [*induct-simp*]: *(induct-true \implies PROP P) \equiv PROP P*
<proof>

lemma [*induct-simp*]: *(PROP P \implies induct-true) \equiv Trueprop induct-true*
<proof>

lemma [*induct-simp*]: *($\bigwedge x::'a::\{\}. \text{induct-true}$) \equiv Trueprop induct-true*
<proof>

lemma [*induct-simp*]: *induct-implies induct-true P \equiv P*
<proof>

lemma [*induct-simp*]: *x = x \longleftrightarrow True*
<proof>

end

<ML>

2.3.6 Coherent logic

$\langle ML \rangle$

2.3.7 Reorienting equalities

$\langle ML \rangle$

2.4 Other simple lemmas and lemma duplicates

lemma *eq-iff-swap*: $(x = y \longleftrightarrow P) \implies (y = x \longleftrightarrow P)$
 $\langle proof \rangle$

lemma *all-cong1*: $(\bigwedge x. P\ x = P'\ x) \implies (\forall x. P\ x) = (\forall x. P'\ x)$
 $\langle proof \rangle$

lemma *ex-cong1*: $(\bigwedge x. P\ x = P'\ x) \implies (\exists x. P\ x) = (\exists x. P'\ x)$
 $\langle proof \rangle$

lemma *all-cong*: $(\bigwedge x. Q\ x \implies P\ x = P'\ x) \implies (\forall x. Q\ x \longrightarrow P\ x) = (\forall x. Q\ x \longrightarrow P'\ x)$
 $\langle proof \rangle$

lemma *ex-cong*: $(\bigwedge x. Q\ x \implies P\ x = P'\ x) \implies (\exists x. Q\ x \wedge P\ x) = (\exists x. Q\ x \wedge P'\ x)$
 $\langle proof \rangle$

lemma *ex1-eq [iff]*: $\exists!x. x = t \iff \exists!x. t = x$
 $\langle proof \rangle$

lemma *choice-eq*: $(\forall x. \exists!y. P\ x\ y) = (\exists!f. \forall x. P\ x\ (f\ x))$ (**is** *?lhs = ?rhs*)
 $\langle proof \rangle$

lemmas *eq-sym-conv = eq-commute*

lemma *nnf-simps*:

$(\neg (P \wedge Q)) = (\neg P \vee \neg Q)$
 $(\neg (P \vee Q)) = (\neg P \wedge \neg Q)$
 $(P \longrightarrow Q) = (\neg P \vee Q)$
 $(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q))$
 $(\neg (P = Q)) = ((P \wedge \neg Q) \vee (\neg P \wedge Q))$
 $(\neg \neg P) = P$
 $\langle proof \rangle$

2.5 Basic ML bindings

$\langle ML \rangle$

locale *cnf*
begin

lemma *clause2raw-notE*: $\llbracket P; \neg P \rrbracket \Longrightarrow \text{False} \langle \text{proof} \rangle$

lemma *clause2raw-not-disj*: $\llbracket \neg P; \neg Q \rrbracket \Longrightarrow \neg (P \vee Q) \langle \text{proof} \rangle$

lemma *clause2raw-not-not*: $P \Longrightarrow \neg \neg P \langle \text{proof} \rangle$

lemma *iff-refl*: $(P::\text{bool}) = P \langle \text{proof} \rangle$

lemma *iff-trans*: $\llbracket (P::\text{bool}) = Q; Q = R \rrbracket \Longrightarrow P = R \langle \text{proof} \rangle$

lemma *conj-cong*: $\llbracket P = P'; Q = Q' \rrbracket \Longrightarrow (P \wedge Q) = (P' \wedge Q') \langle \text{proof} \rangle$

lemma *disj-cong*: $\llbracket P = P'; Q = Q' \rrbracket \Longrightarrow (P \vee Q) = (P' \vee Q') \langle \text{proof} \rangle$

lemma *make-nnf-imp*: $\llbracket (\neg P) = P'; Q = Q' \rrbracket \Longrightarrow (P \longrightarrow Q) = (P' \vee Q') \langle \text{proof} \rangle$

lemma *make-nnf-iff*: $\llbracket P = P'; (\neg P) = NP; Q = Q'; (\neg Q) = NQ \rrbracket \Longrightarrow (P = Q) = ((P' \vee NQ) \wedge (NP \vee Q')) \langle \text{proof} \rangle$

lemma *make-nnf-not-false*: $(\neg \text{False}) = \text{True} \langle \text{proof} \rangle$

lemma *make-nnf-not-true*: $(\neg \text{True}) = \text{False} \langle \text{proof} \rangle$

lemma *make-nnf-not-conj*: $\llbracket (\neg P) = P'; (\neg Q) = Q' \rrbracket \Longrightarrow (\neg(P \wedge Q)) = (P' \vee Q') \langle \text{proof} \rangle$

lemma *make-nnf-not-disj*: $\llbracket (\neg P) = P'; (\neg Q) = Q' \rrbracket \Longrightarrow (\neg(P \vee Q)) = (P' \wedge Q') \langle \text{proof} \rangle$

lemma *make-nnf-not-imp*: $\llbracket P = P'; (\neg Q) = Q' \rrbracket \Longrightarrow (\neg(P \longrightarrow Q)) = (P' \wedge Q') \langle \text{proof} \rangle$

lemma *make-nnf-not-iff*: $\llbracket P = P'; (\neg P) = NP; Q = Q'; (\neg Q) = NQ \rrbracket \Longrightarrow (\neg(P = Q)) = ((P' \vee Q') \wedge (NP \vee NQ)) \langle \text{proof} \rangle$

lemma *make-nnf-not-not*: $P = P' \Longrightarrow (\neg \neg P) = P' \langle \text{proof} \rangle$

lemma *simp-TF-conj-True-l*: $\llbracket P = \text{True}; Q = Q' \rrbracket \Longrightarrow (P \wedge Q) = Q' \langle \text{proof} \rangle$

lemma *simp-TF-conj-True-r*: $\llbracket P = P'; Q = \text{True} \rrbracket \Longrightarrow (P \wedge Q) = P' \langle \text{proof} \rangle$

lemma *simp-TF-conj-False-l*: $P = \text{False} \Longrightarrow (P \wedge Q) = \text{False} \langle \text{proof} \rangle$

lemma *simp-TF-conj-False-r*: $Q = \text{False} \Longrightarrow (P \wedge Q) = \text{False} \langle \text{proof} \rangle$

lemma *simp-TF-disj-True-l*: $P = \text{True} \Longrightarrow (P \vee Q) = \text{True} \langle \text{proof} \rangle$

lemma *simp-TF-disj-True-r*: $Q = \text{True} \Longrightarrow (P \vee Q) = \text{True} \langle \text{proof} \rangle$

lemma *simp-TF-disj-False-l*: $\llbracket P = \text{False}; Q = Q' \rrbracket \Longrightarrow (P \vee Q) = Q' \langle \text{proof} \rangle$

lemma *simp-TF-disj-False-r*: $\llbracket P = P'; Q = \text{False} \rrbracket \Longrightarrow (P \vee Q) = P' \langle \text{proof} \rangle$

lemma *make-cnf-disj-conj-l*: $\llbracket (P \vee R) = PR; (Q \vee R) = QR \rrbracket \Longrightarrow ((P \wedge Q) \vee R) = (PR \wedge QR) \langle \text{proof} \rangle$

lemma *make-cnf-disj-conj-r*: $\llbracket (P \vee Q) = PQ; (P \vee R) = PR \rrbracket \Longrightarrow (P \vee (Q \wedge R)) = (PQ \wedge PR) \langle \text{proof} \rangle$

lemma *make-cnfx-disj-ex-l*: $((\exists (x::\text{bool}). P x) \vee Q) = (\exists x. P x \vee Q) \langle \text{proof} \rangle$

lemma *make-cnfx-disj-ex-r*: $(P \vee (\exists (x::\text{bool}). Q x)) = (\exists x. P \vee Q x) \langle \text{proof} \rangle$

lemma *make-cnfx-newlit*: $(P \vee Q) = (\exists x. (P \vee x) \wedge (Q \vee \neg x)) \langle \text{proof} \rangle$

lemma *make-cnfx-ex-cong*: $(\forall (x::\text{bool}). P x = Q x) \Longrightarrow (\exists x. P x) = (\exists x. Q x) \langle \text{proof} \rangle$

lemma *weakening-thm*: $\llbracket P; Q \rrbracket \Longrightarrow Q \langle \text{proof} \rangle$

lemma *cnftac-eq-imp*: $\llbracket P = Q; P \rrbracket \Longrightarrow Q \langle \text{proof} \rangle$

end

$\langle ML \rangle$

3 *NO-MATCH* simproc

The simplification procedure can be used to avoid simplification of terms of a certain form.

definition *NO-MATCH* :: 'a \Rightarrow 'b \Rightarrow bool
where *NO-MATCH* pat val \equiv True

lemma *NO-MATCH-cong*[cong]: *NO-MATCH* pat val = *NO-MATCH* pat val
 $\langle proof \rangle$

declare [[*coercion-args NO-MATCH - -*]]

$\langle ML \rangle$

This setup ensures that a rewrite rule of the form *NO-MATCH* pat val \Rightarrow *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 \Rightarrow bool
where *ASSUMPTION* A \equiv A

lemma *ASSUMPTION-cong*[cong]: *ASSUMPTION* A = *ASSUMPTION* A
 $\langle proof \rangle$

lemma *ASSUMPTION-I*: A \Rightarrow *ASSUMPTION* A
 $\langle proof \rangle$

lemma *ASSUMPTION-D*: *ASSUMPTION* A \Rightarrow A
 $\langle proof \rangle$

$\langle ML \rangle$

3.1 Code generator setup

3.1.1 Generic code generator preprocessor setup

lemma *conj-left-cong*: P \longleftrightarrow Q \Rightarrow P \wedge R \longleftrightarrow Q \wedge R
 $\langle proof \rangle$

lemma *disj-left-cong*: P \longleftrightarrow Q \Rightarrow P \vee R \longleftrightarrow Q \vee R
 $\langle proof \rangle$

$\langle ML \rangle$

3.1.2 Generic code generator foundation

Datatype *bool*

code-datatype *True False*

lemma [*code*]:

$P \wedge \text{True} \longleftrightarrow P$
 $P \wedge \text{False} \longleftrightarrow \text{False}$
 $\text{True} \wedge P \longleftrightarrow P$
 $\text{False} \wedge P \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma [*code*]:

$P \vee \text{True} \longleftrightarrow \text{True}$
 $P \vee \text{False} \longleftrightarrow P$
 $\text{True} \vee P \longleftrightarrow \text{True}$
 $\text{False} \vee P \longleftrightarrow P$
 $\langle \text{proof} \rangle$

lemma [*code*]:

$(P \longrightarrow \text{True}) \longleftrightarrow \text{True}$
 $(P \longrightarrow \text{False}) \longleftrightarrow \neg P$
 $(\text{True} \longrightarrow P) \longleftrightarrow P$
 $(\text{False} \longrightarrow P) \longleftrightarrow \text{True}$
 $\langle \text{proof} \rangle$

More about *prop*

lemma [*code nbe*]:

$(P \Longrightarrow R) \equiv \text{Trueprop } (P \longrightarrow R)$
 $(\text{PROP } Q \Longrightarrow \text{True}) \equiv \text{Trueprop } \text{True}$
 $(\text{True} \Longrightarrow \text{PROP } Q) \equiv \text{PROP } Q$
 $\langle \text{proof} \rangle$

lemma *Trueprop-code* [*code*]: $\text{Trueprop } \text{True} \equiv \text{Code-Generator.holds}$

$\langle \text{proof} \rangle$

declare *Trueprop-code* [*symmetric, code-post*]

Cases

lemma *Let-case-cert*:

assumes *CASE* $\equiv (\lambda x. \text{Let } x \text{ } f)$
shows *CASE* $x \equiv f \ x$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

declare $[[\text{code abort: undefined}]]$

3.1.3 Equality

lemma *[code nbe]*:
 $\langle x = x \longleftrightarrow \text{True} \rangle$
 $\langle \text{proof} \rangle$

class *equal* =
fixes *equal* :: 'a \Rightarrow 'a \Rightarrow bool
assumes *equal-eq*: *equal* *x y* \longleftrightarrow *x* = *y*
begin

lemma *eq-equal* *[code]*: (=) \equiv *equal*
 $\langle \text{proof} \rangle$

lemma *equal* *[code-post]*: *equal* = (=)
 $\langle \text{proof} \rangle$

lemma *equal-refl*: *equal* *x x* \longleftrightarrow *True*
 $\langle \text{proof} \rangle$

end

$\langle ML \rangle$

instantiation *itself* :: (*type*) *equal*
begin

definition *equal-itself* :: 'a *itself* \Rightarrow 'a *itself* \Rightarrow bool
where *equal-itself* *x y* \longleftrightarrow *x* = *y*

instance
 $\langle \text{proof} \rangle$

end

lemma *equal-itself-code* *[code]*: *equal* *TYPE*('a) *TYPE*('a) \longleftrightarrow *True*
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *equal-alias-cert*: *OFCLASS*('a, *equal-class*) \equiv (((=) :: 'a \Rightarrow 'a \Rightarrow bool) \equiv *equal*)
 (**is** *?ofclass* \equiv *?equal*)
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

3.1.4 Generic code generator target languages

type *bool*

code-printing

```

type-constructor bool  $\mapsto$ 
  (SML) bool and (OCaml) bool and (Haskell) Bool and (Scala) Boolean
| constant True  $\mapsto$ 
  (SML) true and (OCaml) true and (Haskell) True and (Scala) true
| constant False  $\mapsto$ 
  (SML) false and (OCaml) false and (Haskell) False and (Scala) false

```

code-reserved

```

(SML) bool true false
and (OCaml) bool
and (Scala) Boolean

```

code-printing

```

constant Not  $\mapsto$ 
  (SML) not and (OCaml) not and (Haskell) not and (Scala) ! -
| constant HOL.conj  $\mapsto$ 
  (SML) infixl 1 andalso and (OCaml) infixl 3 && and (Haskell) infixr 3 &&
and (Scala) infixl 3 &&
| constant HOL.disj  $\mapsto$ 
  (SML) infixl 0 orelse and (OCaml) infixl 2 || and (Haskell) infixl 2 || and
(Scala) infixl 1 ||
| constant HOL.implies  $\mapsto$ 
  (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 / })
| constant If  $\mapsto$ 
  (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
and (OCaml) not

```

code-identifier

```

code-module Pure  $\mapsto$ 
  (SML) HOL and (OCaml) HOL and (Haskell) HOL and (Scala) HOL

```

Using built-in Haskell equality.

code-printing

```

type-class equal  $\mapsto$  (Haskell) Eq
| constant HOL.equal  $\mapsto$  (Haskell) infix 4 ==
| constant HOL.eq  $\mapsto$  (Haskell) infix 4 ==

```

undefined

code-printing

```

constant undefined  $\dashv$ 
  (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

$\langle ML \rangle$

3.2 Counterexample Search Units

3.2.1 Quickcheck

```

quickcheck-params [size = 5, iterations = 50]

```

3.2.2 Nitpick setup

```

named-theorems nitpick-unfold alternative definitions of constants as needed by Nitpick

```

```

  and nitpick-simp equational specification of constants as needed by Nitpick
  and nitpick-psimp partial equational specification of constants as needed by Nitpick
  and nitpick-choice-spec choice specification of constants as needed by Nitpick

```

```

declare if-bool-eq-conj [nitpick-unfold, no-atp]
  and if-bool-eq-disj [no-atp]

```

3.3 Preprocessing for the predicate compiler

```

named-theorems code-pred-def alternative definitions of constants for the Predicate Compiler

```

```

  and code-pred-inline inlining definitions for the Predicate Compiler
  and code-pred-simp simplification rules for the optimisations in the Predicate Compiler

```

3.4 Legacy tactics and ML bindings

$\langle ML \rangle$

```

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* :: $\langle 'a \Rightarrow 'a \Rightarrow \text{bool} \rangle$ (**infix** $\langle \leq \rangle$ 50)
 assumes *refl*: $\langle a \leq a \rangle$ — not *iff*: makes problems due to multiple (dual) interpretations
 and *trans*: $\langle a \leq b \Longrightarrow b \leq c \Longrightarrow a \leq c \rangle$

locale *preordering* = *partial-preordering* +
 fixes *less* :: $\langle 'a \Rightarrow 'a \Rightarrow \text{bool} \rangle$ (**infix** $\langle < \rangle$ 50)
 assumes *strict-iff-not*: $\langle a < b \longleftrightarrow a \leq b \wedge \neg b \leq a \rangle$
begin

lemma *strict-implies-order*:
 $\langle a < b \Longrightarrow a \leq b \rangle$
 $\langle \text{proof} \rangle$

lemma *irrefl*: — not *iff*: makes problems due to multiple (dual) interpretations
 $\langle \neg a < a \rangle$
 $\langle \text{proof} \rangle$

lemma *asym*:
 $\langle a < b \Longrightarrow b < a \Longrightarrow \text{False} \rangle$
 $\langle \text{proof} \rangle$

lemma *strict-trans1*:
 $\langle a \leq b \Longrightarrow b < c \Longrightarrow a < c \rangle$
 $\langle \text{proof} \rangle$

lemma *strict-trans2*:
 $\langle a < b \Longrightarrow b \leq c \Longrightarrow a < c \rangle$
 $\langle \text{proof} \rangle$

lemma *strict-trans*:
 $\langle a < b \Longrightarrow b < c \Longrightarrow a < c \rangle$
 $\langle \text{proof} \rangle$

end

lemma *preordering-strictI*: — Alternative introduction rule with bias towards strict order

fixes *less-eq* (**infix** $\langle \leq \rangle$ 50)
 and *less* (**infix** $\langle < \rangle$ 50)
 assumes *less-eq-less*: $\langle \bigwedge a b. a \leq b \longleftrightarrow a < b \vee a = b \rangle$
 assumes *asym*: $\langle \bigwedge a b. a < b \Longrightarrow \neg b < a \rangle$
 assumes *irrefl*: $\langle \bigwedge a. \neg a < a \rangle$
 assumes *trans*: $\langle \bigwedge a b c. a < b \Longrightarrow b < c \Longrightarrow a < c \rangle$
 shows $\langle \text{preordering } (\leq) (<) \rangle$
 $\langle \text{proof} \rangle$

lemma *preordering-dualI*:
fixes *less-eq* (**infix** $\langle \leq \rangle$ 50)
and *less* (**infix** $\langle < \rangle$ 50)
assumes $\langle \text{preordering } (\lambda a \ b. \ b \leq a) \ (\lambda a \ b. \ b < a) \rangle$
shows $\langle \text{preordering } (\leq) \ (\langle \rangle) \rangle$
 $\langle \text{proof} \rangle$

locale *ordering* = *partial-preordering* +
fixes *less* :: $\langle 'a \Rightarrow 'a \Rightarrow \text{bool} \rangle$ (**infix** $\langle < \rangle$ 50)
assumes *strict-iff-order*: $\langle a < b \longleftrightarrow a \leq b \wedge a \neq b \rangle$
assumes *antisym*: $\langle a \leq b \Longrightarrow b \leq a \Longrightarrow a = b \rangle$
begin

sublocale *preordering* $\langle (\leq) \rangle$ $\langle (\langle) \rangle$
 $\langle \text{proof} \rangle$

lemma *strict-implies-not-eq*:
 $\langle a < b \Longrightarrow a \neq b \rangle$
 $\langle \text{proof} \rangle$

lemma *not-eq-order-implies-strict*:
 $\langle a \neq b \Longrightarrow a \leq b \Longrightarrow a < b \rangle$
 $\langle \text{proof} \rangle$

lemma *order-iff-strict*:
 $\langle a \leq b \longleftrightarrow a < b \vee a = b \rangle$
 $\langle \text{proof} \rangle$

lemma *eq-iff*: $\langle a = b \longleftrightarrow a \leq b \wedge b \leq a \rangle$
 $\langle \text{proof} \rangle$

end

lemma *ordering-strictI*: — Alternative introduction rule with bias towards strict order

fixes *less-eq* (**infix** $\langle \leq \rangle$ 50)
and *less* (**infix** $\langle < \rangle$ 50)
assumes *less-eq-less*: $\langle \bigwedge a \ b. \ a \leq b \longleftrightarrow a < b \vee a = b \rangle$
assumes *asym*: $\langle \bigwedge a \ b. \ a < b \Longrightarrow \neg b < a \rangle$
assumes *irrefl*: $\langle \bigwedge a. \ \neg a < a \rangle$
assumes *trans*: $\langle \bigwedge a \ b \ c. \ a < b \Longrightarrow b < c \Longrightarrow a < c \rangle$
shows $\langle \text{ordering } (\leq) \ (\langle \rangle) \rangle$
 $\langle \text{proof} \rangle$

lemma *ordering-dualI*:
fixes *less-eq* (**infix** $\langle \leq \rangle$ 50)
and *less* (**infix** $\langle < \rangle$ 50)
assumes $\langle \text{ordering } (\lambda a \ b. \ b \leq a) \ (\lambda a \ b. \ b < a) \rangle$
shows $\langle \text{ordering } (\leq) \ (\langle \rangle) \rangle$

$\langle proof \rangle$

locale *ordering-top* = *ordering* +
fixes *top* :: $\langle 'a \rangle$ ($\langle \top \rangle$)
assumes *extremum* [*simp*]: $\langle a \leq \top \rangle$
begin

lemma *extremum-uniqueI*:
 $\langle \top \leq a \implies a = \top \rangle$
 $\langle proof \rangle$

lemma *extremum-unique*:
 $\langle \top \leq a \longleftrightarrow a = \top \rangle$
 $\langle proof \rangle$

lemma *extremum-strict* [*simp*]:
 $\langle \neg (\top < a) \rangle$
 $\langle proof \rangle$

lemma *not-eq-extremum*:
 $\langle a \neq \top \longleftrightarrow a < \top \rangle$
 $\langle proof \rangle$

end

4.2 Syntactic orders

class *ord* =
fixes *less-eq* :: $'a \Rightarrow 'a \Rightarrow bool$
and *less* :: $'a \Rightarrow 'a \Rightarrow bool$
begin

notation
less-eq ($\langle '(\leq) \rangle$) **and**
less-eq ($\langle (\langle notation=infix \le \rangle - / \le -) \rangle$ [*51*, *51*] *50*) **and**
less ($\langle '(<) \rangle$) **and**
less ($\langle (\langle notation=infix < \rangle - / < -) \rangle$ [*51*, *51*] *50*)

abbreviation (*input*)
greater-eq (**infix** $\langle \geq \rangle$ *50*)
where $x \geq y \equiv y \leq x$

abbreviation (*input*)
greater (**infix** $\langle > \rangle$ *50*)
where $x > y \equiv y < x$

notation (*ASCII*)
less-eq ($\langle '(<=) \rangle$) **and**
less-eq ($\langle (\langle notation=infix <= \rangle - / <= -) \rangle$ [*51*, *51*] *50*)

notation (*input*)
greater-eq (**infix** $\langle \rangle = \rangle$ 50)

end

4.3 Quasi orders

class *preorder* = *ord* +
 assumes *less-le-not-le*: $x < y \longleftrightarrow x \leq y \wedge \neg (y \leq x)$
 and *order-refl* [*iff*]: $x \leq x$
 and *order-trans*: $x \leq y \implies y \leq z \implies x \leq z$
begin

sublocale *order*: *preordering less-eq less* + *dual-order*: *preordering greater-eq greater*
 $\langle \text{proof} \rangle$

Reflexivity.

lemma *eq-refl*: $x = y \implies x \leq y$
 — This form is useful with the classical reasoner.
 $\langle \text{proof} \rangle$

lemma *less-irrefl* [*iff*]: $\neg x < x$
 $\langle \text{proof} \rangle$

lemma *less-imp-le*: $x < y \implies x \leq y$
 $\langle \text{proof} \rangle$

Asymmetry.

lemma *less-not-sym*: $x < y \implies \neg (y < x)$
 $\langle \text{proof} \rangle$

lemma *less-asy*: $x < y \implies (\neg P \implies y < x) \implies P$
 $\langle \text{proof} \rangle$

Transitivity.

lemma *less-trans*: $x < y \implies y < z \implies x < z$
 $\langle \text{proof} \rangle$

lemma *le-less-trans*: $x \leq y \implies y < z \implies x < z$
 $\langle \text{proof} \rangle$

lemma *less-le-trans*: $x < y \implies y \leq z \implies x < z$
 $\langle \text{proof} \rangle$

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

lemma *less-imp-not-less*: $x < y \implies (\neg y < x) \longleftrightarrow \text{True}$
 $\langle \text{proof} \rangle$

lemma *less-imp-triv*: $x < y \implies (y < x \implies P) \longleftrightarrow \text{True}$
 ⟨proof⟩

Transitivity rules for calculational reasoning

lemma *less-asym'*: $a < b \implies b < a \implies P$
 ⟨proof⟩

Dual order

lemma *dual-preorder*:
 ⟨class.preorder (\geq) $(>)$ ⟩
 ⟨proof⟩

end

lemma *preordering-preorderI*:
 ⟨class.preorder (\leq) $(<)$ ⟩ if ⟨preordering (\leq) $(<)$ ⟩
 for less-eq (infix \leq 50) and less (infix $<$ 50)
 ⟨proof⟩

4.4 Partial orders

class *order* = *preorder* +
 assumes *order-antisym*: $x \leq y \implies y \leq x \implies x = y$
begin

lemma *less-le*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$
 ⟨proof⟩

sublocale *order*: *ordering less-eq less* + *dual-order*: *ordering greater-eq greater*
 ⟨proof⟩

Reflexivity.

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x = y$
 — NOT suitable for iff, since it can cause PROOF FAILED.
 ⟨proof⟩

lemma *le-imp-less-or-eq*: $x \leq y \implies x < y \vee x = y$
 ⟨proof⟩

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

lemma *less-imp-not-eq*: $x < y \implies (x = y) \longleftrightarrow \text{False}$
 ⟨proof⟩

lemma *less-imp-not-eq2*: $x < y \implies (y = x) \longleftrightarrow \text{False}$
 ⟨proof⟩

Transitivity rules for calculational reasoning

lemma *neq-le-trans*: $a \neq b \implies a \leq b \implies a < b$

$\langle \text{proof} \rangle$

lemma *le-neq-trans*: $a \leq b \implies a \neq b \implies a < b$

$\langle \text{proof} \rangle$

Asymmetry.

lemma *order-eq-iff*: $x = y \longleftrightarrow x \leq y \wedge y \leq x$

$\langle \text{proof} \rangle$

lemma *antisym-conv*: $y \leq x \implies x \leq y \longleftrightarrow x = y$

$\langle \text{proof} \rangle$

lemma *less-imp-neq*: $x < y \implies x \neq y$

$\langle \text{proof} \rangle$

lemma *antisym-conv1*: $\neg x < y \implies x \leq y \longleftrightarrow x = y$

$\langle \text{proof} \rangle$

lemma *antisym-conv2*: $x \leq y \implies \neg x < y \longleftrightarrow x = y$

$\langle \text{proof} \rangle$

lemma *leD*: $y \leq x \implies \neg x < y$

$\langle \text{proof} \rangle$

Least value operator

definition (*in ord*)

Least :: ($'a \Rightarrow \text{bool}$) $\Rightarrow 'a$ (**binder** $\langle \text{LEAST} \rangle 10$) **where**

Least $P = (\text{THE } x. P\ x \wedge (\forall y. P\ y \longrightarrow x \leq y))$

lemma *Least-equality*:

assumes $P\ x$

and $\bigwedge y. P\ y \implies x \leq y$

shows $\text{Least } P = x$

$\langle \text{proof} \rangle$

lemma *LeastI2-order*:

assumes $P\ x$

and $\bigwedge y. P\ y \implies x \leq y$

and $\bigwedge x. P\ x \implies \forall y. P\ y \longrightarrow x \leq y \implies Q\ x$

shows $Q\ (\text{Least } P)$

$\langle \text{proof} \rangle$

lemma *Least-ex1*:

assumes $\exists! x. P\ x \wedge (\forall y. P\ y \longrightarrow x \leq y)$

shows $\text{LeastI1}: P\ (\text{Least } P)$ **and** *Least1-le*: $P\ z \implies \text{Least } P \leq z$

$\langle \text{proof} \rangle$

Greatest value operator

definition *Greatest* :: ($'a \Rightarrow \text{bool}$) $\Rightarrow 'a$ (**binder** $\langle \text{GREATEST} \rangle 10$) **where**

$Greatest\ P = (THE\ x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow x \geq y))$

lemma *GreatestI2-order:*

[[$P\ x$;
 $\bigwedge y.\ P\ y \implies x \geq y$;
 $\bigwedge x.\ [P\ x; \forall y.\ P\ y \longrightarrow x \geq y] \implies Q\ x$]
 $\implies Q\ (Greatest\ P)$
 $\langle proof \rangle$

lemma *Greatest-equality:*

[[$P\ x$; $\bigwedge y.\ P\ y \implies x \geq y$] $\implies Greatest\ P = x$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemma *order-strictI:*

fixes *less* (**infix** $<$ 50)
and *less-eq* (**infix** \leq 50)
assumes $\bigwedge a\ b.\ a \leq b \longleftrightarrow a < b \vee a = b$
assumes $\bigwedge a\ b.\ a < b \implies \neg b < a$
assumes $\bigwedge a.\ \neg a < a$
assumes $\bigwedge a\ b\ c.\ a < b \implies b < c \implies a < c$
shows *class.order less-eq less*
 $\langle proof \rangle$

context *order*

begin

Dual order

lemma *dual-order:*

class.order (\geq) ($>$)
 $\langle proof \rangle$

end

4.5 Linear (total) orders

class *linorder* = *order* +

assumes *linear*: $x \leq y \vee y \leq x$

begin

lemma *less-linear*: $x < y \vee x = y \vee y < x$

$\langle \text{proof} \rangle$

lemma *le-less-linear*: $x \leq y \vee y < x$

$\langle \text{proof} \rangle$

lemma *le-cases* [*case-names le ge*]:

$(x \leq y \implies P) \implies (y \leq x \implies P) \implies P$

$\langle \text{proof} \rangle$

lemma (*in linorder*) *le-cases3*:

$\llbracket x \leq y; y \leq z \rrbracket \implies P; \llbracket y \leq x; x \leq z \rrbracket \implies P; \llbracket x \leq z; z \leq y \rrbracket \implies P;$
 $\llbracket z \leq y; y \leq x \rrbracket \implies P; \llbracket y \leq z; z \leq x \rrbracket \implies P; \llbracket z \leq x; x \leq y \rrbracket \implies P \implies P$

$\langle \text{proof} \rangle$

lemma *linorder-cases* [*case-names less equal greater*]:

$(x < y \implies P) \implies (x = y \implies P) \implies (y < x \implies P) \implies P$

$\langle \text{proof} \rangle$

lemma *linorder-wlog* [*case-names le sym*]:

$(\bigwedge a\ b. a \leq b \implies P\ a\ b) \implies (\bigwedge a\ b. P\ b\ a \implies P\ a\ b) \implies P\ a\ b$

$\langle \text{proof} \rangle$

lemma *not-less*: $\neg x < y \longleftrightarrow y \leq x$

$\langle \text{proof} \rangle$

lemma *not-less-iff-gr-or-eq*: $\neg(x < y) \longleftrightarrow (x > y \vee x = y)$

$\langle \text{proof} \rangle$

lemma *not-le*: $\neg x \leq y \longleftrightarrow y < x$

$\langle \text{proof} \rangle$

lemma *neq-iff*: $x \neq y \longleftrightarrow x < y \vee y < x$

$\langle \text{proof} \rangle$

lemma *neqE*: $x \neq y \implies (x < y \implies R) \implies (y < x \implies R) \implies R$

$\langle \text{proof} \rangle$

lemma *antisym-conv3*: $\neg y < x \implies \neg x < y \longleftrightarrow x = y$

$\langle \text{proof} \rangle$

lemma *leI*: $\neg x < y \implies y \leq x$

$\langle \text{proof} \rangle$

lemma *not-le-imp-less*: $\neg y \leq x \implies x < y$

$\langle \text{proof} \rangle$

lemma *linorder-less-wlog* [*case-names less refl sym*]:

$\llbracket \bigwedge a\ b. a < b \implies P\ a\ b; \bigwedge a. P\ a\ a; \bigwedge a\ b. P\ b\ a \implies P\ a\ b \rrbracket \implies P\ a\ b$

$\langle \text{proof} \rangle$

Dual order

```
lemma dual-linorder:
  class.linorder ( $\geq$ ) ( $>$ )
   $\langle proof \rangle$ 
```

end

Alternative introduction rule with bias towards strict order

```
lemma linorder-strictI:
  fixes less-eq (infix  $\leq$  50)
    and less (infix  $<$  50)
  assumes class.order less-eq less
  assumes trichotomy:  $\bigwedge a b. a < b \vee a = b \vee b < a$ 
  shows class.linorder less-eq less
   $\langle proof \rangle$ 
```

4.6 Reasoning tools setup

$\langle ML \rangle$

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.

Declarations to set up transitivity reasoner of partial and linear orders.

context *order*

begin

lemma *nless-le*: $(\neg a < b) \longleftrightarrow (\neg a \leq b) \vee a = b$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

end

context *linorder*

begin

lemma *nle-le*: $(\neg a \leq b) \longleftrightarrow b \leq a \wedge b \neq a$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

end

$\langle ML \rangle$

4.7 Bounded quantifiers

syntax (*ASCII*)

-*All-less* :: $[idt, 'a, bool] \Rightarrow bool$ $(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } ALL \rangle \rangle ALL$
 $-<= ./ -) \rangle [0, 0, 10] 10)$

-*Ex-less* :: $[idt, 'a, bool] \Rightarrow bool$ $(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } EX \rangle \rangle EX$
 $-<= ./ -) \rangle [0, 0, 10] 10)$

-*All-less-eq* :: $[idt, 'a, bool] \Rightarrow bool$ $(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } ALL \rangle \rangle ALL$
 $-<= ./ -) \rangle [0, 0, 10] 10)$

-*Ex-less-eq* :: $[idt, 'a, bool] \Rightarrow bool$ $(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } EX \rangle \rangle EX$
 $-<= ./ -) \rangle [0, 0, 10] 10)$

-*All-greater* :: $[idt, 'a, bool] \Rightarrow bool$ $(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } ALL \rangle \rangle ALL$
 $->= ./ -) \rangle [0, 0, 10] 10)$

-*Ex-greater* :: $[idt, 'a, bool] \Rightarrow bool$ $(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } EX \rangle \rangle EX$
 $->= ./ -) \rangle [0, 0, 10] 10)$

-*All-greater-eq* :: $[idt, 'a, bool] \Rightarrow bool$ $(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } ALL \rangle \rangle ALL$
 $->= ./ -) \rangle [0, 0, 10] 10)$

-*Ex-greater-eq* :: $[idt, 'a, bool] \Rightarrow bool$ $(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } EX \rangle \rangle EX$
 $->= ./ -) \rangle [0, 0, 10] 10)$

-*All-neq* :: $[idt, 'a, bool] \Rightarrow bool$ $(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } ALL \rangle \rangle ALL$
 $- \sim = ./ -) \rangle [0, 0, 10] 10)$

-*Ex-neq* :: $[idt, 'a, bool] \Rightarrow bool$ $(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } EX \rangle \rangle EX$
 $- \sim = ./ -) \rangle [0, 0, 10] 10)$

syntax

$-All-less :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ \forall \rangle \forall \ -<-./ \ -) \rangle [0, 0, 10] \ 10)$
 $-Ex-less :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ \exists \rangle \exists \ -<-./ \ -) \rangle [0, 0, 10] \ 10)$
 $-All-less-eq :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ \forall \rangle \forall \ -\leq-./ \ -) \rangle [0, 0, 10] \ 10)$
 $-Ex-less-eq :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ \exists \rangle \exists \ -\leq-./ \ -) \rangle [0, 0, 10] \ 10)$

 $-All-greater :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ \forall \rangle \forall \ ->-./ \ -) \rangle [0, 0, 10] \ 10)$
 $-Ex-greater :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ \exists \rangle \exists \ ->-./ \ -) \rangle [0, 0, 10] \ 10)$
 $-All-greater-eq :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ \forall \rangle \forall \ -\geq-./ \ -) \rangle [0, 0, 10] \ 10)$
 $-Ex-greater-eq :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ \exists \rangle \exists \ -\geq-./ \ -) \rangle [0, 0, 10] \ 10)$

 $-All-neq :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ \forall \rangle \forall \ -\neq-./ \ -) \rangle [0, 0, 10] \ 10)$
 $-Ex-neq :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ \exists \rangle \exists \ -\neq-./ \ -) \rangle [0, 0, 10] \ 10)$

syntax (input)

$-All-less :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ ! \rangle ! \ -<-./ \ -) \rangle [0, 0, 10] \ 10)$
 $-Ex-less :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ ? \rangle ? \ -<-./ \ -) \rangle [0, 0, 10] \ 10)$
 $-All-less-eq :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ ! \rangle ! \ -<= -./ \ -) \rangle [0, 0, 10] \ 10)$
 $-Ex-less-eq :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ ? \rangle ? \ -<= -./ \ -) \rangle [0, 0, 10] \ 10)$
 $-All-neq :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ ! \rangle ! \ \sim = -./ \ -) \rangle [0, 0, 10] \ 10)$
 $-Ex-neq :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \ ? \rangle ? \ \sim = -./ \ -) \rangle [0, 0, 10] \ 10)$

syntax-consts

$-All-less \ -All-less-eq \ -All-greater \ -All-greater-eq \ -All-neq \Leftrightarrow All$ **and**
 $-Ex-less \ -Ex-less-eq \ -Ex-greater \ -Ex-greater-eq \ -Ex-neq \Leftrightarrow Ex$

translations

$\forall x < y. P \rightarrow \forall x. x < y \rightarrow P$
 $\exists x < y. P \rightarrow \exists x. x < y \wedge P$
 $\forall x \leq y. P \rightarrow \forall x. x \leq y \rightarrow P$
 $\exists x \leq y. P \rightarrow \exists x. x \leq y \wedge P$
 $\forall x > y. P \rightarrow \forall x. x > y \rightarrow P$
 $\exists x > y. P \rightarrow \exists x. x > y \wedge P$

$$\begin{aligned}
& \forall x \geq y. P \rightarrow \forall x. x \geq y \rightarrow P \\
& \exists x \geq y. P \rightarrow \exists x. x \geq y \wedge P \\
& \forall x \neq y. P \rightarrow \forall x. x \neq y \rightarrow P \\
& \exists x \neq y. P \rightarrow \exists x. x \neq y \wedge P
\end{aligned}$$

$\langle ML \rangle$

4.8 Transitivity reasoning

context *ord*

begin

lemma *ord-le-eq-trans*: $a \leq b \implies b = c \implies a \leq c$
 $\langle proof \rangle$

lemma *ord-eq-le-trans*: $a = b \implies b \leq c \implies a \leq c$
 $\langle proof \rangle$

lemma *ord-less-eq-trans*: $a < b \implies b = c \implies a < c$
 $\langle proof \rangle$

lemma *ord-eq-less-trans*: $a = b \implies b < c \implies a < c$
 $\langle proof \rangle$

end

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

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

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

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

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

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

lemma *order-subst1*: $(a::'a::order) \leq f b \implies (b::'b::order) \leq c \implies$
 $(!!x y. x \leq y \implies f x \leq f y) \implies a \leq f c$
 $\langle proof \rangle$

lemma *order-subst2*: $(a::'a::order) \leq b \implies f b \leq (c::'c::order) \implies$
 $(!!x y. x \leq y \implies f x \leq f y) \implies f a \leq c$
 $\langle proof \rangle$

lemma *ord-le-eq-subst*: $a \leq b \implies f b = c \implies$
 $(!!x y. x \leq y \implies f x \leq f y) \implies f a \leq c$
 $\langle proof \rangle$

lemma *ord-eq-le-subst*: $a = f b \implies b \leq c \implies$
 $(!!x y. x \leq y \implies f x \leq f y) \implies a \leq f c$
 $\langle proof \rangle$

lemma *ord-less-eq-subst*: $a < b \implies f b = c \implies$
 $(!!x y. x < y \implies f x < f y) \implies f a < c$
 $\langle proof \rangle$

lemma *ord-eq-less-subst*: $a = f b \implies b < c \implies$
 $(!!x y. x < y \implies f x < f y) \implies a < f c$
 $\langle proof \rangle$

Note that this list of rules is in reverse order of priorities.

lemmas [*trans*] =
order-less-subst2
order-less-subst1
order-le-less-subst2
order-le-less-subst1
order-less-le-subst2
order-less-le-subst1
order-subst2
order-subst1
ord-le-eq-subst
ord-eq-le-subst
ord-less-eq-subst
ord-eq-less-subst
forw-subst
back-subst
rev-mp
mp

lemmas (**in** *order*) [*trans*] =
neq-le-trans
le-neq-trans

lemmas (**in** *preorder*) [*trans*] =

less-trans
less-asym'
le-less-trans
less-le-trans
order-trans

lemmas (in *order*) [*trans*] =
order.antisym

lemmas (in *ord*) [*trans*] =
ord-le-eq-trans
ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans

lemmas [*trans*] =
trans

lemmas *order-trans-rules* =
order-less-subst2
order-less-subst1
order-le-less-subst2
order-le-less-subst1
order-less-le-subst2
order-less-le-subst1
order-subst2
order-subst1
ord-le-eq-subst
ord-eq-le-subst
ord-less-eq-subst
ord-eq-less-subst
forw-subst
back-subst
rev-mp
mp
neq-le-trans
le-neq-trans
less-trans
less-asym'
le-less-trans
less-le-trans
order-trans
order.antisym
ord-le-eq-trans
ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans
trans

These support proving chains of decreasing inequalities $a \geq b \geq c \dots$ in Isar

proofs.

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

$$\begin{aligned}
 & a = b \implies b > c \implies a > c \\
 & a > b \implies b = c \implies a > c \\
 & a = b \implies b \geq c \implies a \geq c \\
 & a \geq b \implies b = c \implies a \geq c \\
 & (x::'a::order) \geq y \implies y \geq x \implies x = y \\
 & (x::'a::order) \geq y \implies y \geq z \implies x \geq z \\
 & (x::'a::order) > y \implies y \geq z \implies x > z \\
 & (x::'a::order) \geq y \implies y > z \implies x > z \\
 & (a::'a::order) > b \implies b > a \implies P \\
 & (x::'a::order) > y \implies y > z \implies x > z \\
 & (a::'a::order) \geq b \implies a \neq b \implies a > b \\
 & (a::'a::order) \neq b \implies a \geq b \implies a > b \\
 & a = f b \implies b > c \implies (\bigwedge x y. x > y \implies f x > f y) \implies a > f c \\
 & a > b \implies f b = c \implies (\bigwedge x y. x > y \implies f x > f y) \implies f a > c \\
 & a = f b \implies b \geq c \implies (\bigwedge x y. x \geq y \implies f x \geq f y) \implies a \geq f c \\
 & a \geq b \implies f b = c \implies (\bigwedge x y. x \geq y \implies f x \geq f y) \implies f a \geq c \\
 & \langle proof \rangle
 \end{aligned}$$

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

$$\begin{aligned}
 & \text{assumes } (a::'a::order) \geq f b \\
 & \quad \text{and } b \geq c \\
 & \quad \text{and } \bigwedge x y. x \geq y \implies f x \geq f y \\
 & \text{shows } a \geq f c \\
 & \langle proof \rangle
 \end{aligned}$$

lemma *xt3* [*no-atp*]:

$$\begin{aligned}
 & \text{assumes } (a::'a::order) \geq b \\
 & \quad \text{and } (f b::'b::order) \geq c \\
 & \quad \text{and } \bigwedge x y. x \geq y \implies f x \geq f y \\
 & \text{shows } f a \geq c \\
 & \langle proof \rangle
 \end{aligned}$$

lemma *xt4* [*no-atp*]:

$$\begin{aligned}
 & \text{assumes } (a::'a::order) > f b \\
 & \quad \text{and } (b::'b::order) \geq c \\
 & \quad \text{and } \bigwedge x y. x \geq y \implies f x \geq f y \\
 & \text{shows } a > f c \\
 & \langle proof \rangle
 \end{aligned}$$

lemma *xt5* [*no-atp*]:

$$\begin{aligned}
 & \text{assumes } (a::'a::order) > b \\
 & \quad \text{and } (f b::'b::order) \geq c \\
 & \quad \text{and } \bigwedge x y. x > y \implies f x > f y \\
 & \text{shows } f a > c \\
 & \langle proof \rangle
 \end{aligned}$$

lemma *xt6* [*no-atp*]:

assumes $(a::'a::order) \geq f\ b$
and $b > c$
and $\bigwedge x\ y. x > y \implies f\ x > f\ y$
shows $a > f\ c$
 $\langle proof \rangle$

lemma *xt7* [*no-atp*]:
assumes $(a::'a::order) \geq b$
and $(f\ b::'b::order) > c$
and $\bigwedge x\ y. x \geq y \implies f\ x \geq f\ y$
shows $f\ a > c$
 $\langle proof \rangle$

lemma *xt8* [*no-atp*]:
assumes $(a::'a::order) > f\ b$
and $(b::'b::order) > c$
and $\bigwedge x\ y. x > y \implies f\ x > f\ y$
shows $a > f\ c$
 $\langle proof \rangle$

lemma *xt9* [*no-atp*]:
assumes $(a::'a::order) > b$
and $(f\ b::'b::order) > c$
and $\bigwedge x\ y. x > y \implies f\ x > f\ y$
shows $f\ a > c$
 $\langle proof \rangle$

lemmas *xtrans* = *xt1 xt2 xt3 xt4 xt5 xt6 xt7 xt8 xt9*

4.9 min and max – fundamental

definition (*in ord*) *min* :: $'a \Rightarrow 'a \Rightarrow 'a$ **where**
 $min\ a\ b = (if\ a \leq b\ then\ a\ else\ b)$

definition (*in ord*) *max* :: $'a \Rightarrow 'a \Rightarrow 'a$ **where**
 $max\ a\ b = (if\ a \leq b\ then\ b\ else\ a)$

lemma *min-absorb1*: $x \leq y \implies min\ x\ y = x$
 $\langle proof \rangle$

lemma *max-absorb2*: $x \leq y \implies max\ x\ y = y$
 $\langle proof \rangle$

lemma *min-absorb2*: $(y::'a::order) \leq x \implies min\ x\ y = y$
 $\langle proof \rangle$

lemma *max-absorb1*: $(y::'a::order) \leq x \implies max\ x\ y = x$
 $\langle proof \rangle$


```

lemma max-min-same [simp]:
  fixes  $x\ y :: 'a :: \text{linorder}$ 
  shows  $\max\ x\ (\min\ x\ y) = x\ \max\ (\min\ x\ y)\ x = x\ \max\ (\min\ x\ y)\ y = y\ \max\ y\ (\min\ x\ y) = y$ 
   $\langle \text{proof} \rangle$ 

```

4.10 (Unique) top and bottom elements

```

class bot =
  fixes  $\text{bot} :: 'a\ (\lrcorner\ \bot)$ 

```

```

class order-bot = order + bot +
  assumes bot-least:  $\bot \leq a$ 
begin

```

```

sublocale bot: ordering-top greater-eq greater bot
   $\langle \text{proof} \rangle$ 

```

```

lemma le-bot:
   $a \leq \bot \implies a = \bot$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma bot-unique:
   $a \leq \bot \longleftrightarrow a = \bot$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma not-less-bot:
   $\neg a < \bot$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma bot-less:
   $a \neq \bot \longleftrightarrow \bot < a$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma max-bot[simp]:  $\max\ \text{bot}\ x = x$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma max-bot2[simp]:  $\max\ x\ \text{bot} = x$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma min-bot[simp]:  $\min\ \text{bot}\ x = \text{bot}$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma min-bot2[simp]:  $\min\ x\ \text{bot} = \text{bot}$ 
   $\langle \text{proof} \rangle$ 

```

```

end

```

```

class top =

```

```

fixes top :: 'a (⋄⊤⋄)

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

sublocale top: ordering-top less-eq less top
  ⟨proof⟩

lemma top-le:
   $\top \leq a \implies a = \top$ 
  ⟨proof⟩

lemma top-unique:
   $\top \leq a \iff a = \top$ 
  ⟨proof⟩

lemma not-top-less:
   $\neg \top < a$ 
  ⟨proof⟩

lemma less-top:
   $a \neq \top \iff a < \top$ 
  ⟨proof⟩

lemma max-top[simp]:  $\max \top x = \top$ 
  ⟨proof⟩

lemma max-top2[simp]:  $\max x \top = \top$ 
  ⟨proof⟩

lemma min-top[simp]:  $\min \top x = x$ 
  ⟨proof⟩

lemma min-top2[simp]:  $\min x \top = x$ 
  ⟨proof⟩

end

```

4.11 Dense orders

```

class dense-order = order +
  assumes dense:  $x < y \implies (\exists z. x < z \wedge z < y)$ 

class dense-linorder = linorder + dense-order
begin

lemma dense-le:
  fixes y z :: 'a

```

```

assumes  $\bigwedge x. x < y \implies x \leq z$ 
shows  $y \leq z$ 
 $\langle proof \rangle$ 

```

```

lemma dense-le-bounded:
  fixes  $x\ y\ z :: 'a$ 
  assumes  $x < y$ 
  assumes *:  $\bigwedge w. \llbracket x < w ; w < y \rrbracket \implies w \leq z$ 
  shows  $y \leq z$ 
 $\langle proof \rangle$ 

```

```

lemma dense-ge:
  fixes  $y\ z :: 'a$ 
  assumes  $\bigwedge x. z < x \implies y \leq x$ 
  shows  $y \leq z$ 
 $\langle proof \rangle$ 

```

```

lemma dense-ge-bounded:
  fixes  $x\ y\ z :: 'a$ 
  assumes  $z < x$ 
  assumes *:  $\bigwedge w. \llbracket z < w ; w < x \rrbracket \implies y \leq w$ 
  shows  $y \leq z$ 
 $\langle proof \rangle$ 

```

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

```

```

class unbounded-dense-order = dense-order + no-top + no-bot

```

```

instance unbounded-dense-linorder  $\subseteq$  unbounded-dense-order  $\langle proof \rangle$ 

```

4.12 Wellorders

```

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

```

```

lemma wellorder-Least-lemma:
  fixes  $k :: 'a$ 
  assumes  $P\ k$ 
  shows LeastI:  $P\ (\text{LEAST } x. P\ x)$  and Least-le:  $(\text{LEAST } x. P\ x) \leq k$ 

```

$\langle proof \rangle$

lemma *LeastI-ex*: $\exists x. P x \implies P (Least P)$

$\langle proof \rangle$

lemma *LeastI2*:

$P a \implies (\bigwedge x. P x \implies Q x) \implies Q (Least P)$

$\langle proof \rangle$

lemma *LeastI2-ex*:

$\exists a. P a \implies (\bigwedge x. P x \implies Q x) \implies Q (Least P)$

$\langle proof \rangle$

lemma *LeastI2-wellorder*:

assumes $P a$

and $\bigwedge a. \llbracket P a; \forall b. P b \longrightarrow a \leq b \rrbracket \implies Q a$

shows $Q (Least P)$

$\langle proof \rangle$

lemma *LeastI2-wellorder-ex*:

assumes $\exists x. P x$

and $\bigwedge a. \llbracket P a; \forall b. P b \longrightarrow a \leq b \rrbracket \implies Q a$

shows $Q (Least P)$

$\langle proof \rangle$

lemma *not-less-Least*: $k < (LEAST x. P x) \implies \neg P k$

$\langle proof \rangle$

lemma *exists-least-iff*: $(\exists n. P n) \longleftrightarrow (\exists n. P n \wedge (\forall m < n. \neg P m))$ (**is** *?lhs*

$\longleftrightarrow ?rhs$)

$\langle proof \rangle$

lemma *exists-least-iff'*:

shows $(\exists n. P n) \longleftrightarrow P (Least P) \wedge (\forall m < (Least P). \neg P m)$

$\langle proof \rangle$

end

4.13 Order on *bool*

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

begin

definition

le-bool-def [*simp*]: $P \leq Q \longleftrightarrow P \longrightarrow Q$

definition

[*simp*]: $(P::bool) < Q \longleftrightarrow \neg P \wedge Q$

definition

$[simp]: \perp \longleftrightarrow False$

definition

$[simp]: \top \longleftrightarrow True$

instance $\langle proof \rangle$

end

lemma *le-boolI*: $(P \implies Q) \implies P \leq Q$
 $\langle proof \rangle$

lemma *le-boolI'*: $P \longrightarrow Q \implies P \leq Q$
 $\langle proof \rangle$

lemma *le-boolE*: $P \leq Q \implies P \implies (Q \implies R) \implies R$
 $\langle proof \rangle$

lemma *le-boolD*: $P \leq Q \implies P \longrightarrow Q$
 $\langle proof \rangle$

lemma *bot-boolE*: $\perp \implies P$
 $\langle proof \rangle$

lemma *top-boolI*: \top
 $\langle proof \rangle$

lemma *[code]*:
 $False \leq b \longleftrightarrow True$
 $True \leq b \longleftrightarrow b$
 $False < b \longleftrightarrow b$
 $True < b \longleftrightarrow False$
 $\langle proof \rangle$

4.14 Order on $- \Rightarrow -$

instantiation *fun* :: $(type, ord) \rightarrow ord$
begin

definition

le-fun-def: $f \leq g \longleftrightarrow (\forall x. f\ x \leq g\ x)$

definition

$(f :: 'a \Rightarrow 'b) < g \longleftrightarrow f \leq g \wedge \neg (g \leq f)$

instance $\langle proof \rangle$

end

instance *fun* :: (*type*, *preorder*) *preorder* $\langle proof \rangle$

instance *fun* :: (*type*, *order*) *order* $\langle proof \rangle$

instantiation *fun* :: (*type*, *bot*) *bot*
begin

definition

$\perp = (\lambda x. \perp)$

instance $\langle proof \rangle$

end

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

lemma *bot-apply* [*simp*, *code*]:

$\perp x = \perp$

$\langle proof \rangle$

instance $\langle proof \rangle$

end

instantiation *fun* :: (*type*, *top*) *top*
begin

definition

[*no-atp*]: $\top = (\lambda x. \top)$

instance $\langle proof \rangle$

end

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

lemma *top-apply* [*simp*, *code*]:

$\top x = \top$

$\langle proof \rangle$

instance $\langle proof \rangle$

end

lemma *le-funI*: $(\bigwedge x. f x \leq g x) \implies f \leq g$
 $\langle proof \rangle$

lemma *le-funE*: $f \leq g \implies (f\ x \leq g\ x \implies P) \implies P$
 $\langle proof \rangle$

lemma *le-funD*: $f \leq g \implies f\ x \leq g\ x$
 $\langle proof \rangle$

4.15 Order on unary and binary predicates

lemma *predicate1I*:
assumes $PQ: \bigwedge x. P\ x \implies Q\ x$
shows $P \leq Q$
 $\langle proof \rangle$

lemma *predicate1D*:
 $P \leq Q \implies P\ x \implies Q\ x$
 $\langle proof \rangle$

lemma *rev-predicate1D*:
 $P\ x \implies P \leq Q \implies Q\ x$
 $\langle proof \rangle$

lemma *predicate2I*:
assumes $PQ: \bigwedge x\ y. P\ x\ y \implies Q\ x\ y$
shows $P \leq Q$
 $\langle proof \rangle$

lemma *predicate2D*:
 $P \leq Q \implies P\ x\ y \implies Q\ x\ y$
 $\langle proof \rangle$

lemma *rev-predicate2D*:
 $P\ x\ y \implies P \leq Q \implies Q\ x\ y$
 $\langle proof \rangle$

lemma *bot1E* [*no-atp*]: $\bot\ x \implies P$
 $\langle proof \rangle$

lemma *bot2E*: $\bot\ x\ y \implies P$
 $\langle proof \rangle$

lemma *top1I*: $\top\ x$
 $\langle proof \rangle$

lemma *top2I*: $\top\ x\ y$
 $\langle proof \rangle$

4.16 Name duplicates

lemmas *antisym* = *order.antisym*

lemmas *eq-iff* = *order.eq-iff*

```

lemmas order-eq-refl = preorder-class.eq-refl
lemmas order-less-irrefl = preorder-class.less-irrefl
lemmas order-less-imp-le = preorder-class.less-imp-le
lemmas order-less-not-sym = preorder-class.less-not-sym
lemmas order-less-asymp = preorder-class.less-asymp
lemmas order-less-trans = preorder-class.less-trans
lemmas order-le-less-trans = preorder-class.le-less-trans
lemmas order-less-le-trans = preorder-class.less-le-trans
lemmas order-less-imp-not-less = preorder-class.less-imp-not-less
lemmas order-less-imp-triv = preorder-class.less-imp-triv
lemmas order-less-asymp' = preorder-class.less-asymp'

lemmas order-less-le = order-class.less-le
lemmas order-le-less = order-class.le-less
lemmas order-le-imp-less-or-eq = order-class.le-imp-less-or-eq
lemmas order-less-imp-not-eq = order-class.less-imp-not-eq
lemmas order-less-imp-not-eq2 = order-class.less-imp-not-eq2
lemmas order-neq-le-trans = order-class.neq-le-trans
lemmas order-le-neq-trans = order-class.le-neq-trans
lemmas order-eq-iff = order-class.order.eq-iff
lemmas order-antisym-conv = order-class.antisym-conv

lemmas linorder-linear = linorder-class.linear
lemmas linorder-less-linear = linorder-class.less-linear
lemmas linorder-le-less-linear = linorder-class.le-less-linear
lemmas linorder-le-cases = linorder-class.le-cases
lemmas linorder-not-less = linorder-class.not-less
lemmas linorder-not-le = linorder-class.not-le
lemmas linorder-neq-iff = 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 $f :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\langle * \rangle$ 70)

assumes *assoc* [*ac-simps*]: $a * b * c = a * (b * c)$

locale *abel-semigroup* = *semigroup* +

assumes *commute* [*ac-simps*]: $a * b = b * a$

begin

lemma *left-commute* [*ac-simps*]: $b * (a * c) = a * (b * c)$

$\langle proof \rangle$

end

locale *monoid* = *semigroup* +

fixes $z :: 'a$ ($\langle 1 \rangle$)

assumes *left-neutral* [*simp*]: $1 * a = a$

assumes *right-neutral* [*simp*]: $a * 1 = a$

locale *comm-monoid* = *abel-semigroup* +

fixes $z :: 'a$ ($\langle 1 \rangle$)

assumes *comm-neutral*: $a * 1 = a$

begin

sublocale *monoid*

$\langle proof \rangle$

end

locale *group* = *semigroup* +
fixes *z* :: 'a ($\langle 1 \rangle$)
fixes *inverse* :: 'a \Rightarrow 'a
assumes *group-left-neutral*: $1 * a = a$
assumes *left-inverse [simp]*: $inverse\ a * a = 1$
begin

lemma *left-cancel*: $a * b = a * c \longleftrightarrow b = c$
 $\langle proof \rangle$

sublocale *monoid*
 $\langle proof \rangle$

lemma *inverse-unique*:
assumes $a * b = 1$
shows $inverse\ a = b$
 $\langle proof \rangle$

lemma *inverse-neutral [simp]*: $inverse\ 1 = 1$
 $\langle proof \rangle$

lemma *inverse-inverse [simp]*: $inverse\ (inverse\ a) = a$
 $\langle proof \rangle$

lemma *right-inverse [simp]*: $a * inverse\ a = 1$
 $\langle proof \rangle$

lemma *inverse-distrib-swap*: $inverse\ (a * b) = inverse\ b * inverse\ a$
 $\langle proof \rangle$

lemma *right-cancel*: $b * a = c * a \longleftrightarrow b = c$
 $\langle proof \rangle$

end

5.3 Generic operations

class *zero* =
fixes *zero* :: 'a ($\langle 0 \rangle$)

class *one* =
fixes *one* :: 'a ($\langle 1 \rangle$)

hide-const (**open**) *zero one*

lemma *Let-0 [simp]*: $Let\ 0\ f = f\ 0$
 $\langle proof \rangle$

lemma *Let-1* [*simp*]: *Let* $1\ f = f\ 1$

<proof>

<ML>

class *plus* =

fixes *plus* :: '*a* ⇒ '*a* ⇒ '*a* (**infixl** <+> 65)

class *minus* =

fixes *minus* :: '*a* ⇒ '*a* ⇒ '*a* (**infixl** <-> 65)

class *uminus* =

fixes *uminus* :: '*a* ⇒ '*a* (*<(open-block notation=prefix ->-)>* [81] 80)

class *times* =

fixes *times* :: '*a* ⇒ '*a* ⇒ '*a* (**infixl** <*> 70)

bundle *uminus-syntax*

begin

notation *uminus* (*<(open-block notation=prefix ->-)>* [81] 80)

end

5.4 Semigroups and Monoids

class *semigroup-add* = *plus* +

assumes *add-assoc*: $(a + b) + c = a + (b + c)$

begin

sublocale *add*: *semigroup plus*

<proof>

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*

<proof>

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
  ⟨proof⟩

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

sublocale mult: ab-semigroup times
  ⟨proof⟩

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

sublocale add: monoid plus 0
  ⟨proof⟩

end

lemma zero-reorient:  $0 = x \longleftrightarrow x = 0$ 

```

```

    <proof>

class comm-monoid-add = zero + ab-semigroup-add +
    assumes add-0:  $0 + a = a$ 
begin

subclass monoid-add
    <proof>

sublocale add: comm-monoid plus 0
    <proof>

end

class monoid-mult = one + semigroup-mult +
    assumes mult-1-left:  $1 * a = a$ 
    and mult-1-right:  $a * 1 = a$ 
begin

sublocale mult: monoid times 1
    <proof>

end

lemma one-reorient:  $1 = x \longleftrightarrow x = 1$ 
    <proof>

class comm-monoid-mult = one + ab-semigroup-mult +
    assumes mult-1:  $1 * a = a$ 
begin

subclass monoid-mult
    <proof>

sublocale mult: comm-monoid times 1
    <proof>

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]:  $a + b = a + c \longleftrightarrow b = c$ 
    <proof>

lemma add-right-cancel [simp]:  $b + a = c + a \longleftrightarrow b = c$ 
    <proof>

```

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*]: $(a + b) - b = a$
 $\langle proof \rangle$

subclass *cancel-semigroup-add*
 $\langle proof \rangle$

lemma *add-diff-cancel-left* [*simp*]: $(c + a) - (c + b) = a - b$
 $\langle proof \rangle$

lemma *add-diff-cancel-right* [*simp*]: $(a + c) - (b + c) = a - b$
 $\langle proof \rangle$

lemma *diff-right-commute*: $a - c - b = a - b - c$
 $\langle proof \rangle$

end

class *cancel-comm-monoid-add* = *cancel-ab-semigroup-add* + *comm-monoid-add*
begin

lemma *diff-zero* [*simp*]: $a - 0 = a$
 $\langle proof \rangle$

lemma *diff-cancel* [*simp*]: $a - a = 0$
 $\langle proof \rangle$

lemma *add-implies-diff*:
assumes $c + b = a$
shows $c = a - b$
 $\langle proof \rangle$

lemma *add-cancel-right-right* [*simp*]: $a = a + b \longleftrightarrow b = 0$
 $(\text{is } ?P \longleftrightarrow ?Q)$
 $\langle proof \rangle$

lemma *add-cancel-right-left* [*simp*]: $a = b + a \longleftrightarrow b = 0$
 $\langle proof \rangle$

lemma *add-cancel-left-right* [*simp*]: $a + b = a \longleftrightarrow b = 0$
 $\langle proof \rangle$

lemma *add-cancel-left-left* [*simp*]: $b + a = a \longleftrightarrow b = 0$
 $\langle \text{proof} \rangle$

end

class *comm-monoid-diff* = *cancel-comm-monoid-add* +
assumes *zero-diff* [*simp*]: $0 - a = 0$
begin

lemma *diff-add-zero* [*simp*]: $a - (a + b) = 0$
 $\langle \text{proof} \rangle$

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*: $a - b = a + (- b)$
 $\langle \text{proof} \rangle$

sublocale *add*: *group plus 0 uminus*
 $\langle \text{proof} \rangle$

lemma *minus-unique*: $a + b = 0 \implies - a = b$
 $\langle \text{proof} \rangle$

lemma *minus-zero*: $- 0 = 0$
 $\langle \text{proof} \rangle$

lemma *minus-minus*: $- (- a) = a$
 $\langle \text{proof} \rangle$

lemma *right-minus*: $a + - a = 0$
 $\langle \text{proof} \rangle$

lemma *diff-self* [*simp*]: $a - a = 0$
 $\langle \text{proof} \rangle$

subclass *cancel-semigroup-add*
 $\langle \text{proof} \rangle$

lemma *minus-add-cancel* [*simp*]: $- a + (a + b) = b$
 $\langle \text{proof} \rangle$

lemma *add-minus-cancel* [simp]: $a + (-a + b) = b$
 ⟨proof⟩

lemma *diff-add-cancel* [simp]: $a - b + b = a$
 ⟨proof⟩

lemma *add-diff-cancel* [simp]: $a + b - b = a$
 ⟨proof⟩

lemma *minus-add*: $-(a + b) = -b + -a$
 ⟨proof⟩

lemma *right-minus-eq* [simp]: $a - b = 0 \longleftrightarrow a = b$
 ⟨proof⟩

lemma *eq-iff-diff-eq-0*: $a = b \longleftrightarrow a - b = 0$
 ⟨proof⟩

lemma *diff-0* [simp]: $0 - a = -a$
 ⟨proof⟩

lemma *diff-0-right* [simp]: $a - 0 = a$
 ⟨proof⟩

lemma *diff-minus-eq-add* [simp]: $a - -b = a + b$
 ⟨proof⟩

lemma *neg-equal-iff-equal* [simp]: $-a = -b \longleftrightarrow a = b$
 ⟨proof⟩

lemma *neg-equal-0-iff-equal* [simp]: $-a = 0 \longleftrightarrow a = 0$
 ⟨proof⟩

lemma *neg-0-equal-iff-equal* [simp]: $0 = -a \longleftrightarrow 0 = a$
 ⟨proof⟩

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*: $a = -b \longleftrightarrow b = -a$
 ⟨proof⟩

lemma *minus-equation-iff*: $-a = b \longleftrightarrow -b = a$
 ⟨proof⟩

lemma *eq-neg-iff-add-eq-0*: $a = -b \longleftrightarrow a + b = 0$
 ⟨proof⟩

lemma *add-eq-0-iff2*: $a + b = 0 \longleftrightarrow a = -b$
 ⟨proof⟩

lemma *neg-eq-iff-add-eq-0*: $- a = b \longleftrightarrow a + b = 0$
 ⟨*proof*⟩

lemma *add-eq-0-iff*: $a + b = 0 \longleftrightarrow b = - a$
 ⟨*proof*⟩

lemma *minus-diff-eq [simp]*: $-(a - b) = b - a$
 ⟨*proof*⟩

lemma *add-diff-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]*:
 $a + (b - c) = (a + b) - c$
 ⟨*proof*⟩

lemma *diff-add-eq-diff-diff-swap*: $a - (b + c) = a - c - b$
 ⟨*proof*⟩

lemma *diff-eq-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]*:
 $a - b = c \longleftrightarrow a = c + b$
 ⟨*proof*⟩

lemma *eq-diff-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]*:
 $a = c - b \longleftrightarrow a + b = c$
 ⟨*proof*⟩

lemma *diff-diff-eq2 [algebra-simps, algebra-split-simps, field-simps, field-split-simps]*:
 $a - (b - c) = (a + c) - b$
 ⟨*proof*⟩

lemma *diff-eq-diff-eq*: $a - b = c - d \implies a = b \longleftrightarrow c = d$
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

subclass *cancel-comm-monoid-add*
 ⟨*proof*⟩

lemma *uminus-add-conv-diff [simp]*: $- a + b = b - a$
 ⟨*proof*⟩

lemma *minus-add-distrib [simp]*: $-(a + b) = - a + - b$
 ⟨*proof*⟩

lemma *diff-add-eq* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps*]:

$$(a - b) + c = (a + c) - b$$

<proof>

lemma *minus-diff-commute*:

$$- b - a = - a - b$$

<proof>

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*: $a \leq b \implies a + c \leq b + c$
<proof>

non-strict, in both arguments

lemma *add-mono*: $a \leq b \implies c \leq d \implies a + c \leq b + d$
<proof>

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*: $a < b \implies c + a < c + b$

$\langle proof \rangle$

lemma *add-strict-right-mono*: $a < b \implies a + c < b + c$
 $\langle proof \rangle$

subclass *strict-ordered-ab-semigroup-add*
 $\langle proof \rangle$

lemma *add-less-le-mono*: $a < b \implies c \leq d \implies a + c < b + d$
 $\langle proof \rangle$

lemma *add-le-less-mono*: $a \leq b \implies c < d \implies a + c < b + d$
 $\langle proof \rangle$

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 $a < b$
 $\langle proof \rangle$

lemma *add-less-imp-less-right*: $a + c < b + c \implies a < b$
 $\langle proof \rangle$

lemma *add-less-cancel-left* [*simp*]: $c + a < c + b \longleftrightarrow a < b$
 $\langle proof \rangle$

lemma *add-less-cancel-right* [*simp*]: $a + c < b + c \longleftrightarrow a < b$
 $\langle proof \rangle$

lemma *add-le-cancel-left* [*simp*]: $c + a \leq c + b \longleftrightarrow a \leq b$
 $\langle proof \rangle$

lemma *add-le-cancel-right* [*simp*]: $a + c \leq b + c \longleftrightarrow a \leq b$
 $\langle proof \rangle$

lemma *add-le-imp-le-right*: $a + c \leq b + c \implies a \leq b$
 $\langle proof \rangle$

lemma *max-add-distrib-left*: $\max x y + z = \max (x + z) (y + z)$
 $\langle proof \rangle$

lemma *min-add-distrib-left*: $\min x y + z = \min (x + z) (y + z)$
 $\langle proof \rangle$

lemma *max-add-distrib-right*: $x + \max y z = \max (x + y) (x + z)$
 $\langle \text{proof} \rangle$

lemma *min-add-distrib-right*: $x + \min y z = \min (x + y) (x + z)$
 $\langle \text{proof} \rangle$

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]*: $0 \leq a \implies 0 \leq b \implies 0 \leq a + b$
 $\langle \text{proof} \rangle$

lemma *add-nonpos-nonpos*: $a \leq 0 \implies b \leq 0 \implies a + b \leq 0$
 $\langle \text{proof} \rangle$

lemma *add-nonneg-eq-0-iff*: $0 \leq x \implies 0 \leq y \implies x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
 $\langle \text{proof} \rangle$

lemma *add-nonpos-eq-0-iff*: $x \leq 0 \implies y \leq 0 \implies x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
 $\langle \text{proof} \rangle$

lemma *add-increasing*: $0 \leq a \implies b \leq c \implies b \leq a + c$
 $\langle \text{proof} \rangle$

lemma *add-increasing2*: $0 \leq c \implies b \leq a \implies b \leq a + c$
 $\langle \text{proof} \rangle$

lemma *add-decreasing*: $a \leq 0 \implies c \leq b \implies a + c \leq b$
 $\langle \text{proof} \rangle$

lemma *add-decreasing2*: $c \leq 0 \implies a \leq b \implies a + c \leq b$
 $\langle \text{proof} \rangle$

lemma *add-pos-nonneg*: $0 < a \implies 0 \leq b \implies 0 < a + b$
 $\langle \text{proof} \rangle$

lemma *add-pos-pos*: $0 < a \implies 0 < b \implies 0 < a + b$
 $\langle \text{proof} \rangle$

lemma *add-nonneg-pos*: $0 \leq a \implies 0 < b \implies 0 < a + b$
 $\langle \text{proof} \rangle$

lemma *add-neg-nonpos*: $a < 0 \implies b \leq 0 \implies a + b < 0$
 $\langle \text{proof} \rangle$

lemma *add-neg-neg*: $a < 0 \implies b < 0 \implies a + b < 0$
 ⟨proof⟩

lemma *add-nonpos-neg*: $a \leq 0 \implies b < 0 \implies a + b < 0$
 ⟨proof⟩

lemmas *add-sign-intros* =
add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonneg
add-neg-nonpos add-neg-neg add-nonpos-neg add-nonpos-nonpos

end

class *strict-ordered-comm-monoid-add* = *comm-monoid-add* + *strict-ordered-ab-semigroup-add*
begin

lemma *pos-add-strict*: $0 < a \implies b < c \implies b < a + c$
 ⟨proof⟩

end

class *ordered-cancel-comm-monoid-add* = *ordered-comm-monoid-add* + *cancel-ab-semigroup-add*
begin

subclass *ordered-cancel-ab-semigroup-add* ⟨proof⟩
subclass *strict-ordered-comm-monoid-add* ⟨proof⟩

lemma *add-strict-increasing*: $0 < a \implies b \leq c \implies b < a + c$
 ⟨proof⟩

lemma *add-strict-increasing2*: $0 \leq a \implies b < c \implies b < a + c$
 ⟨proof⟩

end

class *ordered-ab-semigroup-monoid-add-imp-le* = *monoid-add* + *ordered-ab-semigroup-add-imp-le*
begin

lemma *add-less-same-cancel1* [*simp*]: $b + a < b \longleftrightarrow a < 0$
 ⟨proof⟩

lemma *add-less-same-cancel2* [*simp*]: $a + b < b \longleftrightarrow a < 0$
 ⟨proof⟩

lemma *less-add-same-cancel1* [*simp*]: $a < a + b \longleftrightarrow 0 < b$
 ⟨proof⟩

lemma *less-add-same-cancel2* [*simp*]: $a < b + a \longleftrightarrow 0 < b$
 ⟨proof⟩

lemma *add-le-same-cancel1* [*simp*]: $b + a \leq b \longleftrightarrow a \leq 0$
 ⟨*proof*⟩

lemma *add-le-same-cancel2* [*simp*]: $a + b \leq b \longleftrightarrow a \leq 0$
 ⟨*proof*⟩

lemma *le-add-same-cancel1* [*simp*]: $a \leq a + b \longleftrightarrow 0 \leq b$
 ⟨*proof*⟩

lemma *le-add-same-cancel2* [*simp*]: $a \leq b + a \longleftrightarrow 0 \leq b$
 ⟨*proof*⟩

subclass *cancel-comm-monoid-add*
 ⟨*proof*⟩

subclass *ordered-cancel-comm-monoid-add*
 ⟨*proof*⟩

end

class *ordered-ab-group-add* = *ab-group-add* + *ordered-ab-semigroup-add*
begin

subclass *ordered-cancel-ab-semigroup-add* ⟨*proof*⟩

subclass *ordered-ab-semigroup-monoid-add-imp-le*
 ⟨*proof*⟩

lemma *max-diff-distrib-left*: $\max x y - z = \max (x - z) (y - z)$
 ⟨*proof*⟩

lemma *min-diff-distrib-left*: $\min x y - z = \min (x - z) (y - z)$
 ⟨*proof*⟩

lemma *le-imp-neg-le*:
assumes $a \leq b$
shows $-b \leq -a$
 ⟨*proof*⟩

lemma *neg-le-iff-le* [*simp*]: $-b \leq -a \longleftrightarrow a \leq b$
 ⟨*proof*⟩

lemma *neg-le-0-iff-le* [*simp*]: $-a \leq 0 \longleftrightarrow 0 \leq a$
 ⟨*proof*⟩

lemma *neg-0-le-iff-le* [*simp*]: $0 \leq -a \longleftrightarrow a \leq 0$
 ⟨*proof*⟩

lemma *neg-less-iff-less* [*simp*]: $-b < -a \longleftrightarrow a < b$

$\langle \text{proof} \rangle$

lemma *neg-less-0-iff-less* [simp]: $- a < 0 \longleftrightarrow 0 < a$
 $\langle \text{proof} \rangle$

lemma *neg-0-less-iff-less* [simp]: $0 < - a \longleftrightarrow a < 0$
 $\langle \text{proof} \rangle$

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $a < - b \longleftrightarrow b < - a$
 $\langle \text{proof} \rangle$

lemma *minus-less-iff*: $- a < b \longleftrightarrow - b < a$
 $\langle \text{proof} \rangle$

lemma *le-minus-iff*: $a \leq - b \longleftrightarrow b \leq - a$
 $\langle \text{proof} \rangle$

lemma *minus-le-iff*: $- a \leq b \longleftrightarrow - b \leq a$
 $\langle \text{proof} \rangle$

lemma *diff-less-0-iff-less* [simp]: $a - b < 0 \longleftrightarrow a < b$
 $\langle \text{proof} \rangle$

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]:
 $a - b < c \longleftrightarrow a < c + b$
 $\langle \text{proof} \rangle$

lemma *less-diff-eq* [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
 $a < c - b \longleftrightarrow a + b < c$
 $\langle \text{proof} \rangle$

lemma *diff-gt-0-iff-gt* [simp]: $a - b > 0 \longleftrightarrow a > b$
 $\langle \text{proof} \rangle$

lemma *diff-le-eq* [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
 $a - b \leq c \longleftrightarrow a \leq c + b$
 $\langle \text{proof} \rangle$

lemma *le-diff-eq* [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
 $a \leq c - b \longleftrightarrow a + b \leq c$
 $\langle \text{proof} \rangle$

lemma *diff-le-0-iff-le* [simp]: $a - b \leq 0 \longleftrightarrow a \leq b$
 $\langle \text{proof} \rangle$

lemmas *le-iff-diff-le-0* = *diff-le-0-iff-le* [symmetric]

lemma *diff-ge-0-iff-ge* [*simp*]: $a - b \geq 0 \longleftrightarrow a \geq b$
 ⟨*proof*⟩

lemma *diff-eq-diff-less*: $a - b = c - d \implies a < b \longleftrightarrow c < d$
 ⟨*proof*⟩

lemma *diff-eq-diff-less-eq*: $a - b = c - d \implies a \leq b \longleftrightarrow c \leq d$
 ⟨*proof*⟩

lemma *diff-mono*: $a \leq b \implies d \leq c \implies a - c \leq b - d$
 ⟨*proof*⟩

lemma *diff-left-mono*: $b \leq a \implies c - a \leq c - b$
 ⟨*proof*⟩

lemma *diff-right-mono*: $a \leq b \implies a - c \leq b - c$
 ⟨*proof*⟩

lemma *diff-strict-mono*: $a < b \implies d < c \implies a - c < b - d$
 ⟨*proof*⟩

lemma *diff-strict-left-mono*: $b < a \implies c - a < c - b$
 ⟨*proof*⟩

lemma *diff-strict-right-mono*: $a < b \implies a - c < b - c$
 ⟨*proof*⟩

end

locale *group-cancel*

begin

lemma *add1*: $(A::'a::comm-monoid-add) \equiv k + a \implies A + b \equiv k + (a + b)$
 ⟨*proof*⟩

lemma *add2*: $(B::'a::comm-monoid-add) \equiv k + b \implies a + B \equiv k + (a + b)$
 ⟨*proof*⟩

lemma *sub1*: $(A::'a::ab-group-add) \equiv k + a \implies A - b \equiv k + (a - b)$
 ⟨*proof*⟩

lemma *sub2*: $(B::'a::ab-group-add) \equiv k + b \implies a - B \equiv -k + (a - b)$
 ⟨*proof*⟩

lemma *neg1*: $(A::'a::ab-group-add) \equiv k + a \implies -A \equiv -k + -a$
 ⟨*proof*⟩

lemma *rule0*: $(a::'a::comm-monoid-add) \equiv a + 0$


```

    <proof>

end

<ML>

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 <proof>

subclass ordered-ab-semigroup-add-imp-le
  <proof>

end

class linordered-ab-group-add = linorder + ordered-ab-group-add
begin

subclass linordered-cancel-ab-semigroup-add <proof>

lemma equal-neg-zero [simp]:  $a = - a \longleftrightarrow a = 0$ 
  <proof>

lemma neg-equal-zero [simp]:  $- a = a \longleftrightarrow a = 0$ 
  <proof>

lemma neg-less-eq-nonneg [simp]:  $- a \leq a \longleftrightarrow 0 \leq a$ 
  <proof>

lemma neg-less-pos [simp]:  $- a < a \longleftrightarrow 0 < a$ 
  <proof>

lemma less-eq-neg-nonpos [simp]:  $a \leq - a \longleftrightarrow a \leq 0$ 
  <proof>

lemma less-neg-neg [simp]:  $a < - a \longleftrightarrow a < 0$ 
  <proof>

lemma double-zero [simp]:  $a + a = 0 \longleftrightarrow a = 0$ 
  <proof>

lemma double-zero-sym [simp]:  $0 = a + a \longleftrightarrow a = 0$ 
  <proof>

```

lemma *zero-less-double-add-iff-zero-less-single-add* [simp]: $0 < a + a \longleftrightarrow 0 < a$
 ⟨proof⟩

lemma *zero-le-double-add-iff-zero-le-single-add* [simp]: $0 \leq a + a \longleftrightarrow 0 \leq a$
 ⟨proof⟩

lemma *double-add-less-zero-iff-single-add-less-zero* [simp]: $a + a < 0 \longleftrightarrow a < 0$
 ⟨proof⟩

lemma *double-add-le-zero-iff-single-add-le-zero* [simp]: $a + a \leq 0 \longleftrightarrow a \leq 0$
 ⟨proof⟩

lemma *minus-max-eq-min*: $- \max x y = \min (-x) (-y)$
 ⟨proof⟩

lemma *minus-min-eq-max*: $- \min x y = \max (-x) (-y)$
 ⟨proof⟩

end

class *abs* =
 fixes *abs* :: 'a \Rightarrow 'a ($\langle \langle \text{open-block notation} = \langle \text{mixfix abs} \rangle \rangle | \cdot \rangle \rangle$)

bundle *abs-syntax*

begin

notation *abs* ($\langle \langle \text{open-block notation} = \langle \text{mixfix abs} \rangle \rangle | \cdot \rangle \rangle$)

end

class *sgn* =
 fixes *sgn* :: 'a \Rightarrow 'a

class *ordered-ab-group-add-abs* = *ordered-ab-group-add* + *abs* +
 assumes *abs-ge-zero* [simp]: $|a| \geq 0$
 and *abs-ge-self*: $a \leq |a|$
 and *abs-leI*: $a \leq b \implies -a \leq b \implies |a| \leq b$
 and *abs-minus-cancel* [simp]: $|-a| = |a|$
 and *abs-triangle-ineq*: $|a + b| \leq |a| + |b|$

begin

lemma *abs-minus-le-zero*: $-|a| \leq 0$
 ⟨proof⟩

lemma *abs-of-nonneg* [simp]:
 assumes *nonneg*: $0 \leq a$
 shows $|a| = a$
 ⟨proof⟩

lemma *abs-idempotent* [simp]: $||a|| = |a|$
 ⟨proof⟩

lemma *abs-eq-0* [*simp*]: $|a| = 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *abs-zero* [*simp*]: $|0| = 0$
 $\langle \text{proof} \rangle$

lemma *abs-0-eq* [*simp*]: $0 = |a| \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *abs-le-zero-iff* [*simp*]: $|a| \leq 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *abs-le-self-iff* [*simp*]: $|a| \leq a \longleftrightarrow 0 \leq a$
 $\langle \text{proof} \rangle$

lemma *zero-less-abs-iff* [*simp*]: $0 < |a| \longleftrightarrow a \neq 0$
 $\langle \text{proof} \rangle$

lemma *abs-not-less-zero* [*simp*]: $\neg |a| < 0$
 $\langle \text{proof} \rangle$

lemma *abs-ge-minus-self*: $-a \leq |a|$
 $\langle \text{proof} \rangle$

lemma *abs-minus-commute*: $|a - b| = |b - a|$
 $\langle \text{proof} \rangle$

lemma *abs-of-pos*: $0 < a \implies |a| = a$
 $\langle \text{proof} \rangle$

lemma *abs-of-nonpos* [*simp*]:
 assumes $a \leq 0$
 shows $|a| = -a$
 $\langle \text{proof} \rangle$

lemma *abs-of-neg*: $a < 0 \implies |a| = -a$
 $\langle \text{proof} \rangle$

lemma *abs-le-D1*: $|a| \leq b \implies a \leq b$
 $\langle \text{proof} \rangle$

lemma *abs-le-D2*: $|a| \leq b \implies -a \leq b$
 $\langle \text{proof} \rangle$

lemma *abs-le-iff*: $|a| \leq b \longleftrightarrow a \leq b \wedge -a \leq b$
 $\langle \text{proof} \rangle$

lemma *abs-triangle-ineq2*: $|a| - |b| \leq |a - b|$

$\langle proof \rangle$

lemma *abs-triangle-ineq2-sym*: $|a| - |b| \leq |b - a|$
 $\langle proof \rangle$

lemma *abs-triangle-ineq3*: $||a| - |b|| \leq |a - b|$
 $\langle proof \rangle$

lemma *abs-triangle-ineq4*: $|a - b| \leq |a| + |b|$
 $\langle proof \rangle$

lemma *abs-diff-triangle-ineq*: $|a + b - (c + d)| \leq |a - c| + |b - d|$
 $\langle proof \rangle$

lemma *abs-add-abs* [simp]: $||a| + |b|| = |a| + |b|$
 (is ?L = ?R)
 $\langle proof \rangle$

end

lemma *dense-eq0-I*:
 fixes $x::'a::\{dense-linorder,ordered-ab-group-add-abs\}$
 assumes $\bigwedge e. 0 < e \implies |x| \leq e$
 shows $x = 0$
 $\langle proof \rangle$

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*: $a \leq b \longleftrightarrow (\exists c. b = a + c)$
begin

lemma *zero-le*[simp]: $0 \leq x$
 $\langle proof \rangle$

lemma *le-zero-eq*[simp]: $n \leq 0 \longleftrightarrow n = 0$
 $\langle proof \rangle$

lemma *not-less-zero*[simp]: $\neg n < 0$
 $\langle proof \rangle$

lemma *zero-less-iff-neq-zero*: $0 < n \longleftrightarrow n \neq 0$
 ⟨proof⟩

This theorem is useful with *blast*

lemma *gr-zeroI*: $(n = 0 \implies \text{False}) \implies 0 < n$
 ⟨proof⟩

lemma *not-gr-zero[simp]*: $\neg 0 < n \longleftrightarrow n = 0$
 ⟨proof⟩

subclass *ordered-comm-monoid-add*
 ⟨proof⟩

lemma *gr-implies-not-zero*: $m < n \implies n \neq 0$
 ⟨proof⟩

lemma *add-eq-0-iff-both-eq-0[simp]*: $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
 ⟨proof⟩

lemma *zero-eq-add-iff-both-eq-0[simp]*: $0 = x + y \longleftrightarrow x = 0 \wedge y = 0$
 ⟨proof⟩

lemma *less-eqE*:
 assumes $\langle a \leq b \rangle$
 obtains c where $\langle b = a + c \rangle$
 ⟨proof⟩

lemma *lessE*:
 assumes $\langle a < b \rangle$
 obtains c where $\langle b = a + c \rangle$ and $\langle c \neq 0 \rangle$
 ⟨proof⟩

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*: $a + (b - a) = b$
 ⟨proof⟩

lemma *add-diff-assoc*: $c + (b - a) = c + b - a$
 ⟨proof⟩

lemma *add-diff-assoc2*: $b - a + c = b + c - a$
 ⟨proof⟩

lemma *diff-add-assoc*: $c + b - a = c + (b - a)$
 ⟨proof⟩

lemma *diff-add-assoc2*: $b + c - a = b - a + c$
 ⟨proof⟩

lemma *diff-diff-right*: $c - (b - a) = c + a - b$
 ⟨proof⟩

lemma *diff-add*: $b - a + a = b$
 ⟨proof⟩

lemma *le-add-diff*: $c \leq b + c - a$
 ⟨proof⟩

lemma *le-imp-diff-is-add*: $a \leq b \implies b - a = c \longleftrightarrow b = c + a$
 ⟨proof⟩

lemma *le-diff-conv2*: $c \leq b - a \longleftrightarrow c + a \leq b$
 (is $?P \longleftrightarrow ?Q$)
 ⟨proof⟩

end

end

5.9 Tools setup

lemma *add-mono-thms-linordered-semiring*:

fixes $i\ j\ k :: 'a::\text{ordered-ab-semigroup-add}$

shows $i \leq j \wedge k \leq l \implies i + k \leq j + l$

and $i = j \wedge k \leq l \implies i + k \leq j + l$

and $i \leq j \wedge k = l \implies i + k \leq j + l$

and $i = j \wedge k = l \implies i + k = j + l$

⟨proof⟩

lemma *add-mono-thms-linordered-field*:

fixes $i\ j\ k :: 'a::\text{ordered-cancel-ab-semigroup-add}$

shows $i < j \wedge k = l \implies i + k < j + l$

and $i = j \wedge k < l \implies i + k < j + l$

and $i < j \wedge k \leq l \implies i + k < j + l$

and $i \leq j \wedge k < l \implies i + k < j + l$

and $i < j \wedge k < l \implies i + k < j + l$

$\langle proof \rangle$

code-identifier

code-module *Groups* \multimap (*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*]: $a * (a * b) = a * b$

$\langle proof \rangle$

lemma *right-idem* [*simp*]: $(a * b) * b = a * b$

$\langle proof \rangle$

end

locale *semilattice-neutr* = *semilattice* + *comm-monoid*

locale *semilattice-order* = *semilattice* +

fixes *less-eq* :: $'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** $\langle \leq \rangle$ 50)

and *less* :: $'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** $\langle < \rangle$ 50)

assumes *order-iff*: $a \leq b \longleftrightarrow a = a * b$

and *strict-order-iff*: $a < b \longleftrightarrow a = a * b \wedge a \neq b$

begin

lemma *orderI*: $a = a * b \implies a \leq b$

$\langle proof \rangle$

lemma *orderE*:

assumes $a \leq b$

obtains $a = a * b$

$\langle proof \rangle$

sublocale *ordering* *less-eq* *less*

$\langle proof \rangle$

lemma *cobounded1* [simp]: $a * b \leq a$
 $\langle proof \rangle$

lemma *cobounded2* [simp]: $a * b \leq b$
 $\langle proof \rangle$

lemma *boundedI*:
 assumes $a \leq b$ and $a \leq c$
 shows $a \leq b * c$
 $\langle proof \rangle$

lemma *boundedE*:
 assumes $a \leq b * c$
 obtains $a \leq b$ and $a \leq c$
 $\langle proof \rangle$

lemma *bounded-iff* [simp]: $a \leq b * c \longleftrightarrow a \leq b \wedge a \leq c$
 $\langle proof \rangle$

lemma *strict-boundedE*:
 assumes $a < b * c$
 obtains $a < b$ and $a < c$
 $\langle proof \rangle$

lemma *coboundedI1*: $a \leq c \implies a * b \leq c$
 $\langle proof \rangle$

lemma *coboundedI2*: $b \leq c \implies a * b \leq c$
 $\langle proof \rangle$

lemma *strict-coboundedI1*: $a < c \implies a * b < c$
 $\langle proof \rangle$

lemma *strict-coboundedI2*: $b < c \implies a * b < c$
 $\langle proof \rangle$

lemma *mono*: $a \leq c \implies b \leq d \implies a * b \leq c * d$
 $\langle proof \rangle$

lemma *absorb1*: $a \leq b \implies a * b = a$
 $\langle proof \rangle$

lemma *absorb2*: $b \leq a \implies a * b = b$
 $\langle proof \rangle$

lemma *absorb3*: $a < b \implies a * b = a$
 $\langle proof \rangle$

lemma *absorb4*: $b < a \implies a * b = b$
 $\langle \text{proof} \rangle$

lemma *absorb-iff1*: $a \leq b \longleftrightarrow a * b = a$
 $\langle \text{proof} \rangle$

lemma *absorb-iff2*: $b \leq a \longleftrightarrow a * b = b$
 $\langle \text{proof} \rangle$

end

locale *semilattice-neutr-order* = *semilattice-neutr* + *semilattice-order*
begin

sublocale *ordering-top less-eq less 1*
 $\langle \text{proof} \rangle$

lemma *eq-neutr-iff* [simp]: $\langle a * b = 1 \longleftrightarrow a = 1 \wedge b = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *neutr-eq-iff* [simp]: $\langle 1 = a * b \longleftrightarrow a = 1 \wedge b = 1 \rangle$
 $\langle \text{proof} \rangle$

end

Interpretations for boolean operators

interpretation *conj*: *semilattice-neutr* $\langle (\wedge) \rangle$ *True*
 $\langle \text{proof} \rangle$

interpretation *disj*: *semilattice-neutr* $\langle (\vee) \rangle$ *False*
 $\langle \text{proof} \rangle$

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** $\langle \sqcap \rangle$ 70)

class *sup* =
fixes *sup* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\langle \sqcup \rangle$ 65)

6.3 Concrete lattices

class *semilattice-inf* = *order* + *inf* +
assumes *inf-le1* [simp]: $x \sqcap y \leq x$
and *inf-le2* [simp]: $x \sqcap y \leq y$

```

and inf-greatest:  $x \leq y \implies x \leq z \implies x \leq y \sqcap z$ 

class semilattice-sup = order + sup +
  assumes sup-ge1 [simp]:  $x \leq x \sqcup y$ 
  and sup-ge2 [simp]:  $y \leq x \sqcup y$ 
  and sup-least:  $y \leq x \implies z \leq x \implies y \sqcup z \leq x$ 
begin

Dual lattice.

lemma dual-semilattice: class.semilattice-inf sup greater-eq greater
  <proof>

end

class lattice = semilattice-inf + semilattice-sup

6.3.1 Intro and elim rules

context semilattice-inf
begin

lemma le-infI1:  $a \leq x \implies a \sqcap b \leq x$ 
  <proof>

lemma le-infI2:  $b \leq x \implies a \sqcap b \leq x$ 
  <proof>

lemma le-infI:  $x \leq a \implies x \leq b \implies x \leq a \sqcap b$ 
  <proof>

lemma le-infE:  $x \leq a \sqcap b \implies (x \leq a \implies x \leq b \implies P) \implies P$ 
  <proof>

lemma le-inf-iff:  $x \leq y \sqcap z \iff x \leq y \wedge x \leq z$ 
  <proof>

lemma le-iff-inf:  $x \leq y \iff x \sqcap y = x$ 
  <proof>

lemma inf-mono:  $a \leq c \implies b \leq d \implies a \sqcap b \leq c \sqcap d$ 
  <proof>

end

context semilattice-sup
begin

lemma le-supI1:  $x \leq a \implies x \leq a \sqcup b$ 
  <proof>

```

lemma *le-supI2*: $x \leq b \implies x \leq a \sqcup b$
 ⟨proof⟩

lemma *le-supI*: $a \leq x \implies b \leq x \implies a \sqcup b \leq x$
 ⟨proof⟩

lemma *le-supE*: $a \sqcup b \leq x \implies (a \leq x \implies b \leq x \implies P) \implies P$
 ⟨proof⟩

lemma *le-sup-iff*: $x \sqcup y \leq z \iff x \leq z \wedge y \leq z$
 ⟨proof⟩

lemma *le-iff-sup*: $x \leq y \iff x \sqcup y = y$
 ⟨proof⟩

lemma *sup-mono*: $a \leq c \implies b \leq d \implies a \sqcup b \leq c \sqcup d$
 ⟨proof⟩

end

6.3.2 Equational laws

context *semilattice-inf*
begin

sublocale *inf*: *semilattice inf*
 ⟨proof⟩

sublocale *inf*: *semilattice-order inf less-eq less*
 ⟨proof⟩

lemma *inf-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
 ⟨proof⟩

lemma *inf-commute*: $(x \sqcap y) = (y \sqcap x)$
 ⟨proof⟩

lemma *inf-left-commute*: $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$
 ⟨proof⟩

lemma *inf-idem*: $x \sqcap x = x$
 ⟨proof⟩

lemma *inf-left-idem*: $x \sqcap (x \sqcap y) = x \sqcap y$
 ⟨proof⟩

lemma *inf-right-idem*: $(x \sqcap y) \sqcap y = x \sqcap y$
 ⟨proof⟩

lemma *inf-absorb1*: $x \leq y \implies x \sqcap y = x$
 ⟨*proof*⟩

lemma *inf-absorb2*: $y \leq x \implies x \sqcap y = y$
 ⟨*proof*⟩

lemmas *inf-aci = inf-commute inf-assoc inf-left-commute inf-left-idem*

end

context *semilattice-sup*
begin

sublocale *sup*: *semilattice sup*
 ⟨*proof*⟩

sublocale *sup*: *semilattice-order sup greater-eq greater*
 ⟨*proof*⟩

lemma *sup-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
 ⟨*proof*⟩

lemma *sup-commute*: $(x \sqcup y) = (y \sqcup x)$
 ⟨*proof*⟩

lemma *sup-left-commute*: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$
 ⟨*proof*⟩

lemma *sup-idem*: $x \sqcup x = x$
 ⟨*proof*⟩

lemma *sup-left-idem [simp]*: $x \sqcup (x \sqcup y) = x \sqcup y$
 ⟨*proof*⟩

lemma *sup-absorb1*: $y \leq x \implies x \sqcup y = x$
 ⟨*proof*⟩

lemma *sup-absorb2*: $x \leq y \implies x \sqcup y = y$
 ⟨*proof*⟩

lemmas *sup-aci = sup-commute sup-assoc sup-left-commute sup-left-idem*

end

context *lattice*
begin

lemma *dual-lattice*: *class.lattice sup (\geq) ($>$) inf*

$\langle proof \rangle$

lemma *inf-sup-absorb* [simp]: $x \sqcap (x \sqcup y) = x$
 $\langle proof \rangle$

lemma *sup-inf-absorb* [simp]: $x \sqcup (x \sqcap y) = x$
 $\langle proof \rangle$

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*: $x \sqcup (y \sqcap z) \leq (x \sqcup y) \sqcap (x \sqcup z)$
 $\langle proof \rangle$

lemma *distrib-inf-le*: $(x \sqcap y) \sqcup (x \sqcap z) \leq x \sqcap (y \sqcup z)$
 $\langle proof \rangle$

If you have one of them, you have them all.

lemma *distrib-imp1*:
assumes *distrib*: $\bigwedge x y z. x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
shows $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
 $\langle proof \rangle$

lemma *distrib-imp2*:
assumes *distrib*: $\bigwedge x y z. x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
shows $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
 $\langle proof \rangle$

end

6.3.3 Strict order

context *semilattice-inf*
begin

lemma *less-infI1*: $a < x \implies a \sqcap b < x$
 $\langle proof \rangle$

lemma *less-infI2*: $b < x \implies a \sqcap b < x$
 $\langle proof \rangle$

end

context *semilattice-sup*
begin

lemma *less-supI1*: $x < a \implies x < a \sqcup b$

$\langle proof \rangle$

lemma *less-supI2*: $x < b \implies x < a \sqcup b$
 $\langle proof \rangle$

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*: $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
 $\langle proof \rangle$

lemma *inf-sup-distrib1*: $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
 $\langle proof \rangle$

lemma *inf-sup-distrib2*: $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
 $\langle proof \rangle$

lemma *dual-distrib-lattice*: *class.distrib-lattice* *sup* (\geq) ($>$) *inf*
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemma *inf-top-left*: $\top \sqcap x = x$
 $\langle proof \rangle$

lemma *inf-top-right*: $x \sqcap \top = x$
 $\langle proof \rangle$

lemma *inf-eq-top-iff*: $x \sqcap y = \top \longleftrightarrow x = \top \wedge y = \top$
 $\langle proof \rangle$

lemma *top-eq-inf-iff*: $\top = x \sqcap y \longleftrightarrow x = \top \wedge y = \top$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemma *sup-bot-left*: $\perp \sqcup x = x$
 $\langle proof \rangle$

lemma *sup-bot-right*: $x \sqcup \perp = x$
 $\langle proof \rangle$

lemma *sup-eq-bot-iff*: $x \sqcup y = \perp \longleftrightarrow x = \perp \wedge y = \perp$
 $\langle proof \rangle$

lemma *bot-eq-sup-iff*: $\perp = x \sqcup y \longleftrightarrow x = \perp \wedge y = \perp$
 $\langle proof \rangle$

end

class *bounded-lattice-bot* = *lattice* + *order-bot*
begin

subclass *bounded-semilattice-sup-bot* $\langle proof \rangle$

lemma *inf-bot-left [simp]*: $\perp \sqcap x = \perp$
 $\langle proof \rangle$

lemma *inf-bot-right [simp]*: $x \sqcap \perp = \perp$
 $\langle proof \rangle$

end

class *bounded-lattice-top* = *lattice* + *order-top*
begin

subclass *bounded-semilattice-inf-top* $\langle proof \rangle$

lemma *sup-top-left [simp]*: $\top \sqcup x = \top$
 $\langle proof \rangle$

```

lemma sup-top-right [simp]:  $x \sqcup \top = \top$ 
  ⟨proof⟩

end

class bounded-lattice = lattice + order-bot + order-top
begin

subclass bounded-lattice-bot ⟨proof⟩
subclass bounded-lattice-top ⟨proof⟩

lemma dual-bounded-lattice: class.bounded-lattice sup greater-eq greater inf  $\top \perp$ 
  ⟨proof⟩

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
  ⟨proof⟩

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:  $\min x y \leq z \longleftrightarrow x \leq z \vee y \leq z$ 
  ⟨proof⟩

lemma le-max-iff-disj:  $z \leq \max x y \longleftrightarrow z \leq x \vee z \leq y$ 
  ⟨proof⟩

lemma min-less-iff-disj:  $\min x y < z \longleftrightarrow x < z \vee y < z$ 
  ⟨proof⟩

lemma less-max-iff-disj:  $z < \max x y \longleftrightarrow z < x \vee z < y$ 
  ⟨proof⟩

lemma min-less-iff-conj [simp]:  $z < \min x y \longleftrightarrow z < x \wedge z < y$ 
  ⟨proof⟩

lemma max-less-iff-conj [simp]:  $\max x y < z \longleftrightarrow x < z \wedge y < z$ 
  ⟨proof⟩

```


lemma *min-max-distrib1*: $\min (\max b c) a = \max (\min b a) (\min c a)$
 $\langle \text{proof} \rangle$

lemma *min-max-distrib2*: $\min a (\max b c) = \max (\min a b) (\min a c)$
 $\langle \text{proof} \rangle$

lemma *max-min-distrib1*: $\max (\min b c) a = \min (\max b a) (\max c a)$
 $\langle \text{proof} \rangle$

lemma *max-min-distrib2*: $\max a (\min b c) = \min (\max a b) (\max a c)$
 $\langle \text{proof} \rangle$

lemmas *min-max-distribs* = *min-max-distrib1 min-max-distrib2 max-min-distrib1 max-min-distrib2*

lemma *split-min* [no-atp]: $P (\min i j) \longleftrightarrow (i \leq j \longrightarrow P i) \wedge (\neg i \leq j \longrightarrow P j)$
 $\langle \text{proof} \rangle$

lemma *split-max* [no-atp]: $P (\max i j) \longleftrightarrow (i \leq j \longrightarrow P j) \wedge (\neg i \leq j \longrightarrow P i)$
 $\langle \text{proof} \rangle$

lemma *split-min-lin* [no-atp]:
 $\langle P (\min a b) \longleftrightarrow (b = a \longrightarrow P a) \wedge (a < b \longrightarrow P a) \wedge (b < a \longrightarrow P b) \rangle$
 $\langle \text{proof} \rangle$

lemma *split-max-lin* [no-atp]:
 $\langle P (\max a b) \longleftrightarrow (b = a \longrightarrow P a) \wedge (a < b \longrightarrow P b) \wedge (b < a \longrightarrow P a) \rangle$
 $\langle \text{proof} \rangle$

end

lemma *inf-min*: $\text{inf} = (\min :: 'a :: \{\text{semilattice-inf}, \text{linorder}\} \Rightarrow 'a \Rightarrow 'a)$
 $\langle \text{proof} \rangle$

lemma *sup-max*: $\text{sup} = (\max :: 'a :: \{\text{semilattice-sup}, \text{linorder}\} \Rightarrow 'a \Rightarrow 'a)$
 $\langle \text{proof} \rangle$

6.7 Uniqueness of inf and sup

lemma (in *semilattice-inf*) *inf-unique*:
fixes *f* (infixl \triangleleft 70)
assumes *le1*: $\bigwedge x y. x \triangleleft y \leq x$
and *le2*: $\bigwedge x y. x \triangleleft y \leq y$
and *greatest*: $\bigwedge x y z. x \leq y \implies x \leq z \implies x \leq y \triangleleft z$
shows $x \sqcap y = x \triangleleft y$
 $\langle \text{proof} \rangle$

lemma (in *semilattice-sup*) *sup-unique*:
fixes *f* (infixl \triangleright 70)

```

assumes ge1 [simp]:  $\bigwedge x y. x \leq x \nabla y$ 
and ge2:  $\bigwedge x y. y \leq x \nabla y$ 
and least:  $\bigwedge x y z. y \leq x \implies z \leq x \implies y \nabla z \leq x$ 
shows  $x \sqcup y = x \nabla y$ 
 $\langle proof \rangle$ 

```

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]:  $(f \sqcup g) x = f x \sqcup g x$ 
 $\langle proof \rangle$ 

```

```

instance
 $\langle proof \rangle$ 

```

```

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]:  $(f \sqcap g) x = f x \sqcap g x$ 
 $\langle proof \rangle$ 

```

```

instance  $\langle proof \rangle$ 

```

```

end

```

```

instance fun :: (type, lattice) lattice  $\langle proof \rangle$ 

```

```

instance fun :: (type, distrib-lattice) distrib-lattice
 $\langle proof \rangle$ 

```

```

instance fun :: (type, bounded-lattice) bounded-lattice  $\langle proof \rangle$ 

```

```

instantiation fun :: (type, uminus) uminus
begin

```

```

definition fun-Compl-def:  $- A = (\lambda x. - A x)$ 

```

```

lemma uminus-apply [simp, code]:  $(- A) x = - (A x)$ 
 $\langle proof \rangle$ 

```

```

instance  $\langle proof \rangle$ 

```

end

instantiation *fun* :: (*type*, *minus*) *minus*
begin

definition *fun-diff-def*: $A - B = (\lambda x. A\ x - B\ x)$

lemma *minus-apply* [*simp*, *code*]: $(A - B)\ x = A\ x - B\ x$
 $\langle proof \rangle$

instance $\langle proof \rangle$

end

end

7 Boolean Algebras

theory *Boolean-Algebras*
imports *Lattices*
begin

7.1 Abstract boolean algebra

locale *abstract-boolean-algebra* = *conj*: *abel-semigroup* $\langle (\sqcap) \rangle$ + *disj*: *abel-semigroup* $\langle (\sqcup) \rangle$

for *conj* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$ (**infixr** $\langle \sqcap \rangle$ 70)
and *disj* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$ (**infixr** $\langle \sqcup \rangle$ 65) +
fixes *compl* :: $\langle 'a \Rightarrow 'a \rangle$ ($\langle (\langle open-block\ notation = \langle prefix - \rangle - \rangle) [81]\ 80 \rangle$)
and *zero* :: $\langle 'a \rangle$ ($\langle 0 \rangle$)
and *one* :: $\langle 'a \rangle$ ($\langle 1 \rangle$)
assumes *conj-disj-distrib*: $\langle x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z) \rangle$
and *disj-conj-distrib*: $\langle x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z) \rangle$
and *conj-one-right*: $\langle x \sqcap 1 = x \rangle$
and *disj-zero-right*: $\langle x \sqcup 0 = x \rangle$
and *conj-cancel-right* [*simp*]: $\langle x \sqcap -\ x = 0 \rangle$
and *disj-cancel-right* [*simp*]: $\langle x \sqcup -\ x = 1 \rangle$

begin

sublocale *conj*: *semilattice-neutr* $\langle (\sqcap) \rangle$ $\langle 1 \rangle$
 $\langle proof \rangle$

sublocale *disj*: *semilattice-neutr* $\langle (\sqcup) \rangle$ $\langle 0 \rangle$
 $\langle proof \rangle$

7.1.1 Complement

lemma *complement-unique*:

assumes 1: $a \sqcap x = \mathbf{0}$
assumes 2: $a \sqcup x = \mathbf{1}$
assumes 3: $a \sqcap y = \mathbf{0}$
assumes 4: $a \sqcup y = \mathbf{1}$
shows $x = y$
 $\langle \text{proof} \rangle$

lemma *compl-unique*: $x \sqcap y = \mathbf{0} \implies x \sqcup y = \mathbf{1} \implies \neg x = y$
 $\langle \text{proof} \rangle$

lemma *double-compl* [simp]: $\neg (\neg x) = x$
 $\langle \text{proof} \rangle$

lemma *compl-eq-compl-iff* [simp]:
 $\langle \neg x = \neg y \iff x = y \rangle$ (**is** $\langle ?P \iff ?Q \rangle$)
 $\langle \text{proof} \rangle$

7.1.2 Conjunction

lemma *conj-zero-right* [simp]: $x \sqcap \mathbf{0} = \mathbf{0}$
 $\langle \text{proof} \rangle$

lemma *compl-one* [simp]: $\neg \mathbf{1} = \mathbf{0}$
 $\langle \text{proof} \rangle$

lemma *conj-zero-left* [simp]: $\mathbf{0} \sqcap x = \mathbf{0}$
 $\langle \text{proof} \rangle$

lemma *conj-cancel-left* [simp]: $\neg x \sqcap x = \mathbf{0}$
 $\langle \text{proof} \rangle$

lemma *conj-disj-distrib2*: $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
 $\langle \text{proof} \rangle$

lemmas *conj-disj-distrib* = *conj-disj-distrib conj-disj-distrib2*

7.1.3 Disjunction

context
begin

interpretation *dual*: *abstract-boolean-algebra* $\langle (\sqcup) \rangle \langle (\sqcap) \rangle$ *compl* $\langle \mathbf{1} \rangle \langle \mathbf{0} \rangle$
 $\langle \text{proof} \rangle$

lemma *disj-one-right* [simp]: $x \sqcup \mathbf{1} = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *compl-zero* [simp]: $\neg \mathbf{0} = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *disj-cancel-left* [*simp*]: $\neg x \sqcup x = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *disj-conj-distrib2*: $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
 $\langle \text{proof} \rangle$

lemmas *disj-conj-distrib* = *disj-conj-distrib* *disj-conj-distrib2*

end

7.1.4 De Morgan’s Laws

lemma *de-Morgan-conj* [*simp*]: $\neg (x \sqcap y) = \neg x \sqcup \neg y$
 $\langle \text{proof} \rangle$

context
begin

interpretation *dual*: *abstract-boolean-algebra* $\langle (\sqcup) \rangle \langle (\sqcap) \rangle$ *compl* $\langle \mathbf{1} \rangle \langle \mathbf{0} \rangle$
 $\langle \text{proof} \rangle$

lemma *de-Morgan-disj* [*simp*]: $\neg (x \sqcup y) = \neg x \sqcap \neg y$
 $\langle \text{proof} \rangle$

end

end

7.2 Symmetric Difference

locale *abstract-boolean-algebra-sym-diff* = *abstract-boolean-algebra* +
fixes *xor* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$ (**infixr** $\langle \ominus \rangle$ 65)
assumes *xor-def* : $\langle x \ominus y = (x \sqcap \neg y) \sqcup (\neg x \sqcap y) \rangle$
begin

sublocale *xor*: *comm-monoid xor* $\langle \mathbf{0} \rangle$
 $\langle \text{proof} \rangle$

lemma *xor-def2*:
 $\langle x \ominus y = (x \sqcup y) \sqcap (\neg x \sqcup \neg y) \rangle$
 $\langle \text{proof} \rangle$

lemma *xor-one-right* [*simp*]: $x \ominus \mathbf{1} = \neg x$
 $\langle \text{proof} \rangle$

lemma *xor-one-left* [*simp*]: $\mathbf{1} \ominus x = \neg x$
 $\langle \text{proof} \rangle$

lemma *xor-self* [*simp*]: $x \ominus x = \mathbf{0}$
 $\langle \text{proof} \rangle$

lemma *xor-left-self* [*simp*]: $x \ominus (x \ominus y) = y$
 $\langle \text{proof} \rangle$

lemma *xor-compl-left* [*simp*]: $\neg x \ominus y = \neg (x \ominus y)$
 $\langle \text{proof} \rangle$

lemma *xor-compl-right* [*simp*]: $x \ominus \neg y = \neg (x \ominus y)$
 $\langle \text{proof} \rangle$

lemma *xor-cancel-right* [*simp*]: $x \ominus \neg x = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *xor-cancel-left* [*simp*]: $\neg x \ominus x = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *conj-xor-distrib*: $x \sqcap (y \ominus z) = (x \sqcap y) \ominus (x \sqcap z)$
 $\langle \text{proof} \rangle$

lemma *conj-xor-distrib2*: $(y \ominus z) \sqcap x = (y \sqcap x) \ominus (z \sqcap x)$
 $\langle \text{proof} \rangle$

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*: $\langle x \sqcap \neg x = \perp \rangle$
and *sup-compl-top*: $\langle x \sqcup \neg x = \top \rangle$
assumes *diff-eq*: $\langle x - y = x \sqcap \neg y \rangle$
begin

sublocale *boolean-algebra*: *abstract-boolean-algebra* $\langle (\sqcap) \rangle \langle (\sqcup) \rangle$ *uminus* $\perp \top$
 $\langle \text{proof} \rangle$

lemma *compl-inf-bot*: $\neg x \sqcap x = \perp$
 $\langle \text{proof} \rangle$

lemma *compl-sup-top*: $\neg x \sqcup x = \top$
 $\langle \text{proof} \rangle$

lemma *compl-unique*:
assumes $x \sqcap y = \perp$
and $x \sqcup y = \top$

shows $\neg x = y$
 $\langle proof \rangle$

lemma *double-compl*: $\neg(\neg x) = x$
 $\langle proof \rangle$

lemma *compl-eq-compl-iff*: $\neg x = \neg y \longleftrightarrow x = y$
 $\langle proof \rangle$

lemma *compl-bot-eq*: $\neg \perp = \top$
 $\langle proof \rangle$

lemma *compl-top-eq*: $\neg \top = \perp$
 $\langle proof \rangle$

lemma *compl-inf*: $\neg(x \sqcap y) = \neg x \sqcup \neg y$
 $\langle proof \rangle$

lemma *compl-sup*: $\neg(x \sqcup y) = \neg x \sqcap \neg y$
 $\langle proof \rangle$

lemma *compl-mono*:
assumes $x \leq y$
shows $\neg y \leq \neg x$
 $\langle proof \rangle$

lemma *compl-le-compl-iff* [*simp*]: $\neg x \leq \neg y \longleftrightarrow y \leq x$
 $\langle proof \rangle$

lemma *compl-le-swap1*:
assumes $y \leq \neg x$
shows $x \leq \neg y$
 $\langle proof \rangle$

lemma *compl-le-swap2*:
assumes $\neg y \leq x$
shows $\neg x \leq y$
 $\langle proof \rangle$

lemma *compl-less-compl-iff* [*simp*]: $\neg x < \neg y \longleftrightarrow y < x$
 $\langle proof \rangle$

lemma *compl-less-swap1*:
assumes $y < \neg x$
shows $x < \neg y$
 $\langle proof \rangle$

lemma *compl-less-swap2*:
assumes $\neg y < x$

shows $- x < y$
 $\langle proof \rangle$

lemma *sup-cancel-left1*: $\langle x \sqcup a \sqcup (- x \sqcup b) = \top \rangle$
 $\langle proof \rangle$

lemma *sup-cancel-left2*: $\langle - x \sqcup a \sqcup (x \sqcup b) = \top \rangle$
 $\langle proof \rangle$

lemma *inf-cancel-left1*: $\langle x \sqcap a \sqcap (- x \sqcap b) = \perp \rangle$
 $\langle proof \rangle$

lemma *inf-cancel-left2*: $\langle - x \sqcap a \sqcap (x \sqcap b) = \perp \rangle$
 $\langle proof \rangle$

lemma *sup-compl-top-left1* [simp]: $\langle - x \sqcup (x \sqcup y) = \top \rangle$
 $\langle proof \rangle$

lemma *sup-compl-top-left2* [simp]: $\langle x \sqcup (- x \sqcup y) = \top \rangle$
 $\langle proof \rangle$

lemma *inf-compl-bot-left1* [simp]: $\langle - x \sqcap (x \sqcap y) = \perp \rangle$
 $\langle proof \rangle$

lemma *inf-compl-bot-left2* [simp]: $\langle x \sqcap (- x \sqcap y) = \perp \rangle$
 $\langle proof \rangle$

lemma *inf-compl-bot-right* [simp]: $\langle x \sqcap (y \sqcap - x) = \perp \rangle$
 $\langle proof \rangle$

end

7.4 Lattice on *bool*

instantiation *bool* :: *boolean-algebra*
begin

definition *bool-Compl-def* [simp]: *uminus* = *Not*

definition *bool-diff-def* [simp]: $A - B \longleftrightarrow A \wedge \neg B$

definition [simp]: $P \sqcap Q \longleftrightarrow P \wedge Q$

definition [simp]: $P \sqcup Q \longleftrightarrow P \vee Q$

instance $\langle proof \rangle$

end

lemma *sup-boolI1*: $P \Longrightarrow P \sqcup Q$
 $\langle \text{proof} \rangle$

lemma *sup-boolI2*: $Q \Longrightarrow P \sqcup Q$
 $\langle \text{proof} \rangle$

lemma *sup-boolE*: $P \sqcup Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
 $\langle \text{proof} \rangle$

instance *fun* :: (*type*, *boolean-algebra*) *boolean-algebra*
 $\langle \text{proof} \rangle$

7.5 Lattice on unary and binary predicates

lemma *inf1I*: $A\ x \Longrightarrow B\ x \Longrightarrow (A \sqcap B)\ x$
 $\langle \text{proof} \rangle$

lemma *inf2I*: $A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow (A \sqcap B)\ x\ y$
 $\langle \text{proof} \rangle$

lemma *inf1E*: $(A \sqcap B)\ x \Longrightarrow (A\ x \Longrightarrow B\ x \Longrightarrow P) \Longrightarrow P$
 $\langle \text{proof} \rangle$

lemma *inf2E*: $(A \sqcap B)\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow P) \Longrightarrow P$
 $\langle \text{proof} \rangle$

lemma *inf1D1*: $(A \sqcap B)\ x \Longrightarrow A\ x$
 $\langle \text{proof} \rangle$

lemma *inf2D1*: $(A \sqcap B)\ x\ y \Longrightarrow A\ x\ y$
 $\langle \text{proof} \rangle$

lemma *inf1D2*: $(A \sqcap B)\ x \Longrightarrow B\ x$
 $\langle \text{proof} \rangle$

lemma *inf2D2*: $(A \sqcap B)\ x\ y \Longrightarrow B\ x\ y$
 $\langle \text{proof} \rangle$

lemma *sup1I1*: $A\ x \Longrightarrow (A \sqcup B)\ x$
 $\langle \text{proof} \rangle$

lemma *sup2I1*: $A\ x\ y \Longrightarrow (A \sqcup B)\ x\ y$
 $\langle \text{proof} \rangle$

lemma *sup1I2*: $B\ x \Longrightarrow (A \sqcup B)\ x$
 $\langle \text{proof} \rangle$

lemma *sup2I2*: $B\ x\ y \Longrightarrow (A \sqcup B)\ x\ y$
 $\langle \text{proof} \rangle$

lemma *sup1E*: $(A \sqcup B) \ x \Longrightarrow (A \ x \Longrightarrow P) \Longrightarrow (B \ x \Longrightarrow P) \Longrightarrow P$
 $\langle proof \rangle$

lemma *sup2E*: $(A \sqcup B) \ x \ y \Longrightarrow (A \ x \ y \Longrightarrow P) \Longrightarrow (B \ x \ y \Longrightarrow P) \Longrightarrow P$
 $\langle proof \rangle$

Classical introduction rule: no commitment to A vs B .

lemma *sup1CI*: $(\neg B \ x \Longrightarrow A \ x) \Longrightarrow (A \sqcup B) \ x$
 $\langle proof \rangle$

lemma *sup2CI*: $(\neg B \ x \ y \Longrightarrow A \ x \ y) \Longrightarrow (A \sqcup B) \ x \ y$
 $\langle proof \rangle$

7.6 Simproc setup

locale *boolean-algebra-cancel*
begin

lemma *sup1*: $(A::'a::semilattice-sup) \equiv sup \ k \ a \Longrightarrow sup \ A \ b \equiv sup \ k \ (sup \ a \ b)$
 $\langle proof \rangle$

lemma *sup2*: $(B::'a::semilattice-sup) \equiv sup \ k \ b \Longrightarrow sup \ a \ B \equiv sup \ k \ (sup \ a \ b)$
 $\langle proof \rangle$

lemma *sup0*: $(a::'a::bounded-semilattice-sup-bot) \equiv sup \ a \ bot$
 $\langle proof \rangle$

lemma *inf1*: $(A::'a::semilattice-inf) \equiv inf \ k \ a \Longrightarrow inf \ A \ b \equiv inf \ k \ (inf \ a \ b)$
 $\langle proof \rangle$

lemma *inf2*: $(B::'a::semilattice-inf) \equiv inf \ k \ b \Longrightarrow inf \ a \ B \equiv inf \ k \ (inf \ a \ b)$
 $\langle proof \rangle$

lemma *inf0*: $(a::'a::bounded-semilattice-inf-top) \equiv inf \ a \ top$
 $\langle proof \rangle$

end

$\langle ML \rangle$

context *boolean-algebra*
begin

lemma *shunt1*: $(x \sqcap y \leq z) \longleftrightarrow (x \leq -y \sqcup z)$
 $\langle proof \rangle$

lemma *shunt2*: $(x \sqcap -y \leq z) \longleftrightarrow (x \leq y \sqcup z)$

$\langle proof \rangle$

lemma *inf-shunt*: $(x \sqcap y = \perp) \longleftrightarrow (x \leq - y)$
 $\langle proof \rangle$

lemma *sup-shunt*: $(x \sqcup y = \top) \longleftrightarrow (- x \leq y)$
 $\langle proof \rangle$

lemma *diff-shunt-var[simp]*: $(x - y = \perp) \longleftrightarrow (x \leq y)$
 $\langle proof \rangle$

lemma *diff-shunt[simp]*: $(\perp = x - y) \longleftrightarrow (x \leq y)$
 $\langle proof \rangle$

lemma *sup-neg-inf*:
 $\langle p \leq q \sqcup r \longleftrightarrow p \sqcap -q \leq r \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)
 $\langle proof \rangle$

end

end

8 Set theory for higher-order logic

theory *Set*

imports *Lattices Boolean-Algebras*

begin

8.1 Sets as predicates

typed decl *'a set*

axiomatization *Collect* :: $('a \Rightarrow bool) \Rightarrow 'a \text{ set}$ — comprehension
and *member* :: $'a \Rightarrow 'a \text{ set} \Rightarrow bool$ — membership
where *mem-Collect-eq* [*iff*, *code-unfold*]: $\text{member } a \text{ (Collect } P) = P \ a$
and *Collect-mem-eq* [*simp*, *code-unfold*]: $\text{Collect } (\lambda x. \text{member } x \ A) = A$

notation

member $\langle \langle '(\in) \rangle \rangle$ **and**

member $\langle \langle \langle notation = \langle infix \in \rangle \rangle - / \in - \rangle \rangle$ [*51*, *51*] *50*)

abbreviation *not-member*

where *not-member* $x \ A \equiv \neg (x \in A)$ — non-membership

notation

not-member $\langle \langle '(\notin) \rangle \rangle$ **and**

not-member $\langle \langle \langle notation = \langle infix \notin \rangle \rangle - / \notin - \rangle \rangle$ [*51*, *51*] *50*)

open-bundle *member-ASCII-syntax*

begin

notation (*ASCII*)
 member ($\langle \langle ' \langle : \rangle \rangle \rangle$) **and**
 member ($\langle \langle \langle \text{notation} = \langle \text{infix} : \rangle \rangle - / : - \rangle \rangle [51, 51] 50 \rangle$) **and**
 not-member ($\langle \langle ' \langle \sim : \rangle \rangle \rangle$) **and**
 not-member ($\langle \langle \langle \text{notation} = \langle \text{infix} \sim : \rangle \rangle - / \sim : - \rangle \rangle [51, 51] 50 \rangle$)
end

Set comprehensions

syntax
 -Coll :: pptrn \Rightarrow bool \Rightarrow 'a set ($\langle \langle \langle \text{indent} = 1 \text{ notation} = \langle \text{mixfix set comprehension} \rangle \rangle \{ - / - \} \rangle \rangle$)
syntax-consts
 -Coll \Rightarrow Collect
translations
 $\{x. P\} \Rightarrow \text{CONST Collect } (\lambda x. P)$

syntax (*ASCII*)
 -Collect :: pptrn \Rightarrow 'a set \Rightarrow bool \Rightarrow 'a set ($\langle \langle \langle \text{indent} = 1 \text{ notation} = \langle \text{mixfix set comprehension} \rangle \rangle \{ - / - \} \rangle \rangle$)
syntax
 -Collect :: pptrn \Rightarrow 'a set \Rightarrow bool \Rightarrow 'a set ($\langle \langle \langle \text{indent} = 1 \text{ notation} = \langle \text{mixfix set comprehension} \rangle \rangle \{ - / \in - \} \rangle \rangle$)
translations
 $\{p:A. P\} \mapsto \text{CONST Collect } (\lambda p. p \in A \wedge P)$

$\langle ML \rangle$

lemma CollectI: $P a \Longrightarrow a \in \{x. P x\}$
 $\langle \text{proof} \rangle$

lemma CollectD: $a \in \{x. P x\} \Longrightarrow P a$
 $\langle \text{proof} \rangle$

lemma Collect-cong: $(\bigwedge x. P x = Q x) \Longrightarrow \{x. P x\} = \{x. Q x\}$
 $\langle \text{proof} \rangle$

Simproc for pulling $x = t$ in $\{x. \dots \wedge x = t \wedge \dots\}$ to the front (and similarly for $t = x$):

$\langle ML \rangle$

lemmas CollectE = CollectD [elim-format]

lemma set-eqI:
 assumes $\bigwedge x. x \in A \longleftrightarrow x \in B$
 shows $A = B$
 $\langle \text{proof} \rangle$

lemma set-eq-iff: $A = B \longleftrightarrow (\forall x. x \in A \longleftrightarrow x \in B)$
 $\langle \text{proof} \rangle$

lemma *Collect-eqI*:

assumes $\bigwedge x. P\ x = Q\ x$
 shows $\text{Collect } P = \text{Collect } Q$
 $\langle \text{proof} \rangle$

Lifting of predicate class instances

instantiation *set* :: (type) boolean-algebra
begin

definition *less-eq-set*

where $A \leq B \longleftrightarrow (\lambda x. \text{member } x\ A) \leq (\lambda x. \text{member } x\ B)$

definition *less-set*

where $A < B \longleftrightarrow (\lambda x. \text{member } x\ A) < (\lambda x. \text{member } x\ B)$

definition *inf-set*

where $A \sqcap B = \text{Collect } ((\lambda x. \text{member } x\ A) \sqcap (\lambda x. \text{member } x\ B))$

definition *sup-set*

where $A \sqcup B = \text{Collect } ((\lambda x. \text{member } x\ A) \sqcup (\lambda x. \text{member } x\ B))$

definition *bot-set*

where $\perp = \text{Collect } \perp$

definition *top-set*

where $\top = \text{Collect } \top$

definition *uminus-set*

where $- A = \text{Collect } (- (\lambda x. \text{member } x\ A))$

definition *minus-set*

where $A - B = \text{Collect } ((\lambda x. \text{member } x\ A) - (\lambda x. \text{member } x\ B))$

instance

$\langle \text{proof} \rangle$

end

Set enumerations

abbreviation *empty* :: 'a set ($\langle \{\} \rangle$)

where $\{\} \equiv \text{bot}$

definition *insert* :: 'a \Rightarrow 'a set \Rightarrow 'a set

where *insert-compr*: $\text{insert } a\ B = \{x. x = a \vee x \in B\}$

open-bundle *set-enumeration-syntax*

begin

syntax

-*Finset* :: *args* \Rightarrow '*a set* ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix set enumeration} \rangle \{ - \} \rangle$)

syntax-consts

-*Finset* \equiv *insert*

translations

$\{x, xs\} \equiv \text{CONST insert } x \{xs\}$

$\{x\} \equiv \text{CONST insert } x \{\}$

end

8.2 Subsets and bounded quantifiers

abbreviation *subset* :: '*a set* \Rightarrow '*a set* \Rightarrow *bool*

where *subset* \equiv *less*

abbreviation *subset-eq* :: '*a set* \Rightarrow '*a set* \Rightarrow *bool*

where *subset-eq* \equiv *less-eq*

notation

subset ($\langle '(\subset) \rangle$) **and**

subset ($\langle \langle \text{notation}=\langle \text{infix } \subset \rangle \rangle - / \subset - \rangle$ [51, 51] 50) **and**

subset-eq ($\langle '(\subseteq) \rangle$) **and**

subset-eq ($\langle \langle \text{notation}=\langle \text{infix } \subseteq \rangle \rangle - / \subseteq - \rangle$ [51, 51] 50)

abbreviation (*input*)

supset :: '*a set* \Rightarrow '*a set* \Rightarrow *bool* **where**

supset \equiv *greater*

abbreviation (*input*)

supset-eq :: '*a set* \Rightarrow '*a set* \Rightarrow *bool* **where**

supset-eq \equiv *greater-eq*

notation

supset ($\langle '(\supset) \rangle$) **and**

supset ($\langle \langle \text{notation}=\langle \text{infix } \supset \rangle \rangle - / \supset - \rangle$ [51, 51] 50) **and**

supset-eq ($\langle '(\supseteq) \rangle$) **and**

supset-eq ($\langle \langle \text{notation}=\langle \text{infix } \supseteq \rangle \rangle - / \supseteq - \rangle$ [51, 51] 50)

notation (*ASCII output*)

subset ($\langle '(<) \rangle$) **and**

subset ($\langle \langle \text{notation}=\langle \text{infix } < \rangle \rangle - / < - \rangle$ [51, 51] 50) **and**

subset-eq ($\langle '(<=) \rangle$) **and**

subset-eq ($\langle \langle \text{notation}=\langle \text{infix } <= \rangle \rangle - / <= - \rangle$ [51, 51] 50)

definition *Ball* :: '*a set* \Rightarrow ('*a* \Rightarrow *bool*) \Rightarrow *bool*

where *Ball* *A P* $\longleftrightarrow (\forall x. x \in A \longrightarrow P x)$ — bounded universal quantifiers

definition *Bex* :: '*a set* \Rightarrow ('*a* \Rightarrow *bool*) \Rightarrow *bool*

where *Bex* *A P* $\longleftrightarrow (\exists x. x \in A \wedge P x)$ — bounded existential quantifiers

syntax (ASCII)

$-Ball :: pttrn \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder$
 $ALL : \gg ALL (-/:-)./ - \rangle [0, 0, 10] 10)$
 $-Bex :: pttrn \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder$
 $EX : \gg EX (-/:-)./ - \rangle [0, 0, 10] 10)$
 $-Bex1 :: pttrn \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder$
 $EX! : \gg EX! (-/:-)./ - \rangle [0, 0, 10] 10)$
 $-Bleat :: id \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow 'a$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder$
 $LEAST : \gg LEAST (-/:-)./ - \rangle [0, 0, 10] 10)$

syntax (input)

$-Ball :: pttrn \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder !$
 $: \gg ! (-/:-)./ - \rangle [0, 0, 10] 10)$
 $-Bex :: pttrn \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder ?$
 $: \gg ? (-/:-)./ - \rangle [0, 0, 10] 10)$
 $-Bex1 :: pttrn \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder$
 $?! : \gg ?! (-/:-)./ - \rangle [0, 0, 10] 10)$

syntax

$-Ball :: pttrn \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder$
 $\forall : \gg \forall (-/\in-)./ - \rangle [0, 0, 10] 10)$
 $-Bex :: pttrn \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder$
 $\exists : \gg \exists (-/\in-)./ - \rangle [0, 0, 10] 10)$
 $-Bex1 :: pttrn \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder$
 $\exists! : \gg \exists! (-/\in-)./ - \rangle [0, 0, 10] 10)$
 $-Bleat :: id \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow 'a$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder$
 $LEAST : \gg LEAST (-/\in-)./ - \rangle [0, 0, 10] 10)$

syntax-consts

$-Ball \Rightarrow Ball \text{ and}$
 $-Bex \Rightarrow Bex \text{ and}$
 $-Bex1 \Rightarrow Ex1 \text{ and}$
 $-Bleat \Rightarrow Least$

translations

$\forall x \in A. P \Leftrightarrow CONST \text{ Ball } A (\lambda x. P)$
 $\exists x \in A. P \Leftrightarrow CONST \text{ Bex } A (\lambda x. P)$
 $\exists! x \in A. P \rightarrow \exists! x. x \in A \wedge P$
 $LEAST x:A. P \rightarrow LEAST x. x \in A \wedge P$

syntax (ASCII output)

$-setlessAll :: [idt, 'a, bool] \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder ALL \rangle \gg ALL$
 $-<-./ - \rangle [0, 0, 10] 10)$
 $-setlessEx :: [idt, 'a, bool] \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder EX \rangle \gg EX -<-./$
 $- \rangle [0, 0, 10] 10)$
 $-setleAll :: [idt, 'a, bool] \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder ALL \rangle \gg ALL$
 $-<= -./ - \rangle [0, 0, 10] 10)$
 $-setleEx :: [idt, 'a, bool] \Rightarrow bool$ $(\langle \langle \text{indent}=3 \text{ notation}=\langle binder EX \rangle \gg EX -<= -./$

-)> [0, 0, 10] 10)
 -settleEx1 :: [idt, 'a, bool] ⇒ bool (⟨(⟨indent=3 notation=⟨binder EX!⟩EX!
 -<= -./ -)⟩ [0, 0, 10] 10)

syntax

-setlessAll :: [idt, 'a, bool] ⇒ bool (⟨(⟨indent=3 notation=⟨binder ∀⟩∀ -< -./ -)⟩
 [0, 0, 10] 10)
 -setlessEx :: [idt, 'a, bool] ⇒ bool (⟨(⟨indent=3 notation=⟨binder ∃⟩∃ -< -./ -)⟩
 [0, 0, 10] 10)
 -setleAll :: [idt, 'a, bool] ⇒ bool (⟨(⟨indent=3 notation=⟨binder ∀⟩∀ -<= -./ -)⟩
 [0, 0, 10] 10)
 -setleEx :: [idt, 'a, bool] ⇒ bool (⟨(⟨indent=3 notation=⟨binder ∃⟩∃ -<= -./ -)⟩
 [0, 0, 10] 10)
 -setleEx1 :: [idt, 'a, bool] ⇒ bool (⟨(⟨indent=3 notation=⟨binder ∃!⟩∃! -<= -./
 -)⟩ [0, 0, 10] 10)

syntax-consts

-setlessAll -setleAll ⇒ All and
 -setlessEx -setleEx ⇒ Ex and
 -setleEx1 ⇒ Ex1

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
 ∃! A ⊆ B. P → ∃! A. A ⊆ B ∧ P

⟨ML⟩

Translate between {e | x1...xn. P} and {u. ∃ x1...xn. u = e ∧ P}; {y.
 ∃ x1...xn. y = e ∧ P} is only translated if [0..n] ⊆ bus e.

syntax

-Setcompr :: 'a ⇒ idts ⇒ bool ⇒ 'a set
 (⟨(⟨indent=1 notation=⟨mixfix set comprehension⟩{- |/-/ -})⟩)

syntax-consts

-Setcompr ⇒ Collect

⟨ML⟩

lemma ballI [intro!]: (∧ x. x ∈ A ⇒ P x) ⇒ ∀ x ∈ A. P x
 ⟨proof⟩

lemmas strip = impI allI ballI

lemma bspec [dest?]: ∀ x ∈ A. P x ⇒ x ∈ A ⇒ P x
 ⟨proof⟩

Gives better instantiation for bound:

$\langle ML \rangle$

lemma *ballE* [*elim*]: $\forall x \in A. P x \implies (P x \implies Q) \implies (x \notin A \implies Q) \implies Q$
 $\langle proof \rangle$

lemma *bexI* [*intro*]: $P x \implies x \in A \implies \exists x \in A. P x$
 — Normally the best argument order: $P x$ constrains the choice of $x \in A$.
 $\langle proof \rangle$

lemma *rev-bexI* [*intro?*]: $x \in A \implies P x \implies \exists x \in A. P x$
 — The best argument order when there is only one $x \in A$.
 $\langle proof \rangle$

lemma *bexCI*: $(\forall x \in A. \neg P x \implies P a) \implies a \in A \implies \exists x \in A. P x$
 $\langle proof \rangle$

lemma *bexE* [*elim!*]: $\exists x \in A. P x \implies (\bigwedge x. x \in A \implies P x \implies Q) \implies Q$
 $\langle proof \rangle$

lemma *ball-triv* [*simp*]: $(\forall x \in A. P) \longleftrightarrow ((\exists x. x \in A) \longrightarrow P)$
 — trivial rewrite rule.
 $\langle proof \rangle$

lemma *bex-triv* [*simp*]: $(\exists x \in A. P) \longleftrightarrow ((\exists x. x \in A) \wedge P)$
 — Dual form for existentials.
 $\langle proof \rangle$

lemma *bex-triv-one-point1* [*simp*]: $(\exists x \in A. x = a) \longleftrightarrow a \in A$
 $\langle proof \rangle$

lemma *bex-triv-one-point2* [*simp*]: $(\exists x \in A. a = x) \longleftrightarrow a \in A$
 $\langle proof \rangle$

lemma *bex-one-point1* [*simp*]: $(\exists x \in A. x = a \wedge P x) \longleftrightarrow a \in A \wedge P a$
 $\langle proof \rangle$

lemma *bex-one-point2* [*simp*]: $(\exists x \in A. a = x \wedge P x) \longleftrightarrow a \in A \wedge P a$
 $\langle proof \rangle$

lemma *ball-one-point1* [*simp*]: $(\forall x \in A. x = a \longrightarrow P x) \longleftrightarrow (a \in A \longrightarrow P a)$
 $\langle proof \rangle$

lemma *ball-one-point2* [*simp*]: $(\forall x \in A. a = x \longrightarrow P x) \longleftrightarrow (a \in A \longrightarrow P a)$
 $\langle proof \rangle$

lemma *ball-conj-distrib*: $(\forall x \in A. P x \wedge Q x) \longleftrightarrow (\forall x \in A. P x) \wedge (\forall x \in A. Q x)$
 $\langle proof \rangle$

lemma *bex-disj-distrib*: $(\exists x \in A. P x \vee Q x) \longleftrightarrow (\exists x \in A. P x) \vee (\exists x \in A. Q x)$

$\langle proof \rangle$

Congruence rules

lemma *ball-cong*:

$$\llbracket A = B; \bigwedge x. x \in B \implies P\ x \longleftrightarrow Q\ x \rrbracket \implies (\forall x \in A. P\ x) \longleftrightarrow (\forall x \in B. Q\ x)$$

$\langle proof \rangle$

lemma *ball-cong-simp* [cong]:

$$\llbracket A = B; \bigwedge x. x \in B =_{simp} \implies P\ x \longleftrightarrow Q\ x \rrbracket \implies (\forall x \in A. P\ x) \longleftrightarrow (\forall x \in B. Q\ x)$$

$\langle proof \rangle$

lemma *bex-cong*:

$$\llbracket A = B; \bigwedge x. x \in B \implies P\ x \longleftrightarrow Q\ x \rrbracket \implies (\exists x \in A. P\ x) \longleftrightarrow (\exists x \in B. Q\ x)$$

$\langle proof \rangle$

lemma *bex-cong-simp* [cong]:

$$\llbracket A = B; \bigwedge x. x \in B =_{simp} \implies P\ x \longleftrightarrow Q\ x \rrbracket \implies (\exists x \in A. P\ x) \longleftrightarrow (\exists x \in B. Q\ x)$$

$\langle proof \rangle$

lemma *bex1-def*: $(\exists !x \in X. P\ x) \longleftrightarrow (\exists x \in X. P\ x) \wedge (\forall x \in X. \forall y \in X. P\ x \longrightarrow P\ y \longrightarrow x = y)$

$\langle proof \rangle$

8.3 Basic operations

8.3.1 Subsets

lemma *subsetI* [intro!]: $(\bigwedge x. x \in A \implies x \in B) \implies A \subseteq B$

$\langle proof \rangle$

Map the type *'a set \Rightarrow anything* to just *'a*; for overloading constants whose first argument has type *'a set*.

lemma *subsetD* [elim, intro?]: $A \subseteq B \implies c \in A \implies c \in B$

$\langle proof \rangle$

lemma *rev-subsetD* [intro?, no-atp]: $c \in A \implies A \subseteq B \implies c \in B$

— The same, with reversed premises for use with *erule* – cf. $\llbracket ?P; ?P \longrightarrow ?Q \rrbracket \implies ?Q$.

$\langle proof \rangle$

lemma *subsetCE* [elim, no-atp]: $A \subseteq B \implies (c \notin A \implies P) \implies (c \in B \implies P) \implies P$

— Classical elimination rule.

$\langle proof \rangle$

lemma *subset-eq*: $A \subseteq B \longleftrightarrow (\forall x \in A. x \in B)$
 $\langle proof \rangle$

lemma *contra-subsetD* [*no-atp*]: $A \subseteq B \implies c \notin B \implies c \notin A$
 $\langle proof \rangle$

lemma *subset-refl*: $A \subseteq A$
 $\langle proof \rangle$

lemma *subset-trans*: $A \subseteq B \implies B \subseteq C \implies A \subseteq C$
 $\langle proof \rangle$

lemma *subset-not-subset-eq* [*code*]: $A \subset B \longleftrightarrow A \subseteq B \wedge \neg B \subseteq A$
 $\langle proof \rangle$

lemma *eq-mem-trans*: $a = b \implies b \in A \implies a \in A$
 $\langle proof \rangle$

lemmas *basic-trans-rules* [*trans*] =
order-trans-rules rev-subsetD subsetD eq-mem-trans

8.3.2 Equality

lemma *subset-antisym* [*intro!*]: $A \subseteq B \implies B \subseteq A \implies A = B$
 — Anti-symmetry of the subset relation.
 $\langle proof \rangle$

Equality rules from ZF set theory – are they appropriate here?

lemma *equalityD1*: $A = B \implies A \subseteq B$
 $\langle proof \rangle$

lemma *equalityD2*: $A = B \implies B \subseteq A$
 $\langle proof \rangle$

Be careful when adding this to the claset as *subset-empty* is in the simpset:
 $A = \{\}$ goes to $\{\} \subseteq A$ and $A \subseteq \{\}$ and then back to $A = \{\}$!

lemma *equalityE*: $A = B \implies (A \subseteq B \implies B \subseteq A \implies P) \implies P$
 $\langle proof \rangle$

lemma *equalityCE* [*elim*]: $A = B \implies (c \in A \implies c \in B \implies P) \implies (c \notin A \implies c \notin B \implies P) \implies P$
 $\langle proof \rangle$

lemma *eqset-imp-iff*: $A = B \implies x \in A \longleftrightarrow x \in B$
 $\langle proof \rangle$

lemma *equelem-imp-iff*: $x = y \implies x \in A \longleftrightarrow y \in A$
 $\langle proof \rangle$

8.3.3 The empty set

lemma *empty-def*: $\{\} = \{x. \text{False}\}$
 $\langle \text{proof} \rangle$

lemma *empty-iff* [*simp*]: $c \in \{\} \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *emptyE* [*elim!*]: $a \in \{\} \implies P$
 $\langle \text{proof} \rangle$

lemma *empty-subsetI* [*iff*]: $\{\} \subseteq A$
 — One effect is to delete the ASSUMPTION $\{\} \subseteq A$
 $\langle \text{proof} \rangle$

lemma *equals0I*: $(\bigwedge y. y \in A \implies \text{False}) \implies A = \{\}$
 $\langle \text{proof} \rangle$

lemma *equals0D*: $A = \{\} \implies a \notin A$
 — Use for reasoning about disjointness: $A \cap B = \{\}$
 $\langle \text{proof} \rangle$

lemma *ball-empty* [*simp*]: $\text{Ball } \{\} P \longleftrightarrow \text{True}$
 $\langle \text{proof} \rangle$

lemma *bex-empty* [*simp*]: $\text{Bex } \{\} P \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

8.3.4 The universal set – UNIV

abbreviation *UNIV* :: ‘a set
 where *UNIV* $\equiv \text{top}$

lemma *UNIV-def*: $\text{UNIV} = \{x. \text{True}\}$
 $\langle \text{proof} \rangle$

lemma *UNIV-I* [*simp*]: $x \in \text{UNIV}$
 $\langle \text{proof} \rangle$

declare *UNIV-I* [*intro*] — unsafe makes it less likely to cause problems

lemma *UNIV-witness* [*intro?*]: $\exists x. x \in \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *subset-UNIV*: $A \subseteq \text{UNIV}$
 $\langle \text{proof} \rangle$

Eta-contracting these two rules (to remove P) causes them to be ignored because of their interaction with congruence rules.

lemma *ball-UNIV* [*simp*]: $Ball\ UNIV\ P \longleftrightarrow All\ P$
 $\langle proof \rangle$

lemma *bex-UNIV* [*simp*]: $Bex\ UNIV\ P \longleftrightarrow Ex\ P$
 $\langle proof \rangle$

lemma *UNIV-eq-I*: $(\bigwedge x. x \in A) \implies UNIV = A$
 $\langle proof \rangle$

lemma *UNIV-not-empty* [*iff*]: $UNIV \neq \{\}$
 $\langle proof \rangle$

lemma *empty-not-UNIV* [*simp*]: $\{\} \neq UNIV$
 $\langle proof \rangle$

8.3.5 The Powerset operator – Pow

definition *Pow* :: $'a\ set \Rightarrow 'a\ set\ set$
where *Pow-def*: $Pow\ A = \{B. B \subseteq A\}$

lemma *Pow-iff* [*iff*]: $A \in Pow\ B \longleftrightarrow A \subseteq B$
 $\langle proof \rangle$

lemma *PowI*: $A \subseteq B \implies A \in Pow\ B$
 $\langle proof \rangle$

lemma *PowD*: $A \in Pow\ B \implies A \subseteq B$
 $\langle proof \rangle$

lemma *Pow-bottom*: $\{\} \in Pow\ B$
 $\langle proof \rangle$

lemma *Pow-top*: $A \in Pow\ A$
 $\langle proof \rangle$

lemma *Pow-not-empty*: $Pow\ A \neq \{\}$
 $\langle proof \rangle$

8.3.6 Set complement

lemma *Compl-iff* [*simp*]: $c \in -\ A \longleftrightarrow c \notin A$
 $\langle proof \rangle$

lemma *ComplI* [*intro!*]: $(c \in A \implies False) \implies c \in -\ A$
 $\langle proof \rangle$

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* [*dest!*]: $c \in - A \implies c \notin A$
 $\langle proof \rangle$

lemmas *ComplE* = *ComplD* [*elim-format*]

lemma *Compl-eq*: $- A = \{x. \neg x \in A\}$
 $\langle proof \rangle$

8.3.7 Binary intersection

abbreviation *inter* :: 'a set \Rightarrow 'a set \Rightarrow 'a set (**infixl** $\langle \cap \rangle$ 70)
where $(\cap) \equiv inf$

notation (*ASCII*)
inter (**infixl** $\langle Int \rangle$ 70)

lemma *Int-def*: $A \cap B = \{x. x \in A \wedge x \in B\}$
 $\langle proof \rangle$

lemma *Int-iff* [*simp*]: $c \in A \cap B \longleftrightarrow c \in A \wedge c \in B$
 $\langle proof \rangle$

lemma *IntI* [*intro!*]: $c \in A \implies c \in B \implies c \in A \cap B$
 $\langle proof \rangle$

lemma *IntD1*: $c \in A \cap B \implies c \in A$
 $\langle proof \rangle$

lemma *IntD2*: $c \in A \cap B \implies c \in B$
 $\langle proof \rangle$

lemma *IntE* [*elim!*]: $c \in A \cap B \implies (c \in A \implies c \in B \implies P) \implies P$
 $\langle proof \rangle$

8.3.8 Binary union

abbreviation *union* :: 'a set \Rightarrow 'a set \Rightarrow 'a set (**infixl** $\langle \cup \rangle$ 65)
where *union* $\equiv sup$

notation (*ASCII*)
union (**infixl** $\langle Un \rangle$ 65)

lemma *Un-def*: $A \cup B = \{x. x \in A \vee x \in B\}$
 $\langle proof \rangle$

lemma *Un-iff* [*simp*]: $c \in A \cup B \longleftrightarrow c \in A \vee c \in B$
 $\langle proof \rangle$

lemma *UnI1* [*elim?*]: $c \in A \implies c \in A \cup B$
 $\langle proof \rangle$

lemma *UnI2* [*elim?*]: $c \in B \implies c \in A \cup B$
 $\langle \text{proof} \rangle$

Classical introduction rule: no commitment to A vs. B .

lemma *UnCI* [*intro!*]: $(c \notin B \implies c \in A) \implies c \in A \cup B$
 $\langle \text{proof} \rangle$

lemma *UnE* [*elim!*]: $c \in A \cup B \implies (c \in A \implies P) \implies (c \in B \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *insert-def*: $\text{insert } a \ B = \{x. x = a\} \cup B$
 $\langle \text{proof} \rangle$

8.3.9 Set difference

lemma *Diff-iff* [*simp*]: $c \in A - B \longleftrightarrow c \in A \wedge c \notin B$
 $\langle \text{proof} \rangle$

lemma *DiffI* [*intro!*]: $c \in A \implies c \notin B \implies c \in A - B$
 $\langle \text{proof} \rangle$

lemma *DiffD1*: $c \in A - B \implies c \in A$
 $\langle \text{proof} \rangle$

lemma *DiffD2*: $c \in A - B \implies c \in B \implies P$
 $\langle \text{proof} \rangle$

lemma *DiffE* [*elim!*]: $c \in A - B \implies (c \in A \implies c \notin B \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *set-diff-eq*: $A - B = \{x. x \in A \wedge x \notin B\}$
 $\langle \text{proof} \rangle$

lemma *Compl-eq-Diff-UNIV*: $- A = (UNIV - A)$
 $\langle \text{proof} \rangle$

abbreviation *sym-diff* :: $'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$ **where**
 $\text{sym-diff } A \ B \equiv ((A - B) \cup (B - A))$

8.3.10 Augmenting a set – insert

lemma *insert-iff* [*simp*]: $a \in \text{insert } b \ A \longleftrightarrow a = b \vee a \in A$
 $\langle \text{proof} \rangle$

lemma *insertI1*: $a \in \text{insert } a \ B$
 $\langle \text{proof} \rangle$

lemma *insertI2*: $a \in B \implies a \in \text{insert } b \ B$

$\langle \text{proof} \rangle$

lemma *insertE* [*elim!*]: $a \in \text{insert } b \ A \implies (a = b \implies P) \implies (a \in A \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *insertCI* [*intro!*]: $(a \notin B \implies a = b) \implies a \in \text{insert } b \ B$
 — Classical introduction rule.
 $\langle \text{proof} \rangle$

lemma *subset-insert-iff*: $A \subseteq \text{insert } x \ B \longleftrightarrow (\text{if } x \in A \text{ then } A - \{x\} \subseteq B \text{ else } A \subseteq B)$
 $\langle \text{proof} \rangle$

lemma *set-insert*:
 assumes $x \in A$
 obtains B where $A = \text{insert } x \ B$ and $x \notin B$
 $\langle \text{proof} \rangle$

lemma *insert-ident*: $x \notin A \implies x \notin B \implies \text{insert } x \ A = \text{insert } x \ B \longleftrightarrow A = B$
 $\langle \text{proof} \rangle$

lemma *insert-eq-iff*:
 assumes $a \notin A \ b \notin B$
 shows $\text{insert } a \ A = \text{insert } b \ B \longleftrightarrow$
 ($\text{if } a = b \text{ then } A = B \text{ else } \exists C. A = \text{insert } b \ C \wedge b \notin C \wedge B = \text{insert } a \ C \wedge a \notin C$)
 (is $?L \longleftrightarrow ?R$)
 $\langle \text{proof} \rangle$

lemma *insert-UNIV*[*simp*]: $\text{insert } x \ UNIV = UNIV$
 $\langle \text{proof} \rangle$

8.3.11 Singletons, using insert

lemma *singletonI* [*intro!*]: $a \in \{a\}$
 — Redundant? But unlike *insertCI*, it proves the subgoal immediately!
 $\langle \text{proof} \rangle$

lemma *singletonD* [*dest!*]: $b \in \{a\} \implies b = a$
 $\langle \text{proof} \rangle$

lemmas $\text{singletonE} = \text{singletonD}$ [*elim-format*]

lemma *singleton-iff*: $b \in \{a\} \longleftrightarrow b = a$
 $\langle \text{proof} \rangle$

lemma *singleton-inject* [*dest!*]: $\{a\} = \{b\} \implies a = b$
 $\langle \text{proof} \rangle$

lemma *singleton-insert-inj-eq* [iff]: $\{b\} = \text{insert } a \ A \longleftrightarrow a = b \wedge A \subseteq \{b\}$
 ⟨proof⟩

lemma *singleton-insert-inj-eq'* [iff]: $\text{insert } a \ A = \{b\} \longleftrightarrow a = b \wedge A \subseteq \{b\}$
 ⟨proof⟩

lemma *subset-singletonD*: $A \subseteq \{x\} \implies A = \{\} \vee A = \{x\}$
 ⟨proof⟩

lemma *subset-singleton-iff*: $X \subseteq \{a\} \longleftrightarrow X = \{\} \vee X = \{a\}$
 ⟨proof⟩

lemma *subset-singleton-iff-Uniq*: $(\exists a. A \subseteq \{a\}) \longleftrightarrow (\exists \leq_1 x. x \in A)$
 ⟨proof⟩

lemma *singleton-conv* [simp]: $\{x. x = a\} = \{a\}$
 ⟨proof⟩

lemma *singleton-conv2* [simp]: $\{x. a = x\} = \{a\}$
 ⟨proof⟩

lemma *Diff-single-insert*: $A - \{x\} \subseteq B \implies A \subseteq \text{insert } x \ B$
 ⟨proof⟩

lemma *subset-Diff-insert*: $A \subseteq B - \text{insert } x \ C \longleftrightarrow A \subseteq B - C \wedge x \notin A$
 ⟨proof⟩

lemma *doubleton-eq-iff*: $\{a, b\} = \{c, d\} \longleftrightarrow a = c \wedge b = d \vee a = d \wedge b = c$
 ⟨proof⟩

lemma *Un-singleton-iff*: $A \cup B = \{x\} \longleftrightarrow A = \{\} \wedge B = \{x\} \vee A = \{x\} \wedge B = \{\} \vee A = \{x\} \wedge B = \{x\}$
 ⟨proof⟩

lemma *singleton-Un-iff*: $\{x\} = A \cup B \longleftrightarrow A = \{\} \wedge B = \{x\} \vee A = \{x\} \wedge B = \{\} \vee A = \{x\} \wedge B = \{x\}$
 ⟨proof⟩

8.3.12 Image of a set under a function

Frequently b does not have the syntactic form of $f \ x$.

definition *image* :: $('a \Rightarrow 'b) \Rightarrow 'a \ \text{set} \Rightarrow 'b \ \text{set}$ (infixr \hookrightarrow 90)
 where $f \hookrightarrow A = \{y. \exists x \in A. y = f \ x\}$

lemma *image-eqI* [simp, intro]: $b = f \ x \implies x \in A \implies b \in f \hookrightarrow A$
 ⟨proof⟩

lemma *imageI*: $x \in A \implies f \ x \in f \hookrightarrow A$

$\langle proof \rangle$

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 .

$\langle proof \rangle$

lemma *imageE* [*elim!*]:

assumes $b \in (\lambda x. f\ x)\ ' A$ — The eta-expansion gives variable-name preservation.

obtains x **where** $b = f\ x$ **and** $x \in A$

$\langle proof \rangle$

lemma *Compr-image-eq*: $\{x \in f\ ' A. P\ x\} = f\ '\{x \in A. P\ (f\ x)\}$

$\langle proof \rangle$

lemma *image-Un*: $f\ '(A \cup B) = f\ ' A \cup f\ ' B$

$\langle proof \rangle$

lemma *image-iff*: $z \in f\ ' A \longleftrightarrow (\exists x \in A. z = f\ x)$

$\langle proof \rangle$

lemma *image-subsetI*: $(\bigwedge x. x \in A \implies f\ x \in B) \implies f\ ' A \subseteq B$

— Replaces the three steps *subsetI*, *imageE*, *hypsubst*, but breaks too many existing proofs.

$\langle proof \rangle$

lemma *image-subset-iff*: $f\ ' A \subseteq B \longleftrightarrow (\forall x \in A. f\ x \in B)$

— This rewrite rule would confuse users if made default.

$\langle proof \rangle$

lemma *subset-imageE*:

assumes $B \subseteq f\ ' A$

obtains C **where** $C \subseteq A$ **and** $B = f\ ' C$

$\langle proof \rangle$

lemma *subset-image-iff*: $B \subseteq f\ ' A \longleftrightarrow (\exists A A \subseteq A. B = f\ ' A A)$

$\langle proof \rangle$

lemma *image-ident* [*simp*]: $(\lambda x. x)\ ' Y = Y$

$\langle proof \rangle$

lemma *image-empty* [*simp*]: $f\ '\{\} = \{\}$

$\langle proof \rangle$

lemma *image-insert* [*simp*]: $f\ '\text{insert } a\ B = \text{insert } (f\ a)\ (f\ ' B)$

$\langle proof \rangle$

lemma *image-constant*: $x \in A \implies (\lambda x. c)\ ' A = \{c\}$

$\langle proof \rangle$

lemma *image-constant-conv*: $(\lambda x. c) \text{ ' } A = (\text{if } A = \{\} \text{ then } \{\} \text{ else } \{c\})$
 ⟨proof⟩

lemma *image-image*: $f \text{ ' } (g \text{ ' } A) = (\lambda x. f (g x)) \text{ ' } A$
 ⟨proof⟩

lemma *insert-image [simp]*: $x \in A \implies \text{insert } (f x) (f \text{ ' } A) = f \text{ ' } A$
 ⟨proof⟩

lemma *image-is-empty [iff]*: $f \text{ ' } A = \{\} \longleftrightarrow A = \{\}$
 ⟨proof⟩

lemma *empty-is-image [iff]*: $\{\} = f \text{ ' } A \longleftrightarrow A = \{\}$
 ⟨proof⟩

lemma *image-Collect*: $f \text{ ' } \{x. P x\} = \{f x \mid x. P x\}$

— 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.

⟨proof⟩

lemma *if-image-distrib [simp]*:
 $(\lambda x. \text{if } P x \text{ then } f x \text{ else } g x) \text{ ' } S = f \text{ ' } (S \cap \{x. P x\}) \cup g \text{ ' } (S \cap \{x. \neg P x\})$
 ⟨proof⟩

lemma *image-cong*:

$f \text{ ' } M = g \text{ ' } N \text{ if } M = N \bigwedge x. x \in N \implies f x = g x$
 ⟨proof⟩

lemma *image-cong-simp [cong]*:

$f \text{ ' } M = g \text{ ' } N \text{ if } M = N \bigwedge x. x \in N =_{\text{simp}} \implies f x = g x$
 ⟨proof⟩

lemma *image-Int-subset*: $f \text{ ' } (A \cap B) \subseteq f \text{ ' } A \cap f \text{ ' } B$
 ⟨proof⟩

lemma *image-diff-subset*: $f \text{ ' } A - f \text{ ' } B \subseteq f \text{ ' } (A - B)$
 ⟨proof⟩

lemma *Setcompr-eq-image*: $\{f x \mid x. x \in A\} = f \text{ ' } A$
 ⟨proof⟩

lemma *setcompr-eq-image*: $\{f x \mid x. P x\} = f \text{ ' } \{x. P x\}$
 ⟨proof⟩

lemma *ball-imageD*: $\forall x \in f \text{ ' } A. P x \implies \forall x \in A. P (f x)$
 ⟨proof⟩

lemma *bex-imageD*: $\exists x \in f \text{ ' } A. P x \implies \exists x \in A. P (f x)$

$\langle \text{proof} \rangle$

lemma *image-add-0* [simp]: $(+) (0 :: 'a :: \text{comm-monoid-add}) \text{ ` } S = S$
 $\langle \text{proof} \rangle$

theorem *Cantors-theorem*: $\nexists f. f \text{ ` } A = \text{Pow } A$
 $\langle \text{proof} \rangle$

Range of a function – just an abbreviation for image!

abbreviation *range* :: $('a \Rightarrow 'b) \Rightarrow 'b \text{ set}$ — of function
where *range* $f \equiv f \text{ ` } \text{UNIV}$

lemma *range-eqI*: $b = f \ x \Longrightarrow b \in \text{range } f$
 $\langle \text{proof} \rangle$

lemma *rangeI*: $f \ x \in \text{range } f$
 $\langle \text{proof} \rangle$

lemma *rangeE* [elim?]: $b \in \text{range } (\lambda x. f \ x) \Longrightarrow (\bigwedge x. b = f \ x \Longrightarrow P) \Longrightarrow P$
 $\langle \text{proof} \rangle$

lemma *range-subsetD*: $\text{range } f \subseteq B \Longrightarrow f \ i \in B$
 $\langle \text{proof} \rangle$

lemma *full-SetCompr-eq*: $\{u. \exists x. u = f \ x\} = \text{range } f$
 $\langle \text{proof} \rangle$

lemma *range-composition*: $\text{range } (\lambda x. f \ (g \ x)) = f \text{ ` } \text{range } g$
 $\langle \text{proof} \rangle$

lemma *range-constant* [simp]: $\text{range } (\lambda _. x) = \{x\}$
 $\langle \text{proof} \rangle$

lemma *range-eq-singletonD*: $\text{range } f = \{a\} \Longrightarrow f \ x = a$
 $\langle \text{proof} \rangle$

8.3.13 Some rules with *if*

Elimination of $\{x. \dots \wedge x = t \wedge \dots\}$.

lemma *Collect-conv-if*: $\{x. x = a \wedge P \ x\} = (\text{if } P \ a \text{ then } \{a\} \text{ else } \{\})$
 $\langle \text{proof} \rangle$

lemma *Collect-conv-if2*: $\{x. a = x \wedge P \ x\} = (\text{if } P \ a \text{ then } \{a\} \text{ else } \{\})$
 $\langle \text{proof} \rangle$

Rewrite rules for boolean case-splitting: faster than *if-split* [split].

lemma *if-split-eq1*: $(\text{if } Q \text{ then } x \text{ else } y) = b \longleftrightarrow (Q \longrightarrow x = b) \wedge (\neg Q \longrightarrow y = b)$

$\langle \text{proof} \rangle$

lemma *if-split-eq2*: $a = (\text{if } Q \text{ then } x \text{ else } y) \longleftrightarrow (Q \longrightarrow a = x) \wedge (\neg Q \longrightarrow a = y)$

$\langle \text{proof} \rangle$

Split ifs on either side of the membership relation. Not for *[simp]* – can cause goals to blow up!

lemma *if-split-mem1*: $(\text{if } Q \text{ then } x \text{ else } y) \in b \longleftrightarrow (Q \longrightarrow x \in b) \wedge (\neg Q \longrightarrow y \in b)$

$\langle \text{proof} \rangle$

lemma *if-split-mem2*: $(a \in (\text{if } Q \text{ then } x \text{ else } y)) \longleftrightarrow (Q \longrightarrow a \in x) \wedge (\neg Q \longrightarrow a \in y)$

$\langle \text{proof} \rangle$

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!*]: $A \subseteq B \Longrightarrow A \neq B \Longrightarrow A \subset B$

$\langle \text{proof} \rangle$

lemma *psubsetE* [*elim!*]: $A \subset B \Longrightarrow (A \subseteq B \Longrightarrow \neg B \subseteq A \Longrightarrow R) \Longrightarrow R$

$\langle \text{proof} \rangle$

lemma *psubset-insert-iff*:

$A \subset \text{insert } x \ B \longleftrightarrow (\text{if } x \in B \text{ then } A \subset B \text{ else if } x \in A \text{ then } A - \{x\} \subset B \text{ else } A \subseteq B)$

$\langle \text{proof} \rangle$

lemma *psubset-eq*: $A \subset B \longleftrightarrow A \subseteq B \wedge A \neq B$

$\langle \text{proof} \rangle$

lemma *psubset-imp-subset*: $A \subset B \Longrightarrow A \subseteq B$

$\langle \text{proof} \rangle$

lemma *psubset-trans*: $A \subset B \Longrightarrow B \subset C \Longrightarrow A \subset C$

$\langle \text{proof} \rangle$

lemma *psubsetD*: $A \subset B \Longrightarrow c \in A \Longrightarrow c \in B$

$\langle \text{proof} \rangle$

lemma *psubset-subset-trans*: $A \subset B \Longrightarrow B \subseteq C \Longrightarrow A \subset C$

$\langle \text{proof} \rangle$

lemma *subset-psubset-trans*: $A \subseteq B \Longrightarrow B \subset C \Longrightarrow A \subset C$

$\langle \text{proof} \rangle$

lemma *psubset-imp-ex-mem*: $A \subseteq B \implies \exists b. b \in B - A$
 $\langle proof \rangle$

lemma *atomize-ball*: $(\bigwedge x. x \in A \implies P x) \equiv Trueprop (\forall x \in A. P x)$
 $\langle proof \rangle$

lemmas $[symmetric, rulify] = atomize-ball$
and $[symmetric, defn] = atomize-ball$

lemma *image-Pow-mono*: $f \text{ ‘ } A \subseteq B \implies image\ f \text{ ‘ } Pow\ A \subseteq Pow\ B$
 $\langle proof \rangle$

lemma *image-Pow-surj*: $f \text{ ‘ } A = B \implies image\ f \text{ ‘ } Pow\ A = Pow\ B$
 $\langle proof \rangle$

8.4.2 Derived rules involving subsets.

insert.

lemma *subset-insertI*: $B \subseteq insert\ a\ B$
 $\langle proof \rangle$

lemma *subset-insertI2*: $A \subseteq B \implies A \subseteq insert\ b\ B$
 $\langle proof \rangle$

lemma *subset-insert*: $x \notin A \implies A \subseteq insert\ x\ B \longleftrightarrow A \subseteq B$
 $\langle proof \rangle$

Finite Union – the least upper bound of two sets.

lemma *Un-upper1*: $A \subseteq A \cup B$
 $\langle proof \rangle$

lemma *Un-upper2*: $B \subseteq A \cup B$
 $\langle proof \rangle$

lemma *Un-least*: $A \subseteq C \implies B \subseteq C \implies A \cup B \subseteq C$
 $\langle proof \rangle$

Finite Intersection – the greatest lower bound of two sets.

lemma *Int-lower1*: $A \cap B \subseteq A$
 $\langle proof \rangle$

lemma *Int-lower2*: $A \cap B \subseteq B$
 $\langle proof \rangle$

lemma *Int-greatest*: $C \subseteq A \implies C \subseteq B \implies C \subseteq A \cap B$
 $\langle proof \rangle$

Set difference.

lemma *Diff-subset[simp]*: $A - B \subseteq A$
 $\langle proof \rangle$

lemma *Diff-subset-conv*: $A - B \subseteq C \longleftrightarrow A \subseteq B \cup C$
 $\langle proof \rangle$

8.4.3 Equalities involving union, intersection, inclusion, etc.

$\{\}$.

lemma *Collect-const [simp]*: $\{s. P\} = (\text{if } P \text{ then } UNIV \text{ else } \{\})$
 — supersedes *Collect-False-empty*
 $\langle proof \rangle$

lemma *subset-empty [simp]*: $A \subseteq \{\} \longleftrightarrow A = \{\}$
 $\langle proof \rangle$

lemma *not-psubset-empty [iff]*: $\neg (A < \{\})$
 $\langle proof \rangle$

lemma *Collect-subset [simp]*: $\{x \in A. P\ x\} \subseteq A \langle proof \rangle$

lemma *Collect-empty-eq [simp]*: $\text{Collect } P = \{\} \longleftrightarrow (\forall x. \neg P\ x)$
 $\langle proof \rangle$

lemma *empty-Collect-eq [simp]*: $\{\} = \text{Collect } P \longleftrightarrow (\forall x. \neg P\ x)$
 $\langle proof \rangle$

lemma *Collect-neg-eq*: $\{x. \neg P\ x\} = - \{x. P\ x\}$
 $\langle proof \rangle$

lemma *Collect-disj-eq*: $\{x. P\ x \vee Q\ x\} = \{x. P\ x\} \cup \{x. Q\ x\}$
 $\langle proof \rangle$

lemma *Collect-imp-eq*: $\{x. P\ x \longrightarrow Q\ x\} = - \{x. P\ x\} \cup \{x. Q\ x\}$
 $\langle proof \rangle$

lemma *Collect-conj-eq*: $\{x. P\ x \wedge Q\ x\} = \{x. P\ x\} \cap \{x. Q\ x\}$
 $\langle proof \rangle$

lemma *Collect-conj-eq2*: $\{x \in A. P\ x \wedge Q\ x\} = \{x \in A. P\ x\} \cap \{x \in A. Q\ x\}$
 $\langle proof \rangle$

lemma *Collect-mono-iff*: $\text{Collect } P \subseteq \text{Collect } Q \longleftrightarrow (\forall x. P\ x \longrightarrow Q\ x)$
 $\langle proof \rangle$

insert.

lemma *insert-is-Un*: $\text{insert } a\ A = \{a\} \cup A$

— NOT SUITABLE FOR REWRITING since $\{a\} \equiv \text{insert } a \ \{\}$
 $\langle \text{proof} \rangle$

lemma *insert-not-empty* [simp]: $\text{insert } a \ A \neq \{\}$
and *empty-not-insert* [simp]: $\{\} \neq \text{insert } a \ A$
 $\langle \text{proof} \rangle$

lemma *insert-absorb*: $a \in A \implies \text{insert } a \ A = A$
 — [simp] causes recursive calls when there are nested inserts
 — with *quadratic* running time
 $\langle \text{proof} \rangle$

lemma *insert-absorb2* [simp]: $\text{insert } x \ (\text{insert } x \ A) = \text{insert } x \ A$
 $\langle \text{proof} \rangle$

lemma *insert-commute*: $\text{insert } x \ (\text{insert } y \ A) = \text{insert } y \ (\text{insert } x \ A)$
 $\langle \text{proof} \rangle$

lemma *insert-subset* [simp]: $\text{insert } x \ A \subseteq B \longleftrightarrow x \in B \wedge A \subseteq B$
 $\langle \text{proof} \rangle$

lemma *mk-disjoint-insert*: $a \in A \implies \exists B. A = \text{insert } a \ B \wedge a \notin B$
 — use new B rather than $A - \{a\}$ to avoid infinite unfolding
 $\langle \text{proof} \rangle$

lemma *insert-Collect*: $\text{insert } a \ (\text{Collect } P) = \{u. u \neq a \longrightarrow P \ u\}$
 $\langle \text{proof} \rangle$

lemma *insert-inter-insert* [simp]: $\text{insert } a \ A \cap \text{insert } a \ B = \text{insert } a \ (A \cap B)$
 $\langle \text{proof} \rangle$

lemma *insert-disjoint* [simp]:
 $\text{insert } a \ A \cap B = \{\} \longleftrightarrow a \notin B \wedge A \cap B = \{\}$
 $\{\} = \text{insert } a \ A \cap B \longleftrightarrow a \notin B \wedge \{\} = A \cap B$
 $\langle \text{proof} \rangle$

lemma *disjoint-insert* [simp]:
 $B \cap \text{insert } a \ A = \{\} \longleftrightarrow a \notin B \wedge B \cap A = \{\}$
 $\{\} = A \cap \text{insert } b \ B \longleftrightarrow b \notin A \wedge \{\} = A \cap B$
 $\langle \text{proof} \rangle$

Int

lemma *Int-absorb*: $A \cap A = A$
 $\langle \text{proof} \rangle$

lemma *Int-left-absorb*: $A \cap (A \cap B) = A \cap B$
 $\langle \text{proof} \rangle$

lemma *Int-commute*: $A \cap B = B \cap A$

$\langle proof \rangle$

lemma *Int-left-commute*: $A \cap (B \cap C) = B \cap (A \cap C)$
 $\langle proof \rangle$

lemma *Int-assoc*: $(A \cap B) \cap C = A \cap (B \cap C)$
 $\langle proof \rangle$

lemmas *Int-ac = Int-assoc Int-left-absorb Int-commute Int-left-commute*
 — Intersection is an AC-operator

lemma *Int-absorb1*: $B \subseteq A \implies A \cap B = B$
 $\langle proof \rangle$

lemma *Int-absorb2*: $A \subseteq B \implies A \cap B = A$
 $\langle proof \rangle$

lemma *Int-empty-left*: $\{\} \cap B = \{\}$
 $\langle proof \rangle$

lemma *Int-empty-right*: $A \cap \{\} = \{\}$
 $\langle proof \rangle$

lemma *disjoint-eq-subset-Compl*: $A \cap B = \{\} \longleftrightarrow A \subseteq - B$
 $\langle proof \rangle$

lemma *disjoint-iff*: $A \cap B = \{\} \longleftrightarrow (\forall x. x \in A \longrightarrow x \notin B)$
 $\langle proof \rangle$

lemma *disjoint-iff-not-equal*: $A \cap B = \{\} \longleftrightarrow (\forall x \in A. \forall y \in B. x \neq y)$
 $\langle proof \rangle$

lemma *Int-UNIV-left*: $UNIV \cap B = B$
 $\langle proof \rangle$

lemma *Int-UNIV-right*: $A \cap UNIV = A$
 $\langle proof \rangle$

lemma *Int-Un-distrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
 $\langle proof \rangle$

lemma *Int-Un-distrib2*: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
 $\langle proof \rangle$

lemma *Int-UNIV*: $A \cap B = UNIV \longleftrightarrow A = UNIV \wedge B = UNIV$
 $\langle proof \rangle$

lemma *Int-subset-iff*: $C \subseteq A \cap B \longleftrightarrow C \subseteq A \wedge C \subseteq B$
 $\langle proof \rangle$

lemma *Int-Collect*: $x \in A \cap \{x. P\ x\} \longleftrightarrow x \in A \wedge P\ x$
 $\langle proof \rangle$

Un.

lemma *Un-absorb*: $A \cup A = A$
 $\langle proof \rangle$

lemma *Un-left-absorb*: $A \cup (A \cup B) = A \cup B$
 $\langle proof \rangle$

lemma *Un-commute*: $A \cup B = B \cup A$
 $\langle proof \rangle$

lemma *Un-left-commute*: $A \cup (B \cup C) = B \cup (A \cup C)$
 $\langle proof \rangle$

lemma *Un-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
 $\langle proof \rangle$

lemmas *Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute*
 — Union is an AC-operator

lemma *Un-absorb1*: $A \subseteq B \implies A \cup B = B$
 $\langle proof \rangle$

lemma *Un-absorb2*: $B \subseteq A \implies A \cup B = A$
 $\langle proof \rangle$

lemma *Un-empty-left*: $\{\} \cup B = B$
 $\langle proof \rangle$

lemma *Un-empty-right*: $A \cup \{\} = A$
 $\langle proof \rangle$

lemma *Un-UNIV-left*: $UNIV \cup B = UNIV$
 $\langle proof \rangle$

lemma *Un-UNIV-right*: $A \cup UNIV = UNIV$
 $\langle proof \rangle$

lemma *Un-insert-left [simp]*: $(insert\ a\ B) \cup C = insert\ a\ (B \cup C)$
 $\langle proof \rangle$

lemma *Un-insert-right [simp]*: $A \cup (insert\ a\ B) = insert\ a\ (A \cup B)$
 $\langle proof \rangle$

lemma *Int-insert-left*: $(insert\ a\ B) \cap C = (if\ a \in C\ then\ insert\ a\ (B \cap C)\ else\ B \cap C)$

$\langle \text{proof} \rangle$

lemma *Int-insert-left-if0* [simp]: $a \notin C \implies (\text{insert } a \ B) \cap C = B \cap C$
 $\langle \text{proof} \rangle$

lemma *Int-insert-left-if1* [simp]: $a \in C \implies (\text{insert } a \ B) \cap C = \text{insert } a \ (B \cap C)$
 $\langle \text{proof} \rangle$

lemma *Int-insert-right*: $A \cap (\text{insert } a \ B) = (\text{if } a \in A \text{ then } \text{insert } a \ (A \cap B) \text{ else } A \cap B)$
 $\langle \text{proof} \rangle$

lemma *Int-insert-right-if0* [simp]: $a \notin A \implies A \cap (\text{insert } a \ B) = A \cap B$
 $\langle \text{proof} \rangle$

lemma *Int-insert-right-if1* [simp]: $a \in A \implies A \cap (\text{insert } a \ B) = \text{insert } a \ (A \cap B)$
 $\langle \text{proof} \rangle$

lemma *Un-Int-distrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
 $\langle \text{proof} \rangle$

lemma *Un-Int-distrib2*: $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$
 $\langle \text{proof} \rangle$

lemma *Un-Int-crazy*: $(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
 $\langle \text{proof} \rangle$

lemma *subset-Un-eq*: $A \subseteq B \longleftrightarrow A \cup B = B$
 $\langle \text{proof} \rangle$

lemma *Un-empty* [iff]: $A \cup B = \{\} \longleftrightarrow A = \{\} \wedge B = \{\}$
 $\langle \text{proof} \rangle$

lemma *Un-subset-iff*: $A \cup B \subseteq C \longleftrightarrow A \subseteq C \wedge B \subseteq C$
 $\langle \text{proof} \rangle$

lemma *Un-Diff-Int*: $(A - B) \cup (A \cap B) = A$
 $\langle \text{proof} \rangle$

lemma *Diff-Int2*: $A \cap C - B \cap C = A \cap C - B$
 $\langle \text{proof} \rangle$

lemma *subset-UnE*:

assumes $C \subseteq A \cup B$

obtains $A' \ B'$ **where** $A' \subseteq A \ B' \subseteq B \ C = A' \cup B'$

$\langle \text{proof} \rangle$

lemma *Un-Int-eq* [*simp*]: $(S \cup T) \cap S = S$ $(S \cup T) \cap T = T$ $S \cap (S \cup T) = S$
 $T \cap (S \cup T) = T$
 ⟨*proof*⟩

lemma *Int-Un-eq* [*simp*]: $(S \cap T) \cup S = S$ $(S \cap T) \cup T = T$ $S \cup (S \cap T) = S$
 $T \cup (S \cap T) = T$
 ⟨*proof*⟩

Set complement

lemma *Compl-disjoint* [*simp*]: $A \cap -A = \{\}$
 ⟨*proof*⟩

lemma *Compl-disjoint2* [*simp*]: $-A \cap A = \{\}$
 ⟨*proof*⟩

lemma *Compl-partition*: $A \cup -A = UNIV$
 ⟨*proof*⟩

lemma *Compl-partition2*: $-A \cup A = UNIV$
 ⟨*proof*⟩

lemma *double-complement*: $-(-A) = A$ **for** $A :: 'a \text{ set}$
 ⟨*proof*⟩

lemma *Compl-Un*: $-(A \cup B) = (-A) \cap (-B)$
 ⟨*proof*⟩

lemma *Compl-Int*: $-(A \cap B) = (-A) \cup (-B)$
 ⟨*proof*⟩

lemma *subset-Compl-self-eq*: $A \subseteq -A \longleftrightarrow A = \{\}$
 ⟨*proof*⟩

lemma *Un-Int-assoc-eq*: $(A \cap B) \cup C = A \cap (B \cup C) \longleftrightarrow C \subseteq A$
 — Halmos, Naive Set Theory, page 16.
 ⟨*proof*⟩

lemma *Compl-UNIV-eq*: $-UNIV = \{\}$
 ⟨*proof*⟩

lemma *Compl-empty-eq*: $-\{\} = UNIV$
 ⟨*proof*⟩

lemma *Compl-subset-Compl-iff* [*iff*]: $-A \subseteq -B \longleftrightarrow B \subseteq A$
 ⟨*proof*⟩

lemma *Compl-eq-Compl-iff* [*iff*]: $-A = -B \longleftrightarrow A = B$
for $A B :: 'a \text{ set}$
 ⟨*proof*⟩

lemma *Compl-insert*: $- \text{insert } x \ A = (- \ A) - \{x\}$
 $\langle \text{proof} \rangle$

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*: $(\forall x \in A \cup B. P \ x) \longleftrightarrow (\forall x \in A. P \ x) \wedge (\forall x \in B. P \ x)$
 $\langle \text{proof} \rangle$

lemma *ball-Un*: $(\exists x \in A \cup B. P \ x) \longleftrightarrow (\exists x \in A. P \ x) \vee (\exists x \in B. P \ x)$
 $\langle \text{proof} \rangle$

Set difference.

lemma *Diff-eq*: $A - B = A \cap (- \ B)$
 $\langle \text{proof} \rangle$

lemma *Diff-eq-empty-iff*: $A - B = \{\} \longleftrightarrow A \subseteq B$
 $\langle \text{proof} \rangle$

lemma *Diff-cancel [simp]*: $A - A = \{\}$
 $\langle \text{proof} \rangle$

lemma *Diff-idemp [simp]*: $(A - B) - B = A - B$
 for $A \ B :: 'a \ \text{set}$
 $\langle \text{proof} \rangle$

lemma *Diff-triv*: $A \cap B = \{\} \implies A - B = A$
 $\langle \text{proof} \rangle$

lemma *empty-Diff [simp]*: $\{\} - A = \{\}$
 $\langle \text{proof} \rangle$

lemma *Diff-empty [simp]*: $A - \{\} = A$
 $\langle \text{proof} \rangle$

lemma *Diff-UNIV [simp]*: $A - \text{UNIV} = \{\}$
 $\langle \text{proof} \rangle$

lemma *Diff-insert0 [simp]*: $x \notin A \implies A - \text{insert } x \ B = A - B$
 $\langle \text{proof} \rangle$

lemma *Diff-insert*: $A - \text{insert } a \ B = A - B - \{a\}$
 — NOT SUITABLE FOR REWRITING since $\{a\} \equiv \text{insert } a \ 0$
 $\langle \text{proof} \rangle$

lemma *Diff-insert2*: $A - \text{insert } a \ B = A - \{a\} - B$
 — NOT SUITABLE FOR REWRITING since $\{a\} \equiv \text{insert } a \ 0$

$\langle proof \rangle$

lemma *insert-Diff-if*: $insert\ x\ A - B = (if\ x \in B\ then\ A - B\ else\ insert\ x\ (A - B))$

$\langle proof \rangle$

lemma *insert-Diff1 [simp]*: $x \in B \implies insert\ x\ A - B = A - B$

$\langle proof \rangle$

lemma *insert-Diff-single[simp]*: $insert\ a\ (A - \{a\}) = insert\ a\ A$

$\langle proof \rangle$

lemma *insert-Diff*: $a \in A \implies insert\ a\ (A - \{a\}) = A$

$\langle proof \rangle$

lemma *Diff-insert-absorb*: $x \notin A \implies (insert\ x\ A) - \{x\} = A$

$\langle proof \rangle$

lemma *Diff-disjoint [simp]*: $A \cap (B - A) = \{\}$

$\langle proof \rangle$

lemma *Diff-partition*: $A \subseteq B \implies A \cup (B - A) = B$

$\langle proof \rangle$

lemma *double-diff*: $A \subseteq B \implies B \subseteq C \implies B - (C - A) = A$

$\langle proof \rangle$

lemma *Un-Diff-cancel [simp]*: $A \cup (B - A) = A \cup B$

$\langle proof \rangle$

lemma *Un-Diff-cancel2 [simp]*: $(B - A) \cup A = B \cup A$

$\langle proof \rangle$

lemma *Diff-Un*: $A - (B \cup C) = (A - B) \cap (A - C)$

$\langle proof \rangle$

lemma *Diff-Int*: $A - (B \cap C) = (A - B) \cup (A - C)$

$\langle proof \rangle$

lemma *Diff-Diff-Int*: $A - (A - B) = A \cap B$

$\langle proof \rangle$

lemma *Un-Diff*: $(A \cup B) - C = (A - C) \cup (B - C)$

$\langle proof \rangle$

lemma *Int-Diff*: $(A \cap B) - C = A \cap (B - C)$

$\langle proof \rangle$

lemma *Diff-Int-distrib*: $C \cap (A - B) = (C \cap A) - (C \cap B)$

$\langle proof \rangle$

lemma *Diff-Int-distrib2*: $(A - B) \cap C = (A \cap C) - (B \cap C)$
 $\langle proof \rangle$

lemma *Diff-Compl [simp]*: $A - (- B) = A \cap B$
 $\langle proof \rangle$

lemma *Compl-Diff-eq [simp]*: $- (A - B) = - A \cup B$
 $\langle proof \rangle$

lemma *subset-Compl-singleton [simp]*: $A \subseteq - \{b\} \longleftrightarrow b \notin A$
 $\langle proof \rangle$

Quantification over type *bool*.

lemma *bool-induct*: $P \text{ True} \Longrightarrow P \text{ False} \Longrightarrow P x$
 $\langle proof \rangle$

lemma *all-bool-eq*: $(\forall b. P b) \longleftrightarrow P \text{ True} \wedge P \text{ False}$
 $\langle proof \rangle$

lemma *bool-contrapos*: $P x \Longrightarrow \neg P \text{ False} \Longrightarrow P \text{ True}$
 $\langle proof \rangle$

lemma *ex-bool-eq*: $(\exists b. P b) \longleftrightarrow P \text{ True} \vee P \text{ False}$
 $\langle proof \rangle$

lemma *UNIV-bool*: $UNIV = \{\text{False}, \text{True}\}$
 $\langle proof \rangle$

Pow

lemma *Pow-empty [simp]*: $Pow \{\} = \{\{\}\}$
 $\langle proof \rangle$

lemma *Pow-singleton-iff [simp]*: $Pow X = \{Y\} \longleftrightarrow X = \{\} \wedge Y = \{\}$
 $\langle proof \rangle$

lemma *Pow-insert*: $Pow (\text{insert } a \ A) = Pow A \cup (\text{insert } a \ ' Pow A)$
 $\langle proof \rangle$

lemma *Pow-Compl*: $Pow (- A) = \{- B \mid B. A \in Pow B\}$
 $\langle proof \rangle$

lemma *Pow-UNIV [simp]*: $Pow UNIV = UNIV$
 $\langle proof \rangle$

lemma *Un-Pow-subset*: $Pow A \cup Pow B \subseteq Pow (A \cup B)$
 $\langle proof \rangle$

lemma *Pow-Int-eq* [*simp*]: $\text{Pow } (A \cap B) = \text{Pow } A \cap \text{Pow } B$
 $\langle \text{proof} \rangle$

Miscellany.

lemma *Int-Diff-disjoint*: $A \cap B \cap (A - B) = \{\}$
 $\langle \text{proof} \rangle$

lemma *Int-Diff-Un*: $A \cap B \cup (A - B) = A$
 $\langle \text{proof} \rangle$

lemma *set-eq-subset*: $A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
 $\langle \text{proof} \rangle$

lemma *subset-iff*: $A \subseteq B \longleftrightarrow (\forall t. t \in A \longrightarrow t \in B)$
 $\langle \text{proof} \rangle$

lemma *subset-iff-psubset-eq*: $A \subseteq B \longleftrightarrow A \subset B \vee A = B$
 $\langle \text{proof} \rangle$

lemma *all-not-in-conv* [*simp*]: $(\forall x. x \notin A) \longleftrightarrow A = \{\}$
 $\langle \text{proof} \rangle$

lemma *ex-in-conv*: $(\exists x. x \in A) \longleftrightarrow A \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *ball-simps* [*simp*, *no-atp*]:
 $\bigwedge A P Q. (\forall x \in A. P x \vee Q) \longleftrightarrow ((\forall x \in A. P x) \vee Q)$
 $\bigwedge A P Q. (\forall x \in A. P \vee Q x) \longleftrightarrow (P \vee (\forall x \in A. Q x))$
 $\bigwedge A P Q. (\forall x \in A. P \longrightarrow Q x) \longleftrightarrow (P \longrightarrow (\forall x \in A. Q x))$
 $\bigwedge A P Q. (\forall x \in A. P x \longrightarrow Q) \longleftrightarrow ((\exists x \in A. P x) \longrightarrow Q)$
 $\bigwedge P. (\forall x \in \{\}. P x) \longleftrightarrow \text{True}$
 $\bigwedge P. (\forall x \in \text{UNIV}. P x) \longleftrightarrow (\forall x. P x)$
 $\bigwedge a B P. (\forall x \in \text{insert } a B. P x) \longleftrightarrow (P a \wedge (\forall x \in B. P x))$
 $\bigwedge P Q. (\forall x \in \text{Collect } Q. P x) \longleftrightarrow (\forall x. Q x \longrightarrow P x)$
 $\bigwedge A P f. (\forall x \in f'A. P x) \longleftrightarrow (\forall x \in A. P (f x))$
 $\bigwedge A P. (\neg (\forall x \in A. P x)) \longleftrightarrow (\exists x \in A. \neg P x)$
 $\langle \text{proof} \rangle$

lemma *bex-simps* [*simp*, *no-atp*]:
 $\bigwedge A P Q. (\exists x \in A. P x \wedge Q) \longleftrightarrow ((\exists x \in A. P x) \wedge Q)$
 $\bigwedge A P Q. (\exists x \in A. P \wedge Q x) \longleftrightarrow (P \wedge (\exists x \in A. Q x))$
 $\bigwedge P. (\exists x \in \{\}. P x) \longleftrightarrow \text{False}$
 $\bigwedge P. (\exists x \in \text{UNIV}. P x) \longleftrightarrow (\exists x. P x)$
 $\bigwedge a B P. (\exists x \in \text{insert } a B. P x) \longleftrightarrow (P a \vee (\exists x \in B. P x))$
 $\bigwedge P Q. (\exists x \in \text{Collect } Q. P x) \longleftrightarrow (\exists x. Q x \wedge P x)$
 $\bigwedge A P f. (\exists x \in f'A. P x) \longleftrightarrow (\exists x \in A. P (f x))$
 $\bigwedge A P. (\neg (\exists x \in A. P x)) \longleftrightarrow (\forall x \in A. \neg P x)$
 $\langle \text{proof} \rangle$

lemma *ex-image-cong-iff* [*simp, no-atp*]:

$$(\exists x. x \in f^{\ast}A) \longleftrightarrow A \neq \{\} \quad (\exists x. x \in f^{\ast}A \wedge P x) \longleftrightarrow (\exists x \in A. P (f x))$$

<proof>

8.4.4 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B \implies f^{\ast} A \subseteq f^{\ast} B$

<proof>

lemma *Pow-mono*: $A \subseteq B \implies \text{Pow } A \subseteq \text{Pow } B$

<proof>

lemma *insert-mono*: $C \subseteq D \implies \text{insert } a \ C \subseteq \text{insert } a \ D$

<proof>

lemma *Un-mono*: $A \subseteq C \implies B \subseteq D \implies A \cup B \subseteq C \cup D$

<proof>

lemma *Int-mono*: $A \subseteq C \implies B \subseteq D \implies A \cap B \subseteq C \cap D$

<proof>

lemma *Diff-mono*: $A \subseteq C \implies D \subseteq B \implies A - B \subseteq C - D$

<proof>

lemma *Compl-anti-mono*: $A \subseteq B \implies - B \subseteq - A$

<proof>

Monotonicity of implications.

lemma *in-mono*: $A \subseteq B \implies x \in A \longrightarrow x \in B$

<proof>

lemma *conj-mono*: $P1 \longrightarrow Q1 \implies P2 \longrightarrow Q2 \implies (P1 \wedge P2) \longrightarrow (Q1 \wedge Q2)$

<proof>

lemma *disj-mono*: $P1 \longrightarrow Q1 \implies P2 \longrightarrow Q2 \implies (P1 \vee P2) \longrightarrow (Q1 \vee Q2)$

<proof>

lemma *imp-mono*: $Q1 \longrightarrow P1 \implies P2 \longrightarrow Q2 \implies (P1 \longrightarrow P2) \longrightarrow (Q1 \longrightarrow Q2)$

<proof>

lemma *imp-refl*: $P \longrightarrow P$ *<proof>*

lemma *not-mono*: $Q \longrightarrow P \implies \neg P \longrightarrow \neg Q$

<proof>

lemma *ex-mono*: $(\bigwedge x. P x \longrightarrow Q x) \implies (\exists x. P x) \longrightarrow (\exists x. Q x)$

<proof>

lemma *all-mono*: $(\bigwedge x. P\ x \longrightarrow Q\ x) \Longrightarrow (\forall x. P\ x) \longrightarrow (\forall x. Q\ x)$
 $\langle proof \rangle$

lemma *Collect-mono*: $(\bigwedge x. P\ x \longrightarrow Q\ x) \Longrightarrow Collect\ P \subseteq Collect\ Q$
 $\langle proof \rangle$

lemma *Int-Collect-mono*: $A \subseteq B \Longrightarrow (\bigwedge x. x \in A \Longrightarrow P\ x \longrightarrow Q\ x) \Longrightarrow A \cap Collect\ P \subseteq B \cap Collect\ Q$
 $\langle proof \rangle$

lemmas *basic-monos* =
subset-refl imp-refl disj-mono conj-mono ex-mono Collect-mono in-mono

lemma *eq-to-mono*: $a = b \Longrightarrow c = d \Longrightarrow b \longrightarrow d \Longrightarrow a \longrightarrow c$
 $\langle proof \rangle$

8.4.5 Inverse image of a function

definition *vimage* :: $('a \Rightarrow 'b) \Rightarrow 'b\ set \Rightarrow 'a\ set$ (**infixr** \leftarrow' 90)
 where $f\ \leftarrow'\ B \equiv \{x. f\ x \in B\}$

lemma *vimage-eq [simp]*: $a \in f\ \leftarrow'\ B \longleftrightarrow f\ a \in B$
 $\langle proof \rangle$

lemma *vimage-singleton-eq*: $a \in f\ \leftarrow'\ \{b\} \longleftrightarrow f\ a = b$
 $\langle proof \rangle$

lemma *vimageI [intro]*: $f\ a = b \Longrightarrow b \in B \Longrightarrow a \in f\ \leftarrow'\ B$
 $\langle proof \rangle$

lemma *vimageI2*: $f\ a \in A \Longrightarrow a \in f\ \leftarrow'\ A$
 $\langle proof \rangle$

lemma *vimageE [elim!]*: $a \in f\ \leftarrow'\ B \Longrightarrow (\bigwedge x. f\ a = x \Longrightarrow x \in B \Longrightarrow P) \Longrightarrow P$
 $\langle proof \rangle$

lemma *vimageD*: $a \in f\ \leftarrow'\ A \Longrightarrow f\ a \in A$
 $\langle proof \rangle$

lemma *vimage-empty [simp]*: $f\ \leftarrow'\ \{\} = \{\}$
 $\langle proof \rangle$

lemma *vimage-Compl*: $f\ \leftarrow'\ (\neg A) = \neg (f\ \leftarrow'\ A)$
 $\langle proof \rangle$

lemma *vimage-Un [simp]*: $f\ \leftarrow'\ (A \cup B) = (f\ \leftarrow'\ A) \cup (f\ \leftarrow'\ B)$
 $\langle proof \rangle$

lemma *vimage-Int* [simp]: $f -' (A \cap B) = (f -' A) \cap (f -' B)$
 ⟨proof⟩

lemma *vimage-Collect-eq* [simp]: $f -' \text{Collect } P = \{y. P (f y)\}$
 ⟨proof⟩

lemma *vimage-Collect*: $(\bigwedge x. P (f x) = Q x) \implies f -' (\text{Collect } P) = \text{Collect } Q$
 ⟨proof⟩

lemma *vimage-insert*: $f -' (\text{insert } a B) = (f -' \{a\}) \cup (f -' B)$
 — NOT suitable for rewriting because of the recurrence of $\{a\}$.
 ⟨proof⟩

lemma *vimage-Diff*: $f -' (A - B) = (f -' A) - (f -' B)$
 ⟨proof⟩

lemma *vimage-UNIV* [simp]: $f -' \text{UNIV} = \text{UNIV}$
 ⟨proof⟩

lemma *vimage-mono*: $A \subseteq B \implies f -' A \subseteq f -' B$
 — monotonicity
 ⟨proof⟩

lemma *vimage-image-eq*: $f -' (f ' A) = \{y. \exists x \in A. f x = y\}$
 ⟨proof⟩

lemma *image-vimage-subset*: $f ' (f -' A) \subseteq A$
 ⟨proof⟩

lemma *image-vimage-eq* [simp]: $f ' (f -' A) = A \cap \text{range } f$
 ⟨proof⟩

lemma *image-subset-iff-subset-vimage*: $f ' A \subseteq B \longleftrightarrow A \subseteq f -' B$
 ⟨proof⟩

lemma *subset-vimage-iff*: $A \subseteq f -' B \longleftrightarrow (\forall x \in A. f x \in B)$
 ⟨proof⟩

lemma *vimage-const* [simp]: $((\lambda x. c) -' A) = (\text{if } c \in A \text{ then } \text{UNIV} \text{ else } \{\})$
 ⟨proof⟩

lemma *vimage-if* [simp]: $((\lambda x. \text{if } x \in B \text{ then } c \text{ else } d) -' A) =$
 $(\text{if } c \in A \text{ then } (\text{if } d \in A \text{ then } \text{UNIV} \text{ else } B)$
 $\text{else if } d \in A \text{ then } - B \text{ else } \{\})$
 ⟨proof⟩

lemma *vimage-inter-cong*: $(\bigwedge w. w \in S \implies f w = g w) \implies f -' y \cap S = g -' y \cap S$
 ⟨proof⟩

lemma *image-ident* [*simp*]: $(\lambda x. x) -' Y = Y$
 $\langle \text{proof} \rangle$

8.4.6 Singleton sets

definition *is-singleton* :: $'a \text{ set} \Rightarrow \text{bool}$
where *is-singleton* $A \longleftrightarrow (\exists x. A = \{x\})$

lemma *is-singletonI* [*simp*, *intro!*]: *is-singleton* $\{x\}$
 $\langle \text{proof} \rangle$

lemma *is-singletonI'*: $A \neq \{\} \Longrightarrow (\bigwedge x y. x \in A \Longrightarrow y \in A \Longrightarrow x = y) \Longrightarrow$
is-singleton A
 $\langle \text{proof} \rangle$

lemma *is-singletonE*: *is-singleton* $A \Longrightarrow (\bigwedge x. A = \{x\} \Longrightarrow P) \Longrightarrow P$
 $\langle \text{proof} \rangle$

lemma *is-singleton-iff-ex1*:
 $\langle \text{is-singleton } A \longleftrightarrow (\exists! x. x \in A) \rangle$
 $\langle \text{proof} \rangle$

8.4.7 Getting the contents of a singleton set

definition *the-elem* :: $'a \text{ set} \Rightarrow 'a$
where *the-elem* $X = (\text{THE } x. X = \{x\})$

lemma *the-elem-eq* [*simp*]: *the-elem* $\{x\} = x$
 $\langle \text{proof} \rangle$

lemma *is-singleton-the-elem*: *is-singleton* $A \longleftrightarrow A = \{\text{the-elem } A\}$
 $\langle \text{proof} \rangle$

lemma *the-elem-image-unique*:
assumes $A \neq \{\}$
and $*$: $\bigwedge y. y \in A \Longrightarrow f y = a$
shows *the-elem* $(f -' A) = a$
 $\langle \text{proof} \rangle$

8.4.8 Monad operation

definition *bind* :: $'a \text{ set} \Rightarrow ('a \Rightarrow 'b \text{ set}) \Rightarrow 'b \text{ set}$
where *bind* $A f = \{x. \exists B \in f'A. x \in B\}$

hide-const (**open**) *bind*

lemma *bind-bind*: *Set.bind* (*Set.bind* $A B$) $C = \text{Set.bind } A (\lambda x. \text{Set.bind } (B x) C)$
for $A :: 'a \text{ set}$

$\langle proof \rangle$

lemma *empty-bind* [simp]: $Set.bind \ \{\} \ f = \{\}$
 $\langle proof \rangle$

lemma *nonempty-bind-const*: $A \neq \{\} \implies Set.bind \ A \ (\lambda-. \ B) = B$
 $\langle proof \rangle$

lemma *bind-const*: $Set.bind \ A \ (\lambda-. \ B) = (if \ A = \{\} \ then \ \{\} \ else \ B)$
 $\langle proof \rangle$

lemma *bind-singleton-conv-image*: $Set.bind \ A \ (\lambda x. \ \{f \ x\}) = f \ ' \ A$
 $\langle proof \rangle$

8.4.9 Operations for execution

Use those operations only for generating executable / efficient code. Otherwise use the RHSs directly.

context
begin

qualified definition *is-empty* :: $'a \ set \Rightarrow bool$ — only for code generation
where *is-empty-iff* [code-abbrev, simp]: $is-empty \ A \longleftrightarrow A = \{\}$

qualified definition *remove* :: $'a \Rightarrow 'a \ set \Rightarrow 'a \ set$ — only for code generation
where *remove-eq* [code-abbrev, simp]: $remove \ x \ A = A - \{x\}$

qualified definition *filter* :: $('a \Rightarrow bool) \Rightarrow 'a \ set \Rightarrow 'a \ set$ — only for code generation
where *filter-eq* [code-abbrev, simp]: $filter \ P \ A = \{a \in A. \ P \ a\}$

qualified definition *can-select* :: $('a \Rightarrow bool) \Rightarrow 'a \ set \Rightarrow bool$ — only for code generation
where *can-select-iff* [code-abbrev, simp]: $can-select \ P \ A = (\exists !x \in A. \ P \ x)$

qualified lemma *can-select-iff-is-singleton*:
 $\langle Set.can-select \ P \ A \longleftrightarrow is-singleton \ (Set.filter \ P \ A) \rangle$
 $\langle proof \rangle$

end

instantiation *set* :: (*equal*) *equal*
begin

definition *HOL.equal* $A \ B \longleftrightarrow A \subseteq B \wedge B \subseteq A$

instance
 $\langle proof \rangle$

end

Misc

definition *pairwise* :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a set \Rightarrow bool
where *pairwise* R S $\longleftrightarrow (\forall x \in S. \forall y \in S. x \neq y \longrightarrow R\ x\ y)$

lemma *pairwise-alt*: *pairwise* R S $\longleftrightarrow (\forall x \in S. \forall y \in S - \{x\}. R\ x\ y)$
 <proof>

lemma *pairwise-trivial* [simp]: *pairwise* ($\lambda i\ j. j \neq i$) I
 <proof>

lemma *pairwiseI* [intro?]:
pairwise R S **if** $\bigwedge x\ y. x \in S \Longrightarrow y \in S \Longrightarrow x \neq y \Longrightarrow R\ x\ y$
 <proof>

lemma *pairwiseD*:
 R x y **and** R y x
if *pairwise* R S x \in S **and** y \in S **and** x \neq y
 <proof>

lemma *pairwise-empty* [simp]: *pairwise* P {}
 <proof>

lemma *pairwise-singleton* [simp]: *pairwise* P {A}
 <proof>

lemma *pairwise-insert*:
pairwise r (insert x s) $\longleftrightarrow (\forall y. y \in s \wedge y \neq x \longrightarrow r\ x\ y \wedge r\ y\ x) \wedge \text{pairwise}\ r\ s$
 <proof>

lemma *pairwise-subset*: *pairwise* P S $\Longrightarrow T \subseteq S \Longrightarrow \text{pairwise}\ P\ T$
 <proof>

lemma *pairwise-mono*: $\llbracket \text{pairwise}\ P\ A; \bigwedge x\ y. P\ x\ y \Longrightarrow Q\ x\ y; B \subseteq A \rrbracket \Longrightarrow \text{pairwise}\ Q\ B$
 <proof>

lemma *pairwise-imageI*:
pairwise P (f ' A)
if $\bigwedge x\ y. x \in A \Longrightarrow y \in A \Longrightarrow x \neq y \Longrightarrow f\ x \neq f\ y \Longrightarrow P\ (f\ x)\ (f\ y)$
 <proof>

lemma *pairwise-image*: *pairwise* r (f ' s) $\longleftrightarrow \text{pairwise}\ (\lambda x\ y. (f\ x \neq f\ y) \longrightarrow r\ (f\ x)\ (f\ y))\ s$
 <proof>

definition *disjnt* :: 'a set \Rightarrow 'a set \Rightarrow bool
where *disjnt* A B $\longleftrightarrow A \cap B = \{\}$

lemma *disjnt-self-iff-empty* [simp]: $\text{disjnt } S \ S \longleftrightarrow S = \{\}$
 ⟨proof⟩

lemma *disjnt-commute*: $\text{disjnt } A \ B = \text{disjnt } B \ A$
 ⟨proof⟩

lemma *disjnt-iff*: $\text{disjnt } A \ B \longleftrightarrow (\forall x. \neg (x \in A \wedge x \in B))$
 ⟨proof⟩

lemma *disjnt-sym*: $\text{disjnt } A \ B \implies \text{disjnt } B \ A$
 ⟨proof⟩

lemma *disjnt-empty1* [simp]: $\text{disjnt } \{\} \ A$ **and** *disjnt-empty2* [simp]: $\text{disjnt } A \ \{\}$
 ⟨proof⟩

lemma *disjnt-insert1* [simp]: $\text{disjnt } (\text{insert } a \ X) \ Y \longleftrightarrow a \notin Y \wedge \text{disjnt } X \ Y$
 ⟨proof⟩

lemma *disjnt-insert2* [simp]: $\text{disjnt } Y \ (\text{insert } a \ X) \longleftrightarrow a \notin Y \wedge \text{disjnt } Y \ X$
 ⟨proof⟩

lemma *disjnt-subset1* : $\llbracket \text{disjnt } X \ Y; Z \subseteq X \rrbracket \implies \text{disjnt } Z \ Y$
 ⟨proof⟩

lemma *disjnt-subset2* : $\llbracket \text{disjnt } X \ Y; Z \subseteq Y \rrbracket \implies \text{disjnt } X \ Z$
 ⟨proof⟩

lemma *disjnt-Un1* [simp]: $\text{disjnt } (A \cup B) \ C \longleftrightarrow \text{disjnt } A \ C \wedge \text{disjnt } B \ C$
 ⟨proof⟩

lemma *disjnt-Un2* [simp]: $\text{disjnt } C \ (A \cup B) \longleftrightarrow \text{disjnt } C \ A \wedge \text{disjnt } C \ B$
 ⟨proof⟩

lemma *disjnt-Diff1*: $\text{disjnt } (X - Y) \ (U - V)$ **and** *disjnt-Diff2*: $\text{disjnt } (U - V) \ (X - Y)$
if $X \subseteq V$
 ⟨proof⟩

lemma *disjoint-image-subset*: $\llbracket \text{pairwise disjnt } \mathcal{A}; \bigwedge X. X \in \mathcal{A} \implies f \ X \subseteq X \rrbracket \implies \text{pairwise disjnt } (f \ \mathcal{A})$
 ⟨proof⟩

lemma *pairwise-disjnt-iff*: $\text{pairwise disjnt } \mathcal{A} \longleftrightarrow (\forall x. \exists_{\leq 1} X. X \in \mathcal{A} \wedge x \in X)$
 ⟨proof⟩

lemma *disjnt-insert*:
 $\langle \text{disjnt } (\text{insert } x \ M) \ N \rangle$ **if** $\langle x \notin N \rangle \langle \text{disjnt } M \ N \rangle$
 ⟨proof⟩

lemma *Int-emptyI*: $(\bigwedge x. x \in A \implies x \in B \implies \text{False}) \implies A \cap B = \{\}$
 $\langle \text{proof} \rangle$

lemma *in-image-insert-iff*:
assumes $\bigwedge C. C \in B \implies x \notin C$
shows $A \in \text{insert } x \text{ ` } B \longleftrightarrow x \in A \wedge A - \{x\} \in B$ (**is** $?P \longleftrightarrow ?Q$)
 $\langle \text{proof} \rangle$

hide-const (**open**) *member not-member*

lemmas *equalityI* = *subset-antisym*

lemmas *set-mp* = *subsetD*

lemmas *set-rev-mp* = *rev-subsetD*

end

9 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*: $\text{Rep } x \in A$
and *Rep-inverse*: $\text{Abs } (\text{Rep } x) = x$
and *Abs-inverse*: $y \in A \implies \text{Rep } (\text{Abs } y) = y$
— This will be axiomatized for each typedef!
begin

lemma *Rep-inject*: $\text{Rep } x = \text{Rep } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *Abs-inject*:
assumes $x \in A$ **and** $y \in A$
shows $\text{Abs } x = \text{Abs } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *Rep-cases* [*cases set*]:
assumes $y \in A$
and *hyp*: $\bigwedge x. y = \text{Rep } x \implies P$
shows P
 $\langle \text{proof} \rangle$

lemma *Abs-cases* [*cases type*]:
assumes r : $\bigwedge y. x = \text{Abs } y \implies y \in A \implies P$

shows P
 $\langle proof \rangle$

lemma *Rep-induct* [*induct set*]:
 assumes $y: y \in A$
 and *hyp*: $\bigwedge x. P (Rep\ x)$
 shows $P\ y$
 $\langle proof \rangle$

lemma *Abs-induct* [*induct type*]:
 assumes $r: \bigwedge y. y \in A \implies P (Abs\ y)$
 shows $P\ x$
 $\langle proof \rangle$

lemma *Rep-range*: $range\ Rep = A$
 $\langle proof \rangle$

lemma *Abs-image*: $Abs\ `A = UNIV$
 $\langle proof \rangle$

end

$\langle ML \rangle$

end

10 Notions about functions

theory *Fun*
 imports *Set*
 keywords *functor* :: *thy-goal-defn*
begin

lemma *apply-inverse*: $f\ x = u \implies (\bigwedge x. P\ x \implies g\ (f\ x) = x) \implies P\ x \implies x = g\ u$
 $\langle proof \rangle$

Uniqueness, so NOT the axiom of choice.

lemma *uniq-choice*: $\forall x. \exists! y. Q\ x\ y \implies \exists f. \forall x. Q\ x\ (f\ x)$
 $\langle proof \rangle$

lemma *b-uniq-choice*: $\forall x \in S. \exists! y. Q\ x\ y \implies \exists f. \forall x \in S. Q\ x\ (f\ x)$
 $\langle proof \rangle$

10.1 The Identity Function *id*

definition *id* :: $'a \Rightarrow 'a$
 where $id = (\lambda x. x)$

lemma *id-apply* [*simp*]: $id\ x = x$
 $\langle proof \rangle$

lemma *image-id* [*simp*]: $image\ id = id$
 $\langle proof \rangle$

lemma *vimage-id* [*simp*]: $vimage\ id = id$
 $\langle proof \rangle$

lemma *eq-id-iff*: $(\forall x. f\ x = x) \longleftrightarrow f = id$
 $\langle proof \rangle$

code-printing

constant *id* $\rightarrow (Haskell)\ id$

10.2 The Composition Operator $f \circ g$

definition *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*)
comp (**infixl** \circ 55)

lemma *comp-apply* [*simp*]: $(f \circ g)\ x = f\ (g\ x)$
 $\langle proof \rangle$

lemma *comp-assoc*: $(f \circ g) \circ h = f \circ (g \circ h)$
 $\langle proof \rangle$

lemma *id-comp* [*simp*]: $id \circ g = g$
 $\langle proof \rangle$

lemma *comp-id* [*simp*]: $f \circ id = f$
 $\langle proof \rangle$

lemma *comp-eq-dest*: $a \circ b = c \circ d \Longrightarrow a\ (b\ v) = c\ (d\ v)$
 $\langle proof \rangle$

lemma *comp-eq-elim*: $a \circ b = c \circ d \Longrightarrow ((\bigwedge v. a\ (b\ v) = c\ (d\ v)) \Longrightarrow R) \Longrightarrow R$
 $\langle proof \rangle$

lemma *comp-eq-dest-lhs*: $a \circ b = c \Longrightarrow a\ (b\ v) = c\ v$
 $\langle proof \rangle$

lemma *comp-eq-id-dest*: $a \circ b = id \circ c \Longrightarrow a\ (b\ v) = c\ v$
 $\langle proof \rangle$

lemma *image-comp*: $f\ ' (g\ ' r) = (f \circ g)\ ' r$
 $\langle proof \rangle$

lemma *image-comp*: $f -' (g -' x) = (g \circ f) -' x$
 $\langle \text{proof} \rangle$

lemma *image-eq-imp-comp*: $f -' A = g -' B \implies (h \circ f) -' A = (h \circ g) -' B$
 $\langle \text{proof} \rangle$

lemma *image-bind*: $f -' (\text{Set.bind } A \ g) = \text{Set.bind } A \ ((\cdot) f \circ g)$
 $\langle \text{proof} \rangle$

lemma *bind-image*: $\text{Set.bind } (f -' A) \ g = \text{Set.bind } A \ (g \circ f)$
 $\langle \text{proof} \rangle$

lemma (in *group-add*) *minus-comp-minus* [simp]: $\text{uminus} \circ \text{uminus} = \text{id}$
 $\langle \text{proof} \rangle$

lemma (in *boolean-algebra*) *minus-comp-minus* [simp]: $\text{uminus} \circ \text{uminus} = \text{id}$
 $\langle \text{proof} \rangle$

code-printing

constant *comp* \rightarrow (*SML*) **infixl** 5 *o* and (*Haskell*) **infixr** 9 .

10.3 The Forward Composition Operator *fcomp*

definition *fcomp* :: $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'c$ (**infixl** $\circ>$ 60)
 where $f \circ> g = (\lambda x. g (f x))$

lemma *fcomp-apply* [simp]: $(f \circ> g) x = g (f x)$
 $\langle \text{proof} \rangle$

lemma *fcomp-assoc*: $(f \circ> g) \circ> h = f \circ> (g \circ> h)$
 $\langle \text{proof} \rangle$

lemma *id-fcomp* [simp]: $\text{id} \circ> g = g$
 $\langle \text{proof} \rangle$

lemma *fcomp-id* [simp]: $f \circ> \text{id} = f$
 $\langle \text{proof} \rangle$

lemma *fcomp-comp*: $fcomp \ f \ g = comp \ g \ f$
 $\langle \text{proof} \rangle$

code-printing

constant *fcomp* \rightarrow (*Eval*) **infixl** 1 $\#>$

no-notation *fcomp* (**infixl** $\circ>$ 60)

10.4 Mapping functions

definition *map-fun* :: $('c \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'd) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow 'd$

where $\text{map-fun } f \ g \ h = g \circ h \circ f$

lemma $\text{map-fun-apply [simp]: map-fun } f \ g \ h \ x = g \ (h \ (f \ x))$
 $\langle \text{proof} \rangle$

10.5 Injectivity and Bijectivity

definition $\text{inj-on} :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \text{ — injective}$
where $\text{inj-on } f \ A \longleftrightarrow (\forall x \in A. \forall y \in A. f \ x = f \ y \longrightarrow x = y)$

definition $\text{bij-betw} :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow \text{bool} \text{ — bijective}$
where $\text{bij-betw } f \ A \ B \longleftrightarrow \text{inj-on } f \ A \wedge f \ 'A = B$

A common special case: functions injective, surjective or bijective over the entire domain type.

abbreviation $\text{inj} :: ('a \Rightarrow 'b) \Rightarrow \text{bool}$
where $\text{inj } f \equiv \text{inj-on } f \ \text{UNIV}$

abbreviation $\text{surj} :: ('a \Rightarrow 'b) \Rightarrow \text{bool}$
where $\text{surj } f \equiv \text{range } f = \text{UNIV}$

translations — The negated case:
 $\neg \text{CONST surj } f \leftarrow \text{CONST range } f \neq \text{CONST UNIV}$

abbreviation $\text{bij} :: ('a \Rightarrow 'b) \Rightarrow \text{bool}$
where $\text{bij } f \equiv \text{bij-betw } f \ \text{UNIV} \ \text{UNIV}$

lemma $\text{inj-def: inj } f \longleftrightarrow (\forall x \ y. f \ x = f \ y \longrightarrow x = y)$
 $\langle \text{proof} \rangle$

lemma $\text{injI: } (\bigwedge x \ y. f \ x = f \ y \Longrightarrow x = y) \Longrightarrow \text{inj } f$
 $\langle \text{proof} \rangle$

theorem $\text{range-ex1-eq: inj } f \Longrightarrow b \in \text{range } f \longleftrightarrow (\exists !x. b = f \ x)$
 $\langle \text{proof} \rangle$

lemma $\text{injD: inj } f \Longrightarrow f \ x = f \ y \Longrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma $\text{inj-on-eq-iff: inj-on } f \ A \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow f \ x = f \ y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma $\text{inj-on-cong: } (\bigwedge a. a \in A \Longrightarrow f \ a = g \ a) \Longrightarrow \text{inj-on } f \ A \longleftrightarrow \text{inj-on } g \ A$
 $\langle \text{proof} \rangle$

lemma $\text{image-strict-mono: inj-on } f \ B \Longrightarrow A \subset B \Longrightarrow f \ 'A \subset f \ 'B$
 $\langle \text{proof} \rangle$

lemma $\text{inj-compose: inj } f \Longrightarrow \text{inj } g \Longrightarrow \text{inj } (f \circ g)$

$\langle proof \rangle$

lemma *inj-fun*: $inj\ f \implies inj\ (\lambda x\ y.\ f\ x)$
 $\langle proof \rangle$

lemma *inj-eq*: $inj\ f \implies f\ x = f\ y \longleftrightarrow x = y$
 $\langle proof \rangle$

lemma *inj-on-iff-Uniq*: $inj-on\ f\ A \longleftrightarrow (\forall x \in A.\ \exists_{\leq 1} y.\ y \in A \wedge f\ x = f\ y)$
 $\langle proof \rangle$

lemma *inj-on-id[simp]*: $inj-on\ id\ A$
 $\langle proof \rangle$

lemma *inj-on-id2[simp]*: $inj-on\ (\lambda x.\ x)\ A$
 $\langle proof \rangle$

lemma *inj-on-Int*: $inj-on\ f\ A \vee inj-on\ f\ B \implies inj-on\ f\ (A \cap B)$
 $\langle proof \rangle$

lemma *surj-id*: $surj\ id$
 $\langle proof \rangle$

lemma *bij-id[simp]*: $bij\ id$
 $\langle proof \rangle$

lemma *bij-uminus*: $bij\ (uminus :: 'a \Rightarrow 'a::group-add)$
 $\langle proof \rangle$

lemma *bij-betwE*: $bij-betw\ f\ A\ B \implies \forall a \in A.\ f\ a \in B$
 $\langle proof \rangle$

lemma *inj-onI [intro?]*: $(\bigwedge x\ y.\ x \in A \implies y \in A \implies f\ x = f\ y \implies x = y) \implies inj-on\ f\ A$
 $\langle proof \rangle$

For those frequent proofs by contradiction

lemma *inj-onCI*: $(\bigwedge x\ y.\ x \in A \implies y \in A \implies f\ x = f\ y \implies x \neq y \implies False) \implies inj-on\ f\ A$
 $\langle proof \rangle$

lemma *inj-on-inverseI*: $(\bigwedge x.\ x \in A \implies g\ (f\ x) = x) \implies inj-on\ f\ A$
 $\langle proof \rangle$

lemma *inj-onD*: $inj-on\ f\ A \implies f\ x = f\ y \implies x \in A \implies y \in A \implies x = y$
 $\langle proof \rangle$

lemma *inj-on-subset*:
 $\llbracket inj-on\ f\ A; B \subseteq A \rrbracket \implies inj-on\ f\ B$

$\langle proof \rangle$

lemma *comp-inj-on*: $inj\text{-}on\ f\ A \implies inj\text{-}on\ g\ (f\ ' A) \implies inj\text{-}on\ (g \circ f)\ A$
 $\langle proof \rangle$

lemma *inj-on-imageI*: $inj\text{-}on\ (g \circ f)\ A \implies inj\text{-}on\ g\ (f\ ' A)$
 $\langle proof \rangle$

lemma *inj-on-image-iff*:
 $\forall x \in A. \forall y \in A. g\ (f\ x) = g\ (f\ y) \longleftrightarrow g\ x = g\ y \implies inj\text{-}on\ f\ A \implies inj\text{-}on\ g\ (f\ ' A) \longleftrightarrow inj\text{-}on\ g\ A$
 $\langle proof \rangle$

lemma *inj-on-contrad*: $inj\text{-}on\ f\ A \implies x \neq y \implies x \in A \implies y \in A \implies f\ x \neq f\ y$
 $\langle proof \rangle$

lemma *inj-singleton [simp]*: $inj\text{-}on\ (\lambda x. \{x\})\ A$
 $\langle proof \rangle$

lemma *inj-on-empty[iff]*: $inj\text{-}on\ f\ \{\}$
 $\langle proof \rangle$

lemma *inj-on-Un*: $inj\text{-}on\ f\ (A \cup B) \longleftrightarrow inj\text{-}on\ f\ A \wedge inj\text{-}on\ f\ B \wedge f\ ' (A - B) \cap f\ ' (B - A) = \{\}$
 $\langle proof \rangle$

lemma *inj-on-insert [iff]*: $inj\text{-}on\ f\ (insert\ a\ A) \longleftrightarrow inj\text{-}on\ f\ A \wedge f\ a \notin f\ ' (A - \{a\})$
 $\langle proof \rangle$

lemma *inj-on-diff*: $inj\text{-}on\ f\ A \implies inj\text{-}on\ f\ (A - B)$
 $\langle proof \rangle$

lemma *comp-inj-on-iff*: $inj\text{-}on\ f\ A \implies inj\text{-}on\ f'\ (f\ ' A) \longleftrightarrow inj\text{-}on\ (f' \circ f)\ A$
 $\langle proof \rangle$

lemma *inj-on-imageI2*: $inj\text{-}on\ (f' \circ f)\ A \implies inj\text{-}on\ f\ A$
 $\langle proof \rangle$

lemma *inj-img-insertE*:
 assumes $inj\text{-}on\ f\ A$
 assumes $x \notin B$
 and $insert\ x\ B = f\ ' A$
 obtains $x' A'$ where $x' \notin A'$ and $A = insert\ x'\ A'$ and $x = f\ x'$ and $B = f\ ' A'$
 $\langle proof \rangle$

lemma *linorder-inj-onI*:
 fixes $A :: 'a::order\ set$

assumes $ne: \bigwedge x y. \llbracket x < y; x \in A; y \in A \rrbracket \implies f x \neq f y$ **and** $lin: \bigwedge x y. \llbracket x \in A; y \in A \rrbracket \implies x \leq y \vee y \leq x$
shows $inj\text{-}on\ f\ A$
 $\langle proof \rangle$

lemma $linorder\text{-}inj\text{-}onI'$:
fixes $A :: 'a :: linorder\ set$
assumes $\bigwedge i j. i \in A \implies j \in A \implies i < j \implies f i \neq f j$
shows $inj\text{-}on\ f\ A$
 $\langle proof \rangle$

lemma $linorder\text{-}injI$:
assumes $\bigwedge x y :: 'a :: linorder. x < y \implies f x \neq f y$
shows $inj\ f$
— Courtesy of Stephan Merz
 $\langle proof \rangle$

lemma $inj\text{-}on\text{-}image\text{-}Pow$: $inj\text{-}on\ f\ A \implies inj\text{-}on\ (image\ f)\ (Pow\ A)$
 $\langle proof \rangle$

lemma $inj\text{-}on\text{-}vimage\text{-}image$: $inj\text{-}on\ (\lambda b. f\ -' \{b\})\ (f\ ' A)$
 $\langle proof \rangle$

lemma $bij\text{-}betw\text{-}image\text{-}Pow$: $bij\text{-}betw\ f\ A\ B \implies bij\text{-}betw\ (image\ f)\ (Pow\ A)\ (Pow\ B)$
 $\langle proof \rangle$

lemma $surj\text{-}def$: $surj\ f \longleftrightarrow (\forall y. \exists x. y = f x)$
 $\langle proof \rangle$

lemma $surjI$:
assumes $\bigwedge x. g\ (f\ x) = x$
shows $surj\ g$
 $\langle proof \rangle$

lemma $surjD$: $surj\ f \implies \exists x. y = f x$
 $\langle proof \rangle$

lemma $surjE$: $surj\ f \implies (\bigwedge x. y = f x \implies C) \implies C$
 $\langle proof \rangle$

lemma $comp\text{-}surj$: $surj\ f \implies surj\ g \implies surj\ (g \circ f)$
 $\langle proof \rangle$

lemma $bij\text{-}betw\text{-}imageI$: $inj\text{-}on\ f\ A \implies f\ ' A = B \implies bij\text{-}betw\ f\ A\ B$
 $\langle proof \rangle$

lemma $bij\text{-}betw\text{-}imp\text{-}surj\text{-}on$: $bij\text{-}betw\ f\ A\ B \implies f\ ' A = B$
 $\langle proof \rangle$

lemma *bij-betw-imp-surj*: $\text{bij-betw } f \ A \ UNIV \Longrightarrow \text{surj } f$
 $\langle \text{proof} \rangle$

lemma *bij-betw-empty1*: $\text{bij-betw } f \ \{\} \ A \Longrightarrow A = \{\}$
 $\langle \text{proof} \rangle$

lemma *bij-betw-empty2*: $\text{bij-betw } f \ A \ \{\} \Longrightarrow A = \{\}$
 $\langle \text{proof} \rangle$

lemma *inj-on-imp-bij-betw*: $\text{inj-on } f \ A \Longrightarrow \text{bij-betw } f \ A \ (f \text{ ‘ } A)$
 $\langle \text{proof} \rangle$

lemma *bij-betw-DiffI*:
assumes $\text{bij-betw } f \ A \ B \ \text{bij-betw } f \ C \ D \ C \subseteq A \ D \subseteq B$
shows $\text{bij-betw } f \ (A - C) \ (B - D)$
 $\langle \text{proof} \rangle$

lemma *bij-betw-singleton-iff* [simp]: $\text{bij-betw } f \ \{x\} \ \{y\} \longleftrightarrow f \ x = y$
 $\langle \text{proof} \rangle$

lemma *bij-betw-singletonI* [intro]: $f \ x = y \Longrightarrow \text{bij-betw } f \ \{x\} \ \{y\}$
 $\langle \text{proof} \rangle$

lemma *bij-betw-imp-empty-iff*: $\text{bij-betw } f \ A \ B \Longrightarrow A = \{\} \longleftrightarrow B = \{\}$
 $\langle \text{proof} \rangle$

lemma *bij-betw-imp-Ex-iff*: $\text{bij-betw } f \ \{x. P \ x\} \ \{x. Q \ x\} \Longrightarrow (\exists x. P \ x) \longleftrightarrow (\exists x. Q \ x)$
 $\langle \text{proof} \rangle$

lemma *bij-betw-imp-Bex-iff*: $\text{bij-betw } f \ \{x \in A. P \ x\} \ \{x \in B. Q \ x\} \Longrightarrow (\exists x \in A. P \ x) \longleftrightarrow (\exists x \in B. Q \ x)$
 $\langle \text{proof} \rangle$

lemma *bij-betw-apply*: $\llbracket \text{bij-betw } f \ A \ B; a \in A \rrbracket \Longrightarrow f \ a \in B$
 $\langle \text{proof} \rangle$

lemma *bij-def*: $\text{bij } f \longleftrightarrow \text{inj } f \wedge \text{surj } f$
 $\langle \text{proof} \rangle$

lemma *bijI*: $\text{inj } f \Longrightarrow \text{surj } f \Longrightarrow \text{bij } f$
 $\langle \text{proof} \rangle$

lemma *bij-is-inj*: $\text{bij } f \Longrightarrow \text{inj } f$
 $\langle \text{proof} \rangle$

lemma *bij-is-surj*: $\text{bij } f \Longrightarrow \text{surj } f$
 $\langle \text{proof} \rangle$

lemma *bij-betw-imp-inj-on*: $\text{bij-betw } f \ A \ B \implies \text{inj-on } f \ A$
 ⟨proof⟩

lemma *bij-betw-trans*: $\text{bij-betw } f \ A \ B \implies \text{bij-betw } g \ B \ C \implies \text{bij-betw } (g \circ f) \ A \ C$
 ⟨proof⟩

lemma *bij-comp*: $\text{bij } f \implies \text{bij } g \implies \text{bij } (g \circ f)$
 ⟨proof⟩

lemma *bij-betw-comp-iff*: $\text{bij-betw } f \ A \ A' \implies \text{bij-betw } f' \ A' \ A'' \longleftrightarrow \text{bij-betw } (f' \circ f) \ A \ A''$
 ⟨proof⟩

lemma *bij-betw-Collect*:
 assumes $\text{bij-betw } f \ A \ B \ \bigwedge x. x \in A \implies Q \ (f \ x) \longleftrightarrow P \ x$
 shows $\text{bij-betw } f \ \{x \in A. P \ x\} \ \{y \in B. Q \ y\}$
 ⟨proof⟩

lemma *bij-betw-comp-iff2*:
 assumes $\text{bij}: \text{bij-betw } f' \ A' \ A''$
 and $\text{img}: f \ ' \ A \leq A'$
 shows $\text{bij-betw } f \ A \ A' \longleftrightarrow \text{bij-betw } (f' \circ f) \ A \ A''$ (is ?L \longleftrightarrow ?R)
 ⟨proof⟩

lemma *bij-betw-inv*:
 assumes $\text{bij-betw } f \ A \ B$
 shows $\exists g. \text{bij-betw } g \ B \ A$
 ⟨proof⟩

lemma *bij-betw-cong*: $(\bigwedge a. a \in A \implies f \ a = g \ a) \implies \text{bij-betw } f \ A \ A' = \text{bij-betw } g \ A \ A'$
 ⟨proof⟩

lemma *bij-betw-id[intro, simp]*: $\text{bij-betw } \text{id} \ A \ A$
 ⟨proof⟩

lemma *bij-betw-id-iff*: $\text{bij-betw } \text{id} \ A \ B \longleftrightarrow A = B$
 ⟨proof⟩

lemma *bij-betw-combine*:
 $\text{bij-betw } f \ A \ B \implies \text{bij-betw } f \ C \ D \implies B \cap D = \{\} \implies \text{bij-betw } f \ (A \cup C) \ (B \cup D)$
 ⟨proof⟩

lemma *bij-betw-subset*: $\text{bij-betw } f \ A \ A' \implies B \subseteq A \implies f \ ' \ B = B' \implies \text{bij-betw } f \ B \ B'$
 ⟨proof⟩

lemma *bij-betw-ball*: $\text{bij-betw } f \ A \ B \implies (\forall b \in B. \text{ phi } b) = (\forall a \in A. \text{ phi } (f \ a))$
 $\langle \text{proof} \rangle$

lemma *bij-pointE*:
assumes *bij f*
obtains *x* **where** $y = f \ x$ **and** $\bigwedge x'. y = f \ x' \implies x' = x$
 $\langle \text{proof} \rangle$

lemma *bij-iff*:
 $\langle \text{bij } f \longleftrightarrow (\forall x. \exists ! y. f \ y = x) \rangle$ **(is** $\langle ?P \longleftrightarrow ?Q \rangle$
 $\langle \text{proof} \rangle$

lemma *bij-betw-partition*:
 $\langle \text{bij-betw } f \ A \ B \rangle$
if $\langle \text{bij-betw } f \ (A \cup C) \ (B \cup D) \rangle$ $\langle \text{bij-betw } f \ C \ D \rangle$ $\langle A \cap C = \{\} \rangle$ $\langle B \cap D = \{\} \rangle$
 $\langle \text{proof} \rangle$

lemma *surj-image-vimage-eq*: $\text{surj } f \implies f \text{ ` } (f \text{ ` } A) = A$
 $\langle \text{proof} \rangle$

lemma *surj-vimage-empty*:
assumes *surj f*
shows $f \text{ ` } A = \{\} \longleftrightarrow A = \{\}$
 $\langle \text{proof} \rangle$

lemma *inj-vimage-image-eq*: $\text{inj } f \implies f \text{ ` } (f \text{ ` } A) = A$
 $\langle \text{proof} \rangle$

lemma *vimage-subsetD*: $\text{surj } f \implies f \text{ ` } B \subseteq A \implies B \subseteq f \text{ ` } A$
 $\langle \text{proof} \rangle$

lemma *vimage-subsetI*: $\text{inj } f \implies B \subseteq f \text{ ` } A \implies f \text{ ` } B \subseteq A$
 $\langle \text{proof} \rangle$

lemma *vimage-subset-eq*: $\text{bij } f \implies f \text{ ` } B \subseteq A \longleftrightarrow B \subseteq f \text{ ` } A$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-eq-iff*: $\text{inj-on } f \ C \implies A \subseteq C \implies B \subseteq C \implies f \text{ ` } A = f \text{ ` } B$
 $\longleftrightarrow A = B$
 $\langle \text{proof} \rangle$

lemma *inj-on-Un-image-eq-iff*: $\text{inj-on } f \ (A \cup B) \implies f \text{ ` } A = f \text{ ` } B \longleftrightarrow A = B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-Int*: $\text{inj-on } f \ C \implies A \subseteq C \implies B \subseteq C \implies f \text{ ` } (A \cap B) = f \text{ ` } A \cap f \text{ ` } B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-set-diff*: $\text{inj-on } f \ C \implies A - B \subseteq C \implies B \subseteq C \implies f \text{ ` } (A$

— $B) = f \circ A - f \circ B$
 $\langle proof \rangle$

lemma *image-Int*: $inj\ f \implies f \circ (A \cap B) = f \circ A \cap f \circ B$
 $\langle proof \rangle$

lemma *image-set-diff*: $inj\ f \implies f \circ (A - B) = f \circ A - f \circ B$
 $\langle proof \rangle$

lemma *inj-on-image-mem-iff*: $inj\text{-on}\ f\ B \implies a \in B \implies A \subseteq B \implies f\ a \in f \circ A$
 $\longleftrightarrow a \in A$
 $\langle proof \rangle$

lemma *inj-image-mem-iff*: $inj\ f \implies f\ a \in f \circ A \longleftrightarrow a \in A$
 $\langle proof \rangle$

lemma *inj-image-subset-iff*: $inj\ f \implies f \circ A \subseteq f \circ B \longleftrightarrow A \subseteq B$
 $\langle proof \rangle$

lemma *inj-image-eq-iff*: $inj\ f \implies f \circ A = f \circ B \longleftrightarrow A = B$
 $\langle proof \rangle$

lemma *surj-Compl-image-subset*: $surj\ f \implies - (f \circ A) \subseteq f \circ (- A)$
 $\langle proof \rangle$

lemma *inj-image-Compl-subset*: $inj\ f \implies f \circ (- A) \subseteq - (f \circ A)$
 $\langle proof \rangle$

lemma *bij-image-Compl-eq*: $bij\ f \implies f \circ (- A) = - (f \circ A)$
 $\langle proof \rangle$

lemma *inj-vimage-singleton*: $inj\ f \implies f^{-1} \{a\} \subseteq \{THE\ x. f\ x = a\}$
 — The inverse image of a singleton under an injective function is included in a singleton.
 $\langle proof \rangle$

lemma *inj-on-vimage-singleton*: $inj\text{-on}\ f\ A \implies f^{-1} \{a\} \cap A \subseteq \{THE\ x. x \in A$
 $\wedge f\ x = a\}$
 $\langle proof \rangle$

lemma *bij-betw-byWitness*:
 assumes *left*: $\forall a \in A. f' (f\ a) = a$
 and *right*: $\forall a' \in A'. f (f'\ a') = a'$
 and $f \circ A \subseteq A'$
 and *img2*: $f' \circ A' \subseteq A$
 shows *bij-betw* $f\ A\ A'$
 $\langle proof \rangle$

corollary *notIn-Un-bij-betw*:

assumes $b \notin A$
and $f b \notin A'$
and $\text{bij-betw } f A A'$
shows $\text{bij-betw } f (A \cup \{b\}) (A' \cup \{f b\})$
 $\langle \text{proof} \rangle$

lemma *notIn-Un-bij-betw3*:
assumes $b \notin A$
and $f b \notin A'$
shows $\text{bij-betw } f A A' = \text{bij-betw } f (A \cup \{b\}) (A' \cup \{f b\})$
 $\langle \text{proof} \rangle$

lemma *inj-on-disjoint-Un*:
assumes $\text{inj-on } f A$ **and** $\text{inj-on } g B$
and $f ' A \cap g ' B = \{\}$
shows $\text{inj-on } (\lambda x. \text{ if } x \in A \text{ then } f x \text{ else } g x) (A \cup B)$
 $\langle \text{proof} \rangle$

lemma *bij-betw-disjoint-Un*:
assumes $\text{bij-betw } f A C$ **and** $\text{bij-betw } g B D$
and $A \cap B = \{\}$
and $C \cap D = \{\}$
shows $\text{bij-betw } (\lambda x. \text{ if } x \in A \text{ then } f x \text{ else } g x) (A \cup B) (C \cup D)$
 $\langle \text{proof} \rangle$

lemma *involuntary-imp-bij*:
 $\langle \text{bij } f \rangle$ **if** $\langle \bigwedge x. f (f x) = x \rangle$
 $\langle \text{proof} \rangle$

10.5.1 Inj/surj/bij of Algebraic Operations

context *cancel-semigroup-add*
begin

lemma *inj-on-add [simp]*:
 $\text{inj-on } ((+) a) A$
 $\langle \text{proof} \rangle$

lemma *inj-on-add' [simp]*:
 $\text{inj-on } (\lambda b. b + a) A$
 $\langle \text{proof} \rangle$

lemma *bij-betw-add [simp]*:
 $\text{bij-betw } ((+) a) A B \longleftrightarrow (+) a ' A = B$
 $\langle \text{proof} \rangle$

end

context *group-add*

begin

lemma *diff-left-imp-eq*: $a - b = a - c \implies b = c$
 $\langle \text{proof} \rangle$

lemma *inj-uminus*[*simp*, *intro*]: *inj-on uminus A*
 $\langle \text{proof} \rangle$

lemma *surj-uminus*[*simp*]: *surj uminus*
 $\langle \text{proof} \rangle$

lemma *surj-plus* [*simp*]:
surj $((+) a)$
 $\langle \text{proof} \rangle$

lemma *surj-plus-right* [*simp*]:
surj $(\lambda b. b + a)$
 $\langle \text{proof} \rangle$

lemma *inj-on-diff-left* [*simp*]:
 $\langle \text{inj-on } ((-) a) A \rangle$
 $\langle \text{proof} \rangle$

lemma *inj-on-diff-right* [*simp*]:
 $\langle \text{inj-on } (\lambda b. b - a) A \rangle$
 $\langle \text{proof} \rangle$

lemma *surj-diff* [*simp*]:
surj $((-) a)$
 $\langle \text{proof} \rangle$

lemma *surj-diff-right* [*simp*]:
surj $(\lambda x. x - a)$
 $\langle \text{proof} \rangle$

lemma shows *bij-plus*: *bij* $((+) a)$ **and** *bij-plus-right*: *bij* $(\lambda x. x + a)$
and *bij-uminus*: *bij uminus*
and *bij-diff*: *bij* $((-) a)$ **and** *bij-diff-right*: *bij* $(\lambda x. x - a)$
 $\langle \text{proof} \rangle$

lemma *translation-subtract-Compl*:
 $(\lambda x. x - a) ' (- t) = - ((\lambda x. x - a) ' t)$
 $\langle \text{proof} \rangle$

lemma *translation-diff*:
 $(+) a ' (s - t) = ((+) a ' s) - ((+) a ' t)$
 $\langle \text{proof} \rangle$

lemma *translation-subtract-diff*:

$(\lambda x. x - a) \text{ ' } (s - t) = ((\lambda x. x - a) \text{ ' } s) - ((\lambda x. x - a) \text{ ' } t)$
 $\langle \text{proof} \rangle$

lemma *translation-Int:*

$(+) a \text{ ' } (s \cap t) = ((+) a \text{ ' } s) \cap ((+) a \text{ ' } t)$
 $\langle \text{proof} \rangle$

lemma *translation-subtract-Int:*

$(\lambda x. x - a) \text{ ' } (s \cap t) = ((\lambda x. x - a) \text{ ' } s) \cap ((\lambda x. x - a) \text{ ' } t)$
 $\langle \text{proof} \rangle$

lemma *translation-Compl:*

$(+) a \text{ ' } (- t) = - ((+) a \text{ ' } t)$
 $\langle \text{proof} \rangle$

end

10.6 Function Updating

definition *fun-upd* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$
where *fun-upd* $f a b = (\lambda x. \text{if } x = a \text{ then } b \text{ else } f x)$

nonterminal *updbinds* and *updbind*

open-bundle *update-syntax*
begin

syntax

-updbind :: $'a \Rightarrow 'a \Rightarrow \text{updbind}$ $(\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix update} \rangle \rangle -$
 $:= / - \rangle \rangle$
 $(\langle - \rangle)$
 $(\langle - \rangle)$
-updbinds :: $\text{updbind} \Rightarrow \text{updbinds} \Rightarrow \text{updbinds}$ $(\langle -, / - \rangle)$
-Update :: $'a \Rightarrow \text{updbinds} \Rightarrow 'a$
 $(\langle \langle \text{open-block notation}=\langle \text{mixfix function update} \rangle \rangle - /'((2-)) \rangle [1000, 0] 900)$

syntax-consts

-Update $\equiv \text{fun-upd}$

translations

-Update $f (-\text{updbinds } b \text{ bs}) \equiv -\text{Update } (-\text{Update } f b) \text{ bs}$
 $f(x:=y) \equiv \text{CONST } \text{fun-upd } f x y$

end

lemma *fun-upd-idem-iff*: $f(x:=y) = f \longleftrightarrow f x = y$
 $\langle \text{proof} \rangle$

lemma *fun-upd-idem*: $f x = y \Longrightarrow f(x := y) = f$
 $\langle \text{proof} \rangle$

lemma *fun-upd-triv* [iff]: $f(x := f x) = f$
 ⟨proof⟩

lemma *fun-upd-apply* [simp]: $(f(x := y)) z = (\text{if } z = x \text{ then } y \text{ else } f z)$
 ⟨proof⟩

lemma *fun-upd-same*: $(f(x := y)) x = y$
 ⟨proof⟩

lemma *fun-upd-other*: $z \neq x \implies (f(x := y)) z = f z$
 ⟨proof⟩

lemma *fun-upd-upd* [simp]: $f(x := y, x := z) = f(x := z)$
 ⟨proof⟩

lemma *fun-upd-twist*: $a \neq c \implies (m(a := b))(c := d) = (m(c := d))(a := b)$
 ⟨proof⟩

lemma *inj-on-fun-updI*: $\text{inj-on } f A \implies y \notin f^{\text{'}} A \implies \text{inj-on } (f(x := y)) A$
 ⟨proof⟩

lemma *fun-upd-image*: $f(x := y)^{\text{'}} A = (\text{if } x \in A \text{ then insert } y (f^{\text{'}} (A - \{x\})) \text{ else } f^{\text{'}} A)$
 ⟨proof⟩

lemma *fun-upd-comp*: $f \circ (g(x := y)) = (f \circ g)(x := f y)$
 ⟨proof⟩

lemma *fun-upd-eqD*: $f(x := y) = g(x := z) \implies y = z$
 ⟨proof⟩

10.7 override-on

definition *override-on* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow 'a \Rightarrow 'b$
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]: $\text{override-on } f g \{\} = f$
 ⟨proof⟩

lemma *override-on-apply-notin*[simp]: $a \notin A \implies (\text{override-on } f g A) a = f a$
 ⟨proof⟩

lemma *override-on-apply-in*[simp]: $a \in A \implies (\text{override-on } f g A) a = g a$
 ⟨proof⟩

lemma *override-on-insert*: $\text{override-on } f g (\text{insert } x X) = (\text{override-on } f g X)(x := g x)$

$\langle \text{proof} \rangle$

lemma *override-on-insert'*: $\text{override-on } f \ g \ (\text{insert } x \ X) = (\text{override-on } (f(x:=g \ x)) \ g \ X)$
 $\langle \text{proof} \rangle$

10.8 Inversion of injective functions

definition *the-inv-into* :: $'a \text{ set} \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$
where *the-inv-into* $A \ f = (\lambda x. \text{THE } y. y \in A \wedge f \ y = x)$

lemma *the-inv-into-f-f*: $\text{inj-on } f \ A \Longrightarrow x \in A \Longrightarrow \text{the-inv-into } A \ f \ (f \ x) = x$
 $\langle \text{proof} \rangle$

lemma *f-the-inv-into-f*: $\text{inj-on } f \ A \Longrightarrow y \in f \ ' \ A \Longrightarrow f \ (\text{the-inv-into } A \ f \ y) = y$
 $\langle \text{proof} \rangle$

lemma *f-the-inv-into-f-bij-betw*:
 $\text{bij-betw } f \ A \ B \Longrightarrow (\text{bij-betw } f \ A \ B \Longrightarrow x \in B) \Longrightarrow f \ (\text{the-inv-into } A \ f \ x) = x$
 $\langle \text{proof} \rangle$

lemma *the-inv-into-into*: $\text{inj-on } f \ A \Longrightarrow x \in f \ ' \ A \Longrightarrow A \subseteq B \Longrightarrow \text{the-inv-into } A \ f \ x \in B$
 $\langle \text{proof} \rangle$

lemma *the-inv-into-onto* [simp]: $\text{inj-on } f \ A \Longrightarrow \text{the-inv-into } A \ f \ ' \ (f \ ' \ A) = A$
 $\langle \text{proof} \rangle$

lemma *the-inv-into-f-eq*: $\text{inj-on } f \ A \Longrightarrow f \ x = y \Longrightarrow x \in A \Longrightarrow \text{the-inv-into } A \ f \ y = x$
 $\langle \text{proof} \rangle$

lemma *the-inv-into-comp*:
 $\text{inj-on } f \ (g \ ' \ A) \Longrightarrow \text{inj-on } g \ A \Longrightarrow x \in f \ ' \ g \ ' \ A \Longrightarrow$
 $\text{the-inv-into } A \ (f \circ g) \ x = (\text{the-inv-into } A \ g \circ \text{the-inv-into } (g \ ' \ A) \ f) \ x$
 $\langle \text{proof} \rangle$

lemma *inj-on-the-inv-into*: $\text{inj-on } f \ A \Longrightarrow \text{inj-on } (\text{the-inv-into } A \ f) \ (f \ ' \ A)$
 $\langle \text{proof} \rangle$

lemma *bij-betw-the-inv-into*: $\text{bij-betw } f \ A \ B \Longrightarrow \text{bij-betw } (\text{the-inv-into } A \ f) \ B \ A$
 $\langle \text{proof} \rangle$

lemma *bij-betw-iff-bijections*:
 $\text{bij-betw } f \ A \ B \longleftrightarrow (\exists g. (\forall x \in A. f \ x \in B \wedge g(f \ x) = x) \wedge (\forall y \in B. g \ y \in A \wedge f(g \ y) = y))$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

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
 <proof>

10.9 Monotonicity

definition *monotone-on* :: 'a set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool
where *monotone-on* A *orda ordb* f \longleftrightarrow ($\forall x \in A. \forall y \in A. \text{orda } x \ y \longrightarrow \text{ordb } (f \ x) \ (f \ y)$)

abbreviation *monotone* :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool
where *monotone* \equiv *monotone-on* UNIV

lemma *monotone-def[no-atp]*: *monotone* *orda ordb* f \longleftrightarrow ($\forall x \ y. \text{orda } x \ y \longrightarrow \text{ordb } (f \ x) \ (f \ y)$)
 <proof>

Lemma *monotone-def* is provided for backward compatibility.

lemma *monotone-onI*:
 ($\bigwedge x \ y. x \in A \implies y \in A \implies \text{orda } x \ y \implies \text{ordb } (f \ x) \ (f \ y)$) \implies *monotone-on* A *orda ordb* f
 <proof>

lemma *monotoneI[intro?]*: ($\bigwedge x \ y. \text{orda } x \ y \implies \text{ordb } (f \ x) \ (f \ y)$) \implies *monotone* *orda ordb* f
 <proof>

lemma *monotone-onD*:
monotone-on A *orda ordb* f $\implies x \in A \implies y \in A \implies \text{orda } x \ y \implies \text{ordb } (f \ x) \ (f \ y)$
 <proof>

lemma *monotoneD[dest?]*: *monotone* *orda ordb* f $\implies \text{orda } x \ y \implies \text{ordb } (f \ x) \ (f \ y)$
 <proof>

lemma *monotone-on-subset*: *monotone-on* A *orda ordb* f $\implies B \subseteq A \implies$ *monotone-on* B *orda ordb* f
 <proof>

lemma *monotone-on-empty[simp]*: *monotone-on* {} *orda ordb* f
 <proof>

lemma *monotone-on-o*:
assumes
mono-f: *monotone-on* A *orda ordb* f **and**

$\text{mono-g: monotone-on } B \text{ ordc } \text{orda } g \text{ and}$
 $g \text{ ' } B \subseteq A$
shows $\text{monotone-on } B \text{ ordc } \text{ordb } (f \circ g)$
 $\langle \text{proof} \rangle$

10.9.1 Specializations For *ord* Type Class And More

context *ord* **begin**

abbreviation $\text{mono-on} :: 'a \text{ set} \Rightarrow ('a \Rightarrow 'b :: \text{ord}) \Rightarrow \text{bool}$
where $\text{mono-on } A \equiv \text{monotone-on } A (\leq) (\leq)$

abbreviation $\text{strict-mono-on} :: 'a \text{ set} \Rightarrow ('a \Rightarrow 'b :: \text{ord}) \Rightarrow \text{bool}$
where $\text{strict-mono-on } A \equiv \text{monotone-on } A (<) (<)$

abbreviation $\text{antimono-on} :: 'a \text{ set} \Rightarrow ('a \Rightarrow 'b :: \text{ord}) \Rightarrow \text{bool}$
where $\text{antimono-on } A \equiv \text{monotone-on } A (\leq) (\lambda x y. y \leq x)$

abbreviation $\text{strict-antimono-on} :: 'a \text{ set} \Rightarrow ('a \Rightarrow 'b :: \text{ord}) \Rightarrow \text{bool}$
where $\text{strict-antimono-on } A \equiv \text{monotone-on } A (<) (\lambda x y. y < x)$

lemma $\text{mono-on-def}[\text{no-atp}]: \text{mono-on } A f \longleftrightarrow (\forall r s. r \in A \wedge s \in A \wedge r \leq s \longrightarrow f r \leq f s)$
 $\langle \text{proof} \rangle$

lemma $\text{strict-mono-on-def}[\text{no-atp}]:$
 $\text{strict-mono-on } A f \longleftrightarrow (\forall r s. r \in A \wedge s \in A \wedge r < s \longrightarrow f r < f s)$
 $\langle \text{proof} \rangle$

Lemmas *mono-on-def* and *strict-mono-on-def* are provided for backward compatibility.

lemma *mono-onI*:
 $(\bigwedge r s. r \in A \Longrightarrow s \in A \Longrightarrow r \leq s \Longrightarrow f r \leq f s) \Longrightarrow \text{mono-on } A f$
 $\langle \text{proof} \rangle$

lemma *strict-mono-onI*:
 $(\bigwedge r s. r \in A \Longrightarrow s \in A \Longrightarrow r < s \Longrightarrow f r < f s) \Longrightarrow \text{strict-mono-on } A f$
 $\langle \text{proof} \rangle$

lemma *mono-onD*: $\llbracket \text{mono-on } A f; r \in A; s \in A; r \leq s \rrbracket \Longrightarrow f r \leq f s$
 $\langle \text{proof} \rangle$

lemma *strict-mono-onD*: $\llbracket \text{strict-mono-on } A f; r \in A; s \in A; r < s \rrbracket \Longrightarrow f r < f s$
 $\langle \text{proof} \rangle$

lemma *mono-on-subset*: $\text{mono-on } A f \Longrightarrow B \subseteq A \Longrightarrow \text{mono-on } B f$
 $\langle \text{proof} \rangle$

end

context *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)
 ⟨proof⟩

lemma *strict-mono-def[no-atp]*: *strict-mono* f ⟷ (∀ x y. x < y ⟶ f x < f y)
 ⟨proof⟩

lemma *antimono-def[no-atp]*: *antimono* f ⟷ (∀ x y. x ≤ y ⟶ f x ≥ f y)
 ⟨proof⟩

Lemmas *mono-def*, *strict-mono-def*, and *antimono-def* are provided for backward compatibility.

lemma *monoI [intro?]*: (λx y. x ≤ y ⟹ f x ≤ f y) ⟹ *mono* f
 ⟨proof⟩

lemma *strict-monoI [intro?]*: (λx y. x < y ⟹ f x < f y) ⟹ *strict-mono* f
 ⟨proof⟩

lemma *antimonoI [intro?]*: (λx y. x ≤ y ⟹ f x ≥ f y) ⟹ *antimono* f
 ⟨proof⟩

lemma *monoD [dest?]*: *mono* f ⟹ x ≤ y ⟹ f x ≤ f y
 ⟨proof⟩

lemma *strict-monoD [dest?]*: *strict-mono* f ⟹ x < y ⟹ f x < f y
 ⟨proof⟩

lemma *antimonoD [dest?]*: *antimono* f ⟹ x ≤ y ⟹ f x ≥ f y
 ⟨proof⟩

lemma *monoE*:
assumes *mono* f
assumes x ≤ y
obtains f x ≤ f y
 ⟨proof⟩

lemma *antimonoE*:
fixes f :: 'a ⇒ 'b::order

```

assumes antimono f
assumes  $x \leq y$ 
obtains  $f\ x \geq f\ y$ 
 $\langle proof \rangle$ 

```

```

end

```

```

lemma mono-imp-mono-on:  $mono\ f \implies mono-on\ A\ f$ 
 $\langle proof \rangle$ 

```

```

lemma strict-mono-on-imp-mono-on:  $strict-mono-on\ A\ f \implies mono-on\ A\ f$ 
for  $f :: 'a::order \Rightarrow 'b::preorder$ 
 $\langle proof \rangle$ 

```

```

lemma strict-mono-mono [dest?]:
 $strict-mono\ f \implies mono\ f$ 
 $\langle proof \rangle$ 

```

```

lemma mono-on-ident:  $mono-on\ S\ (\lambda x. x)$ 
 $\langle proof \rangle$ 

```

```

lemma mono-on-id:  $mono-on\ S\ id$ 
 $\langle proof \rangle$ 

```

```

lemma strict-mono-on-ident:  $strict-mono-on\ S\ (\lambda x. x)$ 
 $\langle proof \rangle$ 

```

```

lemma strict-mono-on-id:  $strict-mono-on\ S\ id$ 
 $\langle proof \rangle$ 

```

```

lemma mono-on-const:
fixes  $a :: 'b::preorder$  shows  $mono-on\ S\ (\lambda x. a)$ 
 $\langle proof \rangle$ 

```

```

lemma antimono-on-const:
fixes  $a :: 'b::preorder$  shows  $antimono-on\ S\ (\lambda x. a)$ 
 $\langle proof \rangle$ 

```

```

context linorder begin

```

```

lemma mono-on-strict-invE:
fixes  $f :: 'a \Rightarrow 'b::preorder$ 
assumes  $mono-on\ S\ f$ 
assumes  $x \in S\ y \in S$ 
assumes  $f\ x < f\ y$ 
obtains  $x < y$ 
 $\langle proof \rangle$ 

```

corollary *mono-on-invE*:

fixes $f :: 'a \Rightarrow 'b::preorder$
assumes *mono-on* S f
assumes $x \in S$ $y \in S$
assumes $f\ x < f\ y$
obtains $x \leq y$
 $\langle proof \rangle$

lemma *strict-mono-on-eq*:

assumes *strict-mono-on* S $(f::'a \Rightarrow 'b::preorder)$
assumes $x \in S$ $y \in S$
shows $f\ x = f\ y \longleftrightarrow x = y$
 $\langle proof \rangle$

lemma *strict-mono-on-less-eq*:

assumes *strict-mono-on* S $(f::'a \Rightarrow 'b::preorder)$
assumes $x \in S$ $y \in S$
shows $f\ x \leq f\ y \longleftrightarrow x \leq y$
 $\langle proof \rangle$

lemma *strict-mono-on-less*:

assumes *strict-mono-on* S $(f::'a \Rightarrow 'b::preorder)$
assumes $x \in S$ $y \in S$
shows $f\ x < f\ y \longleftrightarrow x < y$
 $\langle proof \rangle$

lemmas *mono-invE* = *mono-on-invE*[*OF* - *UNIV-I UNIV-I*]

lemmas *mono-strict-invE* = *mono-on-strict-invE*[*OF* - *UNIV-I UNIV-I*]

lemmas *strict-mono-eq* = *strict-mono-on-eq*[*OF* - *UNIV-I UNIV-I*]

lemmas *strict-mono-less-eq* = *strict-mono-on-less-eq*[*OF* - *UNIV-I UNIV-I*]

lemmas *strict-mono-less* = *strict-mono-on-less*[*OF* - *UNIV-I UNIV-I*]

end

lemma *strict-mono-inv*:

fixes $f :: ('a::linorder) \Rightarrow ('b::linorder)$
assumes *strict-mono* f **and** *surj* f **and** *inv*: $\bigwedge x. g\ (f\ x) = x$
shows *strict-mono* g
 $\langle proof \rangle$

lemma *strict-mono-on-imp-inj-on*:

fixes $f :: 'a::linorder \Rightarrow 'b::preorder$
assumes *strict-mono-on* A f
shows *inj-on* f A
 $\langle proof \rangle$

lemma *strict-mono-on-leD*:

fixes $f :: 'a::order \Rightarrow 'b::preorder$
assumes *strict-mono-on* A f $x \in A$ $y \in A$ $x \leq y$

shows $f\ x \leq f\ y$
 $\langle proof \rangle$

lemma *strict-mono-on-eqD*:
fixes $f :: 'c::linorder \Rightarrow 'd::preorder$
assumes *strict-mono-on* $A\ f\ f\ x = f\ y\ x \in A\ y \in A$
shows $y = x$
 $\langle proof \rangle$

lemma *mono-imp-strict-mono*:
fixes $f :: 'a::order \Rightarrow 'b::order$
shows $\llbracket mono-on\ S\ f; inj-on\ f\ S \rrbracket \Longrightarrow strict-mono-on\ S\ f$
 $\langle proof \rangle$

lemma *strict-mono-iff-mono*:
fixes $f :: 'a::linorder \Rightarrow 'b::order$
shows $strict-mono-on\ S\ f \longleftrightarrow mono-on\ S\ f \wedge inj-on\ f\ S$
 $\langle proof \rangle$

lemma *antimono-imp-strict-antimono*:
fixes $f :: 'a::order \Rightarrow 'b::order$
shows $\llbracket antimono-on\ S\ f; inj-on\ f\ S \rrbracket \Longrightarrow strict-antimono-on\ S\ f$
 $\langle proof \rangle$

lemma *strict-antimono-iff-antimono*:
fixes $f :: 'a::linorder \Rightarrow 'b::order$
shows $strict-antimono-on\ S\ f \longleftrightarrow antimono-on\ S\ f \wedge inj-on\ f\ S$
 $\langle proof \rangle$

lemma *mono-compose*: $mono\ Q \Longrightarrow mono\ (\lambda i\ x. Q\ i\ (f\ x))$
 $\langle proof \rangle$

lemma *mono-add*:
fixes $a :: 'a::ordered-ab-semigroup-add$
shows $mono\ ((+)\ a)$
 $\langle proof \rangle$

lemma (**in** *semilattice-inf*) *mono-inf*: $mono\ f \Longrightarrow f\ (A \sqcap B) \leq f\ A \sqcap f\ B$
for $f :: 'a \Rightarrow 'b::semilattice-inf$
 $\langle proof \rangle$

lemma (**in** *semilattice-sup*) *mono-sup*: $mono\ f \Longrightarrow f\ A \sqcup f\ B \leq f\ (A \sqcup B)$
for $f :: 'a \Rightarrow 'b::semilattice-sup$
 $\langle proof \rangle$

lemma *monotone-on-sup-fun*:
fixes $f\ g :: - \Rightarrow -::semilattice-sup$
shows $monotone-on\ A\ P\ (\leq)\ f \Longrightarrow monotone-on\ A\ P\ (\leq)\ g \Longrightarrow monotone-on\ A\ P\ (\leq)\ (f \sqcup g)$

<proof>

lemma *monotone-on-inf-fun:*

fixes $f\ g :: - \Rightarrow - :: \text{semilattice-inf}$

shows $\text{monotone-on } A\ P\ (\leq)\ f \Longrightarrow \text{monotone-on } A\ P\ (\leq)\ g \Longrightarrow \text{monotone-on } A\ P\ (\leq)\ (f \sqcap g)$

<proof>

lemma *antimonotone-on-sup-fun:*

fixes $f\ g :: - \Rightarrow - :: \text{semilattice-sup}$

shows $\text{monotone-on } A\ P\ (\geq)\ f \Longrightarrow \text{monotone-on } A\ P\ (\geq)\ g \Longrightarrow \text{monotone-on } A\ P\ (\geq)\ (f \sqcup g)$

<proof>

lemma *antimonotone-on-inf-fun:*

fixes $f\ g :: - \Rightarrow - :: \text{semilattice-inf}$

shows $\text{monotone-on } A\ P\ (\geq)\ f \Longrightarrow \text{monotone-on } A\ P\ (\geq)\ g \Longrightarrow \text{monotone-on } A\ P\ (\geq)\ (f \sqcap g)$

<proof>

lemma (**in** *linorder*) *min-of-mono:* $\text{mono } f \Longrightarrow \min (f\ m)\ (f\ n) = f\ (\min\ m\ n)$

<proof>

lemma (**in** *linorder*) *max-of-mono:* $\text{mono } f \Longrightarrow \max (f\ m)\ (f\ n) = f\ (\max\ m\ n)$

<proof>

lemma (**in** *linorder*)

max-of-antimono: $\text{antimono } f \Longrightarrow \max (f\ x)\ (f\ y) = f\ (\min\ x\ y)$ **and**

min-of-antimono: $\text{antimono } f \Longrightarrow \min (f\ x)\ (f\ y) = f\ (\max\ x\ y)$

<proof>

lemma (**in** *linorder*) *strict-mono-imp-inj-on:* $\text{strict-mono } f \Longrightarrow \text{inj-on } f\ A$

<proof>

lemma *mono-Int:* $\text{mono } f \Longrightarrow f\ (A \cap B) \subseteq f\ A \cap f\ B$

<proof>

lemma *mono-Un:* $\text{mono } f \Longrightarrow f\ A \cup f\ B \subseteq f\ (A \cup B)$

<proof>

10.9.2 Least value operator

lemma *Least-mono:* $\text{mono } f \Longrightarrow \exists x \in S. \forall y \in S. x \leq y \Longrightarrow (\text{LEAST } y. y \in f\ 'S) = f\ (\text{LEAST } x. x \in S)$

for $f :: 'a::\text{order} \Rightarrow 'b::\text{order}$

— Courtesy of Stephan Merz

<proof>

10.10 Setup

10.10.1 Proof tools

Simplify terms of the form $f(\dots, x:=y, \dots, x:=z, \dots)$ to $f(\dots, x:=z, \dots)$

$\langle ML \rangle$

10.10.2 Functorial structure of types

$\langle ML \rangle$

functor *map-fun*: *map-fun*
 $\langle proof \rangle$

functor *vimage*
 $\langle proof \rangle$

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 \Rightarrow 'a ($\langle \langle open-block notation = \langle prefix \sqcap \rangle \rangle \sqcap - \rangle$) [900] 900)

class *Sup* =
fixes *Sup* :: 'a set \Rightarrow 'a ($\langle \langle open-block notation = \langle prefix \sqcup \rangle \rangle \sqcup - \rangle$) [900] 900)

syntax
 $-INF1$:: *pttrns* \Rightarrow 'b \Rightarrow 'b ($\langle \langle indent = 3 notation = \langle binder INF \rangle \rangle INF$
 $-./ - \rangle$) [0, 10] 10)
 $-INF$:: *pttrn* \Rightarrow 'a set \Rightarrow 'b \Rightarrow 'b ($\langle \langle indent = 3 notation = \langle binder INF \rangle \rangle INF$
 $-\in ./ - \rangle$) [0, 0, 10] 10)

$-SUP1 \quad :: \text{pttrns} \Rightarrow 'b \Rightarrow 'b \quad (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } SUP \rangle \rangle SUP$
 $-./ -) \rangle [0, 10] 10)$
 $-SUP \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b \quad (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } SUP \rangle \rangle SUP$
 $-\in -./ -) \rangle [0, 0, 10] 10)$

syntax

$-INF1 \quad :: \text{pttrns} \Rightarrow 'b \Rightarrow 'b \quad (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } \sqcap \rangle \rangle \sqcap -./$
 $-) \rangle [0, 10] 10)$
 $-INF \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b \quad (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } \sqcap \rangle \rangle \sqcap -\in -./$
 $-) \rangle [0, 0, 10] 10)$
 $-SUP1 \quad :: \text{pttrns} \Rightarrow 'b \Rightarrow 'b \quad (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } \sqcup \rangle \rangle \sqcup -./$
 $-) \rangle [0, 10] 10)$
 $-SUP \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b \quad (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } \sqcup \rangle \rangle \sqcup -\in -./$
 $-) \rangle [0, 0, 10] 10)$

syntax-consts

$-INF1 -INF \Rightarrow Inf \text{ and}$
 $-SUP1 -SUP \Rightarrow Sup$

translations

$\sqcap x y. f \Rightarrow \sqcap x. \sqcap y. f$
 $\sqcap x. f \Rightarrow \sqcap (CONST \text{ range } (\lambda x. f))$
 $\sqcap x \in A. f \Rightarrow CONST Inf ((\lambda x. f) ' A)$
 $\sqcup x y. f \Rightarrow \sqcup x. \sqcup y. f$
 $\sqcup x. f \Rightarrow \sqcup (CONST \text{ range } (\lambda x. f))$
 $\sqcup x \in A. f \Rightarrow CONST Sup ((\lambda x. f) ' A)$

context *Inf***begin**

lemma *INF-image*: $\sqcap (g ' f ' A) = \sqcap ((g \circ f) ' A)$
 $\langle \text{proof} \rangle$

lemma *INF-identity-eq [simp]*: $(\sqcap x \in A. x) = \sqcap A$
 $\langle \text{proof} \rangle$

lemma *INF-id-eq [simp]*: $\sqcap (id ' A) = \sqcap A$
 $\langle \text{proof} \rangle$

lemma *INF-cong*: $A = B \Longrightarrow (\bigwedge x. x \in B \Longrightarrow C x = D x) \Longrightarrow \sqcap (C ' A) = \sqcap (D ' B)$
 $\langle \text{proof} \rangle$

lemma *INF-cong-simp*:

$A = B \Longrightarrow (\bigwedge x. x \in B = \text{simp} \Rightarrow C x = D x) \Longrightarrow \sqcap (C ' A) = \sqcap (D ' B)$
 $\langle \text{proof} \rangle$

end

context *Sup*
begin

lemma *SUP-image*: $\bigsqcup (g \text{ ‘ } f \text{ ‘ } A) = \bigsqcup ((g \circ f) \text{ ‘ } A)$
 $\langle \text{proof} \rangle$

lemma *SUP-identity-eq* [*simp*]: $(\bigsqcup x \in A. x) = \bigsqcup A$
 $\langle \text{proof} \rangle$

lemma *SUP-id-eq* [*simp*]: $\bigsqcup (id \text{ ‘ } A) = \bigsqcup A$
 $\langle \text{proof} \rangle$

lemma *SUP-cong*: $A = B \implies (\bigwedge x. x \in B \implies C x = D x) \implies \bigsqcup (C \text{ ‘ } A) = \bigsqcup (D \text{ ‘ } B)$
 $\langle \text{proof} \rangle$

lemma *SUP-cong-simp*:
 $A = B \implies (\bigwedge x. x \in B =_{\text{simp}} C x = D x) \implies \bigsqcup (C \text{ ‘ } A) = \bigsqcup (D \text{ ‘ } B)$
 $\langle \text{proof} \rangle$

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*: $x \in A \implies \bigcap A \leq x$
and *Inf-greatest*: $(\bigwedge x. x \in A \implies z \leq x) \implies z \leq \bigcap A$
and *Sup-upper*: $x \in A \implies x \leq \bigsqcup A$
and *Sup-least*: $(\bigwedge x. x \in A \implies x \leq z) \implies \bigsqcup A \leq z$
and *Inf-empty* [*simp*]: $\bigcap \{\} = \top$
and *Sup-empty* [*simp*]: $\bigsqcup \{\} = \perp$
begin

subclass *bounded-lattice*
 $\langle \text{proof} \rangle$

lemma *dual-complete-lattice*: *class.complete-lattice* *Sup* *Inf* *sup* (\geq) ($>$) *inf* \top \perp
 $\langle \text{proof} \rangle$

end

context *complete-lattice*
begin

lemma *Sup-eqI*:

$$\begin{aligned}
& (\bigwedge y. y \in A \implies y \leq x) \implies (\bigwedge y. (\bigwedge z. z \in A \implies z \leq y) \implies x \leq y) \implies \bigsqcup A \\
& = x \\
& \langle proof \rangle
\end{aligned}$$

lemma *Inf-eqI*:

$$\begin{aligned}
& (\bigwedge i. i \in A \implies x \leq i) \implies (\bigwedge y. (\bigwedge i. i \in A \implies y \leq i) \implies y \leq x) \implies \bigcap A = x \\
& \langle proof \rangle
\end{aligned}$$

lemma *SUP-eqI*:

$$\begin{aligned}
& (\bigwedge i. i \in A \implies f i \leq x) \implies (\bigwedge y. (\bigwedge i. i \in A \implies f i \leq y) \implies x \leq y) \implies \\
& (\bigsqcup_{i \in A} f i) = x \\
& \langle proof \rangle
\end{aligned}$$

lemma *INF-eqI*:

$$\begin{aligned}
& (\bigwedge i. i \in A \implies x \leq f i) \implies (\bigwedge y. (\bigwedge i. i \in A \implies f i \geq y) \implies x \geq y) \implies \\
& (\bigcap_{i \in A} f i) = x \\
& \langle proof \rangle
\end{aligned}$$

$$\begin{aligned}
& \textbf{lemma } \textit{INF-lower}: i \in A \implies (\bigcap_{i \in A} f i) \leq f i \\
& \langle proof \rangle
\end{aligned}$$

$$\begin{aligned}
& \textbf{lemma } \textit{INF-greatest}: (\bigwedge i. i \in A \implies u \leq f i) \implies u \leq (\bigcap_{i \in A} f i) \\
& \langle proof \rangle
\end{aligned}$$

$$\begin{aligned}
& \textbf{lemma } \textit{SUP-upper}: i \in A \implies f i \leq (\bigsqcup_{i \in A} f i) \\
& \langle proof \rangle
\end{aligned}$$

$$\begin{aligned}
& \textbf{lemma } \textit{SUP-least}: (\bigwedge i. i \in A \implies f i \leq u) \implies (\bigsqcup_{i \in A} f i) \leq u \\
& \langle proof \rangle
\end{aligned}$$

$$\begin{aligned}
& \textbf{lemma } \textit{Inf-lower2}: u \in A \implies u \leq v \implies \bigcap A \leq v \\
& \langle proof \rangle
\end{aligned}$$

$$\begin{aligned}
& \textbf{lemma } \textit{INF-lower2}: i \in A \implies f i \leq u \implies (\bigcap_{i \in A} f i) \leq u \\
& \langle proof \rangle
\end{aligned}$$

$$\begin{aligned}
& \textbf{lemma } \textit{Sup-upper2}: u \in A \implies v \leq u \implies v \leq \bigsqcup A \\
& \langle proof \rangle
\end{aligned}$$

$$\begin{aligned}
& \textbf{lemma } \textit{SUP-upper2}: i \in A \implies u \leq f i \implies u \leq (\bigsqcup_{i \in A} f i) \\
& \langle proof \rangle
\end{aligned}$$

$$\begin{aligned}
& \textbf{lemma } \textit{le-Inf-iff}: b \leq \bigcap A \longleftrightarrow (\forall a \in A. b \leq a) \\
& \langle proof \rangle
\end{aligned}$$

$$\begin{aligned}
& \textbf{lemma } \textit{le-INF-iff}: u \leq (\bigcap_{i \in A} f i) \longleftrightarrow (\forall i \in A. u \leq f i) \\
& \langle proof \rangle
\end{aligned}$$

$$\textbf{lemma } \textit{Sup-le-iff}: \bigsqcup A \leq b \longleftrightarrow (\forall a \in A. a \leq b)$$

$\langle proof \rangle$

lemma *SUP-le-iff*: $(\bigsqcup_{i \in A} f\ i) \leq u \iff (\forall i \in A. f\ i \leq u)$
 $\langle proof \rangle$

lemma *Inf-insert [simp]*: $\prod (insert\ a\ A) = a \sqcap \prod A$
 $\langle proof \rangle$

lemma *INF-insert*: $(\prod_{x \in insert\ a\ A} f\ x) = f\ a \sqcap \prod (f\ ` A)$
 $\langle proof \rangle$

lemma *Sup-insert [simp]*: $\bigsqcup (insert\ a\ A) = a \sqcup \bigsqcup A$
 $\langle proof \rangle$

lemma *SUP-insert*: $(\bigsqcup_{x \in insert\ a\ A} f\ x) = f\ a \sqcup \bigsqcup (f\ ` A)$
 $\langle proof \rangle$

lemma *INF-empty*: $(\prod_{x \in \{\}} f\ x) = \top$
 $\langle proof \rangle$

lemma *SUP-empty*: $(\bigsqcup_{x \in \{\}} f\ x) = \perp$
 $\langle proof \rangle$

lemma *Inf-UNIV [simp]*: $\prod UNIV = \perp$
 $\langle proof \rangle$

lemma *Sup-UNIV [simp]*: $\bigsqcup UNIV = \top$
 $\langle proof \rangle$

lemma *Inf-eq-Sup*: $\prod A = \bigsqcup \{b. \forall a \in A. b \leq a\}$
 $\langle proof \rangle$

lemma *Sup-eq-Inf*: $\bigsqcup A = \prod \{b. \forall a \in A. a \leq b\}$
 $\langle proof \rangle$

lemma *Inf-superset-mono*: $B \subseteq A \implies \prod A \leq \prod B$
 $\langle proof \rangle$

lemma *Sup-subset-mono*: $A \subseteq B \implies \bigsqcup A \leq \bigsqcup B$
 $\langle proof \rangle$

lemma *Inf-mono*:
assumes $\bigwedge b. b \in B \implies \exists a \in A. a \leq b$
shows $\prod A \leq \prod B$
 $\langle proof \rangle$

lemma *INF-mono*: $(\bigwedge m. m \in B \implies \exists n \in A. f\ n \leq g\ m) \implies (\prod_{n \in A} f\ n) \leq (\prod_{n \in B} g\ n)$
 $\langle proof \rangle$

lemma *INF-mono'*: $(\bigwedge x. f\ x \leq g\ x) \implies (\prod x \in A. f\ x) \leq (\prod x \in A. g\ x)$
 ⟨proof⟩

lemma *Sup-mono*:
 assumes $\bigwedge a. a \in A \implies \exists b \in B. a \leq b$
 shows $\bigsqcup A \leq \bigsqcup B$
 ⟨proof⟩

lemma *SUP-mono*: $(\bigwedge n. n \in A \implies \exists m \in B. f\ n \leq g\ m) \implies (\bigsqcup n \in A. f\ n) \leq (\bigsqcup n \in B. g\ n)$
 ⟨proof⟩

lemma *SUP-mono'*: $(\bigwedge x. f\ x \leq g\ x) \implies (\bigsqcup x \in A. f\ x) \leq (\bigsqcup x \in A. g\ x)$
 ⟨proof⟩

lemma *INF-superset-mono*: $B \subseteq A \implies (\bigwedge x. x \in B \implies f\ x \leq g\ x) \implies (\prod x \in A. f\ x) \leq (\prod x \in B. g\ x)$
 — The last inclusion is POSITIVE!
 ⟨proof⟩

lemma *SUP-subset-mono*: $A \subseteq B \implies (\bigwedge x. x \in A \implies f\ x \leq g\ x) \implies (\bigsqcup x \in A. f\ x) \leq (\bigsqcup x \in B. g\ x)$
 ⟨proof⟩

lemma *Inf-less-eq*:
 assumes $\bigwedge v. v \in A \implies v \leq u$
 and $A \neq \{\}$
 shows $\prod A \leq u$
 ⟨proof⟩

lemma *less-eq-Sup*:
 assumes $\bigwedge v. v \in A \implies u \leq v$
 and $A \neq \{\}$
 shows $u \leq \bigsqcup A$
 ⟨proof⟩

lemma *INF-eq*:
 assumes $\bigwedge i. i \in A \implies \exists j \in B. f\ i \geq g\ j$
 and $\bigwedge j. j \in B \implies \exists i \in A. g\ j \geq f\ i$
 shows $\prod (f\ ' A) = \prod (g\ ' B)$
 ⟨proof⟩

lemma *SUP-eq*:
 assumes $\bigwedge i. i \in A \implies \exists j \in B. f\ i \leq g\ j$
 and $\bigwedge j. j \in B \implies \exists i \in A. g\ j \leq f\ i$
 shows $\bigsqcup (f\ ' A) = \bigsqcup (g\ ' B)$
 ⟨proof⟩

lemma *less-eq-Inf-inter*: $\prod A \sqcup \prod B \leq \prod (A \cap B)$
 $\langle \text{proof} \rangle$

lemma *Sup-inter-less-eq*: $\sqcup (A \cap B) \leq \sqcup A \cap \sqcup B$
 $\langle \text{proof} \rangle$

lemma *Inf-union-distrib*: $\prod (A \cup B) = \prod A \cap \prod B$
 $\langle \text{proof} \rangle$

lemma *INF-union*: $(\prod i \in A \cup B. M i) = (\prod i \in A. M i) \cap (\prod i \in B. M i)$
 $\langle \text{proof} \rangle$

lemma *Sup-union-distrib*: $\sqcup (A \cup B) = \sqcup A \sqcup \sqcup B$
 $\langle \text{proof} \rangle$

lemma *SUP-union*: $(\sqcup i \in A \cup B. M i) = (\sqcup i \in A. M i) \sqcup (\sqcup i \in B. M i)$
 $\langle \text{proof} \rangle$

lemma *INF-inf-distrib*: $(\prod a \in A. f a) \cap (\prod a \in A. g a) = (\prod a \in A. f a \cap g a)$
 $\langle \text{proof} \rangle$

lemma *SUP-sup-distrib*: $(\sqcup a \in A. f a) \sqcup (\sqcup a \in A. g a) = (\sqcup a \in A. f a \sqcup g a)$
 $(\text{is } ?L = ?R)$
 $\langle \text{proof} \rangle$

lemma *Inf-top-conv* [simp]:
 $\prod A = \top \longleftrightarrow (\forall x \in A. x = \top)$
 $\top = \prod A \longleftrightarrow (\forall x \in A. x = \top)$
 $\langle \text{proof} \rangle$

lemma *INF-top-conv* [simp]:
 $(\prod x \in A. B x) = \top \longleftrightarrow (\forall x \in A. B x = \top)$
 $\top = (\prod x \in A. B x) \longleftrightarrow (\forall x \in A. B x = \top)$
 $\langle \text{proof} \rangle$

lemma *Sup-bot-conv* [simp]:
 $\sqcup A = \perp \longleftrightarrow (\forall x \in A. x = \perp)$
 $\perp = \sqcup A \longleftrightarrow (\forall x \in A. x = \perp)$
 $\langle \text{proof} \rangle$

lemma *SUP-bot-conv* [simp]:
 $(\sqcup x \in A. B x) = \perp \longleftrightarrow (\forall x \in A. B x = \perp)$
 $\perp = (\sqcup x \in A. B x) \longleftrightarrow (\forall x \in A. B x = \perp)$
 $\langle \text{proof} \rangle$

lemma *INF-constant*: $(\prod y \in A. c) = (\text{if } A = \{\} \text{ then } \top \text{ else } c)$
 $\langle \text{proof} \rangle$

lemma *SUP-constant*: $(\sqcup y \in A. c) = (\text{if } A = \{\} \text{ then } \perp \text{ else } c)$

$\langle proof \rangle$

lemma *INF-const* [simp]: $A \neq \{\}$ $\implies (\bigcap i \in A. f) = f$
 $\langle proof \rangle$

lemma *SUP-const* [simp]: $A \neq \{\}$ $\implies (\bigcup i \in A. f) = f$
 $\langle proof \rangle$

lemma *INF-top* [simp]: $(\bigcap x \in A. \top) = \top$
 $\langle proof \rangle$

lemma *SUP-bot* [simp]: $(\bigcup x \in A. \bot) = \bot$
 $\langle proof \rangle$

lemma *INF-commute*: $(\bigcap i \in A. \bigcap j \in B. f\ i\ j) = (\bigcap j \in B. \bigcap i \in A. f\ i\ j)$
 $\langle proof \rangle$

lemma *SUP-commute*: $(\bigcup i \in A. \bigcup j \in B. f\ i\ j) = (\bigcup j \in B. \bigcup i \in A. f\ i\ j)$
 $\langle proof \rangle$

lemma *INF-absorb*:
assumes $k \in I$
shows $A\ k \sqcap (\bigcap i \in I. A\ i) = (\bigcap i \in I. A\ i)$
 $\langle proof \rangle$

lemma *SUP-absorb*:
assumes $k \in I$
shows $A\ k \sqcup (\bigcup i \in I. A\ i) = (\bigcup i \in I. A\ i)$
 $\langle proof \rangle$

lemma *INF-inf-const1*: $I \neq \{\}$ $\implies (\bigcap i \in I. \inf\ x\ (f\ i)) = \inf\ x\ (\bigcap i \in I. f\ i)$
 $\langle proof \rangle$

lemma *INF-inf-const2*: $I \neq \{\}$ $\implies (\bigcap i \in I. \inf\ (f\ i)\ x) = \inf\ (\bigcap i \in I. f\ i)\ x$
 $\langle proof \rangle$

lemma *less-INF-D*:
assumes $y < (\bigcap i \in A. f\ i)$ $i \in A$
shows $y < f\ i$
 $\langle proof \rangle$

lemma *SUP-lessD*:
assumes $(\bigcup i \in A. f\ i) < y$ $i \in A$
shows $f\ i < y$
 $\langle proof \rangle$

lemma *INF-UNIV-bool-expand*: $(\bigcap b. A\ b) = A\ True \sqcap A\ False$
 $\langle proof \rangle$

lemma *SUP-UNIV-bool-expand*: $(\bigsqcup b. A \ b) = A \ \text{True} \sqcup A \ \text{False}$
 ⟨proof⟩

lemma *Inf-le-Sup*: $A \neq \{\} \implies \text{Inf } A \leq \text{Sup } A$
 ⟨proof⟩

lemma *INF-le-SUP*: $A \neq \{\} \implies \bigcap (f \text{ ‘ } A) \leq \bigsqcup (f \text{ ‘ } A)$
 ⟨proof⟩

lemma *INF-eq-const*: $I \neq \{\} \implies (\bigwedge i. i \in I \implies f \ i = x) \implies \bigcap (f \text{ ‘ } I) = x$
 ⟨proof⟩

lemma *SUP-eq-const*: $I \neq \{\} \implies (\bigwedge i. i \in I \implies f \ i = x) \implies \bigsqcup (f \text{ ‘ } I) = x$
 ⟨proof⟩

lemma *INF-eq-iff*: $I \neq \{\} \implies (\bigwedge i. i \in I \implies f \ i \leq c) \implies \bigcap (f \text{ ‘ } I) = c \longleftrightarrow (\forall i \in I. f \ i = c)$
 ⟨proof⟩

lemma *SUP-eq-iff*: $I \neq \{\} \implies (\bigwedge i. i \in I \implies c \leq f \ i) \implies \bigsqcup (f \text{ ‘ } I) = c \longleftrightarrow (\forall i \in I. f \ i = c)$
 ⟨proof⟩

end

context *complete-lattice*

begin

lemma *Sup-Inf-le*: $\text{Sup } (\text{Inf } \text{ ‘ } \{f \text{ ‘ } A \mid f . (\forall Y \in A . f \ Y \in Y)\}) \leq \text{Inf } (\text{Sup } \text{ ‘ } A)$
 ⟨proof⟩

end

class *complete-distrib-lattice* = *complete-lattice* +

assumes *Inf-Sup-le*: $\text{Inf } (\text{Sup } \text{ ‘ } A) \leq \text{Sup } (\text{Inf } \text{ ‘ } \{f \text{ ‘ } A \mid f . (\forall Y \in A . f \ Y \in Y)\})$

begin

lemma *Inf-Sup*: $\text{Inf } (\text{Sup } \text{ ‘ } A) = \text{Sup } (\text{Inf } \text{ ‘ } \{f \text{ ‘ } A \mid f . (\forall Y \in A . f \ Y \in Y)\})$
 ⟨proof⟩

subclass *distrib-lattice*

⟨proof⟩

end

context *complete-lattice*

begin

context

fixes $f :: 'a \Rightarrow 'b :: \text{complete-lattice}$

assumes *mono f*

begin

lemma *mono-Inf*: $f (\bigcap A) \leq (\bigcap x \in A. f x)$
 $\langle proof \rangle$

lemma *mono-Sup*: $(\bigcup x \in A. f x) \leq f (\bigcup A)$
 $\langle proof \rangle$

lemma *mono-INF*: $f (\bigcap i \in I. A i) \leq (\bigcap x \in I. f (A x))$
 $\langle proof \rangle$

lemma *mono-SUP*: $(\bigcup x \in I. f (A x)) \leq f (\bigcup i \in I. A i)$
 $\langle proof \rangle$

end

end

class *complete-boolean-algebra* = *boolean-algebra* + *complete-distrib-lattice*
begin

lemma *uminus-Inf*: $-\ (\bigcap A) = \bigcup (uminus \text{ ` } A)$
 $\langle proof \rangle$

lemma *uminus-INF*: $-\ (\bigcap x \in A. B x) = (\bigcup x \in A. - B x)$
 $\langle proof \rangle$

lemma *uminus-Sup*: $-\ (\bigcup A) = \bigcap (uminus \text{ ` } A)$
 $\langle proof \rangle$

lemma *uminus-SUP*: $-\ (\bigcup x \in A. B x) = (\bigcap x \in A. - B x)$
 $\langle proof \rangle$

end

class *complete-linorder* = *linorder* + *complete-lattice*
begin

lemma *dual-complete-linorder*:
class.*complete-linorder* *Sup Inf sup* (\geq) ($>$) *inf* $\top \perp$
 $\langle proof \rangle$

lemma *complete-linorder-inf-min*: $inf = min$
 $\langle proof \rangle$

lemma *complete-linorder-sup-max*: $sup = max$
 $\langle proof \rangle$

lemma *Inf-less-iff*: $\bigcap S < a \longleftrightarrow (\exists x \in S. x < a)$
 $\langle proof \rangle$

lemma *INF-less-iff*: $(\bigcap i \in A. f\ i) < a \longleftrightarrow (\exists x \in A. f\ x < a)$
 $\langle proof \rangle$

lemma *less-Sup-iff*: $a < \bigcup S \longleftrightarrow (\exists x \in S. a < x)$
 $\langle proof \rangle$

lemma *less-SUP-iff*: $a < (\bigcup i \in A. f\ i) \longleftrightarrow (\exists x \in A. a < f\ x)$
 $\langle proof \rangle$

lemma *Sup-eq-top-iff* [simp]: $\bigcup A = \top \longleftrightarrow (\forall x < \top. \exists i \in A. x < i)$
 $\langle proof \rangle$

lemma *SUP-eq-top-iff* [simp]: $(\bigcup i \in A. f\ i) = \top \longleftrightarrow (\forall x < \top. \exists i \in A. x < f\ i)$
 $\langle proof \rangle$

lemma *Inf-eq-bot-iff* [simp]: $\bigcap A = \bot \longleftrightarrow (\forall x > \bot. \exists i \in A. i < x)$
 $\langle proof \rangle$

lemma *INF-eq-bot-iff* [simp]: $(\bigcap i \in A. f\ i) = \bot \longleftrightarrow (\forall x > \bot. \exists i \in A. f\ i < x)$
 $\langle proof \rangle$

lemma *Inf-le-iff*: $\bigcap A \leq x \longleftrightarrow (\forall y > x. \exists a \in A. y > a)$
 $\langle proof \rangle$

lemma *INF-le-iff*: $\bigcap (f\ ` A) \leq x \longleftrightarrow (\forall y > x. \exists i \in A. y > f\ i)$
 $\langle proof \rangle$

lemma *le-Sup-iff*: $x \leq \bigcup A \longleftrightarrow (\forall y < x. \exists a \in A. y < a)$
 $\langle proof \rangle$

lemma *le-SUP-iff*: $x \leq \bigcup (f\ ` A) \longleftrightarrow (\forall y < x. \exists i \in A. y < f\ i)$
 $\langle proof \rangle$

end

11.3 Complete lattice on *bool*

instantiation *bool* :: *complete-lattice*

begin

definition [simp, code]: $\bigcap A \longleftrightarrow \text{False} \notin A$

definition [simp, code]: $\bigcup A \longleftrightarrow \text{True} \in A$

instance

$\langle proof \rangle$

end

lemma *not-False-in-image-Ball* [simp]: $\text{False} \notin P \text{ ' } A \longleftrightarrow \text{Ball } A \ P$
 ⟨proof⟩

lemma *True-in-image-Bex* [simp]: $\text{True} \in P \text{ ' } A \longleftrightarrow \text{Bex } A \ P$
 ⟨proof⟩

lemma *INF-bool-eq* [simp]: $(\lambda A \ f. \bigcap (f \text{ ' } A)) = \text{Ball}$
 ⟨proof⟩

lemma *SUP-bool-eq* [simp]: $(\lambda A \ f. \bigcup (f \text{ ' } A)) = \text{Bex}$
 ⟨proof⟩

instance *bool* :: *complete-boolean-algebra*
 ⟨proof⟩

11.4 Complete lattice on $- \Rightarrow -$

instantiation *fun* :: (*type*, *Inf*) *Inf*
begin

definition $\bigcap A = (\lambda x. \bigcap f \in A. f \ x)$

lemma *Inf-apply* [simp, code]: $(\bigcap A) \ x = (\bigcap f \in A. f \ x)$
 ⟨proof⟩

instance ⟨proof⟩

end

instantiation *fun* :: (*type*, *Sup*) *Sup*
begin

definition $\bigcup A = (\lambda x. \bigcup f \in A. f \ x)$

lemma *Sup-apply* [simp, code]: $(\bigcup A) \ x = (\bigcup f \in A. f \ x)$
 ⟨proof⟩

instance ⟨proof⟩

end

instantiation *fun* :: (*type*, *complete-lattice*) *complete-lattice*
begin

instance
 ⟨proof⟩

end

lemma *INF-apply [simp]*: $(\bigcap y \in A. f y) x = (\bigcap y \in A. f y x)$
 $\langle proof \rangle$

lemma *SUP-apply [simp]*: $(\bigcup y \in A. f y) x = (\bigcup y \in A. f y x)$
 $\langle proof \rangle$

11.5 Complete lattice on unary and binary predicates

lemma *Inf1-I*: $(\bigwedge P. P \in A \implies P a) \implies (\bigcap A) a$
 $\langle proof \rangle$

lemma *INF1-I*: $(\bigwedge x. x \in A \implies B x b) \implies (\bigcap x \in A. B x) b$
 $\langle proof \rangle$

lemma *INF2-I*: $(\bigwedge x. x \in A \implies B x b c) \implies (\bigcap x \in A. B x) b c$
 $\langle proof \rangle$

lemma *Inf2-I*: $(\bigwedge r. r \in A \implies r a b) \implies (\bigcap A) a b$
 $\langle proof \rangle$

lemma *Inf1-D*: $(\bigcap A) a \implies P \in A \implies P a$
 $\langle proof \rangle$

lemma *INF1-D*: $(\bigcap x \in A. B x) b \implies a \in A \implies B a b$
 $\langle proof \rangle$

lemma *Inf2-D*: $(\bigcap A) a b \implies r \in A \implies r a b$
 $\langle proof \rangle$

lemma *INF2-D*: $(\bigcap x \in A. B x) b c \implies a \in A \implies B a b c$
 $\langle proof \rangle$

lemma *Inf1-E*:
assumes $(\bigcap A) a$
obtains $P a \mid P \notin A$
 $\langle proof \rangle$

lemma *INF1-E*:
assumes $(\bigcap x \in A. B x) b$
obtains $B a b \mid a \notin A$
 $\langle proof \rangle$

lemma *Inf2-E*:
assumes $(\bigcap A) a b$
obtains $r a b \mid r \notin A$
 $\langle proof \rangle$

lemma *INF2-E*:

assumes $(\prod x \in A. B\ x)\ b\ c$
obtains $B\ a\ b\ c \mid a \notin A$
 $\langle proof \rangle$

lemma *Sup1-I*: $P \in A \implies P\ a \implies (\bigsqcup A)\ a$
 $\langle proof \rangle$

lemma *SUP1-I*: $a \in A \implies B\ a\ b \implies (\bigsqcup x \in A. B\ x)\ b$
 $\langle proof \rangle$

lemma *Sup2-I*: $r \in A \implies r\ a\ b \implies (\bigsqcup A)\ a\ b$
 $\langle proof \rangle$

lemma *SUP2-I*: $a \in A \implies B\ a\ b\ c \implies (\bigsqcup x \in A. B\ x)\ b\ c$
 $\langle proof \rangle$

lemma *Sup1-E*:
assumes $(\bigsqcup A)\ a$
obtains P **where** $P \in A$ **and** $P\ a$
 $\langle proof \rangle$

lemma *SUP1-E*:
assumes $(\bigsqcup x \in A. B\ x)\ b$
obtains x **where** $x \in A$ **and** $B\ x\ b$
 $\langle proof \rangle$

lemma *Sup2-E*:
assumes $(\bigsqcup A)\ a\ b$
obtains r **where** $r \in A$ $r\ a\ b$
 $\langle proof \rangle$

lemma *SUP2-E*:
assumes $(\bigsqcup x \in A. B\ x)\ b\ c$
obtains x **where** $x \in A$ $B\ x\ b\ c$
 $\langle proof \rangle$

11.6 Complete lattice on - set

instantiation *set* :: (type) complete-lattice
begin

definition $\prod A = \{x. \prod ((\lambda B. x \in B) \text{ ‘ } A)\}$

definition $\bigsqcup A = \{x. \bigsqcup ((\lambda B. x \in B) \text{ ‘ } A)\}$

instance
 $\langle proof \rangle$

end

11.6.1 Inter

abbreviation *Inter* :: 'a set set \Rightarrow 'a set (\bigcap)
where $\bigcap S \equiv \bigcap S$

lemma *Inter-eq*: $\bigcap A = \{x. \forall B \in A. x \in B\}$
 $\langle proof \rangle$

lemma *Inter-iff* [*simp*]: $A \in \bigcap C \longleftrightarrow (\forall X \in C. A \in X)$
 $\langle proof \rangle$

lemma *InterI* [*intro!*]: $(\bigwedge X. X \in C \Longrightarrow A \in X) \Longrightarrow A \in \bigcap C$
 $\langle proof \rangle$

A “destruct” rule – every X in C contains A as an element, but $A \in X$ can hold when $X \in C$ does not! This rule is analogous to *spec*.

lemma *InterD* [*elim*, *Pure.elim*]: $A \in \bigcap C \Longrightarrow X \in C \Longrightarrow A \in X$
 $\langle proof \rangle$

lemma *InterE* [*elim*]: $A \in \bigcap C \Longrightarrow (X \notin C \Longrightarrow R) \Longrightarrow (A \in X \Longrightarrow R) \Longrightarrow R$
 — “Classical” elimination rule – does not require proving $X \in C$.
 $\langle proof \rangle$

lemma *Inter-lower*: $B \in A \Longrightarrow \bigcap A \subseteq B$
 $\langle proof \rangle$

lemma *Inter-subset*: $(\bigwedge X. X \in A \Longrightarrow X \subseteq B) \Longrightarrow A \neq \{\} \Longrightarrow \bigcap A \subseteq B$
 $\langle proof \rangle$

lemma *Inter-greatest*: $(\bigwedge X. X \in A \Longrightarrow C \subseteq X) \Longrightarrow C \subseteq \bigcap A$
 $\langle proof \rangle$

lemma *Inter-empty*: $\bigcap \{\} = UNIV$
 $\langle proof \rangle$

lemma *Inter-UNIV*: $\bigcap UNIV = \{\}$
 $\langle proof \rangle$

lemma *Inter-insert*: $\bigcap (\text{insert } a \ B) = a \cap \bigcap B$
 $\langle proof \rangle$

lemma *Inter-Un-subset*: $\bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B)$
 $\langle proof \rangle$

lemma *Inter-Un-distrib*: $\bigcap (A \cup B) = \bigcap A \cap \bigcap B$
 $\langle proof \rangle$

lemma *Inter-UNIV-conv* [*simp*]:
 $\bigcap A = UNIV \longleftrightarrow (\forall x \in A. x = UNIV)$

$$UNIV = \bigcap A \longleftrightarrow (\forall x \in A. x = UNIV)$$

<proof>

lemma *Inter-anti-mono*: $B \subseteq A \implies \bigcap A \subseteq \bigcap B$
<proof>

11.6.2 Intersections of families

syntax (*ASCII*)

-INTER1 :: *pttrns* \Rightarrow '*b set* \Rightarrow '*b set* (*<(<indent=3 notation=<binder*
INT>>INT -./ ->) [0, 10] 10)
-INTER :: *pttrn* \Rightarrow '*a set* \Rightarrow '*b set* \Rightarrow '*b set* (*<(<indent=3 notation=<binder*
INT>>INT -./ ->) [0, 0, 10] 10)

syntax

-INTER1 :: *pttrns* \Rightarrow '*b set* \Rightarrow '*b set* (*<(<indent=3 notation=<binder*
 $\bigcap \gg \bigcap -./ ->$) [0, 10] 10)
-INTER :: *pttrn* \Rightarrow '*a set* \Rightarrow '*b set* \Rightarrow '*b set* (*<(<indent=3 notation=<binder*
 $\bigcap \gg \bigcap -\in -./ ->$) [0, 0, 10] 10)

syntax (*latex output*)

-INTER1 :: *pttrns* \Rightarrow '*b set* \Rightarrow '*b set* (*<(<3 \bigcap (<unbreakable>-.) / ->*) [0,
10] 10)
-INTER :: *pttrn* \Rightarrow '*a set* \Rightarrow '*b set* \Rightarrow '*b set* (*<(<3 \bigcap (<unbreakable>- \in -) / ->*)
[0, 0, 10] 10)

syntax-consts

-INTER1 -INTER \Leftarrow *Inter*

translations

$\bigcap x y. f \Leftarrow \bigcap x. \bigcap y. f$
 $\bigcap x. f \Leftarrow \bigcap (CONST\ range\ (\lambda x. f))$
 $\bigcap x \in A. f \Leftarrow CONST\ Inter\ ((\lambda x. f)\ 'A)$

lemma *INTER-eq*: $(\bigcap x \in A. B\ x) = \{y. \forall x \in A. y \in B\ x\}$
<proof>

lemma *INT-iff [simp]*: $b \in (\bigcap x \in A. B\ x) \longleftrightarrow (\forall x \in A. b \in B\ x)$
<proof>

lemma *INT-I [intro!]*: $(\bigwedge x. x \in A \implies b \in B\ x) \implies b \in (\bigcap x \in A. B\ x)$
<proof>

lemma *INT-D [elim, Pure.elim]*: $b \in (\bigcap x \in A. B\ x) \implies a \in A \implies b \in B\ a$
<proof>

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 $a \in A$.

$\langle proof \rangle$

lemma *Collect-ball-eq*: $\{x. \forall y \in A. P\ x\ y\} = (\bigcap y \in A. \{x. P\ x\ y\})$
 $\langle proof \rangle$

lemma *Collect-all-eq*: $\{x. \forall y. P\ x\ y\} = (\bigcap y. \{x. P\ x\ y\})$
 $\langle proof \rangle$

lemma *INT-lower*: $a \in A \implies (\bigcap x \in A. B\ x) \subseteq B\ a$
 $\langle proof \rangle$

lemma *INT-greatest*: $(\bigwedge x. x \in A \implies C \subseteq B\ x) \implies C \subseteq (\bigcap x \in A. B\ x)$
 $\langle proof \rangle$

lemma *INT-empty*: $(\bigcap x \in \{\}. B\ x) = UNIV$
 $\langle proof \rangle$

lemma *INT-absorb*: $k \in I \implies A\ k \cap (\bigcap i \in I. A\ i) = (\bigcap i \in I. A\ i)$
 $\langle proof \rangle$

lemma *INT-subset-iff*: $B \subseteq (\bigcap i \in I. A\ i) \longleftrightarrow (\forall i \in I. B \subseteq A\ i)$
 $\langle proof \rangle$

lemma *INT-insert [simp]*: $(\bigcap x \in insert\ a\ A. B\ x) = B\ a \cap \bigcap (B\ ` A)$
 $\langle proof \rangle$

lemma *INT-Un*: $(\bigcap i \in A \cup B. M\ i) = (\bigcap i \in A. M\ i) \cap (\bigcap i \in B. M\ i)$
 $\langle proof \rangle$

lemma *INT-insert-distrib*: $u \in A \implies (\bigcap x \in A. insert\ a\ (B\ x)) = insert\ a\ (\bigcap x \in A. B\ x)$
 $\langle proof \rangle$

lemma *INT-constant [simp]*: $(\bigcap y \in A. c) = (if\ A = \{\} then\ UNIV\ else\ c)$
 $\langle proof \rangle$

lemma *INTER-UNIV-conv*:
 $(UNIV = (\bigcap x \in A. B\ x)) = (\forall x \in A. B\ x = UNIV)$
 $((\bigcap x \in A. B\ x) = UNIV) = (\forall x \in A. B\ x = UNIV)$
 $\langle proof \rangle$

lemma *INT-bool-eq*: $(\bigcap b. A\ b) = A\ True \cap A\ False$
 $\langle proof \rangle$

lemma *INT-anti-mono*: $A \subseteq B \implies (\bigwedge x. x \in A \implies f\ x \subseteq g\ x) \implies (\bigcap x \in B. f\ x) \subseteq (\bigcap x \in A. g\ x)$
 — The last inclusion is POSITIVE!
 $\langle proof \rangle$

lemma *Pow-INT-eq*: $\text{Pow } (\bigcap x \in A. B x) = (\bigcap x \in A. \text{Pow } (B x))$
 $\langle \text{proof} \rangle$

lemma *vimage-INT*: $f - ' (\bigcap x \in A. B x) = (\bigcap x \in A. f - ' B x)$
 $\langle \text{proof} \rangle$

11.6.3 Union

abbreviation *Union* :: 'a set set \Rightarrow 'a set (\bigcup)
 where $\bigcup S \equiv \bigsqcup S$

lemma *Union-eq*: $\bigcup A = \{x. \exists B \in A. x \in B\}$
 $\langle \text{proof} \rangle$

lemma *Union-iff [simp]*: $A \in \bigcup C \longleftrightarrow (\exists X \in C. A \in X)$
 $\langle \text{proof} \rangle$

lemma *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.
 $\langle \text{proof} \rangle$

lemma *UnionE [elim!]*: $A \in \bigcup C \Longrightarrow (\bigwedge X. A \in X \Longrightarrow X \in C \Longrightarrow R) \Longrightarrow R$
 $\langle \text{proof} \rangle$

lemma *Union-upper*: $B \in A \Longrightarrow B \subseteq \bigcup A$
 $\langle \text{proof} \rangle$

lemma *Union-least*: $(\bigwedge X. X \in A \Longrightarrow X \subseteq C) \Longrightarrow \bigcup A \subseteq C$
 $\langle \text{proof} \rangle$

lemma *Union-empty*: $\bigcup \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *Union-UNIV*: $\bigcup UNIV = UNIV$
 $\langle \text{proof} \rangle$

lemma *Union-insert*: $\bigcup (\text{insert } a B) = a \cup \bigcup B$
 $\langle \text{proof} \rangle$

lemma *Union-Un-distrib [simp]*: $\bigcup (A \cup B) = \bigcup A \cup \bigcup B$
 $\langle \text{proof} \rangle$

lemma *Union-Int-subset*: $\bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B$
 $\langle \text{proof} \rangle$

lemma *Union-empty-conv*: $(\bigcup A = \{\}) \longleftrightarrow (\forall x \in A. x = \{\})$
 $\langle \text{proof} \rangle$

lemma *empty-Union-conv*: $(\{\} = \bigcup A) \longleftrightarrow (\forall x \in A. x = \{\})$

$\langle proof \rangle$

lemma *subset-Pow-Union*: $A \subseteq Pow (\bigcup A)$
 $\langle proof \rangle$

lemma *Union-Pow-eq [simp]*: $\bigcup (Pow A) = A$
 $\langle proof \rangle$

lemma *Union-mono*: $A \subseteq B \implies \bigcup A \subseteq \bigcup B$
 $\langle proof \rangle$

lemma *Union-subsetI*: $(\bigwedge x. x \in A \implies \exists y. y \in B \wedge x \subseteq y) \implies \bigcup A \subseteq \bigcup B$
 $\langle proof \rangle$

lemma *disjnt-inj-on-iff*:
 $\llbracket inj-on f (\bigcup \mathcal{A}); X \in \mathcal{A}; Y \in \mathcal{A} \rrbracket \implies disjnt (f ` X) (f ` Y) \longleftrightarrow disjnt X Y$
 $\langle proof \rangle$

lemma *disjnt-Union1 [simp]*: $disjnt (\bigcup \mathcal{A}) B \longleftrightarrow (\forall A \in \mathcal{A}. disjnt A B)$
 $\langle proof \rangle$

lemma *disjnt-Union2 [simp]*: $disjnt B (\bigcup \mathcal{A}) \longleftrightarrow (\forall A \in \mathcal{A}. disjnt B A)$
 $\langle proof \rangle$

11.6.4 Unions of families

syntax (*ASCII*)

-UNION1 :: *pttrns* => 'b set => 'b set $(\langle \langle \langle indent=3 \text{ notation}=\langle binder \text{ UN} \rangle \rangle UN \text{ -./ -} \rangle [0, 10] 10)$
 -UNION :: *pttrn* => 'a set => 'b set => 'b set $(\langle \langle \langle indent=3 \text{ notation}=\langle binder \text{ UN} \rangle \rangle UN \text{ -./ -} \rangle [0, 0, 10] 10)$

syntax

-UNION1 :: *pttrns* => 'b set => 'b set $(\langle \langle \langle indent=3 \text{ notation}=\langle binder \text{ U} \rangle \rangle U \text{ -./ -} \rangle [0, 10] 10)$
 -UNION :: *pttrn* => 'a set => 'b set => 'b set $(\langle \langle \langle indent=3 \text{ notation}=\langle binder \text{ U} \rangle \rangle U \text{ -\in-./ -} \rangle [0, 0, 10] 10)$

syntax (*latex output*)

-UNION1 :: *pttrns* => 'b set => 'b set $(\langle \langle \mathcal{B} \bigcup (\langle unbreakable \rangle \text{ -./ -}) \rangle [0, 10] 10)$
 -UNION :: *pttrn* => 'a set => 'b set => 'b set $(\langle \langle \mathcal{B} \bigcup (\langle unbreakable \rangle \text{ -\in-./ -}) \rangle [0, 0, 10] 10)$

syntax-consts

-UNION1 -UNION $\Rightarrow Union$

translations

$\bigcup x y. f \Rightarrow \bigcup x. \bigcup y. f$

$$\begin{aligned}\bigcup x. f &= \bigcup (\text{CONST range } (\lambda x. f)) \\ \bigcup_{x \in A}. f &= \text{CONST Union } ((\lambda x. f) \text{ ‘ } A)\end{aligned}$$

Note the difference between ordinary syntax of indexed unions and intersections (e.g. $\bigcup_{a_1 \in A_1}. B$) and their L^AT_EX rendition: $\bigcup_{a_1 \in A_1} B$.

lemma *disjoint-UN-iff*: $\text{disjnt } A \ (\bigcup_{i \in I}. B \ i) \longleftrightarrow (\forall i \in I. \text{disjnt } A \ (B \ i))$
<proof>

lemma *UNION-eq*: $(\bigcup_{x \in A}. B \ x) = \{y. \exists x \in A. y \in B \ x\}$
<proof>

lemma *bind-UNION* [code]: $\text{Set.bind } A \ f = \bigcup (f \text{ ‘ } A)$
<proof>

lemma *member-bind* [simp]: $x \in \text{Set.bind } A \ f \longleftrightarrow x \in \bigcup (f \text{ ‘ } A)$
<proof>

lemma *Union-SetCompr-eq*: $\bigcup \{f \ x \mid x. P \ x\} = \{a. \exists x. P \ x \wedge a \in f \ x\}$
<proof>

lemma *UN-iff* [simp]: $b \in (\bigcup_{x \in A}. B \ x) \longleftrightarrow (\exists x \in A. b \in B \ x)$
<proof>

lemma *UN-I* [intro]: $a \in A \Longrightarrow b \in B \ a \Longrightarrow b \in (\bigcup_{x \in A}. B \ x)$
 — The order of the premises presupposes that A is rigid; b may be flexible.
<proof>

lemma *UN-E* [elim!]: $b \in (\bigcup_{x \in A}. B \ x) \Longrightarrow (\bigwedge x. x \in A \Longrightarrow b \in B \ x \Longrightarrow R) \Longrightarrow R$
<proof>

lemma *UN-upper*: $a \in A \Longrightarrow B \ a \subseteq (\bigcup_{x \in A}. B \ x)$
<proof>

lemma *UN-least*: $(\bigwedge x. x \in A \Longrightarrow B \ x \subseteq C) \Longrightarrow (\bigcup_{x \in A}. B \ x) \subseteq C$
<proof>

lemma *Collect-bex-eq*: $\{x. \exists y \in A. P \ x \ y\} = (\bigcup_{y \in A}. \{x. P \ x \ y\})$
<proof>

lemma *UN-insert-distrib*: $u \in A \Longrightarrow (\bigcup_{x \in A}. \text{insert } a \ (B \ x)) = \text{insert } a \ (\bigcup_{x \in A}. B \ x)$
<proof>

lemma *UN-empty*: $(\bigcup_{x \in \{\}}. B \ x) = \{\}$
<proof>

lemma *UN-empty2*: $(\bigcup_{x \in A}. \{\}) = \{\}$
<proof>

lemma *UN-absorb*: $k \in I \implies A \ k \cup (\bigcup_{i \in I}. A \ i) = (\bigcup_{i \in I}. A \ i)$
 ⟨proof⟩

lemma *UN-insert [simp]*: $(\bigcup_{x \in \text{insert } a \ A}. B \ x) = B \ a \cup \bigcup (B \ ' \ A)$
 ⟨proof⟩

lemma *UN-Un [simp]*: $(\bigcup_{i \in A \cup B}. M \ i) = (\bigcup_{i \in A}. M \ i) \cup (\bigcup_{i \in B}. M \ i)$
 ⟨proof⟩

lemma *UN-UN-flatten*: $(\bigcup_{x \in (\bigcup_{y \in A}. B \ y)}. C \ x) = (\bigcup_{y \in A}. \bigcup_{x \in B \ y}. C \ x)$
 ⟨proof⟩

lemma *UN-subset-iff*: $((\bigcup_{i \in I}. A \ i) \subseteq B) = (\forall i \in I. A \ i \subseteq B)$
 ⟨proof⟩

lemma *UN-constant [simp]*: $(\bigcup_{y \in A}. c) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } c)$
 ⟨proof⟩

lemma *UNION-singleton-eq-range*: $(\bigcup_{x \in A}. \{f \ x\}) = f \ ' \ A$
 ⟨proof⟩

lemma *image-Union*: $f \ ' \ \bigcup S = (\bigcup_{x \in S}. f \ ' \ x)$
 ⟨proof⟩

lemma *UNION-empty-conv*:
 $\{\} = (\bigcup_{x \in A}. B \ x) \longleftrightarrow (\forall x \in A. B \ x = \{\})$
 $(\bigcup_{x \in A}. B \ x) = \{\} \longleftrightarrow (\forall x \in A. B \ x = \{\})$
 ⟨proof⟩

lemma *Collect-ex-eq*: $\{x. \exists y. P \ x \ y\} = (\bigcup y. \{x. P \ x \ y\})$
 ⟨proof⟩

lemma *ball-UN*: $(\forall z \in \bigcup (B \ ' \ A). P \ z) \longleftrightarrow (\forall x \in A. \forall z \in B \ x. P \ z)$
 ⟨proof⟩

lemma *beX-UN*: $(\exists z \in \bigcup (B \ ' \ A). P \ z) \longleftrightarrow (\exists x \in A. \exists z \in B \ x. P \ z)$
 ⟨proof⟩

lemma *Un-eq-UN*: $A \cup B = (\bigcup b. \text{if } b \text{ then } A \text{ else } B)$
 ⟨proof⟩

lemma *UN-bool-eq*: $(\bigcup b. A \ b) = (A \ \text{True} \cup A \ \text{False})$
 ⟨proof⟩

lemma *UN-Pow-subset*: $(\bigcup_{x \in A}. \text{Pow } (B \ x)) \subseteq \text{Pow } (\bigcup_{x \in A}. B \ x)$
 ⟨proof⟩

lemma *UN-mono*:

$$A \subseteq B \implies (\bigwedge x. x \in A \implies f x \subseteq g x) \implies \\ (\bigcup_{x \in A}. f x) \subseteq (\bigcup_{x \in B}. g x) \\ \langle proof \rangle$$

lemma *vimage-Union*: $f -' (\bigcup A) = (\bigcup X \in A. f -' X)$
 $\langle proof \rangle$

lemma *vimage-UN*: $f -' (\bigcup_{x \in A}. B x) = (\bigcup_{x \in A}. f -' B x)$
 $\langle proof \rangle$

lemma *vimage-eq-UN*: $f -' B = (\bigcup_{y \in B}. f -' \{y\})$
 — NOT suitable for rewriting
 $\langle proof \rangle$

lemma *image-UN*: $f ' \bigcup (B ' A) = (\bigcup_{x \in A}. f ' B x)$
 $\langle proof \rangle$

lemma *UN-singleton [simp]*: $(\bigcup_{x \in A}. \{x\}) = A$
 $\langle proof \rangle$

lemma *inj-on-image*: $inj-on f (\bigcup A) \implies inj-on ((\cdot) f) A$
 $\langle proof \rangle$

11.6.5 Distributive laws

lemma *Int-Union*: $A \cap \bigcup B = (\bigcup C \in B. A \cap C)$
 $\langle proof \rangle$

lemma *Un-Inter*: $A \cup \bigcap B = (\bigcap C \in B. A \cup C)$
 $\langle proof \rangle$

lemma *Int-Union2*: $\bigcup B \cap A = (\bigcup C \in B. C \cap A)$
 $\langle proof \rangle$

lemma *INT-Int-distrib*: $(\bigcap_{i \in I}. A i \cap B i) = (\bigcap_{i \in I}. A i) \cap (\bigcap_{i \in I}. B i)$
 $\langle proof \rangle$

lemma *UN-Un-distrib*: $(\bigcup_{i \in I}. A i \cup B i) = (\bigcup_{i \in I}. A i) \cup (\bigcup_{i \in I}. B i)$
 $\langle proof \rangle$

lemma *Int-Inter-image*: $(\bigcap_{x \in C}. A x \cap B x) = \bigcap (A ' C) \cap \bigcap (B ' C)$
 $\langle proof \rangle$

lemma *Int-Inter-eq*: $A \cap \bigcap \mathcal{B} = (\text{if } \mathcal{B} = \{\} \text{ then } A \text{ else } (\bigcap B \in \mathcal{B}. A \cap B))$
 $\bigcap \mathcal{B} \cap A = (\text{if } \mathcal{B} = \{\} \text{ then } A \text{ else } (\bigcap B \in \mathcal{B}. B \cap A))$
 $\langle proof \rangle$

lemma *Un-Union-image*: $(\bigcup_{x \in C}. A x \cup B x) = \bigcup (A ' C) \cup \bigcup (B ' C)$
 — Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:

— Union of a family of unions
 $\langle proof \rangle$

lemma *Un-INT-distrib*: $B \cup (\bigcap_{i \in I}. A \ i) = (\bigcap_{i \in I}. B \cup A \ i)$
 $\langle proof \rangle$

lemma *Int-UN-distrib*: $B \cap (\bigcup_{i \in I}. A \ i) = (\bigcup_{i \in I}. B \cap A \ i)$
 — Halmos, Naive Set Theory, page 35.
 $\langle proof \rangle$

lemma *Int-UN-distrib2*: $(\bigcup_{i \in I}. A \ i) \cap (\bigcup_{j \in J}. B \ j) = (\bigcup_{i \in I}. \bigcup_{j \in J}. A \ i \cap B \ j)$
 $\langle proof \rangle$

lemma *Un-INT-distrib2*: $(\bigcap_{i \in I}. A \ i) \cup (\bigcap_{j \in J}. B \ j) = (\bigcap_{i \in I}. \bigcap_{j \in J}. A \ i \cup B \ j)$
 $\langle proof \rangle$

lemma *Union-disjoint*: $(\bigcup C \cap A = \{\}) \longleftrightarrow (\forall B \in C. B \cap A = \{\})$
 $\langle proof \rangle$

lemma *SUP-UNION*: $(\bigsqcup_{x \in (\bigcup y \in A. g \ y)}. f \ x) = (\bigsqcup_{y \in A. \bigsqcup_{x \in g \ y}. f \ x} :: \text{complete-lattice})$
 $\langle proof \rangle$

11.7 Injections and bijections

lemma *inj-on-Inter*: $S \neq \{\} \implies (\bigwedge A. A \in S \implies \text{inj-on } f \ A) \implies \text{inj-on } f \ (\bigcap S)$
 $\langle proof \rangle$

lemma *inj-on-INTER*: $I \neq \{\} \implies (\bigwedge i. i \in I \implies \text{inj-on } f \ (A \ i)) \implies \text{inj-on } f \ (\bigcap i \in I. A \ i)$
 $\langle proof \rangle$

lemma *inj-on-UNION-chain*:

assumes *chain*: $\bigwedge i \ j. i \in I \implies j \in I \implies A \ i \leq A \ j \vee A \ j \leq A \ i$

and *inj*: $\bigwedge i. i \in I \implies \text{inj-on } f \ (A \ i)$

shows *inj-on* $f \ (\bigcup i \in I. A \ i)$

$\langle proof \rangle$

lemma *bij-betw-UNION-chain*:

assumes *chain*: $\bigwedge i \ j. i \in I \implies j \in I \implies A \ i \leq A \ j \vee A \ j \leq A \ i$

and *bij*: $\bigwedge i. i \in I \implies \text{bij-betw } f \ (A \ i) \ (A' \ i)$

shows *bij-betw* $f \ (\bigcup i \in I. A \ i) \ (\bigcup i \in I. A' \ i)$

$\langle proof \rangle$

lemma *image-INT*: $\text{inj-on } f \ C \implies \forall x \in A. B \ x \subseteq C \implies j \in A \implies f \ ' \ (\bigcap (B \ ' \ A)) = (\bigcap x \in A. f \ ' \ B \ x)$

$\langle \text{proof} \rangle$

lemma *bij-image-INT*: $\text{bij } f \implies f' (\bigcap (B' A)) = (\bigcap_{x \in A}. f' B x)$
 $\langle \text{proof} \rangle$

lemma *UNION-fun-upd*: $\bigcup (A(i := B)' J) = \bigcup (A' (J - \{i\})) \cup (\text{if } i \in J \text{ then } B \text{ else } \{\})$
 $\langle \text{proof} \rangle$

lemma *bij-betw-Pow*:
assumes *bij-betw* $f A B$
shows *bij-betw* $(\text{image } f) (\text{Pow } A) (\text{Pow } B)$
 $\langle \text{proof} \rangle$

11.7.1 Complement

lemma *Compl-INT* [simp]: $-(\bigcap_{x \in A}. B x) = (\bigcup_{x \in A}. -B x)$
 $\langle \text{proof} \rangle$

lemma *Compl-UN* [simp]: $-(\bigcup_{x \in A}. B x) = (\bigcap_{x \in A}. -B x)$
 $\langle \text{proof} \rangle$

11.7.2 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

lemma *UN-simps* [simp]:

$\bigwedge a B C. (\bigcup_{x \in C}. \text{insert } a (B x)) = (\text{if } C = \{\} \text{ then } \{\} \text{ else insert } a (\bigcup_{x \in C}. B x))$
 $\bigwedge A B C. (\bigcup_{x \in C}. A x \cup B) = ((\text{if } C = \{\} \text{ then } \{\} \text{ else } (\bigcup_{x \in C}. A x) \cup B)$
 $\bigwedge A B C. (\bigcup_{x \in C}. A \cup B x) = ((\text{if } C = \{\} \text{ then } \{\} \text{ else } A \cup (\bigcup_{x \in C}. B x))$
 $\bigwedge A B C. (\bigcup_{x \in C}. A x \cap B) = ((\bigcup_{x \in C}. A x) \cap B)$
 $\bigwedge A B C. (\bigcup_{x \in C}. A \cap B x) = (A \cap (\bigcup_{x \in C}. B x))$
 $\bigwedge A B C. (\bigcup_{x \in C}. A x - B) = ((\bigcup_{x \in C}. A x) - B)$
 $\bigwedge A B C. (\bigcup_{x \in C}. A - B x) = (A - (\bigcap_{x \in C}. B x))$
 $\bigwedge A B. (\bigcup_{x \in \bigcup A}. B x) = (\bigcup_{y \in A}. \bigcup_{x \in y}. B x)$
 $\bigwedge A B C. (\bigcup_{z \in (\bigcup (B' A))}. C z) = (\bigcup_{x \in A}. \bigcup_{z \in B x}. C z)$
 $\bigwedge A B f. (\bigcup_{x \in f' A}. B x) = (\bigcup_{a \in A}. B (f a))$
 $\langle \text{proof} \rangle$

lemma *INT-simps* [simp]:

$\bigwedge A B C. (\bigcap_{x \in C}. A x \cap B) = (\text{if } C = \{\} \text{ then UNIV else } (\bigcap_{x \in C}. A x) \cap B)$
 $\bigwedge A B C. (\bigcap_{x \in C}. A \cap B x) = (\text{if } C = \{\} \text{ then UNIV else } A \cap (\bigcap_{x \in C}. B x))$
 $\bigwedge A B C. (\bigcap_{x \in C}. A x - B) = (\text{if } C = \{\} \text{ then UNIV else } (\bigcap_{x \in C}. A x) - B)$
 $\bigwedge A B C. (\bigcap_{x \in C}. A - B x) = (\text{if } C = \{\} \text{ then UNIV else } A - (\bigcup_{x \in C}. B x))$
 $\bigwedge a B C. (\bigcap_{x \in C}. \text{insert } a (B x)) = \text{insert } a (\bigcap_{x \in C}. B x)$
 $\bigwedge A B C. (\bigcap_{x \in C}. A x \cup B) = ((\bigcap_{x \in C}. A x) \cup B)$
 $\bigwedge A B C. (\bigcap_{x \in C}. A \cup B x) = (A \cup (\bigcap_{x \in C}. B x))$
 $\bigwedge A B. (\bigcap_{x \in \bigcup A}. B x) = (\bigcap_{y \in A}. \bigcap_{x \in y}. B x)$
 $\bigwedge A B C. (\bigcap_{z \in (\bigcup (B' A))}. C z) = (\bigcap_{x \in A}. \bigcap_{z \in B x}. C z)$

$$\bigwedge A B f. (\bigcap x \in f^* A. B x) = (\bigcap a \in A. B (f a))$$

<proof>

lemma *UN-ball-bex-simps* [simp]:

$$\begin{aligned} \bigwedge A P. (\forall x \in \bigcup A. P x) &\longleftrightarrow (\forall y \in A. \forall x \in y. P x) \\ \bigwedge A B P. (\forall x \in (\bigcup (B \text{ ‘ } A)). P x) &= (\forall a \in A. \forall x \in B a. P x) \\ \bigwedge A P. (\exists x \in \bigcup A. P x) &\longleftrightarrow (\exists y \in A. \exists x \in y. P x) \\ \bigwedge A B P. (\exists x \in (\bigcup (B \text{ ‘ } A)). P x) &\longleftrightarrow (\exists a \in A. \exists x \in B a. P x) \end{aligned}$$

<proof>

Maxiscoping: pulling out big Unions and Intersections.

lemma *UN-extend-simps*:

$$\begin{aligned} \bigwedge a B C. \text{insert } a (\bigcup x \in C. B x) &= (\text{if } C = \{\} \text{ then } \{a\} \text{ else } (\bigcup x \in C. \text{insert } a (B x))) \\ \bigwedge A B C. (\bigcup x \in C. A x) \cup B &= (\text{if } C = \{\} \text{ then } B \text{ else } (\bigcup x \in C. A x \cup B)) \\ \bigwedge A B C. A \cup (\bigcup x \in C. B x) &= (\text{if } C = \{\} \text{ then } A \text{ else } (\bigcup x \in C. A \cup B x)) \\ \bigwedge A B C. ((\bigcup x \in C. A x) \cap B) &= (\bigcup x \in C. A x \cap B) \\ \bigwedge A B C. (A \cap (\bigcup x \in C. B x)) &= (\bigcup x \in C. A \cap B x) \\ \bigwedge A B C. ((\bigcup x \in C. A x) - B) &= (\bigcup x \in C. A x - B) \\ \bigwedge A B C. (A - (\bigcup x \in C. B x)) &= (\bigcup x \in C. A - B x) \\ \bigwedge A B. (\bigcup y \in A. \bigcup x \in y. B x) &= (\bigcup x \in \bigcup A. B x) \\ \bigwedge A B C. (\bigcup x \in A. \bigcup z \in B x. C z) &= (\bigcup z \in (\bigcup (B \text{ ‘ } A)). C z) \\ \bigwedge A B f. (\bigcup a \in A. B (f a)) &= (\bigcup x \in f^* A. B x) \end{aligned}$$

<proof>

lemma *INT-extend-simps*:

$$\begin{aligned} \bigwedge A B C. (\bigcap x \in C. A x) \cap B &= (\text{if } C = \{\} \text{ then } B \text{ else } (\bigcap x \in C. A x \cap B)) \\ \bigwedge A B C. A \cap (\bigcap x \in C. B x) &= (\text{if } C = \{\} \text{ then } A \text{ else } (\bigcap x \in C. A \cap B x)) \\ \bigwedge A B C. (\bigcap x \in C. A x) - B &= (\text{if } C = \{\} \text{ then } \text{UNIV} - B \text{ else } (\bigcap x \in C. A x - B)) \\ \bigwedge A B C. A - (\bigcup x \in C. B x) &= (\text{if } C = \{\} \text{ then } A \text{ else } (\bigcap x \in C. A - B x)) \\ \bigwedge a B C. \text{insert } a (\bigcap x \in C. B x) &= (\bigcap x \in C. \text{insert } a (B x)) \\ \bigwedge A B C. ((\bigcap x \in C. A x) \cup B) &= (\bigcap x \in C. A x \cup B) \\ \bigwedge A B C. A \cup (\bigcap x \in C. B x) &= (\bigcap x \in C. A \cup B x) \\ \bigwedge A B. (\bigcap y \in A. \bigcap x \in y. B x) &= (\bigcap x \in \bigcup A. B x) \\ \bigwedge A B C. (\bigcap x \in A. \bigcap z \in B x. C z) &= (\bigcap z \in (\bigcup (B \text{ ‘ } A)). C z) \\ \bigwedge A B f. (\bigcap a \in A. B (f a)) &= (\bigcap x \in f^* A. B x) \end{aligned}$$

<proof>

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  $\Rightarrow$  'a  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'b
  case-nil :: 'a  $\Rightarrow$  'b
  case-cons :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b
  case-elem :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'a  $\Rightarrow$  'b
  case-abs :: ('c  $\Rightarrow$  'b)  $\Rightarrow$  'b

declare [[coercion-args case-guard - + -]]
declare [[coercion-args case-cons - -]]
declare [[coercion-args case-abs -]]
declare [[coercion-args case-elem - +]]

 $\langle ML \rangle$ 

lemma iffI-np:  $\llbracket x \Longrightarrow \neg y; \neg x \Longrightarrow y \rrbracket \Longrightarrow \neg x \longleftrightarrow y$ 
   $\langle proof \rangle$ 

lemma iff-contradict:
   $\neg P \Longrightarrow P \longleftrightarrow Q \Longrightarrow Q \Longrightarrow R$ 
   $\neg Q \Longrightarrow P \longleftrightarrow Q \Longrightarrow P \Longrightarrow R$ 
   $\langle proof \rangle$ 

 $\langle ML \rangle$ 

Coinduction method that avoids some boilerplate compared with coinduct.

 $\langle ML \rangle$ 

end

```

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  $\Rightarrow$  'a)  $\Rightarrow$  'a
  where lfp f = Inf {u. f u  $\leq$  u}

```

```

lemma lfp-lowerbound: f A  $\leq$  A  $\Longrightarrow$  lfp f  $\leq$  A
  <proof>

```

```

lemma lfp-greatest: ( $\bigwedge$  u. f u  $\leq$  u  $\Longrightarrow$  A  $\leq$  u)  $\Longrightarrow$  A  $\leq$  lfp f
  <proof>

```

```

end

```

```

lemma lfp-fixpoint:
  assumes mono f
  shows f (lfp f) = lfp f
  <proof>

```

```

lemma lfp-unfold: mono f  $\Longrightarrow$  lfp f = f (lfp f)
  <proof>

```

```

lemma lfp-const: lfp ( $\lambda$ x. t) = t
  <proof>

```

```

lemma lfp-eqI: mono F  $\Longrightarrow$  F x = x  $\Longrightarrow$  ( $\bigwedge$  z. F z = z  $\Longrightarrow$  x  $\leq$  z)  $\Longrightarrow$  lfp F = x
  <proof>

```

13.2 General induction rules for least fixed points

```

lemma lfp-ordinal-induct [case-names mono step union]:
  fixes f :: 'a::complete-lattice  $\Rightarrow$  'a
  assumes mono: mono f
    and P-f:  $\bigwedge$  S. P S  $\Longrightarrow$  S  $\leq$  lfp f  $\Longrightarrow$  P (f S)
    and P-Union:  $\bigwedge$  M.  $\forall$  S $\in$ M. P S  $\Longrightarrow$  P (Sup M)
  shows P (lfp f)
  <proof>

```

```

theorem lfp-induct:
  assumes mono: mono f
    and ind: f (inf (lfp f) P)  $\leq$  P
  shows lfp f  $\leq$  P
  <proof>

```

lemma *lfp-induct-set*:
assumes *lfp*: $a \in \text{lfp } f$
and *mono*: $\text{mono } f$
and *hyp*: $\bigwedge x. x \in f (\text{lfp } f \cap \{x. P \ x\}) \implies P \ x$
shows $P \ a$
 $\langle \text{proof} \rangle$

lemma *lfp-ordinal-induct-set*:
assumes *mono*: $\text{mono } f$
and *P-f*: $\bigwedge S. P \ S \implies P \ (f \ S)$
and *P-Union*: $\bigwedge M. \forall S \in M. P \ S \implies P \ (\bigcup M)$
shows $P \ (\text{lfp } f)$
 $\langle \text{proof} \rangle$

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding.

lemma *def-lfp-unfold*: $h \equiv \text{lfp } f \implies \text{mono } f \implies h = f \ h$
 $\langle \text{proof} \rangle$

lemma *def-lfp-induct*: $A \equiv \text{lfp } f \implies \text{mono } f \implies f \ (\inf A \ P) \leq P \implies A \leq P$
 $\langle \text{proof} \rangle$

lemma *def-lfp-induct-set*:
 $A \equiv \text{lfp } f \implies \text{mono } f \implies a \in A \implies (\bigwedge x. x \in f (A \cap \{x. P \ x\}) \implies P \ x) \implies P \ a$
 $\langle \text{proof} \rangle$

Monotonicity of *lfp*!

lemma *lfp-mono*: $(\bigwedge Z. f \ Z \leq g \ Z) \implies \text{lfp } f \leq \text{lfp } g$
 $\langle \text{proof} \rangle$

13.3 Greatest fixed points

context *complete-lattice*
begin

definition *gfp* :: $('a \Rightarrow 'a) \Rightarrow 'a$
where $\text{gfp } f = \text{Sup } \{u. u \leq f \ u\}$

lemma *gfp-upperbound*: $X \leq f \ X \implies X \leq \text{gfp } f$
 $\langle \text{proof} \rangle$

lemma *gfp-least*: $(\bigwedge u. u \leq f \ u \implies u \leq X) \implies \text{gfp } f \leq X$
 $\langle \text{proof} \rangle$

end

lemma *lfp-le-gfp*: $\text{mono } f \implies \text{lfp } f \leq \text{gfp } f$
 $\langle \text{proof} \rangle$

lemma *gfp-fixpoint*:

assumes *mono f*

shows $f (gfp f) = fp f$

<proof>

lemma *gfp-unfold*: $mono f \implies fp f = f (fp f)$

<proof>

lemma *gfp-const*: $fp (\lambda x. t) = t$

<proof>

lemma *gfp-eqI*: $mono F \implies F x = x \implies (\bigwedge z. F z = z \implies z \leq x) \implies fp F = x$

<proof>

13.4 Coinduction rules for greatest fixed points

Weak version.

lemma *weak-coinduct*: $a \in X \implies X \subseteq fp f \implies a \in fp f$

<proof>

lemma *weak-coinduct-image*: $a \in X \implies g'X \subseteq f (g'X) \implies g a \in fp f$

<proof>

lemma *coinduct-lemma*: $X \leq f (sup X (fp f)) \implies mono f \implies sup X (fp f) \leq f (sup X (fp f))$

<proof>

Strong version, thanks to Coen and Frost.

lemma *coinduct-set*: $mono f \implies a \in X \implies X \subseteq f (X \cup fp f) \implies a \in fp f$

<proof>

lemma *gfp-fun-UnI2*: $mono f \implies a \in fp f \implies a \in f (X \cup fp f)$

<proof>

lemma *gfp-ordinal-induct*[*case-names mono step union*]:

fixes $f :: 'a :: complete_lattice \Rightarrow 'a$

assumes *mono*: $mono f$

and *P-f*: $\bigwedge S. P S \implies fp f \leq S \implies P (f S)$

and *P-Union*: $\bigwedge M. \forall S \in M. P S \implies P (Inf M)$

shows $P (fp f)$

<proof>

lemma *coinduct*:

assumes *mono*: $mono f$

and *ind*: $X \leq f (sup X (fp f))$

shows $X \leq fp f$

<proof>

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*: $\text{mono } f \implies \text{mono } (\lambda x. f x \cup X \cup B)$
 $\langle \text{proof} \rangle$

lemma *coinduct3-lemma*:

$X \subseteq f (\text{lfp } (\lambda x. f x \cup X \cup \text{gfp } f)) \implies \text{mono } f \implies$
 $\text{lfp } (\lambda x. f x \cup X \cup \text{gfp } f) \subseteq f (\text{lfp } (\lambda x. f x \cup X \cup \text{gfp } f))$
 $\langle \text{proof} \rangle$

lemma *coinduct3*: $\text{mono } f \implies a \in X \implies X \subseteq f (\text{lfp } (\lambda x. f x \cup X \cup \text{gfp } f)) \implies$
 $a \in \text{gfp } f$
 $\langle \text{proof} \rangle$

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding.

lemma *def-gfp-unfold*: $A \equiv \text{gfp } f \implies \text{mono } f \implies A = f A$
 $\langle \text{proof} \rangle$

lemma *def-coinduct*: $A \equiv \text{gfp } f \implies \text{mono } f \implies X \leq f (\text{sup } X A) \implies X \leq A$
 $\langle \text{proof} \rangle$

lemma *def-coinduct-set*: $A \equiv \text{gfp } f \implies \text{mono } f \implies a \in X \implies X \subseteq f (X \cup A)$
 $\implies a \in A$
 $\langle \text{proof} \rangle$

lemma *def-Collect-coinduct*:

$A \equiv \text{gfp } (\lambda w. \text{Collect } (P w)) \implies \text{mono } (\lambda w. \text{Collect } (P w)) \implies a \in X \implies$
 $(\bigwedge z. z \in X \implies P (X \cup A) z) \implies a \in A$
 $\langle \text{proof} \rangle$

lemma *def-coinduct3*: $A \equiv \text{gfp } f \implies \text{mono } f \implies a \in X \implies X \subseteq f (\text{lfp } (\lambda x. f x$
 $\cup X \cup A)) \implies a \in A$
 $\langle \text{proof} \rangle$

Monotonicity of *gfp*!

lemma *gfp-mono*: $(\bigwedge Z. f Z \leq g Z) \implies \text{gfp } f \leq \text{gfp } g$
 $\langle \text{proof} \rangle$

13.6 Rules for fixed point calculus

lemma *lfp-rolling*:

assumes $\text{mono } g \text{ mono } f$
shows $g (\text{lfp } (\lambda x. f (g x))) = \text{lfp } (\lambda x. g (f x))$
 $\langle \text{proof} \rangle$

lemma *lfp-lfp*:

assumes $f: \bigwedge x y w z. x \leq y \implies w \leq z \implies f x w \leq f y z$
shows $\text{lfp } (\lambda x. \text{lfp } (f x)) = \text{lfp } (\lambda x. f x x)$

$\langle proof \rangle$

lemma *gfp-rolling*:

assumes *mono g mono f*

shows $g (gfp (\lambda x. f (g x))) = gfp (\lambda x. g (f x))$

$\langle proof \rangle$

lemma *gfp-gfp*:

assumes $f: \bigwedge x y w z. x \leq y \implies w \leq z \implies f x w \leq f y z$

shows $gfp (\lambda x. gfp (f x)) = gfp (\lambda x. f x x)$

$\langle proof \rangle$

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*: $X \leq Y \longleftrightarrow X \text{ False} \leq Y \text{ False} \wedge X \text{ True} \leq Y \text{ True}$

$\langle proof \rangle$

lemma *imp-conj-iff*: $((P \longrightarrow Q) \wedge P) = (P \wedge Q)$

$\langle proof \rangle$

lemma *meta-fun-cong*: $P \equiv Q \implies P a \equiv Q a$

$\langle proof \rangle$

$\langle ML \rangle$

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/proofofschroederbernsteintheoremusingtarskiknasterttheorem>
- 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 ' A ⊆ B
    and inj2: inj-on g B and sub2: g ' B ⊆ A
  shows ∃ h. bij-betw h A B
<proof>

```

13.9 Inductive datatypes and primitive recursion

Package setup.

<ML>

Lambda-abstractions with pattern matching:

```

syntax (ASCII)
  -lam-pats-syntax :: cases-syn ⇒ 'a ⇒ 'b (⟨⟨notation=abstraction⟩%_⟩ 10)
syntax
  -lam-pats-syntax :: cases-syn ⇒ 'a ⇒ 'b (⟨⟨notation=abstraction⟩λ-⟩ 10)
<ML>

```

end

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
<proof>

```

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

<ML>

```

old-rep-datatype True False <proof>

```

<ML>

But erase the prefix for properties that are not generated by *free-constructors*.

<ML>

```

lemmas induct = old.bool.induct
lemmas inducts = old.bool.inducts
lemmas rec = old.bool.rec
lemmas_simps = bool.distinct bool.case bool.rec

```

<ML>

declare *case-split* [*cases type: bool*]
 — prefer plain propositional version

lemma [*code*]:
 $HOL.equal\ False\ P \longleftrightarrow \neg P$
 $HOL.equal\ True\ P \longleftrightarrow P$
 $HOL.equal\ P\ False \longleftrightarrow \neg P$
 $HOL.equal\ P\ True \longleftrightarrow P$
 $\langle proof \rangle$

lemma [*code nbe*]:
 $HOL.equal\ P\ P \longleftrightarrow True$ **for** $P :: bool$
 $\langle proof \rangle$

lemma *If-case-cert*:
assumes $CASE \equiv (\lambda b. If\ b\ f\ g)$
shows $(CASE\ True \equiv f) \ \&\&\&\ (CASE\ False \equiv g)$
 $\langle proof \rangle$

$\langle ML \rangle$

code-printing
constant $HOL.equal :: bool \Rightarrow bool \Rightarrow bool \rightarrow (Haskell)$ **infix** 4 ==
| class-instance $bool :: equal \rightarrow (Haskell) -$

14.2 The *unit* type

typedef *unit* = { *True* }
 $\langle proof \rangle$

definition *Unity* :: *unit* ($\iota'()$)
where $() = Abs-unit\ True$

lemma *unit-eq* [*no-atp*]: $u = ()$
 $\langle proof \rangle$

Simplification procedure for *unit-eq*. Cannot use this rule directly — it loops!

$\langle ML \rangle$

free-constructors *case-unit* **for** $()$
 $\langle proof \rangle$

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

$\langle ML \rangle$

old-rep-datatype $()$ $\langle proof \rangle$

⟨ML⟩

But erase the prefix for properties that are not generated by *free-constructors*.

⟨ML⟩

```

lemmas induct = old.unit.induct
lemmas inducts = old.unit.inducts
lemmas rec = old.unit.rec
lemmas simps = unit.case unit.rec

```

⟨ML⟩

```

lemma unit-all-eq1: (⋀x::unit. PROP P x) ≡ PROP P ()
  ⟨proof⟩

```

```

lemma unit-all-eq2: (⋀x::unit. PROP P) ≡ PROP P
  ⟨proof⟩

```

This rewrite counters the effect of *simproc unit-eq* on $\lambda u::unit. f u$, replacing it by f rather than by $\lambda u. f ()$.

```

lemma unit-abs-eta-conv [simp]: ( $\lambda u::unit. f ()$ ) = f
  ⟨proof⟩

```

```

lemma UNIV-unit: UNIV = {()}
  ⟨proof⟩

```

```

instantiation unit :: default
begin

```

```

definition default = ()

```

```

instance ⟨proof⟩

```

```

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]:  $u \leq v$  for  $u v :: unit$ 
  ⟨proof⟩

```

```

definition less-unit :: unit ⇒ unit ⇒ bool
  where ( $-::unit$ ) < - ⇔ False

```

```

lemma less-unit [iff]:  $\neg u < v$  for  $u v :: unit$ 
  ⟨proof⟩

```

```

definition bot-unit :: unit
  where [code-unfold]:  $\perp = ()$ 

definition top-unit :: unit
  where [code-unfold]:  $\top = ()$ 

definition inf-unit :: unit  $\Rightarrow$  unit  $\Rightarrow$  unit
  where [simp]:  $- \sqcap - = ()$ 

definition sup-unit :: unit  $\Rightarrow$  unit  $\Rightarrow$  unit
  where [simp]:  $- \sqcup - = ()$ 

definition Inf-unit :: unit set  $\Rightarrow$  unit
  where [simp]:  $\bigsqcap - = ()$ 

definition Sup-unit :: unit set  $\Rightarrow$  unit
  where [simp]:  $\bigsqcup - = ()$ 

definition uminus-unit :: unit  $\Rightarrow$  unit
  where [simp]:  $- - = ()$ 

declare less-eq-unit-def [abs-def, code-unfold]
  less-unit-def [abs-def, code-unfold]
  inf-unit-def [abs-def, code-unfold]
  sup-unit-def [abs-def, code-unfold]
  Inf-unit-def [abs-def, code-unfold]
  Sup-unit-def [abs-def, code-unfold]
  uminus-unit-def [abs-def, code-unfold]

instance
  <proof>

end

lemma [code]: HOL.equal u v  $\longleftrightarrow$  True for u v :: unit
  <proof>

code-printing
  type-constructor unit  $\rightarrow$ 
    (SML) unit
    and (OCaml) unit
    and (Haskell)  $()$ 
    and (Scala) Unit
  | constant Unity  $\rightarrow$ 
    (SML)  $()$ 
    and (OCaml)  $()$ 
    and (Haskell)  $()$ 
    and (Scala)  $()$ 

```

```
| class-instance unit :: equal  $\rightarrow$ 
  (Haskell)  $-$ 
| constant HOL.equal :: unit  $\Rightarrow$  unit  $\Rightarrow$  bool  $\rightarrow$ 
  (Haskell) infix 4 ==
```

```
code-reserved
  (SML) unit
  and (OCaml) unit
  and (Scala) Unit
```

14.3 The product type

14.3.1 Type definition

```
definition Pair-Rep :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  bool
  where Pair-Rep a b = ( $\lambda x y. x = a \wedge y = b$ )
```

```
definition prod = {f.  $\exists a b. f = \text{Pair-Rep } (a::'a) (b::'b)$ }
```

```
typedef ('a, 'b) prod ( $\langle (\langle \text{notation} = \langle \text{infix } \times \rangle - \times / - \rangle [21, 20] 20) = \text{prod} :: ('a$ 
 $\Rightarrow$  'b  $\Rightarrow$  bool) set
```

<proof>

```
type-notation (ASCII)
  prod (infixr  $\langle * \rangle$  20)
```

```
definition Pair :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'a  $\times$  'b
  where Pair a b = Abs-prod (Pair-Rep a b)
```

```
lemma prod-cases: ( $\bigwedge a b. P (\text{Pair } a b)$ )  $\implies P p$ 
  <proof>
```

```
free-constructors case-prod for Pair fst snd
  <proof>
```

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

<ML>

```
old-rep-datatype Pair
  <proof>
```

<ML>

But erase the prefix for properties that are not generated by *free-constructors*.

<ML>

```
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

```

⟨ML⟩

```

declare prod.case [nitpick-simp del]
declare old.prod.case-cong-weak [cong del]
declare prod.case-eq-if [mono]
declare prod.split [no-atp]
declare prod.split-asm [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*

open-bundle *tuple-syntax*

begin

syntax

```

-tuple      :: 'a ⇒ tuple-args ⇒ 'a × 'b      (⟨⟨indent=1 notation=⟨mixfix
tuple⟩⟩'(-,/ -)⟩⟩)
-tuple-arg  :: 'a ⇒ tuple-args                  (⟨-⟩)
-tuple-args :: 'a ⇒ tuple-args ⇒ tuple-args      (⟨-,/ -⟩)
-pattern    :: pttrn ⇒ patterns ⇒ pttrn        (⟨⟨open-block notation=⟨pattern
tuple⟩⟩'(-,/ -)⟩⟩)
              :: pttrn ⇒ patterns                  (⟨-⟩)
-patterns   :: pttrn ⇒ patterns ⇒ patterns      (⟨-,/ -⟩)
-unit       :: pttrn                            (⟨⟨open-block notation=⟨pattern
unit⟩⟩'()'⟩⟩)

```

syntax-consts

```

-pattern -patterns ⇒ case-prod and
-unit ⇒ case-unit

```

translations

```

(x, y) ⇒ CONST Pair x y
-pattern x y ⇒ CONST Pair x y
-patterns x y ⇒ CONST Pair x y
-tuple x (-tuple-args y z) ⇒ -tuple x (-tuple-arg (-tuple y z))
λ(x, y, zs). b ⇒ CONST case-prod (λx (y, zs). b)
λ(x, y). b ⇒ CONST case-prod (λx y. b)
-abs (CONST Pair x y) t ↦ λ(x, y). t

```

— This rule accommodates tuples in *case C ... (x, y) ... ⇒ ...*: The (*x*, *y*) is parsed as *Pair x y* because it is *logic*, not *pttrn*.

```

λ(). b ⇒ CONST case-unit b
-abs (CONST Unity) t → λ(). t

```

end

```

print case-prod f as case-prod f and case-prod f as case-prod f

```

⟨ML⟩

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.

⟨ML⟩

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*: $(a, b) = (a', b') \implies (a = a' \implies b = b' \implies R) \implies R$
 ⟨proof⟩

lemma *surj-pair* [simp]: $\exists x y. p = (x, y)$
 ⟨proof⟩

lemma *fst-eqD*: $\text{fst } (x, y) = a \implies x = a$
 ⟨proof⟩

lemma *snd-eqD*: $\text{snd } (x, y) = a \implies y = a$
 ⟨proof⟩

lemma *case-prod-unfold* [nitpick-unfold]: $\text{case-prod} = (\lambda c p. c (\text{fst } p) (\text{snd } p))$
 ⟨proof⟩

lemma *case-prod-conv* [*simp*, *code*]: $(\text{case } (a, b) \text{ of } (c, d) \Rightarrow f \ c \ d) = f \ a \ b$
 ⟨*proof*⟩

lemmas *surjective-pairing* = *prod.collapse* [*symmetric*]

lemma *prod-eq-iff*: $s = t \longleftrightarrow \text{fst } s = \text{fst } t \wedge \text{snd } s = \text{snd } t$
 ⟨*proof*⟩

lemma *prod-eqI* [*intro?*]: $\text{fst } p = \text{fst } q \Longrightarrow \text{snd } p = \text{snd } q \Longrightarrow p = q$
 ⟨*proof*⟩

lemma *case-prodI*: $f \ a \ b \Longrightarrow \text{case } (a, b) \text{ of } (c, d) \Rightarrow f \ c \ d$
 ⟨*proof*⟩

lemma *case-prodD*: $(\text{case } (a, b) \text{ of } (c, d) \Rightarrow f \ c \ d) \Longrightarrow f \ a \ b$
 ⟨*proof*⟩

lemma *case-prod-Pair* [*simp*]: *case-prod Pair* = *id*
 ⟨*proof*⟩

lemma *case-prod-eta*: $(\lambda(x, y). f \ (x, y)) = f$
 — Subsumes the old *split-Pair* when *f* is the identity function.
 ⟨*proof*⟩

lemma *case-prod-comp*: $(\text{case } x \text{ of } (a, b) \Rightarrow (f \circ g) \ a \ b) = f \ (g \ (\text{fst } x)) \ (\text{snd } x)$
 ⟨*proof*⟩

lemma *The-case-prod*: *The* (*case-prod P*) = (*THE xy. P* (*fst xy*) (*snd xy*))
 ⟨*proof*⟩

lemma *cond-case-prod-eta*: $(\bigwedge x \ y. f \ x \ y = g \ (x, y)) \Longrightarrow (\lambda(x, y). f \ x \ y) = g$
 ⟨*proof*⟩

lemma *split-paired-all* [*no-atp*]: $(\bigwedge x. \text{PROP } P \ x) \equiv (\bigwedge a \ b. \text{PROP } P \ (a, b))$
 ⟨*proof*⟩

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 $\bigwedge a \ b. \dots = ?P(a, b)$ which cannot be solved by reflexivity.

lemmas *split-tupled-all* = *split-paired-all unit-all-eq2*

⟨*ML*⟩

lemma *split-paired-All* [*simp*, *no-atp*]: $(\forall x. P \ x) \longleftrightarrow (\forall a \ b. P \ (a, b))$
 — [*iff*] is not a good idea because it makes *blast* loop
 ⟨*proof*⟩

lemma *split-paired-Ex* [*simp*, *no-atp*]: $(\exists x. P\ x) \longleftrightarrow (\exists a\ b. P\ (a, b))$
 ⟨*proof*⟩

lemma *split-paired-The* [*no-atp*]: $(THE\ x. P\ x) = (THE\ (a, b). P\ (a, b))$
 — Can’t be added to simpset: loops!
 ⟨*proof*⟩

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*⟩

lemma *case-prod-beta'*: $(\lambda(x,y). f\ x\ y) = (\lambda x. f\ (fst\ x)\ (snd\ x))$
 ⟨*proof*⟩

case-prod used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *prod.split* as rewrite.

lemma *case-prodI2*:
 $\bigwedge p. (\bigwedge a\ b. p = (a, b) \implies c\ a\ b) \implies case\ p\ of\ (a, b) \Rightarrow c\ a\ b$
 ⟨*proof*⟩

lemma *case-prodI2'*:
 $\bigwedge p. (\bigwedge a\ b. (a, b) = p \implies c\ a\ b\ x) \implies (case\ p\ of\ (a, b) \Rightarrow c\ a\ b)\ x$
 ⟨*proof*⟩

lemma *case-prodE* [*elim!*]:
 $(case\ p\ of\ (a, b) \Rightarrow c\ a\ b) \implies (\bigwedge x\ y. p = (x, y) \implies c\ x\ y \implies Q) \implies Q$
 ⟨*proof*⟩

lemma *case-prodE'* [*elim!*]:
 $(case\ p\ of\ (a, b) \Rightarrow c\ a\ b)\ z \implies (\bigwedge x\ y. p = (x, y) \implies c\ x\ y\ z \implies Q) \implies Q$
 ⟨*proof*⟩

lemma *case-prodE2*:
 assumes *q*: $Q\ (case\ z\ of\ (a, b) \Rightarrow P\ a\ b)$
 and *r*: $\bigwedge x\ y. z = (x, y) \implies Q\ (P\ x\ y) \implies R$
 shows *R*
 ⟨*proof*⟩

lemma *case-prodD'*: $(case\ (a, b)\ of\ (c, d) \Rightarrow R\ c\ d)\ c \implies R\ a\ b\ c$
 ⟨*proof*⟩

lemma *mem-case-prodI*: $z \in c\ a\ b \implies z \in (case\ (a, b)\ of\ (d, e) \Rightarrow c\ d\ e)$
 ⟨*proof*⟩

lemma *mem-case-prodI2* [*intro!*]:

$\bigwedge p. (\bigwedge a\ b. p = (a, b) \implies z \in c\ a\ b) \implies z \in (case\ p\ of\ (a, b) \Rightarrow c\ a\ b)$
 $\langle proof \rangle$

declare *mem-case-prodI* [*intro!*] — postponed to maintain traditional declaration order!

declare *case-prodI2'* [*intro!*] — postponed to maintain traditional declaration order!

declare *case-prodI2* [*intro!*] — postponed to maintain traditional declaration order!

declare *case-prodI* [*intro!*] — postponed to maintain traditional declaration order!

lemma *mem-case-prodE* [*elim!*]:

assumes $z \in case\text{-}prod\ c\ p$
obtains $x\ y$ **where** $p = (x, y)$ **and** $z \in c\ x\ y$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *split-eta-SetCompr* [*simp*, *no-atp*]: $(\lambda u. \exists x\ y. u = (x, y) \wedge P\ (x, y)) = P$
 $\langle proof \rangle$

lemma *split-eta-SetCompr2* [*simp*, *no-atp*]: $(\lambda u. \exists x\ y. u = (x, y) \wedge P\ x\ y) = case\text{-}prod\ P$
 $\langle proof \rangle$

lemma *split-part* [*simp*]: $(\lambda(a,b). P \wedge Q\ a\ b) = (\lambda ab. P \wedge case\text{-}prod\ Q\ ab)$
 — Allows simplifications of nested splits in case of independent predicates.
 $\langle proof \rangle$

lemma *split-comp-eq*:

fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c$
and $g :: 'd \Rightarrow 'a$
shows $(\lambda u. f\ (g\ (fst\ u))\ (snd\ u)) = case\text{-}prod\ (\lambda x. f\ (g\ x))$
 $\langle proof \rangle$

lemma *pair-imageI* [*intro*]: $(a, b) \in A \implies f\ a\ b \in (\lambda(a, b). f\ a\ b) \text{ ` } A$
 $\langle proof \rangle$

lemma *Collect-const-case-prod*[*simp*]: $\{(a,b). P\} = (if\ P\ then\ UNIV\ else\ \{\})$
 $\langle proof \rangle$

lemma *The-split-eq* [*simp*]: $(THE\ (x', y'). x = x' \wedge y = y') = (x, y)$
 $\langle proof \rangle$

lemma *case-prod-beta*: $case\text{-}prod\ f\ p = f\ (fst\ p)\ (snd\ p)$
 $\langle proof \rangle$

lemma *prod-cases3* [*cases type*]:
obtains (*fields*) *a b c* **where** $y = (a, b, c)$
 $\langle \text{proof} \rangle$

lemma *prod-induct3* [*case-names fields, induct type*]:
 $(\bigwedge a\ b\ c. P\ (a, b, c)) \implies P\ x$
 $\langle \text{proof} \rangle$

lemma *prod-cases4* [*cases type*]:
obtains (*fields*) *a b c d* **where** $y = (a, b, c, d)$
 $\langle \text{proof} \rangle$

lemma *prod-induct4* [*case-names fields, induct type*]:
 $(\bigwedge a\ b\ c\ d. P\ (a, b, c, d)) \implies P\ x$
 $\langle \text{proof} \rangle$

lemma *prod-cases5* [*cases type*]:
obtains (*fields*) *a b c d e* **where** $y = (a, b, c, d, e)$
 $\langle \text{proof} \rangle$

lemma *prod-induct5* [*case-names fields, induct type*]:
 $(\bigwedge a\ b\ c\ d\ e. P\ (a, b, c, d, e)) \implies P\ x$
 $\langle \text{proof} \rangle$

lemma *prod-cases6* [*cases type*]:
obtains (*fields*) *a b c d e f* **where** $y = (a, b, c, d, e, f)$
 $\langle \text{proof} \rangle$

lemma *prod-induct6* [*case-names fields, induct type*]:
 $(\bigwedge a\ b\ c\ d\ e\ f. P\ (a, b, c, d, e, f)) \implies P\ x$
 $\langle \text{proof} \rangle$

lemma *prod-cases7* [*cases type*]:
obtains (*fields*) *a b c d e f g* **where** $y = (a, b, c, d, e, f, g)$
 $\langle \text{proof} \rangle$

lemma *prod-induct7* [*case-names fields, induct type*]:
 $(\bigwedge a\ b\ c\ d\ e\ f\ g. P\ (a, b, c, d, e, f, g)) \implies P\ x$
 $\langle \text{proof} \rangle$

definition *internal-case-prod* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
where *internal-case-prod* \equiv *case-prod*

lemma *internal-case-prod-conv*: *internal-case-prod* *c* (*a*, *b*) = *c a b*
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

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* = ($\lambda c\ x\ y.\ c\ (x, y)$)

lemma *curry-conv* [*simp*, *code*]: *curry* *f* *a* *b* = *f* (*a*, *b*)
 $\langle proof \rangle$

lemma *curryI* [*intro!*]: *f* (*a*, *b*) \Longrightarrow *curry* *f* *a* *b*
 $\langle proof \rangle$

lemma *curryD* [*dest!*]: *curry* *f* *a* *b* \Longrightarrow *f* (*a*, *b*)
 $\langle proof \rangle$

lemma *curryE*: *curry* *f* *a* *b* \Longrightarrow (*f* (*a*, *b*) \Longrightarrow *Q*) \Longrightarrow *Q*
 $\langle proof \rangle$

lemma *curry-case-prod* [*simp*]: *curry* (*case-prod* *f*) = *f*
 $\langle proof \rangle$

lemma *case-prod-curry* [*simp*]: *case-prod* (*curry* *f*) = *f*
 $\langle proof \rangle$

lemma *curry-K*: *curry* ($\lambda x.\ c$) = ($\lambda x\ y.\ c$)
 $\langle proof \rangle$

The composition-uncurry combinator.

definition *scomp* :: ($'a \Rightarrow 'b \times 'c$) $\Rightarrow ('b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$ (**infixl** $\langle \circ \rightarrow \rangle$ 60)

where *f* $\circ \rightarrow$ *g* = ($\lambda x.\ case\text{-}prod\ g\ (f\ x)$)

no-notation *scomp* (**infixl** $\langle \circ \rightarrow \rangle$ 60)

bundle *state-combinator-syntax*

begin

notation *fcomp* (**infixl** $\langle \circ > \rangle$ 60)

notation *scomp* (**infixl** $\langle \circ \rightarrow \rangle$ 60)

end

context

includes *state-combinator-syntax*

begin

lemma *scomp-unfold*: ($\circ \rightarrow$) = ($\lambda f\ g\ x.\ g\ (fst\ (f\ x))\ (snd\ (f\ x))$)
 $\langle proof \rangle$

lemma *scomp-apply* [*simp*]: $(f \circ \rightarrow g) x = \text{case-prod } g (f x)$
 $\langle \text{proof} \rangle$

lemma *Pair-scomp*: $\text{Pair } x \circ \rightarrow f = f x$
 $\langle \text{proof} \rangle$

lemma *scomp-Pair*: $x \circ \rightarrow \text{Pair} = x$
 $\langle \text{proof} \rangle$

lemma *scomp-scomp*: $(f \circ \rightarrow g) \circ \rightarrow h = f \circ \rightarrow (\lambda x. g x \circ \rightarrow h)$
 $\langle \text{proof} \rangle$

lemma *scomp-fcomp*: $(f \circ \rightarrow g) \circ > h = f \circ \rightarrow (\lambda x. g x \circ > h)$
 $\langle \text{proof} \rangle$

lemma *fcomp-scomp*: $(f \circ > g) \circ \rightarrow h = f \circ > (g \circ \rightarrow h)$
 $\langle \text{proof} \rangle$

end

code-printing

constant *scomp* $\rightarrow (\text{Eval})$ **infixl** 3 $\# \rightarrow$

map-prod — action of the product functor upon functions.

definition *map-prod* :: $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'd$
where *map-prod* $f g = (\lambda(x, y). (f x, g y))$

lemma *map-prod-simp* [*simp*, *code*]: $\text{map-prod } f g (a, b) = (f a, g b)$
 $\langle \text{proof} \rangle$

functor *map-prod*: *map-prod*
 $\langle \text{proof} \rangle$

lemma *fst-map-prod* [*simp*]: $\text{fst } (\text{map-prod } f g x) = f (\text{fst } x)$
 $\langle \text{proof} \rangle$

lemma *snd-map-prod* [*simp*]: $\text{snd } (\text{map-prod } f g x) = g (\text{snd } x)$
 $\langle \text{proof} \rangle$

lemma *fst-comp-map-prod* [*simp*]: $\text{fst} \circ \text{map-prod } f g = f \circ \text{fst}$
 $\langle \text{proof} \rangle$

lemma *snd-comp-map-prod* [*simp*]: $\text{snd} \circ \text{map-prod } f g = g \circ \text{snd}$
 $\langle \text{proof} \rangle$

lemma *map-prod-compose*: $\text{map-prod } (f1 \circ f2) (g1 \circ g2) = (\text{map-prod } f1 g1 \circ \text{map-prod } f2 g2)$
 $\langle \text{proof} \rangle$

lemma *map-prod-ident* [*simp*]: $\text{map-prod } (\lambda x. x) (\lambda y. y) = (\lambda z. z)$
 $\langle \text{proof} \rangle$

lemma *map-prod-imageI* [*intro*]: $(a, b) \in R \implies (f a, g b) \in \text{map-prod } f g \text{ ‘ } R$
 $\langle \text{proof} \rangle$

lemma *prod-fun-imageE* [*elim!*]:
 assumes *major*: $c \in \text{map-prod } f g \text{ ‘ } R$
 and *cases*: $\bigwedge x y. c = (f x, g y) \implies (x, y) \in R \implies P$
 shows P
 $\langle \text{proof} \rangle$

definition *apfst* :: $('a \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'b$
 where $\text{apfst } f = \text{map-prod } f \text{ id}$

definition *apsnd* :: $('b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'a \times 'c$
 where $\text{apsnd } f = \text{map-prod } \text{id } f$

lemma *apfst-conv* [*simp*, *code*]: $\text{apfst } f (x, y) = (f x, y)$
 $\langle \text{proof} \rangle$

lemma *apsnd-conv* [*simp*, *code*]: $\text{apsnd } f (x, y) = (x, f y)$
 $\langle \text{proof} \rangle$

lemma *fst-apfst* [*simp*]: $\text{fst } (\text{apfst } f x) = f (\text{fst } x)$
 $\langle \text{proof} \rangle$

lemma *fst-comp-apfst* [*simp*]: $\text{fst} \circ \text{apfst } f = f \circ \text{fst}$
 $\langle \text{proof} \rangle$

lemma *fst-apsnd* [*simp*]: $\text{fst } (\text{apsnd } f x) = \text{fst } x$
 $\langle \text{proof} \rangle$

lemma *fst-comp-apsnd* [*simp*]: $\text{fst} \circ \text{apsnd } f = \text{fst}$
 $\langle \text{proof} \rangle$

lemma *snd-apfst* [*simp*]: $\text{snd } (\text{apfst } f x) = \text{snd } x$
 $\langle \text{proof} \rangle$

lemma *snd-comp-apfst* [*simp*]: $\text{snd} \circ \text{apfst } f = \text{snd}$
 $\langle \text{proof} \rangle$

lemma *snd-apsnd* [*simp*]: $\text{snd } (\text{apsnd } f x) = f (\text{snd } x)$
 $\langle \text{proof} \rangle$

lemma *snd-comp-apsnd* [*simp*]: $\text{snd} \circ \text{apsnd } f = f \circ \text{snd}$
 $\langle \text{proof} \rangle$

lemma *apfst-compose*: $\text{apfst } f \ (\text{apfst } g \ x) = \text{apfst } (f \circ g) \ x$
 $\langle \text{proof} \rangle$

lemma *apsnd-compose*: $\text{apsnd } f \ (\text{apsnd } g \ x) = \text{apsnd } (f \circ g) \ x$
 $\langle \text{proof} \rangle$

lemma *apfst-apsnd [simp]*: $\text{apfst } f \ (\text{apsnd } g \ x) = (f \ (\text{fst } x), g \ (\text{snd } x))$
 $\langle \text{proof} \rangle$

lemma *apsnd-apfst [simp]*: $\text{apsnd } f \ (\text{apfst } g \ x) = (g \ (\text{fst } x), f \ (\text{snd } x))$
 $\langle \text{proof} \rangle$

lemma *apfst-id [simp]*: $\text{apfst } \text{id} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *apsnd-id [simp]*: $\text{apsnd } \text{id} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *apfst-eq-conv [simp]*: $\text{apfst } f \ x = \text{apfst } g \ x \longleftrightarrow f \ (\text{fst } x) = g \ (\text{fst } x)$
 $\langle \text{proof} \rangle$

lemma *apsnd-eq-conv [simp]*: $\text{apsnd } f \ x = \text{apsnd } g \ x \longleftrightarrow f \ (\text{snd } x) = g \ (\text{snd } x)$
 $\langle \text{proof} \rangle$

lemma *apsnd-apfst-commute*: $\text{apsnd } f \ (\text{apfst } g \ p) = \text{apfst } g \ (\text{apsnd } f \ p)$
 $\langle \text{proof} \rangle$

context
begin

$\langle ML \rangle$

definition *swap* :: $'a \times 'b \Rightarrow 'b \times 'a$
where *swap* $p = (\text{snd } p, \text{fst } p)$

end

lemma *swap-simp [simp]*: $\text{prod.swap } (x, y) = (y, x)$
 $\langle \text{proof} \rangle$

lemma *swap-swap [simp]*: $\text{prod.swap } (\text{prod.swap } p) = p$
 $\langle \text{proof} \rangle$

lemma *swap-comp-swap [simp]*: $\text{prod.swap} \circ \text{prod.swap} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *pair-in-swap-image [simp]*: $(y, x) \in \text{prod.swap} ` A \longleftrightarrow (x, y) \in A$
 $\langle \text{proof} \rangle$

lemma *inj-swap* [*simp*]: *inj-on prod.swap A*
 ⟨*proof*⟩

lemma *swap-inj-on*: *inj-on (λ(i, j). (j, i)) A*
 ⟨*proof*⟩

lemma *surj-swap* [*simp*]: *surj prod.swap*
 ⟨*proof*⟩

lemma *bij-swap* [*simp*]: *bij prod.swap*
 ⟨*proof*⟩

lemma *case-swap* [*simp*]: *(case prod.swap p of (y, x) ⇒ f x y) = (case p of (x, y) ⇒ f x y)*
 ⟨*proof*⟩

lemma *fst-swap* [*simp*]: *fst (prod.swap x) = snd x*
 ⟨*proof*⟩

lemma *snd-swap* [*simp*]: *snd (prod.swap x) = fst x*
 ⟨*proof*⟩

lemma *split-pairs*: *(A,B) = X ⟷ fst X = A ∧ snd X = B*
and *split-pairs2*: *X = (A,B) ⟷ fst X = A ∧ snd X = B*
 ⟨*proof*⟩

Disjoint union of a family of sets – Sigma.

definition *Sigma* :: *'a set ⇒ ('a ⇒ 'b set) ⇒ ('a × 'b) set*
where *Sigma A B* ≡ $\bigcup_{x \in A}. \bigcup_{y \in B} x. \{Pair\ x\ y\}$

context

begin

qualified abbreviation *Times* :: *'a set ⇒ 'b set ⇒ ('a × 'b) set* (**infixr** $\ltimes \rtimes$ 80)
where *A × B* ≡ *Sigma A (λ-. B)*

end

bundle *set-product-syntax*

begin

notation *Product-Type.Times* (**infixr** $\ltimes \rtimes$ 80)

end

syntax

-Sigma :: *pttrn ⇒ 'a set ⇒ 'b set ⇒ ('a × 'b) set*
 (⟨⟨indent=3 notation=⟨binder SIGMA⟩SIGMA -:-./ -⟩ [0, 0, 10] 10)

syntax-consts

-Sigma ≡ *Sigma*

translations

SIGMA x:A. B ≡ *CONST Sigma A (λx. B)*

lemma *SigmaI* [intro!]: $a \in A \implies b \in B \implies (a, b) \in \text{Sigma } A \ B$
 ⟨proof⟩

lemma *SigmaE* [elim!]: $c \in \text{Sigma } A \ B \implies (\bigwedge x y. x \in A \implies y \in B \implies c = (x, y) \implies P) \implies P$
 — The general elimination rule.
 ⟨proof⟩

Elimination of $(a, b) \in A \times B$ – introduces no eigenvariables.

lemma *SigmaD1*: $(a, b) \in \text{Sigma } A \ B \implies a \in A$
 ⟨proof⟩

lemma *SigmaD2*: $(a, b) \in \text{Sigma } A \ B \implies b \in B \ a$
 ⟨proof⟩

lemma *SigmaE2*: $(a, b) \in \text{Sigma } A \ B \implies (a \in A \implies b \in B \implies P) \implies P$
 ⟨proof⟩

lemma *Sigma-cong*: $A = B \implies (\bigwedge x. x \in B \implies C \ x = D \ x) \implies (\text{SIGMA } x:A. C \ x) = (\text{SIGMA } x:B. D \ x)$
 ⟨proof⟩

lemma *Sigma-mono*: $A \subseteq C \implies (\bigwedge x. x \in A \implies B \ x \subseteq D \ x) \implies \text{Sigma } A \ B \subseteq \text{Sigma } C \ D$
 ⟨proof⟩

lemma *Sigma-empty1* [simp]: $\text{Sigma } \{\} \ B = \{\}$
 ⟨proof⟩

lemma *Sigma-empty2* [simp]: $A \times \{\} = \{\}$
 ⟨proof⟩

lemma *UNIV-Times-UNIV* [simp]: $\text{UNIV} \times \text{UNIV} = \text{UNIV}$
 ⟨proof⟩

lemma *Compl-Times-UNIV1* [simp]: $-(\text{UNIV} \times A) = \text{UNIV} \times (-A)$
 ⟨proof⟩

lemma *Compl-Times-UNIV2* [simp]: $-(A \times \text{UNIV}) = (-A) \times \text{UNIV}$
 ⟨proof⟩

lemma *mem-Sigma-iff* [iff]: $(a, b) \in \text{Sigma } A \ B \longleftrightarrow a \in A \wedge b \in B \ a$
 ⟨proof⟩

lemma *mem-Times-iff*: $x \in A \times B \longleftrightarrow \text{fst } x \in A \wedge \text{snd } x \in B$
 ⟨proof⟩

lemma *Sigma-empty-iff*: $(\text{SIGMA } i:I. X \ i) = \{\} \longleftrightarrow (\forall i \in I. X \ i = \{\})$
 ⟨proof⟩

lemma *Times-subset-cancel2*: $x \in C \implies A \times C \subseteq B \times C \longleftrightarrow A \subseteq B$
 ⟨proof⟩

lemma *Times-eq-cancel2*: $x \in C \implies A \times C = B \times C \longleftrightarrow A = B$
 ⟨proof⟩

lemma *Collect-case-prod-Sigma*: $\{(x, y). P\ x \wedge Q\ x\ y\} = (\text{SIGMA } x:\text{Collect } P. \text{Collect } (Q\ x))$
 ⟨proof⟩

lemma *Collect-case-prod [simp]*: $\{(a, b). P\ a \wedge Q\ b\} = \text{Collect } P \times \text{Collect } Q$
 ⟨proof⟩

lemma *Collect-case-prodD*: $x \in \text{Collect } (\text{case-prod } A) \implies A\ (\text{fst } x)\ (\text{snd } x)$
 ⟨proof⟩

lemma *Collect-case-prod-mono*: $A \subseteq B \implies \text{Collect } (\text{case-prod } A) \subseteq \text{Collect } (\text{case-prod } B)$
 ⟨proof⟩

lemma *Collect-split-mono-strong*:
 $X = \text{fst } \text{' } A \implies Y = \text{snd } \text{' } A \implies \forall a \in X. \forall b \in Y. P\ a\ b \longrightarrow Q\ a\ b$
 $\implies A \subseteq \text{Collect } (\text{case-prod } P) \implies A \subseteq \text{Collect } (\text{case-prod } Q)$
 ⟨proof⟩

lemma *UN-Times-distrib*: $(\bigcup (a, b) \in A \times B. E\ a \times F\ b) = \bigcup (E\ \text{' } A) \times \bigcup (F\ \text{' } B)$
 — Suggested by Pierre Chartier
 ⟨proof⟩

lemma *split-paired-Ball-Sigma [simp, no-atp]*: $(\forall z \in \text{Sigma } A\ B. P\ z) \longleftrightarrow (\forall x \in A. \forall y \in B\ x. P\ (x, y))$
 ⟨proof⟩

lemma *split-paired-Bex-Sigma [simp, no-atp]*: $(\exists z \in \text{Sigma } A\ B. P\ z) \longleftrightarrow (\exists x \in A. \exists y \in B\ x. P\ (x, y))$
 ⟨proof⟩

lemma *Sigma-Un-distrib1*: $\text{Sigma } (I \cup J)\ C = \text{Sigma } I\ C \cup \text{Sigma } J\ C$
 ⟨proof⟩

lemma *Sigma-Un-distrib2*: $(\text{SIGMA } i:I. A\ i \cup B\ i) = \text{Sigma } I\ A \cup \text{Sigma } I\ B$
 ⟨proof⟩

lemma *Sigma-Int-distrib1*: $\text{Sigma } (I \cap J)\ C = \text{Sigma } I\ C \cap \text{Sigma } J\ C$
 ⟨proof⟩

lemma *Sigma-Int-distrib2*: $(\text{SIGMA } i:I. A\ i \cap B\ i) = \text{Sigma } I\ A \cap \text{Sigma } I\ B$
 ⟨proof⟩

lemma *Sigma-Diff-distrib1*: $\text{Sigma } (I - J) C = \text{Sigma } I C - \text{Sigma } J C$
 $\langle \text{proof} \rangle$

lemma *Sigma-Diff-distrib2*: $(\text{SIGMA } i:I. A i - B i) = \text{Sigma } I A - \text{Sigma } I B$
 $\langle \text{proof} \rangle$

lemma *Sigma-Union*: $\text{Sigma } (\bigcup X) B = (\bigcup A \in X. \text{Sigma } A B)$
 $\langle \text{proof} \rangle$

lemma *Pair-vimage-Sigma*: $\text{Pair } x - ' \text{Sigma } A f = (\text{if } x \in A \text{ then } f x \text{ else } \{\})$
 $\langle \text{proof} \rangle$

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma *Times-Un-distrib1*: $(A \cup B) \times C = A \times C \cup B \times C$
 $\langle \text{proof} \rangle$

lemma *Times-Int-distrib1*: $(A \cap B) \times C = A \times C \cap B \times C$
 $\langle \text{proof} \rangle$

lemma *Times-Diff-distrib1*: $(A - B) \times C = A \times C - B \times C$
 $\langle \text{proof} \rangle$

lemma *Times-empty [simp]*: $A \times B = \{\} \longleftrightarrow A = \{\} \vee B = \{\}$
 $\langle \text{proof} \rangle$

lemma *times-subset-iff*: $A \times C \subseteq B \times D \longleftrightarrow A = \{\} \vee C = \{\} \vee A \subseteq B \wedge C \subseteq D$
 $\langle \text{proof} \rangle$

lemma *times-eq-iff*: $A \times B = C \times D \longleftrightarrow A = C \wedge B = D \vee (A = \{\} \vee B = \{\}) \wedge (C = \{\} \vee D = \{\})$
 $\langle \text{proof} \rangle$

lemma *fst-image-times [simp]*: $\text{fst } ' (A \times B) = (\text{if } B = \{\} \text{ then } \{\} \text{ else } A)$
 $\langle \text{proof} \rangle$

lemma *snd-image-times [simp]*: $\text{snd } ' (A \times B) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } B)$
 $\langle \text{proof} \rangle$

lemma *fst-image-Sigma*: $\text{fst } ' (\text{Sigma } A B) = \{x \in A. B(x) \neq \{\}\}$
 $\langle \text{proof} \rangle$

lemma *snd-image-Sigma*: $\text{snd } ' (\text{Sigma } A B) = (\bigcup x \in A. B x)$
 $\langle \text{proof} \rangle$

lemma *vimage-fst*: $\text{fst } - ' A = A \times \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *vimage-snd*: $\text{snd} -' A = \text{UNIV} \times A$
 $\langle \text{proof} \rangle$

lemma *insert-Times-insert* [simp]:
 $\text{insert } a \ A \times \text{insert } b \ B = \text{insert } (a,b) \ (A \times \text{insert } b \ B \cup \{a\} \times B)$
 $\langle \text{proof} \rangle$

lemma *sing-Times-sing*: $\{x\} \times \{y\} = \{(x,y)\}$
 $\langle \text{proof} \rangle$

lemma *vimage-Times*: $f -' (A \times B) = (\text{fst} \circ f) -' A \cap (\text{snd} \circ f) -' B$
 $\langle \text{proof} \rangle$

lemma *Times-Int-Times*: $A \times B \cap C \times D = (A \cap C) \times (B \cap D)$
 $\langle \text{proof} \rangle$

lemma *image-paired-Times*:
 $(\lambda(x,y). (f \ x, g \ y)) -' (A \times B) = (f -' A) \times (g -' B)$
 $\langle \text{proof} \rangle$

lemma *Times-insert-right*: $A \times \text{insert } y \ B = (\lambda x. (x, y)) -' A \cup A \times B$
 $\langle \text{proof} \rangle$

lemma *Times-insert-left*: $\text{insert } x \ A \times B = (\lambda y. (x, y)) -' B \cup A \times B$
 $\langle \text{proof} \rangle$

lemma *product-swap*: $\text{prod.swap} -' (A \times B) = B \times A$
 $\langle \text{proof} \rangle$

lemma *swap-product*: $(\lambda(i, j). (j, i)) -' (A \times B) = B \times A$
 $\langle \text{proof} \rangle$

lemma *image-split-eq-Sigma*: $(\lambda x. (f \ x, g \ x)) -' A = \text{Sigma } (f -' A) (\lambda x. g -' (f -' \{x\} \cap A))$
 $\langle \text{proof} \rangle$

lemma *subset-fst-snd*: $A \subseteq (\text{fst} -' A \times \text{snd} -' A)$
 $\langle \text{proof} \rangle$

lemma *inj-on-apfst* [simp]: $\text{inj-on } (\text{apfst } f) \ (A \times \text{UNIV}) \longleftrightarrow \text{inj-on } f \ A$
 $\langle \text{proof} \rangle$

lemma *inj-apfst* [simp]: $\text{inj } (\text{apfst } f) \longleftrightarrow \text{inj } f$
 $\langle \text{proof} \rangle$

lemma *inj-on-apsnd* [simp]: $\text{inj-on } (\text{apsnd } f) \ (\text{UNIV} \times A) \longleftrightarrow \text{inj-on } f \ A$
 $\langle \text{proof} \rangle$

lemma *inj-apsnd* [simp]: $\text{inj } (\text{apsnd } f) \longleftrightarrow \text{inj } f$

```

⟨proof⟩

context
begin

qualified definition product :: 'a set ⇒ 'b set ⇒ ('a × 'b) set
  where [code-abbrev]: product A B = A × B

lemma member-product:  $x \in \text{Product-Type.product } A \ B \longleftrightarrow x \in A \times B$ 
  ⟨proof⟩

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 × B)
  ⟨proof⟩

lemma map-prod-surj:
  fixes f :: 'a ⇒ 'b
    and g :: 'c ⇒ 'd
  assumes surj f and surj g
  shows surj (map-prod f g)
  ⟨proof⟩

lemma map-prod-surj-on:
  assumes f ' A = A' and g ' B = B'
  shows map-prod f g ' (A × B) = A' × B'
  ⟨proof⟩

lemma bij-betw-map-prod:
  assumes bij-betw f A C bij-betw g B D
  shows bij-betw (map-prod f g) (A × B) (C × D)
  ⟨proof⟩

```

14.4 Code generator setup for paired and tripled bounded set comprehension

```

context
begin

qualified lemma paired-bounded-Collect-eq-filter [code-unfold, no-atp]:
  ⟨{(x, y). (x, y) ∈ A ∧ P x y} = Set.filter (λ(x, y). P x y) A⟩
  ⟨proof⟩ lemma tripled-bounded-Collect-eq-filter [code-unfold, no-atp]:
  ⟨{(x, y, z). (x, y, z) ∈ A ∧ P x y z} = Set.filter (λ(x, y, z). P x y z) A⟩
  ⟨proof⟩

```

end

14.5 Simproc for rewriting a set comprehension into a point-free expression

$\langle ML \rangle$

14.6 Lemmas about disjointness

lemma *disjnt-Times1-iff* [simp]: $\text{disjnt } (C \times A) (C \times B) \longleftrightarrow C = \{\} \vee \text{disjnt } A\ B$
 $\langle proof \rangle$

lemma *disjnt-Times2-iff* [simp]: $\text{disjnt } (A \times C) (B \times C) \longleftrightarrow C = \{\} \vee \text{disjnt } A\ B$
 $\langle proof \rangle$

lemma *disjnt-Sigma-iff*: $\text{disjnt } (\text{Sigma } A\ C) (\text{Sigma } B\ C) \longleftrightarrow (\forall i \in A \cap B. C\ i = \{\}) \vee \text{disjnt } A\ B$
 $\langle proof \rangle$

14.7 Inductively defined sets

$\langle ML \rangle$

14.8 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* :: $'a \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool} \Rightarrow \text{bool}$

where $Inl\text{-}Rep\ a\ x\ y\ p \longleftrightarrow x = a \wedge p$

definition $Inr\text{-}Rep :: 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool \Rightarrow bool$

where $Inr\text{-}Rep\ b\ x\ y\ p \longleftrightarrow y = b \wedge \neg p$

definition $sum = \{f. (\exists a. f = Inl\text{-}Rep\ (a::'a)) \vee (\exists b. f = Inr\text{-}Rep\ (b::'b))\}$

typedef $('a, 'b)\ sum\ (\text{infixr } \langle + \rangle\ 10) = sum :: ('a \Rightarrow 'b \Rightarrow bool \Rightarrow bool)\ set$
 $\langle proof \rangle$

lemma $Inl\text{-}RepI: Inl\text{-}Rep\ a \in sum$
 $\langle proof \rangle$

lemma $Inr\text{-}RepI: Inr\text{-}Rep\ b \in sum$
 $\langle proof \rangle$

lemma $inj\text{-}on\text{-}Abs\text{-}sum: A \subseteq sum \Longrightarrow inj\text{-}on\ Abs\text{-}sum\ A$
 $\langle proof \rangle$

lemma $Inl\text{-}Rep\text{-}inject: inj\text{-}on\ Inl\text{-}Rep\ A$
 $\langle proof \rangle$

lemma $Inr\text{-}Rep\text{-}inject: inj\text{-}on\ Inr\text{-}Rep\ A$
 $\langle proof \rangle$

lemma $Inl\text{-}Rep\text{-}not\text{-}Inr\text{-}Rep: Inl\text{-}Rep\ a \neq Inr\text{-}Rep\ b$
 $\langle proof \rangle$

definition $Inl :: 'a \Rightarrow 'a + 'b$
where $Inl = Abs\text{-}sum \circ Inl\text{-}Rep$

definition $Inr :: 'b \Rightarrow 'a + 'b$
where $Inr = Abs\text{-}sum \circ Inr\text{-}Rep$

lemma $inj\text{-}Inl\ [simp]: inj\text{-}on\ Inl\ A$
 $\langle proof \rangle$

lemma $Inl\text{-}inject: Inl\ x = Inl\ y \Longrightarrow x = y$
 $\langle proof \rangle$

lemma $inj\text{-}Inr\ [simp]: inj\text{-}on\ Inr\ A$
 $\langle proof \rangle$

lemma $Inr\text{-}inject: Inr\ x = Inr\ y \Longrightarrow x = y$
 $\langle proof \rangle$

lemma $Inl\text{-}not\text{-}Inr: Inl\ a \neq Inr\ b$
 $\langle proof \rangle$

lemma *Inr-not-Inl*: $Inr\ b \neq Inl\ a$
 $\langle proof \rangle$

lemma *sumE*:
 assumes $\bigwedge x::'a. s = Inl\ x \implies P$
 and $\bigwedge y::'b. s = Inr\ y \implies P$
 shows P
 $\langle proof \rangle$

free-constructors *case-sum* **for**
isl: $Inl\ projl$
 $|$ *Inr* $projr$
 $\langle proof \rangle$

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

$\langle ML \rangle$

old-rep-datatype *Inl Inr*
 $\langle proof \rangle$

$\langle ML \rangle$

But erase the prefix for properties that are not generated by *free-constructors*.

$\langle ML \rangle$

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*

$\langle ML \rangle$

primrec *map-sum* :: $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow 'a + 'b \Rightarrow 'c + 'd$
where
 $map-sum\ f1\ f2\ (Inl\ a) = Inl\ (f1\ a)$
 $| map-sum\ f1\ f2\ (Inr\ a) = Inr\ (f2\ a)$

functor *map-sum*: *map-sum*
 $\langle proof \rangle$

lemma *split-sum-all*: $(\forall x. P\ x) \longleftrightarrow (\forall x. P\ (Inl\ x)) \wedge (\forall x. P\ (Inr\ x))$
 $\langle proof \rangle$

lemma *split-sum-ex*: $(\exists x. P\ x) \longleftrightarrow (\exists x. P\ (Inl\ x)) \vee (\exists x. P\ (Inr\ x))$
 $\langle proof \rangle$

15.2 Projections

lemma *case-sum-KK* [*simp*]: $\text{case-sum } (\lambda x. a) (\lambda x. a) = (\lambda x. a)$
 $\langle \text{proof} \rangle$

lemma *surjective-sum*: $\text{case-sum } (\lambda x::'a. f \text{ (Inl } x)) (\lambda y::'b. f \text{ (Inr } y)) = f$
 $\langle \text{proof} \rangle$

lemma *case-sum-inject*:
 assumes $a: \text{case-sum } f1 \ f2 = \text{case-sum } g1 \ g2$
 and $r: f1 = g1 \implies f2 = g2 \implies P$
 shows P
 $\langle \text{proof} \rangle$

primrec *Suml* :: $('a \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$
 where $\text{Suml } f \text{ (Inl } x) = f \ x$

primrec *Sumr* :: $('b \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$
 where $\text{Sumr } f \text{ (Inr } x) = f \ x$

lemma *Suml-inject*:
 assumes $\text{Suml } f = \text{Suml } g$
 shows $f = g$
 $\langle \text{proof} \rangle$

lemma *Sumr-inject*:
 assumes $\text{Sumr } f = \text{Sumr } g$
 shows $f = g$
 $\langle \text{proof} \rangle$

15.3 The Disjoint Sum of Sets

definition *Plus* :: $'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow ('a + 'b) \text{ set}$ (**infixr** $\langle + \rangle$ 65)
 where $A \langle + \rangle B = \text{Inl } ` A \cup \text{Inr } ` B$

hide-const (**open**) *Plus* — Valuable identifier

lemma *InlI* [*intro!*]: $a \in A \implies \text{Inl } a \in A \langle + \rangle B$
 $\langle \text{proof} \rangle$

lemma *InrI* [*intro!*]: $b \in B \implies \text{Inr } b \in A \langle + \rangle B$
 $\langle \text{proof} \rangle$

Exhaustion rule for sums, a degenerate form of induction

lemma *PlusE* [*elim!*]:
 $u \in A \langle + \rangle B \implies (\bigwedge x. x \in A \implies u = \text{Inl } x \implies P) \implies (\bigwedge y. y \in B \implies u = \text{Inr } y \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *Plus-eq-empty-conv* [*simp*]: $A \langle + \rangle B = \{\} \longleftrightarrow A = \{\} \wedge B = \{\}$

$\langle proof \rangle$

lemma *UNIV-Plus-UNIV* [simp]: $UNIV <+> UNIV = UNIV$
 $\langle proof \rangle$

lemma *UNIV-sum*: $UNIV = Inl \cdot UNIV \cup Inr \cdot UNIV$
 $\langle proof \rangle$

hide-const (open) *Suml Sumr sum*

end

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*: $a * e + (b * e + c) = (a + b) * e + c$
 $\langle proof \rangle$

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*: $a * b \neq 0 \implies a \neq 0 \wedge b \neq 0$
 $\langle proof \rangle$

end

class *semiring-0* = *semiring* + *comm-monoid-add* + *mult-zero*

class *semiring-0-cancel* = *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0*

<proof>

end

class *comm-semiring* = *ab-semigroup-add* + *ab-semigroup-mult* +
assumes *distrib*: $(a + b) * c = a * c + b * c$
begin

subclass *semiring*
<proof>

end

class *comm-semiring-0* = *comm-semiring* + *comm-monoid-add* + *mult-zero*
begin

subclass *semiring-0* *<proof>*

end

class *comm-semiring-0-cancel* = *comm-semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* *<proof>*

subclass *comm-semiring-0* *<proof>*

end

class *zero-neq-one* = *zero* + *one* +
assumes *zero-neq-one* [*simp*]: $0 \neq 1$
begin

lemma *one-neq-zero* [*simp*]: $1 \neq 0$
<proof>

definition *of-bool* :: $bool \Rightarrow 'a$
where *of-bool* *p* = (*if* *p* *then* 1 *else* 0)

lemma *of-bool-eq* [*simp*, *code*]:
of-bool False = 0
of-bool True = 1
<proof>

lemma *of-bool-eq-iff*: $of\text{-}bool\ p = of\text{-}bool\ q \longleftrightarrow p = q$
<proof>

lemma *split-of-bool* [*split*]: $P\ (of\text{-}bool\ p) \longleftrightarrow (p \longrightarrow P\ 1) \wedge (\neg p \longrightarrow P\ 0)$
<proof>

lemma *split-of-bool-asm*: $P \text{ (of-bool } p) \longleftrightarrow \neg (p \wedge \neg P \ 1 \vee \neg p \wedge \neg P \ 0)$
 $\langle \text{proof} \rangle$

lemma *of-bool-eq-0-iff* [*simp*]:
 $\langle \text{of-bool } P = 0 \longleftrightarrow \neg P \rangle$
 $\langle \text{proof} \rangle$

lemma *of-bool-eq-1-iff* [*simp*]:
 $\langle \text{of-bool } P = 1 \longleftrightarrow P \rangle$
 $\langle \text{proof} \rangle$

end

class *semiring-1* = *zero-neq-one* + *semiring-0* + *monoid-mult*
begin

lemma *of-bool-conj*:
 $\text{of-bool } (P \wedge Q) = \text{of-bool } P * \text{of-bool } Q$
 $\langle \text{proof} \rangle$

end

lemma *lambda-zero*: $(\lambda h::'a::\text{mult-zero}. 0) = (*) \ 0$
 $\langle \text{proof} \rangle$

lemma *lambda-one*: $(\lambda x::'a::\text{monoid-mult}. x) = (*) \ 1$
 $\langle \text{proof} \rangle$

16.2 Abstract divisibility

class *dvd* = *times*
begin

definition *dvd* :: $'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** $\langle \text{dvd} \rangle \ 50$)
where $b \text{ dvd } a \longleftrightarrow (\exists k. a = b * k)$

lemma *dvdI* [*intro?*]: $a = b * k \Longrightarrow b \text{ dvd } a$
 $\langle \text{proof} \rangle$

lemma *dvdE* [*elim*]: $b \text{ dvd } a \Longrightarrow (\bigwedge k. a = b * k \Longrightarrow P) \Longrightarrow P$
 $\langle \text{proof} \rangle$

end

context *comm-monoid-mult*
begin

subclass *dvd* $\langle \text{proof} \rangle$

lemma *dvd-refl* [*simp*]: $a \text{ dvd } a$
 $\langle \text{proof} \rangle$

lemma *dvd-trans* [*trans*]:
 assumes $a \text{ dvd } b$ and $b \text{ dvd } c$
 shows $a \text{ dvd } c$
 $\langle \text{proof} \rangle$

lemma *subset-divisors-dvd*: $\{c. c \text{ dvd } a\} \subseteq \{c. c \text{ dvd } b\} \longleftrightarrow a \text{ dvd } b$
 $\langle \text{proof} \rangle$

lemma *strict-subset-divisors-dvd*: $\{c. c \text{ dvd } a\} \subset \{c. c \text{ dvd } b\} \longleftrightarrow a \text{ dvd } b \wedge \neg b \text{ dvd } a$
 $\langle \text{proof} \rangle$

lemma *one-dvd* [*simp*]: $1 \text{ dvd } a$
 $\langle \text{proof} \rangle$

lemma *dvd-mult* [*simp*]: $a \text{ dvd } (b * c)$ if $a \text{ dvd } c$
 $\langle \text{proof} \rangle$

lemma *dvd-mult2* [*simp*]: $a \text{ dvd } (b * c)$ if $a \text{ dvd } b$
 $\langle \text{proof} \rangle$

lemma *dvd-triv-right* [*simp*]: $a \text{ dvd } b * a$
 $\langle \text{proof} \rangle$

lemma *dvd-triv-left* [*simp*]: $a \text{ dvd } a * b$
 $\langle \text{proof} \rangle$

lemma *mult-dvd-mono*:
 assumes $a \text{ dvd } b$
 and $c \text{ dvd } d$
 shows $a * c \text{ dvd } b * d$
 $\langle \text{proof} \rangle$

lemma *dvd-mult-left*: $a * b \text{ dvd } c \implies a \text{ dvd } c$
 $\langle \text{proof} \rangle$

lemma *dvd-mult-right*: $a * b \text{ dvd } c \implies b \text{ dvd } c$
 $\langle \text{proof} \rangle$

end

class *comm-semiring-1* = *zero-neg-one* + *comm-semiring-0* + *comm-monoid-mult*
begin

subclass *semiring-1* $\langle \text{proof} \rangle$

lemma *dvd-0-left-iff* [*simp*]: $0 \text{ dvd } a \longleftrightarrow a = 0$
 ⟨*proof*⟩

lemma *dvd-0-right* [*iff*]: $a \text{ dvd } 0$
 ⟨*proof*⟩

lemma *dvd-0-left*: $0 \text{ dvd } a \implies a = 0$
 ⟨*proof*⟩

lemma *dvd-add* [*simp*]:
 assumes $a \text{ dvd } b$ and $a \text{ dvd } c$
 shows $a \text{ dvd } (b + c)$
 ⟨*proof*⟩

end

class *semiring-1-cancel* = *semiring* + *cancel-comm-monoid-add*
 + *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *semiring-1* ⟨*proof*⟩

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* ⟨*proof*⟩

subclass *comm-semiring-0-cancel* ⟨*proof*⟩

subclass *comm-semiring-1* ⟨*proof*⟩

lemma *left-diff-distrib'* [*algebra-simps*, *algebra-split-simps*]:
 $(b - c) * a = b * a - c * a$
 ⟨*proof*⟩

lemma *dvd-add-times-triv-left-iff* [*simp*]: $a \text{ dvd } c * a + b \longleftrightarrow a \text{ dvd } b$
 ⟨*proof*⟩

lemma *dvd-add-times-triv-right-iff* [*simp*]: $a \text{ dvd } b + c * a \longleftrightarrow a \text{ dvd } b$
 ⟨*proof*⟩

lemma *dvd-add-triv-left-iff* [*simp*]: $a \text{ dvd } a + b \longleftrightarrow a \text{ dvd } b$

$\langle proof \rangle$

lemma *dvd-add-triv-right-iff* [simp]: $a \text{ dvd } b + a \longleftrightarrow a \text{ dvd } b$
 $\langle proof \rangle$

lemma *dvd-add-right-iff*:
assumes $a \text{ dvd } b$
shows $a \text{ dvd } b + c \longleftrightarrow a \text{ dvd } c$ (**is** $?P \longleftrightarrow ?Q$)
 $\langle proof \rangle$

lemma *dvd-add-left-iff*: $a \text{ dvd } c \implies a \text{ dvd } b + c \longleftrightarrow a \text{ dvd } b$
 $\langle proof \rangle$

end

class *ring* = *semiring* + *ab-group-add*
begin

subclass *semiring-0-cancel* $\langle proof \rangle$

Distribution rules

lemma *minus-mult-left*: $-(a * b) = -a * b$
 $\langle proof \rangle$

lemma *minus-mult-right*: $-(a * b) = a * -b$
 $\langle proof \rangle$

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]: $-a * -b = a * b$
 $\langle proof \rangle$

lemma *minus-mult-commute*: $-a * b = a * -b$
 $\langle proof \rangle$

lemma *right-diff-distrib* [algebra-simps, algebra-split-simps]:
 $a * (b - c) = a * b - a * c$
 $\langle proof \rangle$

lemma *left-diff-distrib* [algebra-simps, algebra-split-simps]:
 $(a - b) * c = a * c - b * c$
 $\langle proof \rangle$

lemmas *ring-distrib* = *distrib-left distrib-right left-diff-distrib right-diff-distrib*

lemma *eq-add-iff1*: $a * e + c = b * e + d \longleftrightarrow (a - b) * e + c = d$
 $\langle proof \rangle$

lemma *eq-add-iff2*: $a * e + c = b * e + d \longleftrightarrow c = (b - a) * e + d$
 ⟨*proof*⟩

end

lemmas *ring-distrib* = *distrib-left distrib-right left-diff-distrib right-diff-distrib*

class *comm-ring* = *comm-semiring* + *ab-group-add*
begin

subclass *ring* ⟨*proof*⟩

subclass *comm-semiring-0-cancel* ⟨*proof*⟩

lemma *square-diff-square-factored*: $x * x - y * y = (x + y) * (x - y)$
 ⟨*proof*⟩

end

class *ring-1* = *ring* + *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-1-cancel* ⟨*proof*⟩

lemma *of-bool-not-iff*:
 ⟨*of-bool* $(\neg P) = 1 - \text{of-bool } P$ ⟩
 ⟨*proof*⟩

lemma *square-diff-one-factored*: $x * x - 1 = (x + 1) * (x - 1)$
 ⟨*proof*⟩

end

class *comm-ring-1* = *comm-ring* + *zero-neq-one* + *comm-monoid-mult*
begin

subclass *ring-1* ⟨*proof*⟩

subclass *comm-semiring-1-cancel*
 ⟨*proof*⟩

lemma *dvd-minus-iff* [*simp*]: $x \text{ dvd } - y \longleftrightarrow x \text{ dvd } y$
 ⟨*proof*⟩

lemma *minus-dvd-iff* [*simp*]: $- x \text{ dvd } y \longleftrightarrow x \text{ dvd } y$
 ⟨*proof*⟩

lemma *dvd-diff-right-iff*:
assumes $a \text{ dvd } b$
shows $a \text{ dvd } b - c \longleftrightarrow a \text{ dvd } c$ (**is** $?P \longleftrightarrow ?Q$)

$\langle proof \rangle$

lemma *dvd-diff-left-iff*:

shows $a \text{ dvd } c \implies a \text{ dvd } b - c \longleftrightarrow a \text{ dvd } b$

$\langle proof \rangle$

lemma *dvd-diff [simp]*: $x \text{ dvd } y \implies x \text{ dvd } z \implies x \text{ dvd } (y - z)$

$\langle proof \rangle$

end

16.3 Towards integral domains

class *semiring-no-zero-divisors* = *semiring-0* +

assumes *no-zero-divisors*: $a \neq 0 \implies b \neq 0 \implies a * b \neq 0$

begin

lemma *divisors-zero*:

assumes $a * b = 0$

shows $a = 0 \vee b = 0$

$\langle proof \rangle$

lemma *mult-eq-0-iff [simp]*: $a * b = 0 \longleftrightarrow a = 0 \vee b = 0$

$\langle proof \rangle$

end

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]*: $a * c = b * c \longleftrightarrow c = 0 \vee a = b$

and *mult-cancel-left [simp]*: $c * a = c * b \longleftrightarrow c = 0 \vee a = b$

begin

lemma *mult-left-cancel*: $c \neq 0 \implies c * a = c * b \longleftrightarrow a = b$

$\langle proof \rangle$

lemma *mult-right-cancel*: $c \neq 0 \implies a * c = b * c \longleftrightarrow a = b$

$\langle proof \rangle$

end

class *ring-no-zero-divisors* = *ring* + *semiring-no-zero-divisors*

begin

subclass *semiring-no-zero-divisors-cancel*

$\langle proof \rangle$

end

```

class ring-1-no-zero-divisors = ring-1 + ring-no-zero-divisors
begin

subclass semiring-1-no-zero-divisors ⟨proof⟩

lemma square-eq-1-iff:  $x * x = 1 \longleftrightarrow x = 1 \vee x = -1$ 
  ⟨proof⟩

lemma mult-cancel-right1 [simp]:  $c = b * c \longleftrightarrow c = 0 \vee b = 1$ 
  ⟨proof⟩

lemma mult-cancel-right2 [simp]:  $a * c = c \longleftrightarrow c = 0 \vee a = 1$ 
  ⟨proof⟩

lemma mult-cancel-left1 [simp]:  $c = c * b \longleftrightarrow c = 0 \vee b = 1$ 
  ⟨proof⟩

lemma mult-cancel-left2 [simp]:  $c * a = c \longleftrightarrow c = 0 \vee a = 1$ 
  ⟨proof⟩

end

class semidom = comm-semiring-1-cancel + semiring-no-zero-divisors
begin

subclass semiring-1-no-zero-divisors ⟨proof⟩

end

class idom = comm-ring-1 + semiring-no-zero-divisors
begin

subclass semidom ⟨proof⟩

subclass ring-1-no-zero-divisors ⟨proof⟩

lemma dvd-mult-cancel-right [simp]:
   $a * c \text{ dvd } b * c \longleftrightarrow c = 0 \vee a \text{ dvd } b$ 
  ⟨proof⟩

lemma dvd-mult-cancel-left [simp]:
   $c * a \text{ dvd } c * b \longleftrightarrow c = 0 \vee a \text{ dvd } b$ 
  ⟨proof⟩

lemma square-eq-iff:  $a * a = b * b \longleftrightarrow a = b \vee a = -b$ 
  ⟨proof⟩

lemma inj-mult-left [simp]:  $\langle \text{inj } ((*) a) \longleftrightarrow a \neq 0 \rangle \text{ (is } \langle ?P \longleftrightarrow ?Q \rangle)$ 

```


$\langle proof \rangle$

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*: $sgn\ (-\ 1) = -\ 1$
and *sgn-mult*: $sgn\ (a * b) = sgn\ a * sgn\ b$
begin

lemma *sgn-eq-0-iff*:
 $sgn\ a = 0 \longleftrightarrow a = 0$
 $\langle proof \rangle$

lemma *abs-eq-0-iff*:
 $|a| = 0 \longleftrightarrow a = 0$
 $\langle proof \rangle$

lemma *abs-mult-sgn*:
 $|a| * sgn\ a = a$
 $\langle proof \rangle$

lemma *abs-1* [*simp*]:
 $|1| = 1$
 $\langle proof \rangle$

lemma *sgn-abs* [*simp*]:
 $|sgn\ a| = of_bool\ (a \neq 0)$
 $\langle proof \rangle$

lemma *abs-sgn* [*simp*]:
 $sgn\ |a| = of_bool\ (a \neq 0)$
 $\langle proof \rangle$

lemma *abs-mult*:
 $|a * b| = |a| * |b|$
 $\langle proof \rangle$

lemma *sgn-minus* [*simp*]:
 $sgn\ (-\ a) = -\ sgn\ a$
 $\langle proof \rangle$

lemma *abs-minus* [*simp*]:
 $|- a| = |a|$

$\langle proof \rangle$

end

16.4 (Partial) Division

class *divide* =
 fixes *divide* :: 'a \Rightarrow 'a \Rightarrow 'a (**infixl** $\langle div \rangle$ 70)

$\langle ML \rangle$

context *semiring*
begin

lemma [*field-simps*, *field-split-simps*]:
 shows *distrib-left-NO-MATCH*: *NO-MATCH* $(x \text{ div } y) \ a \Longrightarrow a * (b + c) = a * b + a * c$
 and *distrib-right-NO-MATCH*: *NO-MATCH* $(x \text{ div } y) \ c \Longrightarrow (a + b) * c = a * c + b * c$
 $\langle proof \rangle$

end

context *ring*
begin

lemma [*field-simps*, *field-split-simps*]:
 shows *left-diff-distrib-NO-MATCH*: *NO-MATCH* $(x \text{ div } y) \ c \Longrightarrow (a - b) * c = a * c - b * c$
 and *right-diff-distrib-NO-MATCH*: *NO-MATCH* $(x \text{ div } y) \ a \Longrightarrow a * (b - c) = a * b - a * c$
 $\langle proof \rangle$

end

$\langle ML \rangle$

class *divide-trivial* = *zero* + *one* + *divide* +
 assumes *div-by-0* [*simp*]: $\langle a \text{ div } 0 = 0 \rangle$
 and *div-by-1* [*simp*]: $\langle a \text{ div } 1 = a \rangle$
 and *div-0* [*simp*]: $\langle 0 \text{ div } a = 0 \rangle$

Algebraic classes with division

class *semidom-divide* = *semidom* + *divide* +
 assumes *nonzero-mult-div-cancel-right* [*simp*]: $\langle b \neq 0 \Longrightarrow (a * b) \text{ div } b = a \rangle$
 assumes *semidom-div-by-0*: $\langle a \text{ div } 0 = 0 \rangle$
begin

lemma *nonzero-mult-div-cancel-left* [*simp*]: $\langle a \neq 0 \Longrightarrow (a * b) \text{ div } a = b \rangle$

$\langle proof \rangle$

subclass *divide-trivial*

$\langle proof \rangle$

subclass *semiring-no-zero-divisors-cancel*

$\langle proof \rangle$

lemma *div-self* [simp]: $a \neq 0 \implies a \text{ div } a = 1$

$\langle proof \rangle$

lemma *dvd-div-eq-0-iff*:

assumes $b \text{ dvd } a$

shows $a \text{ div } b = 0 \iff a = 0$

$\langle proof \rangle$

lemma *dvd-div-eq-cancel*:

$a \text{ div } c = b \text{ div } c \implies c \text{ dvd } a \implies c \text{ dvd } b \implies a = b$

$\langle proof \rangle$

lemma *dvd-div-eq-iff*:

$c \text{ dvd } a \implies c \text{ dvd } b \implies a \text{ div } c = b \text{ div } c \iff a = b$

$\langle proof \rangle$

lemma *inj-on-mult*:

inj-on $((*) a) A$ **if** $a \neq 0$

$\langle proof \rangle$

end

class *idom-divide* = *idom* + *semidom-divide*

begin

lemma *dvd-neg-div*:

assumes $b \text{ dvd } a$

shows $-a \text{ div } b = -(a \text{ div } b)$

$\langle proof \rangle$

lemma *dvd-div-neg*:

assumes $b \text{ dvd } a$

shows $a \text{ div } -b = -(a \text{ div } b)$

$\langle proof \rangle$

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 $a \neq 0$
 shows $a * b \text{ dvd } a * c \longleftrightarrow b \text{ dvd } c$
 (is ?lhs \longleftrightarrow ?rhs)
 <proof>

lemma *dvd-times-right-cancel-iff* [simp]:
 assumes $a \neq 0$
 shows $b * a \text{ dvd } c * a \longleftrightarrow b \text{ dvd } c$
 <proof>

lemma *div-dvd-iff-mult*:
 assumes $b \neq 0$ and $b \text{ dvd } a$
 shows $a \text{ div } b \text{ dvd } c \longleftrightarrow a \text{ dvd } c * b$
 <proof>

lemma *dvd-div-iff-mult*:
 assumes $c \neq 0$ and $c \text{ dvd } b$
 shows $a \text{ dvd } b \text{ div } c \longleftrightarrow a * c \text{ dvd } b$
 <proof>

lemma *div-dvd-div* [simp]:
 assumes $a \text{ dvd } b$ and $a \text{ dvd } c$
 shows $b \text{ div } a \text{ dvd } c \text{ div } a \longleftrightarrow b \text{ dvd } c$
 <proof>

lemma *div-add* [simp]:
 assumes $c \text{ dvd } a$ and $c \text{ dvd } b$
 shows $(a + b) \text{ div } c = a \text{ div } c + b \text{ div } c$
 <proof>

lemma *div-mult-div-if-dvd*:
 assumes $b \text{ dvd } a$ and $d \text{ dvd } c$
 shows $(a \text{ div } b) * (c \text{ div } d) = (a * c) \text{ div } (b * d)$
 <proof>

lemma *dvd-div-eq-mult*:
 assumes $a \neq 0$ and $a \text{ dvd } b$
 shows $b \text{ div } a = c \longleftrightarrow b = c * a$
 (is ?lhs \longleftrightarrow ?rhs)
 <proof>

lemma *dvd-div-mult-self* [simp]: $a \text{ dvd } b \implies b \text{ div } a * a = b$
 <proof>

lemma *dvd-mult-div-cancel* [simp]: $a \text{ dvd } b \implies a * (b \text{ div } a) = b$
 <proof>

lemma *div-mult-swap*:

assumes $c \text{ dvd } b$

shows $a * (b \text{ div } c) = (a * b) \text{ div } c$

$\langle \text{proof} \rangle$

lemma *dvd-div-mult*: $c \text{ dvd } b \implies b \text{ div } c * a = (b * a) \text{ div } c$

$\langle \text{proof} \rangle$

lemma *dvd-div-mult2-eq*:

assumes $b * c \text{ dvd } a$

shows $a \text{ div } (b * c) = a \text{ div } b \text{ div } c$

$\langle \text{proof} \rangle$

lemma *dvd-div-div-eq-mult*:

assumes $a \neq 0$ $c \neq 0$ **and** $a \text{ dvd } b$ $c \text{ dvd } d$

shows $b \text{ div } a = d \text{ div } c \longleftrightarrow b * c = a * d$

(**is** $?lhs \longleftrightarrow ?rhs$)

$\langle \text{proof} \rangle$

lemma *dvd-mult-imp-div*:

assumes $a * c \text{ dvd } b$

shows $a \text{ dvd } b \text{ div } c$

$\langle \text{proof} \rangle$

lemma *div-div-eq-right*:

assumes $c \text{ dvd } b$ $b \text{ dvd } a$

shows $a \text{ div } (b \text{ div } c) = a \text{ div } b * c$

$\langle \text{proof} \rangle$

lemma *div-div-div-same*:

assumes $d \text{ dvd } b$ $b \text{ dvd } a$

shows $(a \text{ div } d) \text{ div } (b \text{ div } d) = a \text{ div } b$

$\langle \text{proof} \rangle$

Units: invertible elements in a ring

abbreviation *is-unit* :: $'a \Rightarrow \text{bool}$

where $\text{is-unit } a \equiv a \text{ dvd } 1$

lemma *not-is-unit-0* [simp]: $\neg \text{is-unit } 0$

$\langle \text{proof} \rangle$

lemma *unit-imp-dvd* [dest]: $\text{is-unit } b \implies b \text{ dvd } a$

$\langle \text{proof} \rangle$

lemma *unit-dvdE*:

assumes $\text{is-unit } a$

obtains c **where** $a \neq 0$ **and** $b = a * c$

$\langle \text{proof} \rangle$

lemma *dvd-unit-imp-unit*: $a \text{ dvd } b \implies \text{is-unit } b \implies \text{is-unit } a$
 $\langle \text{proof} \rangle$

lemma *unit-div-1-unit* [*simp*, *intro*]:
assumes *is-unit a*
shows *is-unit (1 div a)*
 $\langle \text{proof} \rangle$

lemma *is-unitE* [*elim?*]:
assumes *is-unit a*
obtains *b* **where** $a \neq 0$ **and** $b \neq 0$
and *is-unit b* **and** $1 \text{ div } a = b$ **and** $1 \text{ div } b = a$
and $a * b = 1$ **and** $c \text{ div } a = c * b$
 $\langle \text{proof} \rangle$

lemma *unit-prod* [*intro*]: $\text{is-unit } a \implies \text{is-unit } b \implies \text{is-unit } (a * b)$
 $\langle \text{proof} \rangle$

lemma *is-unit-mult-iff*: $\text{is-unit } (a * b) \longleftrightarrow \text{is-unit } a \wedge \text{is-unit } b$
 $\langle \text{proof} \rangle$

lemma *unit-div* [*intro*]: $\text{is-unit } a \implies \text{is-unit } b \implies \text{is-unit } (a \text{ div } b)$
 $\langle \text{proof} \rangle$

lemma *mult-unit-dvd-iff*:
assumes *is-unit b*
shows $a * b \text{ dvd } c \longleftrightarrow a \text{ dvd } c$
 $\langle \text{proof} \rangle$

lemma *mult-unit-dvd-iff'*: $\text{is-unit } a \implies (a * b) \text{ dvd } c \longleftrightarrow b \text{ dvd } c$
 $\langle \text{proof} \rangle$

lemma *dvd-mult-unit-iff*:
assumes *is-unit b*
shows $a \text{ dvd } c * b \longleftrightarrow a \text{ dvd } c$
 $\langle \text{proof} \rangle$

lemma *dvd-mult-unit-iff'*: $\text{is-unit } b \implies a \text{ dvd } b * c \longleftrightarrow a \text{ dvd } c$
 $\langle \text{proof} \rangle$

lemma *div-unit-dvd-iff*: $\text{is-unit } b \implies a \text{ div } b \text{ dvd } c \longleftrightarrow a \text{ dvd } c$
 $\langle \text{proof} \rangle$

lemma *dvd-div-unit-iff*: $\text{is-unit } b \implies a \text{ dvd } c \text{ div } b \longleftrightarrow a \text{ dvd } c$
 $\langle \text{proof} \rangle$

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]: $\text{is-unit } a \implies b * (1 \text{ div } a) = b \text{ div } a$
 ⟨proof⟩

lemma *unit-div-mult-self* [simp]: $\text{is-unit } a \implies b \text{ div } a * a = b$
 ⟨proof⟩

lemma *unit-div-1-div-1* [simp]: $\text{is-unit } a \implies 1 \text{ div } (1 \text{ div } a) = a$
 ⟨proof⟩

lemma *unit-div-mult-swap*: $\text{is-unit } c \implies a * (b \text{ div } c) = (a * b) \text{ div } c$
 ⟨proof⟩

lemma *unit-div-commute*: $\text{is-unit } b \implies (a \text{ div } b) * c = (a * c) \text{ div } b$
 ⟨proof⟩

lemma *unit-eq-div1*: $\text{is-unit } b \implies a \text{ div } b = c \longleftrightarrow a = c * b$
 ⟨proof⟩

lemma *unit-eq-div2*: $\text{is-unit } b \implies a = c \text{ div } b \longleftrightarrow a * b = c$
 ⟨proof⟩

lemma *unit-mult-left-cancel*: $\text{is-unit } a \implies a * b = a * c \longleftrightarrow b = c$
 ⟨proof⟩

lemma *unit-mult-right-cancel*: $\text{is-unit } a \implies b * a = c * a \longleftrightarrow b = c$
 ⟨proof⟩

lemma *unit-div-cancel*:
 assumes *is-unit a*
 shows $b \text{ div } a = c \text{ div } a \longleftrightarrow b = c$
 ⟨proof⟩

lemma *is-unit-div-mult2-eq*:
 assumes *is-unit b* and *is-unit c*
 shows $a \text{ div } (b * c) = a \text{ div } b \text{ div } c$
 ⟨proof⟩

lemma *is-unit-div-mult-cancel-left*:
 assumes $a \neq 0$ and *is-unit b*
 shows $a \text{ div } (a * b) = 1 \text{ div } b$
 ⟨proof⟩

lemma *is-unit-div-mult-cancel-right*:
 assumes $a \neq 0$ and *is-unit b*
 shows $a \text{ div } (b * a) = 1 \text{ div } b$
 ⟨proof⟩

lemma *unit-div-eq-0-iff*:
assumes *is-unit b*
shows $a \text{ div } b = 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *div-mult-unit2*:
 $\text{is-unit } c \implies b \text{ dvd } a \implies a \text{ div } (b * c) = a \text{ div } b \text{ div } c$
 $\langle \text{proof} \rangle$

Coprimality

definition *coprime* :: $'a \Rightarrow 'a \Rightarrow \text{bool}$
where $\text{coprime } a \ b \longleftrightarrow (\forall c. \ c \text{ dvd } a \longrightarrow c \text{ dvd } b \longrightarrow \text{is-unit } c)$

lemma *coprimeI*:
assumes $\bigwedge c. \ c \text{ dvd } a \implies c \text{ dvd } b \implies \text{is-unit } c$
shows $\text{coprime } a \ b$
 $\langle \text{proof} \rangle$

lemma *not-coprimeI*:
assumes $c \text{ dvd } a$ **and** $c \text{ dvd } b$ **and** $\neg \text{is-unit } c$
shows $\neg \text{coprime } a \ b$
 $\langle \text{proof} \rangle$

lemma *coprime-common-divisor*:
 $\text{is-unit } c$ **if** $\text{coprime } a \ b$ **and** $c \text{ dvd } a$ **and** $c \text{ dvd } b$
 $\langle \text{proof} \rangle$

lemma *not-coprimeE*:
assumes $\neg \text{coprime } a \ b$
obtains c **where** $c \text{ dvd } a$ **and** $c \text{ dvd } b$ **and** $\neg \text{is-unit } c$
 $\langle \text{proof} \rangle$

lemma *coprime-imp-coprime*:
 $\text{coprime } a \ b$ **if** $\text{coprime } c \ d$
and $\bigwedge e. \ \neg \text{is-unit } e \implies e \text{ dvd } a \implies e \text{ dvd } b \implies e \text{ dvd } c$
and $\bigwedge e. \ \neg \text{is-unit } e \implies e \text{ dvd } a \implies e \text{ dvd } b \implies e \text{ dvd } d$
 $\langle \text{proof} \rangle$

lemma *coprime-divisors*:
 $\text{coprime } a \ b$ **if** $a \text{ dvd } c \ b \text{ dvd } d$ **and** $\text{coprime } c \ d$
 $\langle \text{proof} \rangle$

lemma *coprime-self [simp]*:
 $\text{coprime } a \ a \longleftrightarrow \text{is-unit } a$ (**is** $?P \longleftrightarrow ?Q$)
 $\langle \text{proof} \rangle$

lemma *coprime-commute [ac-simps]*:
 $\text{coprime } b \ a \longleftrightarrow \text{coprime } a \ b$
 $\langle \text{proof} \rangle$

lemma *is-unit-left-imp-coprime*:

coprime a b if is-unit a

<proof>

lemma *is-unit-right-imp-coprime*:

coprime a b if is-unit b

<proof>

lemma *coprime-1-left [simp]*:

coprime 1 a

<proof>

lemma *coprime-1-right [simp]*:

coprime a 1

<proof>

lemma *coprime-0-left-iff [simp]*:

coprime 0 a \longleftrightarrow is-unit a

<proof>

lemma *coprime-0-right-iff [simp]*:

coprime a 0 \longleftrightarrow is-unit a

<proof>

lemma *coprime-mult-self-left-iff [simp]*:

*coprime (c * a) (c * b) \longleftrightarrow is-unit c \wedge coprime a b*

<proof>

lemma *coprime-mult-self-right-iff [simp]*:

*coprime (a * c) (b * c) \longleftrightarrow is-unit c \wedge coprime a b*

<proof>

lemma *coprime-absorb-left*:

assumes *x dvd y*

shows *coprime x y \longleftrightarrow is-unit x*

<proof>

lemma *coprime-absorb-right*:

assumes *y dvd x*

shows *coprime x y \longleftrightarrow is-unit y*

<proof>

end

class *unit-factor =*

fixes *unit-factor :: 'a \Rightarrow 'a*

class *semidom-divide-unit-factor = semidom-divide + unit-factor +*

```

assumes unit-factor-0 [simp]: unit-factor 0 = 0
and is-unit-unit-factor:  $a \text{ dvd } 1 \implies \text{unit-factor } a = a$ 
and unit-factor-is-unit:  $a \neq 0 \implies \text{unit-factor } a \text{ dvd } 1$ 
and unit-factor-mult-unit-left:  $a \text{ dvd } 1 \implies \text{unit-factor } (a * b) = a * \text{unit-factor } b$ 
— This fine-grained hierarchy will later on allow lean normalization of polynomials
begin

```

```

lemma unit-factor-mult-unit-right:  $a \text{ dvd } 1 \implies \text{unit-factor } (b * a) = \text{unit-factor } b * a$ 
  <proof>

```

```

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  $\Rightarrow$  'a
  assumes unit-factor-mult-normalize [simp]:  $\text{unit-factor } a * \text{normalize } a = a$ 
  and normalize-0 [simp]:  $\text{normalize } 0 = 0$ 
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]:  $a \neq 0 \implies \text{unit-factor } a \text{ dvd } b$ 
  <proof>

```

```

lemma unit-factor-self [simp]:  $\text{unit-factor } a \text{ dvd } a$ 
  <proof>

```

```

lemma normalize-mult-unit-factor [simp]:  $\text{normalize } a * \text{unit-factor } a = a$ 
  <proof>

```

```

lemma normalize-eq-0-iff [simp]:  $\text{normalize } a = 0 \iff a = 0$ 
  (is ?lhs  $\iff$  ?rhs)
  <proof>

```

```

lemma unit-factor-eq-0-iff [simp]:  $\text{unit-factor } a = 0 \iff a = 0$ 
  (is ?lhs  $\iff$  ?rhs)
  <proof>

```

```

lemma div-unit-factor [simp]:  $a \text{ div } \text{unit-factor } a = \text{normalize } a$ 

```

$\langle \text{proof} \rangle$

lemma *normalize-div* [simp]: *normalize a div a = 1 div unit-factor a*
 $\langle \text{proof} \rangle$

lemma *is-unit-normalize*:
 assumes *is-unit a*
 shows *normalize a = 1*
 $\langle \text{proof} \rangle$

lemma *unit-factor-1* [simp]: *unit-factor 1 = 1*
 $\langle \text{proof} \rangle$

lemma *normalize-1* [simp]: *normalize 1 = 1*
 $\langle \text{proof} \rangle$

lemma *normalize-1-iff*: *normalize a = 1 \longleftrightarrow is-unit a*
 (is ?lhs \longleftrightarrow ?rhs)
 $\langle \text{proof} \rangle$

lemma *div-normalize* [simp]: *a div normalize a = unit-factor a*
 $\langle \text{proof} \rangle$

lemma *mult-one-div-unit-factor* [simp]: *a * (1 div unit-factor b) = a div unit-factor b*
 $\langle \text{proof} \rangle$

lemma *inv-unit-factor-eq-0-iff* [simp]:
1 div unit-factor a = 0 \longleftrightarrow a = 0
 (is ?lhs \longleftrightarrow ?rhs)
 $\langle \text{proof} \rangle$

lemma *unit-factor-idem* [simp]: *unit-factor (unit-factor a) = unit-factor a*
 $\langle \text{proof} \rangle$

lemma *normalize-unit-factor* [simp]: *a \neq 0 \implies normalize (unit-factor a) = 1*
 $\langle \text{proof} \rangle$

lemma *normalize-mult-unit-left* [simp]:
 assumes *a dvd 1*
 shows *normalize (a * b) = normalize b*
 $\langle \text{proof} \rangle$

lemma *normalize-mult-unit-right* [simp]:
 assumes *b dvd 1*
 shows *normalize (a * b) = normalize a*
 $\langle \text{proof} \rangle$

lemma *normalize-idem* [simp]: *normalize (normalize a) = normalize a*

<proof>

lemma *unit-factor-normalize* [simp]:
 assumes $a \neq 0$
 shows $\text{unit-factor } (\text{normalize } a) = 1$
<proof>

lemma *normalize-dvd-iff* [simp]: $\text{normalize } a \text{ dvd } b \longleftrightarrow a \text{ dvd } b$
<proof>

lemma *dvd-normalize-iff* [simp]: $a \text{ dvd } \text{normalize } b \longleftrightarrow a \text{ dvd } b$
<proof>

lemma *normalize-idem-imp-unit-factor-eq*:
 assumes $\text{normalize } a = a$
 shows $\text{unit-factor } a = \text{of-bool } (a \neq 0)$
<proof>

lemma *normalize-idem-imp-is-unit-iff*:
 assumes $\text{normalize } a = a$
 shows $\text{is-unit } a \longleftrightarrow a = 1$
<proof>

lemma *coprime-normalize-left-iff* [simp]:
 $\text{coprime } (\text{normalize } a) \ b \longleftrightarrow \text{coprime } a \ b$
<proof>

lemma *coprime-normalize-right-iff* [simp]:
 $\text{coprime } a \ (\text{normalize } b) \longleftrightarrow \text{coprime } a \ b$
<proof>

We avoid an explicit definition of associated elements but prefer explicit normalisation 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 $a \text{ dvd } b$ and $b \text{ dvd } a$
 shows $\text{normalize } a = \text{normalize } b$
<proof>

lemma *associatedD1*: $\text{normalize } a = \text{normalize } b \implies a \text{ dvd } b$
<proof>

lemma *associatedD2*: $\text{normalize } a = \text{normalize } b \implies b \text{ dvd } a$
<proof>

lemma *associated-unit*: $\text{normalize } a = \text{normalize } b \implies \text{is-unit } a \implies \text{is-unit } b$
<proof>

lemma *associated-iff-dvd*: $\text{normalize } a = \text{normalize } b \longleftrightarrow a \text{ dvd } b \wedge b \text{ dvd } a$
 (is ?lhs \longleftrightarrow ?rhs)
 <proof>

lemma *associated-eqI*:
 assumes $a \text{ dvd } b$ and $b \text{ dvd } a$
 assumes $\text{normalize } a = a$ and $\text{normalize } b = b$
 shows $a = b$
 <proof>

lemma *normalize-unit-factor-eqI*:
 assumes $\text{normalize } a = \text{normalize } b$
 and $\text{unit-factor } a = \text{unit-factor } b$
 shows $a = b$
 <proof>

lemma *normalize-mult-normalize-left [simp]*: $\text{normalize } (\text{normalize } a * b) = \text{normalize } (a * b)$
 <proof>

lemma *normalize-mult-normalize-right [simp]*: $\text{normalize } (a * \text{normalize } b) = \text{normalize } (a * b)$
 <proof>

end

class *normalization-semidom-multiplicative* = *normalization-semidom* +
 assumes *unit-factor-mult*: $\text{unit-factor } (a * b) = \text{unit-factor } a * \text{unit-factor } b$
begin

lemma *normalize-mult*: $\text{normalize } (a * b) = \text{normalize } a * \text{normalize } b$
 <proof>

lemma *dvd-unit-factor-div*:
 assumes $b \text{ dvd } a$
 shows $\text{unit-factor } (a \text{ div } b) = \text{unit-factor } a \text{ div unit-factor } b$
 <proof>

lemma *dvd-normalize-div*:
 assumes $b \text{ dvd } a$
 shows $\text{normalize } (a \text{ div } b) = \text{normalize } a \text{ div normalize } b$
 <proof>

end

Syntactic division remainder operator

class *modulo* = *dvd* + *divide* +

fixes *modulo* :: 'a \Rightarrow 'a \Rightarrow 'a (**infixl** $\langle \text{mod} \rangle$ 70)

Arbitrary quotient and remainder partitions

class *semiring-modulo* = *comm-semiring-1-cancel* + *divide* + *modulo* +
assumes *div-mult-mod-eq*: $\langle a \text{ div } b * b + a \text{ mod } b = a \rangle$
begin

lemma *mod-div-decomp*:

fixes *a b*

obtains *q r* **where** $q = a \text{ div } b$ **and** $r = a \text{ mod } b$

and $a = q * b + r$

$\langle \text{proof} \rangle$

lemma *mult-div-mod-eq*: $b * (a \text{ div } b) + a \text{ mod } b = a$

$\langle \text{proof} \rangle$

lemma *mod-div-mult-eq*: $a \text{ mod } b + a \text{ div } b * b = a$

$\langle \text{proof} \rangle$

lemma *mod-mult-div-eq*: $a \text{ mod } b + b * (a \text{ div } b) = a$

$\langle \text{proof} \rangle$

lemma *minus-div-mult-eq-mod*: $a - a \text{ div } b * b = a \text{ mod } b$

$\langle \text{proof} \rangle$

lemma *minus-mult-div-eq-mod*: $a - b * (a \text{ div } b) = a \text{ mod } b$

$\langle \text{proof} \rangle$

lemma *minus-mod-eq-div-mult*: $a - a \text{ mod } b = a \text{ div } b * b$

$\langle \text{proof} \rangle$

lemma *minus-mod-eq-mult-div*: $a - a \text{ mod } b = b * (a \text{ div } b)$

$\langle \text{proof} \rangle$

lemma *mod-0-imp-dvd* [*dest!*]:

$b \text{ dvd } a$ **if** $a \text{ mod } b = 0$

$\langle \text{proof} \rangle$

lemma [*nitpick-unfold*]:

$a \text{ mod } b = a - a \text{ div } b * b$

$\langle \text{proof} \rangle$

end

class *semiring-modulo-trivial* = *semiring-modulo* + *divide-trivial*

begin

lemma *mod-0* [*simp*]:

$\langle 0 \text{ mod } a = 0 \rangle$

$\langle \text{proof} \rangle$

lemma *mod-by-0* [*simp*]:
 $\langle a \bmod 0 = a \rangle$
 $\langle \text{proof} \rangle$

lemma *mod-by-1* [*simp*]:
 $\langle a \bmod 1 = 0 \rangle$
 $\langle \text{proof} \rangle$

end

16.5 Quotient and remainder in integral domains

class *semidom-modulo* = *algebraic-semidom* + *semiring-modulo*
begin

subclass *semiring-modulo-trivial* $\langle \text{proof} \rangle$

lemma *mod-self* [*simp*]:
 $a \bmod a = 0$
 $\langle \text{proof} \rangle$

lemma *dvd-imp-mod-0* [*simp*]:
 $b \bmod a = 0$ **if** $a \text{ dvd } b$
 $\langle \text{proof} \rangle$

lemma *mod-eq-0-iff-dvd*:
 $a \bmod b = 0 \iff b \text{ dvd } a$
 $\langle \text{proof} \rangle$

lemma *dvd-eq-mod-eq-0* [*nitpick-unfold*, *code*]:
 $a \text{ dvd } b \iff b \bmod a = 0$
 $\langle \text{proof} \rangle$

lemma *dvd-mod-iff*:
assumes $c \text{ dvd } b$
shows $c \text{ dvd } a \bmod b \iff c \text{ dvd } a$
 $\langle \text{proof} \rangle$

lemma *dvd-mod-imp-dvd*:
assumes $c \text{ dvd } a \bmod b$ **and** $c \text{ dvd } b$
shows $c \text{ dvd } a$
 $\langle \text{proof} \rangle$

lemma *dvd-minus-mod* [*simp*]:
 $b \text{ dvd } a - a \bmod b$
 $\langle \text{proof} \rangle$

lemma *cancel-div-mod-rules*:

$$((a \text{ div } b) * b + a \text{ mod } b) + c = a + c$$

$$(b * (a \text{ div } b) + a \text{ mod } b) + c = a + c$$

<proof>

end

class *idom-modulo* = *idom* + *semidom-modulo*

begin

subclass *idom-divide* *<proof>*

lemma *div-diff* [*simp*]:

$$c \text{ dvd } a \implies c \text{ dvd } b \implies (a - b) \text{ div } c = a \text{ div } c - b \text{ div } c$$

<proof>

end

16.6 Interlude: basic tool support for algebraic and arithmetic calculations

named-theorems *arith arith facts* — *only ground formulas*

<ML>

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> by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer

class *ordered-semiring* = *semiring* + *ordered-comm-monoid-add* +

assumes *mult-left-mono*: $a \leq b \implies 0 \leq c \implies c * a \leq c * b$

assumes *mult-right-mono*: $a \leq b \implies 0 \leq c \implies a * c \leq b * c$

begin

lemma *mult-mono*: $a \leq b \implies c \leq d \implies 0 \leq b \implies 0 \leq c \implies a * c \leq b * d$

<proof>

lemma *mult-mono'*: $a \leq b \implies c \leq d \implies 0 \leq a \implies 0 \leq c \implies a * c \leq b * d$


```

    <proof>

end

lemma mono-mult:
  fixes a :: 'a::ordered-semiring
  shows  $a \geq 0 \implies \text{mono } ((*) a)$ 
  <proof>

class ordered-semiring-0 = semiring-0 + ordered-semiring
begin

lemma mult-nonneg-nonneg [simp]:  $0 \leq a \implies 0 \leq b \implies 0 \leq a * b$ 
  <proof>

lemma mult-nonneg-nonpos:  $0 \leq a \implies b \leq 0 \implies a * b \leq 0$ 
  <proof>

lemma mult-nonpos-nonneg:  $a \leq 0 \implies 0 \leq b \implies a * b \leq 0$ 
  <proof>

Legacy – use mult-nonpos-nonneg.

lemma mult-nonneg-nonpos2:  $0 \leq a \implies b \leq 0 \implies b * a \leq 0$ 
  <proof>

lemma split-mult-neg-le:  $(0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b) \implies a * b \leq 0$ 
  <proof>

end

class zero-less-one = order + zero + one +
  assumes zero-less-one [simp]:  $0 < 1$ 
begin

subclass zero-neg-one
  <proof>

lemma zero-le-one [simp]:
   $\langle 0 \leq 1 \rangle$  <proof>

end

class ordered-semiring-1 = ordered-semiring-0 + semiring-1 + zero-less-one
begin

lemma convex-bound-le:
  assumes  $x \leq a$  and  $y \leq a$  and  $0 \leq u$  and  $0 \leq v$  and  $u + v = 1$ 
  shows  $u * x + v * y \leq a$ 
  <proof>

```

end

class *ordered-cancel-semiring* = *ordered-semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* $\langle \text{proof} \rangle$

subclass *ordered-semiring-0* $\langle \text{proof} \rangle$

subclass *ordered-cancel-ab-semigroup-add* $\langle \text{proof} \rangle$

end

class *linordered-semiring* = *ordered-semiring* + *linordered-cancel-ab-semigroup-add*
begin

subclass *ordered-cancel-semiring* $\langle \text{proof} \rangle$

subclass *ordered-cancel-comm-monoid-add* $\langle \text{proof} \rangle$

subclass *ordered-ab-semigroup-monoid-add-imp-le* $\langle \text{proof} \rangle$

lemma *mult-left-less-imp-less*: $c * a < c * b \implies 0 \leq c \implies a < b$
 $\langle \text{proof} \rangle$

lemma *mult-right-less-imp-less*: $a * c < b * c \implies 0 \leq c \implies a < b$
 $\langle \text{proof} \rangle$

end

class *ordered-semiring-strict* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add*
 +

assumes *mult-strict-left-mono*: $a < b \implies 0 < c \implies c * a < c * b$

assumes *mult-strict-right-mono*: $a < b \implies 0 < c \implies a * c < b * c$

begin

subclass *semiring-0-cancel* $\langle \text{proof} \rangle$

subclass *ordered-semiring*
 $\langle \text{proof} \rangle$

lemma *mult-pos-pos[simp]*: $0 < a \implies 0 < b \implies 0 < a * b$
 $\langle \text{proof} \rangle$

lemma *mult-pos-neg*: $0 < a \implies b < 0 \implies a * b < 0$
 $\langle \text{proof} \rangle$

lemma *mult-neg-pos*: $a < 0 \implies 0 < b \implies a * b < 0$
 $\langle proof \rangle$

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
assumes $a < b$ $c < d$ $0 < b$ $0 \leq c$
shows $a * c < b * d$
 $\langle proof \rangle$

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 $a * c < b * d$
 $\langle proof \rangle$

lemma *mult-less-le-imp-less*:
assumes $a < b$ $c \leq d$ $0 \leq a$ $0 < c$
shows $a * c < b * d$
 $\langle proof \rangle$

lemma *mult-le-less-imp-less*:
assumes $a \leq b$ **and** $c < d$ **and** $0 < a$ **and** $0 \leq c$
shows $a * c < b * d$
 $\langle proof \rangle$

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 $u * x + v * y \leq a$
 $\langle proof \rangle$

end

subclass (**in** *linordered-semiring-1*) *ordered-semiring-1* $\langle proof \rangle$

class *linordered-semiring-strict* = *semiring* + *comm-monoid-add* + *linordered-cancel-ab-semigroup-add*
+
assumes *mult-strict-left-mono*: $a < b \implies 0 < c \implies c * a < c * b$
assumes *mult-strict-right-mono*: $a < b \implies 0 < c \implies a * c < b * c$
begin

subclass *semiring-0-cancel* $\langle proof \rangle$

subclass *linordered-semiring*
 $\langle proof \rangle$

subclass (in *linordered-semiring-strict*) *ordered-semiring-strict*
 ⟨proof⟩

lemma *mult-left-le-imp-le*: $c * a \leq c * b \implies 0 < c \implies a \leq b$
 ⟨proof⟩

lemma *mult-right-le-imp-le*: $a * c \leq b * c \implies 0 < c \implies a \leq b$
 ⟨proof⟩

lemma *zero-less-mult-pos*:
 assumes $0 < a * b$ $0 < a$ **shows** $0 < b$
 ⟨proof⟩

lemma *zero-less-mult-pos2*:
 assumes $0 < b * a$ $0 < a$ **shows** $0 < b$
 ⟨proof⟩

end

class *linordered-semiring-1-strict* = *linordered-semiring-strict* + *semiring-1* + *zero-less-one*
begin

subclass *linordered-semiring-1* ⟨proof⟩

lemma *convex-bound-lt*:
 assumes $x < a$ $y < a$ $0 \leq u$ $0 \leq v$ $u + v = 1$
 shows $u * x + v * y < a$
 ⟨proof⟩

end

class *ordered-semiring-1-strict* = *ordered-semiring-strict* + *semiring-1* + *zero-less-one*
 — analogous to *linordered-semiring-1-strict* not requiring a total order
begin

subclass *ordered-semiring-1* ⟨proof⟩

lemma *convex-bound-lt*:
 assumes $x < a$ **and** $y < a$ **and** $0 \leq u$ **and** $0 \leq v$ **and** $u + v = 1$
 shows $u * x + v * y < a$
 ⟨proof⟩

end

subclass (in *linordered-semiring-1-strict*) *ordered-semiring-1-strict* ⟨proof⟩

class *ordered-comm-semiring* = *comm-semiring-0* + *ordered-ab-semigroup-add* +
 assumes *comm-mult-left-mono*: $a \leq b \implies 0 \leq c \implies c * a \leq c * b$

begin

subclass *ordered-semiring*
 $\langle proof \rangle$

end

class *ordered-cancel-comm-semiring* = *ordered-comm-semiring* + *cancel-comm-monoid-add*
begin

subclass *comm-semiring-0-cancel* $\langle proof \rangle$
subclass *ordered-comm-semiring* $\langle proof \rangle$
subclass *ordered-cancel-semiring* $\langle proof \rangle$

end

class *linordered-comm-semiring-strict* = *comm-semiring-0* + *linordered-cancel-ab-semigroup-add*
+
assumes *comm-mult-strict-left-mono*: $a < b \implies 0 < c \implies c * a < c * b$
begin

subclass *linordered-semiring-strict*
 $\langle proof \rangle$

subclass *ordered-cancel-comm-semiring*
 $\langle proof \rangle$

end

class *ordered-comm-semiring-strict* = *comm-semiring-0* + *ordered-cancel-ab-semigroup-add*
+
— analogous to *linordered-comm-semiring-strict* not requiring a total order
assumes *comm-mult-strict-left-mono*: $a < b \implies 0 < c \implies c * a < c * b$
begin

subclass *ordered-semiring-strict*
 $\langle proof \rangle$

subclass *ordered-cancel-comm-semiring*
 $\langle proof \rangle$

end

subclass (**in** *linordered-comm-semiring-strict*) *ordered-comm-semiring-strict*
 $\langle proof \rangle$

class *ordered-ring* = *ring* + *ordered-cancel-semiring*
begin

subclass *ordered-ab-group-add* $\langle \text{proof} \rangle$

lemma *less-add-iff1*: $a * e + c < b * e + d \longleftrightarrow (a - b) * e + c < d$
 $\langle \text{proof} \rangle$

lemma *less-add-iff2*: $a * e + c < b * e + d \longleftrightarrow c < (b - a) * e + d$
 $\langle \text{proof} \rangle$

lemma *le-add-iff1*: $a * e + c \leq b * e + d \longleftrightarrow (a - b) * e + c \leq d$
 $\langle \text{proof} \rangle$

lemma *le-add-iff2*: $a * e + c \leq b * e + d \longleftrightarrow c \leq (b - a) * e + d$
 $\langle \text{proof} \rangle$

lemma *mult-left-mono-neg*: $b \leq a \implies c \leq 0 \implies c * a \leq c * b$
 $\langle \text{proof} \rangle$

lemma *mult-right-mono-neg*: $b \leq a \implies c \leq 0 \implies a * c \leq b * c$
 $\langle \text{proof} \rangle$

lemma *mult-nonpos-nonpos*: $a \leq 0 \implies b \leq 0 \implies 0 \leq a * b$
 $\langle \text{proof} \rangle$

lemma *split-mult-pos-le*: $(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \implies 0 \leq a * b$
 $\langle \text{proof} \rangle$

end

class *abs-if* = *minus* + *uminus* + *ord* + *zero* + *abs* +
assumes *abs-if*: $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$

class *linordered-ring* = *ring* + *linordered-semiring* + *linordered-ab-group-add* +
abs-if
begin

subclass *ordered-ring* $\langle \text{proof} \rangle$

subclass *ordered-ab-group-add-abs*
 $\langle \text{proof} \rangle$

lemma *zero-le-square* [*simp*]: $0 \leq a * a$
 $\langle \text{proof} \rangle$

lemma *not-square-less-zero* [*simp*]: $\neg (a * a < 0)$
 $\langle \text{proof} \rangle$

proposition *abs-eq-iff*: $|x| = |y| \longleftrightarrow x = y \vee x = -y$
 $\langle \text{proof} \rangle$

lemma *abs-eq-iff'*:

$$|a| = b \longleftrightarrow b \geq 0 \wedge (a = b \vee a = -b)$$

<proof>

lemma *eq-abs-iff'*:

$$a = |b| \longleftrightarrow a \geq 0 \wedge (b = a \vee b = -a)$$

<proof>

lemma *sum-squares-ge-zero*: $0 \leq x * x + y * y$

<proof>

lemma *not-sum-squares-lt-zero*: $\neg x * x + y * y < 0$

<proof>

end

class *linordered-ring-strict* = *ring* + *linordered-semiring-strict*
+ *ordered-ab-group-add* + *abs-if*

begin

subclass *linordered-ring* *<proof>*

lemma *mult-strict-left-mono-neg*: $b < a \implies c < 0 \implies c * a < c * b$

<proof>

lemma *mult-strict-right-mono-neg*: $b < a \implies c < 0 \implies a * c < b * c$

<proof>

lemma *mult-neg-neg*: $a < 0 \implies b < 0 \implies 0 < a * b$

<proof>

subclass *ring-no-zero-divisors*

<proof>

lemma *zero-less-mult-iff* [*algebra-split-simps*, *field-split-simps*]:

$$0 < a * b \longleftrightarrow 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$$

<proof>

lemma *zero-le-mult-iff* [*algebra-split-simps*, *field-split-simps*]:

$$0 \leq a * b \longleftrightarrow 0 \leq a \wedge 0 \leq b \vee a \leq 0 \wedge b \leq 0$$

<proof>

lemma *mult-less-0-iff* [*algebra-split-simps*, *field-split-simps*]:

$$a * b < 0 \longleftrightarrow 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$$

<proof>

lemma *mult-le-0-iff* [*algebra-split-simps*, *field-split-simps*]:

$$a * b \leq 0 \longleftrightarrow 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$$

<proof>

Cancellation laws for $c * a < c * b$ and $a * c < b * c$, 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 *mult-less-cancel-right-disj*: $a * c < b * c \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
 $\langle proof \rangle$

lemma *mult-less-cancel-left-disj*: $c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
 $\langle proof \rangle$

The “conjunction of implication” lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

lemma *mult-less-cancel-right*: $a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
 $\langle proof \rangle$

lemma *mult-less-cancel-left*: $c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
 $\langle proof \rangle$

lemma *mult-le-cancel-right*: $a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
 $\langle proof \rangle$

lemma *mult-le-cancel-left*: $c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
 $\langle proof \rangle$

lemma *mult-le-cancel-left-pos*: $0 < c \implies c * a \leq c * b \longleftrightarrow a \leq b$
 $\langle proof \rangle$

lemma *mult-le-cancel-left-neg*: $c < 0 \implies c * a \leq c * b \longleftrightarrow b \leq a$
 $\langle proof \rangle$

lemma *mult-less-cancel-left-pos*: $0 < c \implies c * a < c * b \longleftrightarrow a < b$
 $\langle proof \rangle$

lemma *mult-less-cancel-left-neg*: $c < 0 \implies c * a < c * b \longleftrightarrow b < a$
 $\langle proof \rangle$

lemma *mult-le-cancel-right-pos*: $0 < c \implies a * c \leq b * c \longleftrightarrow a \leq b$
 $\langle proof \rangle$

lemma *mult-le-cancel-right-neg*: $c < 0 \implies a * c \leq b * c \longleftrightarrow b \leq a$
 $\langle proof \rangle$

lemma *mult-less-cancel-right-pos*: $0 < c \implies a * c < b * c \longleftrightarrow a < b$
 ⟨proof⟩

lemma *mult-less-cancel-right-neg*: $c < 0 \implies a * c < b * c \longleftrightarrow b < a$
 ⟨proof⟩

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* ⟨proof⟩
subclass *ordered-cancel-comm-semiring* ⟨proof⟩

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* ⟨proof⟩

subclass *comm-semiring-1*
 ⟨proof⟩

subclass *ordered-semiring-1* ⟨proof⟩

lemma *not-one-le-zero* [*simp*]: $\neg 1 \leq 0$
 ⟨proof⟩

lemma *not-one-less-zero* [*simp*]: $\neg 1 < 0$
 ⟨proof⟩

lemma *of-bool-less-eq-iff* [*simp*]:
 ⟨*of-bool* $P \leq \text{of-bool } Q \longleftrightarrow (P \longrightarrow Q)$ ⟩
 ⟨proof⟩

lemma *of-bool-less-iff* [*simp*]:
 ⟨*of-bool* $P < \text{of-bool } Q \longleftrightarrow \neg P \wedge Q$ ⟩
 ⟨proof⟩

lemma *mult-left-le*: $c \leq 1 \implies 0 \leq a \implies a * c \leq a$

$\langle \text{proof} \rangle$

lemma *mult-le-one*: $a \leq 1 \implies 0 \leq b \implies b \leq 1 \implies a * b \leq 1$
 $\langle \text{proof} \rangle$

lemma *zero-less-two*: $0 < 1 + 1$
 $\langle \text{proof} \rangle$

end

class *linordered-semidom* = *semidom* + *linordered-comm-semiring-strict* + *zero-less-one*
 +
assumes *le-add-diff-inverse2* [*simp*]: $b \leq a \implies a - b + b = a$
begin

subclass *linordered-nonzero-semiring*
 $\langle \text{proof} \rangle$

lemma *zero-less-eq-of-bool* [*simp*]:
 $\langle 0 \leq \text{of-bool } P \rangle$
 $\langle \text{proof} \rangle$

lemma *zero-less-of-bool-iff* [*simp*]:
 $\langle 0 < \text{of-bool } P \longleftrightarrow P \rangle$
 $\langle \text{proof} \rangle$

lemma *of-bool-less-eq-one* [*simp*]:
 $\langle \text{of-bool } P \leq 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *of-bool-less-one-iff* [*simp*]:
 $\langle \text{of-bool } P < 1 \longleftrightarrow \neg P \rangle$
 $\langle \text{proof} \rangle$

lemma *of-bool-or-iff* [*simp*]:
 $\langle \text{of-bool } (P \vee Q) = \max (\text{of-bool } P) (\text{of-bool } Q) \rangle$
 $\langle \text{proof} \rangle$

Addition is the inverse of subtraction.

lemma *le-add-diff-inverse* [*simp*]: $b \leq a \implies b + (a - b) = a$
 $\langle \text{proof} \rangle$

lemma *add-diff-inverse*: $\neg a < b \implies b + (a - b) = a$
 $\langle \text{proof} \rangle$

lemma *add-le-imp-le-diff*:
assumes $i + k \leq n$ **shows** $i \leq n - k$
 $\langle \text{proof} \rangle$

lemma *add-le-add-imp-diff-le*:

assumes $1: i + k \leq n$

and $2: n \leq j + k$

shows $i + k \leq n \implies n \leq j + k \implies n - k \leq j$

$\langle proof \rangle$

lemma *less-1-mult*: $1 < m \implies 1 < n \implies 1 < m * n$

$\langle proof \rangle$

lemma *less-1-mult'*:

shows $1 < a \implies 1 \leq b \implies 1 < a * b$

$\langle proof \rangle$

end

class *linordered-idom* = *comm-ring-1* + *linordered-comm-semiring-strict* +
ordered-ab-group-add + *abs-if* + *sgn* +

assumes *sgn-if*: $sgn\ x = (if\ x = 0\ then\ 0\ else\ if\ 0 < x\ then\ 1\ else\ -1)$

begin

subclass *linordered-ring-strict* $\langle proof \rangle$

subclass *linordered-semiring-1-strict*
 $\langle proof \rangle$

subclass *ordered-comm-ring* $\langle proof \rangle$

subclass *idom* $\langle proof \rangle$

subclass *linordered-semidom*
 $\langle proof \rangle$

subclass *idom-abs-sgn*
 $\langle proof \rangle$

lemma *abs-bool-eq [simp]*:

$\langle |of-bool\ P| = of-bool\ P \rangle$

$\langle proof \rangle$

lemma *linorder-neqE-linordered-idom*:

assumes $x \neq y$

obtains $x < y \mid y < x$

$\langle proof \rangle$

These cancellation simp rules also produce two cases when the comparison is a goal.

lemma *mult-le-cancel-right1*: $c \leq b * c \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$

$\langle proof \rangle$

lemma *mult-le-cancel-right2*: $a * c \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$

<proof>

lemma *mult-le-cancel-left1*: $c \leq c * b \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$

<proof>

lemma *mult-le-cancel-left2*: $c * a \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$

<proof>

lemma *mult-less-cancel-right1*: $c < b * c \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$

<proof>

lemma *mult-less-cancel-right2*: $a * c < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$

<proof>

lemma *mult-less-cancel-left1*: $c < c * b \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$

<proof>

lemma *mult-less-cancel-left2*: $c * a < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$

<proof>

lemma *sgn-0-0*: $\text{sgn } a = 0 \longleftrightarrow a = 0$

<proof>

lemma *sgn-1-pos*: $\text{sgn } a = 1 \longleftrightarrow a > 0$

<proof>

lemma *sgn-1-neg*: $\text{sgn } a = -1 \longleftrightarrow a < 0$

<proof>

lemma *sgn-pos [simp]*: $0 < a \implies \text{sgn } a = 1$

<proof>

lemma *sgn-neg [simp]*: $a < 0 \implies \text{sgn } a = -1$

<proof>

lemma *abs-sgn*: $|k| = k * \text{sgn } k$

<proof>

lemma *sgn-greater [simp]*: $0 < \text{sgn } a \longleftrightarrow 0 < a$

<proof>

lemma *sgn-less* [simp]: $\text{sgn } a < 0 \longleftrightarrow a < 0$
 ⟨proof⟩

lemma *abs-sgn-eq-1* [simp]:
 $a \neq 0 \implies |\text{sgn } a| = 1$
 ⟨proof⟩

lemma *abs-sgn-eq*: $|\text{sgn } a| = (\text{if } a = 0 \text{ then } 0 \text{ else } 1)$
 ⟨proof⟩

lemma *sgn-mult-self-eq* [simp]:
 $\text{sgn } a * \text{sgn } a = \text{of_bool } (a \neq 0)$
 ⟨proof⟩

lemma *left-sgn-mult-self-eq* [simp]:
 $\langle \text{sgn } a * (\text{sgn } a * b) = \text{of_bool } (a \neq 0) * b \rangle$
 ⟨proof⟩

lemma *abs-mult-self-eq* [simp]:
 $|a| * |a| = a * a$
 ⟨proof⟩

lemma *same-sgn-sgn-add*:
 $\text{sgn } (a + b) = \text{sgn } a \text{ if } \text{sgn } b = \text{sgn } a$
 ⟨proof⟩

lemma *same-sgn-abs-add*:
 $|a + b| = |a| + |b| \text{ if } \text{sgn } b = \text{sgn } a$
 ⟨proof⟩

lemma *sgn-not-eq-imp*:
 $\text{sgn } a = - \text{sgn } b \text{ if } \text{sgn } b \neq \text{sgn } a \text{ and } \text{sgn } a \neq 0 \text{ and } \text{sgn } b \neq 0$
 ⟨proof⟩

lemma *abs-dvd-iff* [simp]: $|m| \text{ dvd } k \longleftrightarrow m \text{ dvd } k$
 ⟨proof⟩

lemma *dvd-abs-iff* [simp]: $m \text{ dvd } |k| \longleftrightarrow m \text{ dvd } k$
 ⟨proof⟩

lemma *dvd-if-abs-eq*: $|l| = |k| \implies l \text{ dvd } k$
 ⟨proof⟩

The following lemmas can be proven in more general structures, but are dangerous as simp rules in absence of $(- ?a = ?a) = (?a = 0)$, $(- ?a < ?a) = (0 < ?a)$, $(- ?a \leq ?a) = (0 \leq ?a)$.

lemma *equation-minus-iff-1* [simp, no-atp]: $1 = - a \longleftrightarrow a = - 1$
 ⟨proof⟩

lemma *minus-equation-iff-1* [*simp, no-atp*]: $- a = 1 \longleftrightarrow a = - 1$
 ⟨*proof*⟩

lemma *le-minus-iff-1* [*simp, no-atp*]: $1 \leq - b \longleftrightarrow b \leq - 1$
 ⟨*proof*⟩

lemma *minus-le-iff-1* [*simp, no-atp*]: $- a \leq 1 \longleftrightarrow - 1 \leq a$
 ⟨*proof*⟩

lemma *less-minus-iff-1* [*simp, no-atp*]: $1 < - b \longleftrightarrow b < - 1$
 ⟨*proof*⟩

lemma *minus-less-iff-1* [*simp, no-atp*]: $- a < 1 \longleftrightarrow - 1 < a$
 ⟨*proof*⟩

lemma *add-less-zeroD*:
 shows $x+y < 0 \implies x < 0 \vee y < 0$
 ⟨*proof*⟩

Is this really better than just rewriting with *abs-if*?

lemma *abs-split* [*no-atp*]: $\langle P \mid a \mid \longleftrightarrow (0 \leq a \longrightarrow P a) \wedge (a < 0 \longrightarrow P (- a)) \rangle$
 ⟨*proof*⟩

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*:
 $\langle a \leq b \longleftrightarrow a < b + 1 \rangle$
 ⟨*proof*⟩

end

Reasoning about inequalities with division

context *linordered-semidom*
begin

lemma *less-add-one*: $a < a + 1$
 ⟨*proof*⟩

end

context *linordered-idom*
begin

lemma *mult-right-le-one-le*: $0 \leq x \implies 0 \leq y \implies y \leq 1 \implies x * y \leq x$
 ⟨*proof*⟩

lemma *mult-left-le-one-le*: $0 \leq x \implies 0 \leq y \implies y \leq 1 \implies y * x \leq x$
 $\langle proof \rangle$

end

Absolute Value

context *linordered-idom*

begin

lemma *mult-sgn-abs*: $sgn\ x * |x| = x$
 $\langle proof \rangle$

lemma *abs-one*: $|1| = 1$
 $\langle proof \rangle$

end

class *ordered-ring-abs* = *ordered-ring* + *ordered-ab-group-add-abs* +
assumes *abs-eq-mult*:
 $(0 \leq a \vee a \leq 0) \wedge (0 \leq b \vee b \leq 0) \implies |a * b| = |a| * |b|$

context *linordered-idom*

begin

subclass *ordered-ring-abs*
 $\langle proof \rangle$

lemma *abs-mult-self*: $|a| * |a| = a * a$
 $\langle proof \rangle$

lemma *abs-mult-less*:
assumes *ac*: $|a| < c$
and *bd*: $|b| < d$
shows $|a| * |b| < c * d$
 $\langle proof \rangle$

lemma *abs-less-iff*: $|a| < b \longleftrightarrow a < b \wedge -a < b$
 $\langle proof \rangle$

lemma *abs-mult-pos*: $0 \leq x \implies |y| * x = |y * x|$
 $\langle proof \rangle$

lemma *abs-mult-pos'*: $0 \leq x \implies x * |y| = |x * y|$
 $\langle proof \rangle$

lemma *abs-diff-less-iff*: $|x - a| < r \longleftrightarrow a - r < x \wedge x < a + r$
 $\langle proof \rangle$

lemma *abs-diff-le-iff*: $|x - a| \leq r \longleftrightarrow a - r \leq x \wedge x \leq a + r$
<proof>

lemma *abs-add-one-gt-zero*: $0 < 1 + |x|$
<proof>

end

16.8 Dioids

Dioids are the alternative extensions of semirings, a semiring can either be a ring or a dioid but never both.

class *dioid* = *semiring-1* + *canonically-ordered-monoid-add*
begin

subclass *ordered-semiring*
<proof>

end

hide-fact (**open**) *comm-mult-left-mono comm-mult-strict-left-mono distrib*

code-identifier

code-module *Rings* \rightarrow (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

17 Natural numbers

theory *Nat*
imports *Inductive Typedef Fun Rings*
begin

17.1 Type *ind*

typedecl *ind*

axiomatization *Zero-Rep* :: *ind* **and** *Suc-Rep* :: *ind* \Rightarrow *ind*
 — The axiom of infinity in 2 parts:
where *Suc-Rep-inject*: *Suc-Rep* $x = \text{Suc-Rep } y \Rightarrow x = y$
and *Suc-Rep-not-Zero-Rep*: *Suc-Rep* $x \neq \text{Zero-Rep}$

17.2 Type *nat*

Type definition

inductive *Nat* :: *ind* \Rightarrow *bool*


```

where
  Zero-RepI: Nat Zero-Rep
| Suc-RepI: Nat i  $\implies$  Nat (Suc-Rep i)

typedef nat = {n. Nat n}
morphisms Rep-Nat Abs-Nat
<proof>

lemma Nat-Rep-Nat: Nat (Rep-Nat n)
<proof>

lemma Nat-Abs-Nat-inverse: Nat n  $\implies$  Rep-Nat (Abs-Nat n) = n
<proof>

lemma Nat-Abs-Nat-inject: Nat n  $\implies$  Nat m  $\implies$  Abs-Nat n = Abs-Nat m  $\longleftrightarrow$  n
= m
<proof>

instantiation nat :: zero
begin

definition Zero-nat-def: 0 = Abs-Nat Zero-Rep

instance <proof>

end

definition Suc :: nat  $\Rightarrow$  nat
  where Suc n = Abs-Nat (Suc-Rep (Rep-Nat n))

lemma Suc-not-Zero: Suc m  $\neq$  0
<proof>

lemma Zero-not-Suc: 0  $\neq$  Suc m
<proof>

lemma Suc-Rep-inject': Suc-Rep x = Suc-Rep y  $\longleftrightarrow$  x = y
<proof>

lemma nat-induct0:
  assumes P 0 and  $\bigwedge n. P n \implies P (Suc n)$ 
  shows P n
<proof>

free-constructors case-nat for 0 :: nat | Suc pred
  where pred (0 :: nat) = (0 :: nat)
<proof>
<ML>

```

old-rep-datatype $0 :: \text{nat } \text{Suc}$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

declare $\text{old.nat.inject}[\text{iff } \text{del}]$
and $\text{old.nat.distinct}(1)[\text{simp } \text{del}, \text{induct-simp } \text{del}]$

lemmas $\text{induct} = \text{old.nat.induct}$
lemmas $\text{inducts} = \text{old.nat.inducts}$
lemmas $\text{rec} = \text{old.nat.rec}$
lemmas $\text{sims} = \text{nat.inject nat.distinct nat.case nat.rec}$

$\langle \text{ML} \rangle$

abbreviation $\text{rec-nat} :: 'a \Rightarrow (\text{nat} \Rightarrow 'a \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a$
where $\text{rec-nat} \equiv \text{old.rec-nat}$

declare $\text{nat.sel}[\text{code } \text{del}]$

hide-const (**open**) Nat.pred — hide everything related to the selector

lemma nat-exhaust [$\text{case-names } 0 \text{ Suc}, \text{cases type: nat}$]:
 $(y = 0 \Longrightarrow P) \Longrightarrow (\bigwedge \text{nat. } y = \text{Suc nat} \Longrightarrow P) \Longrightarrow P$
— for backward compatibility – names of variables differ
 $\langle \text{proof} \rangle$

lemma nat-induct [$\text{case-names } 0 \text{ Suc}, \text{induct type: nat}$]:
fixes n
assumes $P \ 0$ **and** $\bigwedge n. P \ n \Longrightarrow P \ (\text{Suc } n)$
shows $P \ n$
— for backward compatibility – names of variables differ
 $\langle \text{proof} \rangle$

hide-fact
 nat-exhaust
 nat-induct0

$\langle \text{ML} \rangle$

Injectiveness and distinctness lemmas

lemma inj-Suc [simp]:
 $\text{inj-on } \text{Suc } N$
 $\langle \text{proof} \rangle$

lemma bij-betw-Suc [simp]:
 $\text{bij-betw } \text{Suc } M \ N \longleftrightarrow \text{Suc } ' M = N$
 $\langle \text{proof} \rangle$

lemma *Suc-neq-Zero*: $Suc\ m = 0 \implies R$
 ⟨proof⟩

lemma *Zero-neq-Suc*: $0 = Suc\ m \implies R$
 ⟨proof⟩

lemma *Suc-inject*: $Suc\ x = Suc\ y \implies x = y$
 ⟨proof⟩

lemma *n-not-Suc-n*: $n \neq Suc\ n$
 ⟨proof⟩

lemma *Suc-n-not-n*: $Suc\ n \neq n$
 ⟨proof⟩

A special form of induction for reasoning about $m < n$ and $m - n$.

lemma *diff-induct*:
 assumes $\bigwedge x. P\ x\ 0$
 and $\bigwedge y. P\ 0\ (Suc\ y)$
 and $\bigwedge x\ y. P\ x\ y \implies P\ (Suc\ x)\ (Suc\ y)$
 shows $P\ m\ n$
 ⟨proof⟩

17.3 Arithmetic operators

instantiation *nat* :: *comm-monoid-diff*
begin

primrec *plus-nat*
 where
 add-0 [code]: $0 + n = (n::nat)$
 | *add-Suc*: $Suc\ m + n = Suc\ (m + n)$

lemma *add-0-right* [simp]: $m + 0 = m$
for $m :: nat$
 ⟨proof⟩

lemma *add-Suc-right* [simp]: $m + Suc\ n = Suc\ (m + n)$
 ⟨proof⟩

lemma *add-Suc-shift* [code]: $Suc\ m + n = m + Suc\ n$
 ⟨proof⟩

primrec *minus-nat*
 where
 diff-0 [code]: $m - 0 = (m::nat)$
 | *diff-Suc*: $m - Suc\ n = (case\ m - n\ of\ 0 \Rightarrow 0 \mid Suc\ k \Rightarrow k)$

declare *diff-Suc* [simp del]

```

lemma diff-0-eq-0 [simp, code]:  $0 - n = 0$ 
  for  $n :: \text{nat}$ 
   $\langle \text{proof} \rangle$ 

lemma diff-Suc-Suc [simp, code]:  $\text{Suc } m - \text{Suc } n = m - n$ 
   $\langle \text{proof} \rangle$ 

instance
   $\langle \text{proof} \rangle$ 

end

hide-fact (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:  $0 * n = (0 :: \text{nat})$ 
  | mult-Suc:  $\text{Suc } m * n = n + (m * n)$ 

lemma mult-0-right [simp]:  $m * 0 = 0$ 
  for  $m :: \text{nat}$ 
   $\langle \text{proof} \rangle$ 

lemma mult-Suc-right [simp]:  $m * \text{Suc } n = m + (m * n)$ 
   $\langle \text{proof} \rangle$ 

lemma add-mult-distrib:  $(m + n) * k = (m * k) + (n * k)$ 
  for  $m \ n \ k :: \text{nat}$ 
   $\langle \text{proof} \rangle$ 

instance
   $\langle \text{proof} \rangle$ 

end

```

17.3.1 Addition

Reasoning about $m + 0 = 0$, etc.

```

lemma add-is-0 [iff]:  $m + n = 0 \longleftrightarrow m = 0 \wedge n = 0$ 
  for  $m \ n :: \text{nat}$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma add-is-1:  $m + n = \text{Suc } 0 \longleftrightarrow m = \text{Suc } 0 \wedge n = 0 \vee m = 0 \wedge n = \text{Suc } 0$ 

```

$\langle proof \rangle$

lemma *one-is-add*: $Suc\ 0 = m + n \longleftrightarrow m = Suc\ 0 \wedge n = 0 \vee m = 0 \wedge n = Suc\ 0$
 $\langle proof \rangle$

lemma *add-eq-self-zero*: $m + n = m \implies n = 0$
for $m\ n :: nat$
 $\langle proof \rangle$

lemma *plus-1-eq-Suc*:
 $plus\ 1 = Suc$
 $\langle proof \rangle$

lemma *Suc-eq-plus1*: $Suc\ n = n + 1$
 $\langle proof \rangle$

lemma *Suc-eq-plus1-left*: $Suc\ n = 1 + n$
 $\langle proof \rangle$

17.3.2 Difference

lemma *Suc-diff-diff* [simp]: $(Suc\ m - n) - Suc\ k = m - n - k$
 $\langle proof \rangle$

lemma *diff-Suc-1*: $Suc\ n - 1 = n$
 $\langle proof \rangle$

lemma *diff-Suc-1'* [simp]: $Suc\ n - Suc\ 0 = n$
 $\langle proof \rangle$

17.3.3 Multiplication

lemma *mult-is-0* [simp]: $m * n = 0 \longleftrightarrow m = 0 \vee n = 0$ **for** $m\ n :: nat$
 $\langle proof \rangle$

lemma *mult-eq-1-iff* [simp]: $m * n = Suc\ 0 \longleftrightarrow m = Suc\ 0 \wedge n = Suc\ 0$
 $\langle proof \rangle$

lemma *one-eq-mult-iff* [simp]: $Suc\ 0 = m * n \longleftrightarrow m = Suc\ 0 \wedge n = Suc\ 0$
 $\langle proof \rangle$

lemma *nat-mult-eq-1-iff* [simp]: $m * n = 1 \longleftrightarrow m = 1 \wedge n = 1$
and *nat-1-eq-mult-iff* [simp]: $1 = m * n \longleftrightarrow m = 1 \wedge n = 1$ **for** $m\ n :: nat$
 $\langle proof \rangle$

lemma *mult-cancel1* [simp]: $k * m = k * n \longleftrightarrow m = n \vee k = 0$
for $k\ m\ n :: nat$
 $\langle proof \rangle$

lemma *mult-cancel2* [*simp*]: $m * k = n * k \longleftrightarrow m = n \vee k = 0$
for $k\ m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *Suc-mult-cancel1*: $\text{Suc } k * m = \text{Suc } k * n \longleftrightarrow m = n$
 $\langle \text{proof} \rangle$

17.4 Orders on *nat*

17.4.1 Operation definition

instantiation *nat* :: *linorder*
begin

primrec *less-eq-nat*

where

$(0 :: \text{nat}) \leq n \longleftrightarrow \text{True}$
 $| \text{Suc } m \leq n \longleftrightarrow (\text{case } n \text{ of } 0 \Rightarrow \text{False} \mid \text{Suc } n \Rightarrow m \leq n)$

declare *less-eq-nat.simps* [*simp del*]

lemma *le0* [*iff*]: $0 \leq n$ **for**
 $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma [*code*]: $0 \leq n \longleftrightarrow \text{True}$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

definition *less-nat*

where *less-eq-Suc-le*: $n < m \longleftrightarrow \text{Suc } n \leq m$

lemma *Suc-le-mono* [*iff*]: $\text{Suc } n \leq \text{Suc } m \longleftrightarrow n \leq m$
 $\langle \text{proof} \rangle$

lemma *Suc-le-eq* [*code*]: $\text{Suc } m \leq n \longleftrightarrow m < n$
 $\langle \text{proof} \rangle$

lemma *le-0-eq* [*iff*]: $n \leq 0 \longleftrightarrow n = 0$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *not-less0* [*iff*]: $\neg n < 0$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *less-nat-zero-code* [*code*]: $n < 0 \longleftrightarrow \text{False}$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *Suc-less-eq* [*iff*]: $Suc\ m < Suc\ n \longleftrightarrow m < n$
 $\langle proof \rangle$

lemma *less-Suc-eq-le* [*code*]: $m < Suc\ n \longleftrightarrow m \leq n$
 $\langle proof \rangle$

lemma *Suc-less-eq2*: $Suc\ n < m \longleftrightarrow (\exists m'. m = Suc\ m' \wedge n < m')$
 $\langle proof \rangle$

lemma *le-SucI*: $m \leq n \implies m \leq Suc\ n$
 $\langle proof \rangle$

lemma *Suc-leD*: $Suc\ m \leq n \implies m \leq n$
 $\langle proof \rangle$

lemma *less-SucI*: $m < n \implies m < Suc\ n$
 $\langle proof \rangle$

lemma *Suc-lessD*: $Suc\ m < n \implies m < n$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

instantiation *nat* :: *order-bot*
begin

definition *bot-nat* :: *nat*
where *bot-nat* = 0

instance
 $\langle proof \rangle$

end

instance *nat* :: *no-top*
 $\langle proof \rangle$

17.4.2 Introduction properties

lemma *lessI* [*iff*]: $n < Suc\ n$
 $\langle proof \rangle$

lemma *zero-less-Suc* [*iff*]: $0 < Suc\ n$
 $\langle proof \rangle$

17.4.3 Elimination properties

lemma *less-not-refl*: $\neg n < n$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *less-not-refl2*: $n < m \implies m \neq n$
for $m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *less-not-refl3*: $s < t \implies s \neq t$
for $s\ t :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *less-irrefl-nat*: $n < n \implies R$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *less-zeroE*: $n < 0 \implies R$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *less-Suc-eq*: $m < \text{Suc } n \longleftrightarrow m < n \vee m = n$
 $\langle \text{proof} \rangle$

lemma *less-Suc0* [iff]: $(n < \text{Suc } 0) = (n = 0)$
 $\langle \text{proof} \rangle$

lemma *less-one* [iff]: $n < 1 \longleftrightarrow n = 0$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *Suc-mono*: $m < n \implies \text{Suc } m < \text{Suc } n$
 $\langle \text{proof} \rangle$

"Less than" is antisymmetric, sort of.

lemma *less-antisym*: $\neg n < m \implies n < \text{Suc } m \implies m = n$
 $\langle \text{proof} \rangle$

lemma *nat-neq-iff*: $m \neq n \longleftrightarrow m < n \vee n < m$
for $m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

17.4.4 Inductive (?) properties

lemma *Suc-lessI*: $m < n \implies \text{Suc } m \neq n \implies \text{Suc } m < n$
 $\langle \text{proof} \rangle$

lemma *lessE*:
assumes *major*: $i < k$

and $1: k = \text{Suc } i \implies P$
and $2: \bigwedge j. i < j \implies k = \text{Suc } j \implies P$
shows P
 $\langle \text{proof} \rangle$

lemma *less-SucE*:
assumes *major*: $m < \text{Suc } n$
and *less*: $m < n \implies P$
and *eq*: $m = n \implies P$
shows P
 $\langle \text{proof} \rangle$

lemma *Suc-lessE*:
assumes *major*: $\text{Suc } i < k$
and *minor*: $\bigwedge j. i < j \implies k = \text{Suc } j \implies P$
shows P
 $\langle \text{proof} \rangle$

lemma *Suc-less-SucD*: $\text{Suc } m < \text{Suc } n \implies m < n$
 $\langle \text{proof} \rangle$

lemma *less-trans-Suc*:
assumes *le*: $i < j$
shows $j < k \implies \text{Suc } i < k$
 $\langle \text{proof} \rangle$

Can be used with *less-Suc-eq* to get $n = m \vee n < m$.

lemma *not-less-eq*: $\neg m < n \longleftrightarrow n < \text{Suc } m$
 $\langle \text{proof} \rangle$

lemma *not-less-eq-eq*: $\neg m \leq n \longleftrightarrow \text{Suc } n \leq m$
 $\langle \text{proof} \rangle$

Properties of "less than or equal".

lemma *le-imp-less-Suc*: $m \leq n \implies m < \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *Suc-n-not-le-n*: $\neg \text{Suc } n \leq n$
 $\langle \text{proof} \rangle$

lemma *le-Suc-eq*: $m \leq \text{Suc } n \longleftrightarrow m \leq n \vee m = \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *le-SucE*: $m \leq \text{Suc } n \implies (m \leq n \implies R) \implies (m = \text{Suc } n \implies R) \implies R$
 $\langle \text{proof} \rangle$

lemma *Suc-leI*: $m < n \implies \text{Suc } m \leq n$
 $\langle \text{proof} \rangle$

Stronger version of *Suc-leD*.

lemma *Suc-le-lessD*: $\text{Suc } m \leq n \implies m < n$
 ⟨proof⟩

lemma *less-imp-le-nat*: $m < n \implies m \leq n$ **for** $m \ n :: \text{nat}$
 ⟨proof⟩

For instance, $(\text{Suc } m < \text{Suc } n) = (\text{Suc } m \leq n) = (m < n)$

lemmas *le-simps* = *less-imp-le-nat less-Suc-eq-le Suc-le-eq*

Equivalence of $m \leq n$ and $m < n \vee m = n$

lemma *less-or-eq-imp-le*: $m < n \vee m = n \implies m \leq n$
for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *le-eq-less-or-eq*: $m \leq n \longleftrightarrow m < n \vee m = n$
for $m \ n :: \text{nat}$
 ⟨proof⟩

Useful with *blast*.

lemma *eq-imp-le*: $m = n \implies m \leq n$
for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *le-refl*: $n \leq n$
for $n :: \text{nat}$
 ⟨proof⟩

lemma *le-trans*: $i \leq j \implies j \leq k \implies i \leq k$
for $i \ j \ k :: \text{nat}$
 ⟨proof⟩

lemma *le-antisym*: $m \leq n \implies n \leq m \implies m = n$
for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *nat-less-le*: $m < n \longleftrightarrow m \leq n \wedge m \neq n$
for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *le-neq-implies-less*: $m \leq n \implies m \neq n \implies m < n$
for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *nat-le-linear*: $m \leq n \vee n \leq m$
for $m \ n :: \text{nat}$
 ⟨proof⟩

lemmas *linorder-neqE-nat* = *linorder-neqE* [**where** $'a = \text{nat}$]

lemma *le-less-Suc-eq*: $m \leq n \implies n < \text{Suc } m \longleftrightarrow n = m$
 ⟨proof⟩

lemma *not-less-less-Suc-eq*: $\neg n < m \implies n < \text{Suc } m \longleftrightarrow n = m$
 ⟨proof⟩

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

lemma *not0-implies-Suc*: $n \neq 0 \implies \exists m. n = \text{Suc } m$
 ⟨proof⟩

lemma *gr0-implies-Suc*: $n > 0 \implies \exists m. n = \text{Suc } m$
 ⟨proof⟩

lemma *gr-implies-not0*: $m < n \implies n \neq 0$
for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *neq0-conv[iff]*: $n \neq 0 \longleftrightarrow 0 < n$
for $n :: \text{nat}$
 ⟨proof⟩

This theorem is useful with *blast*

lemma *gr0I*: $(n = 0 \implies \text{False}) \implies 0 < n$
for $n :: \text{nat}$
 ⟨proof⟩

lemma *gr0-conv-Suc*: $0 < n \longleftrightarrow (\exists m. n = \text{Suc } m)$
 ⟨proof⟩

lemma *not-gr0 [iff]*: $\neg 0 < n \longleftrightarrow n = 0$
for $n :: \text{nat}$
 ⟨proof⟩

lemma *Suc-le-D*: $\text{Suc } n \leq m' \implies \exists m. m' = \text{Suc } m$
 ⟨proof⟩

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: $m < \text{Suc } n \longleftrightarrow m = 0 \vee (\exists j. m = \text{Suc } j \wedge j < n)$
 ⟨proof⟩

lemma *All-less-Suc*: $(\forall i < \text{Suc } n. P \ i) = (P \ n \wedge (\forall i < n. P \ i))$
 ⟨proof⟩

lemma *All-less-Suc2*: $(\forall i < \text{Suc } n. P \ i) = (P \ 0 \wedge (\forall i < n. P(\text{Suc } i)))$
 ⟨proof⟩

lemma *Ex-less-Suc*: $(\exists i < \text{Suc } n. P \ i) = (P \ n \vee (\exists i < n. P \ i))$
 ⟨proof⟩

lemma *Ex-less-Suc2*: $(\exists i < \text{Suc } n. P i) = (P 0 \vee (\exists i < n. P(\text{Suc } i)))$
 ⟨proof⟩

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⟩

17.4.5 Monotonicity of Addition

lemma *Suc-pred* [*simp*]: $n > 0 \implies \text{Suc } (n - \text{Suc } 0) = n$
 ⟨proof⟩

lemma *Suc-diff-1* [*simp*]: $0 < n \implies \text{Suc } (n - 1) = n$
 ⟨proof⟩

lemma *nat-add-left-cancel-le* [*simp*]: $k + m \leq k + n \longleftrightarrow m \leq n$
 for $k m n :: \text{nat}$
 ⟨proof⟩

lemma *nat-add-left-cancel-less* [*simp*]: $k + m < k + n \longleftrightarrow m < n$
 for $k m n :: \text{nat}$
 ⟨proof⟩

lemma *add-gr-0* [*iff*]: $m + n > 0 \longleftrightarrow m > 0 \vee n > 0$
 for $m n :: \text{nat}$
 ⟨proof⟩

strict, in 1st argument

lemma *add-less-mono1*: $i < j \implies i + k < j + k$
 for $i j k :: \text{nat}$
 ⟨proof⟩

strict, in both arguments

lemma *add-less-mono*:
 fixes $i j k l :: \text{nat}$
 assumes $i < j$ $k < l$ **shows** $i + k < j + l$
 ⟨proof⟩

lemma *less-imp-Suc-add*: $m < n \implies \exists k. n = \text{Suc } (m + k)$
 ⟨proof⟩

lemma *le-Suc-ex*: $k \leq l \implies (\exists n. l = k + n)$
 for $k l :: \text{nat}$
 ⟨proof⟩

lemma *less-natE*:

```

assumes  $\langle m < n \rangle$ 
obtains  $q$  where  $\langle n = \text{Suc } (m + q) \rangle$ 
 $\langle \text{proof} \rangle$ 

```

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  $k * i < k * j$ 
 $\langle \text{proof} \rangle$ 

```

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

```

lemma add-diff-inverse-nat:  $\neg m < n \implies n + (m - n) = m$ 
for  $m\ n :: \text{nat}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma nat-le-iff-add:  $m \leq n \longleftrightarrow (\exists k. n = m + k)$ 
for  $m\ n :: \text{nat}$ 
 $\langle \text{proof} \rangle$ 

```

The naturals form an ordered *semidom* and a *diod*.

```

instance nat :: discrete-linordered-semidom
 $\langle \text{proof} \rangle$ 

```

```

instance nat :: diod
 $\langle \text{proof} \rangle$ 

```

```

declare le0[simp del] — This is now  $0 \leq ?x$ 
declare le-0-eq[simp del] — This is now  $(?n \leq 0) = (?n = 0)$ 
declare not-less0[simp del] — This is now  $\neg ?n < 0$ 
declare not-gr0[simp del] — This is now  $(\neg 0 < ?n) = (?n = 0)$ 

```

```

instance nat :: ordered-cancel-comm-monoid-add  $\langle \text{proof} \rangle$ 
instance nat :: ordered-cancel-comm-monoid-diff  $\langle \text{proof} \rangle$ 

```

17.4.6 *min* and *max*

```

global-interpretation bot-nat-0: ordering-top  $\langle (\geq) \rangle \langle (>) \rangle \langle 0 :: \text{nat} \rangle$ 
 $\langle \text{proof} \rangle$ 

```

```

global-interpretation max-nat: semilattice-neutr-order max  $\langle 0 :: \text{nat} \rangle \langle (\geq) \rangle \langle (>) \rangle$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma mono-Suc: mono Suc
 $\langle \text{proof} \rangle$ 

```

```

lemma min-0L [simp]:  $\text{min } 0\ n = 0$ 
for  $n :: \text{nat}$ 
 $\langle \text{proof} \rangle$ 

```

lemma *min-0R* [simp]: $\min n\ 0 = 0$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *min-Suc-Suc* [simp]: $\min (\text{Suc } m) (\text{Suc } n) = \text{Suc } (\min m\ n)$
 $\langle \text{proof} \rangle$

lemma *min-Suc1*: $\min (\text{Suc } n)\ m = (\text{case } m \text{ of } 0 \Rightarrow 0 \mid \text{Suc } m' \Rightarrow \text{Suc}(\min n\ m'))$
 $\langle \text{proof} \rangle$

lemma *min-Suc2*: $\min m (\text{Suc } n) = (\text{case } m \text{ of } 0 \Rightarrow 0 \mid \text{Suc } m' \Rightarrow \text{Suc}(\min m'\ n))$
 $\langle \text{proof} \rangle$

lemma *max-0L* [simp]: $\max 0\ n = n$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *max-0R* [simp]: $\max n\ 0 = n$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *max-Suc-Suc* [simp]: $\max (\text{Suc } m) (\text{Suc } n) = \text{Suc } (\max m\ n)$
 $\langle \text{proof} \rangle$

lemma *max-Suc1*: $\max (\text{Suc } n)\ m = (\text{case } m \text{ of } 0 \Rightarrow \text{Suc } n \mid \text{Suc } m' \Rightarrow \text{Suc } (\max n\ m'))$
 $\langle \text{proof} \rangle$

lemma *max-Suc2*: $\max m (\text{Suc } n) = (\text{case } m \text{ of } 0 \Rightarrow \text{Suc } n \mid \text{Suc } m' \Rightarrow \text{Suc } (\max m'\ n))$
 $\langle \text{proof} \rangle$

lemma *nat-mult-min-left*: $\min m\ n * q = \min (m * q) (n * q)$
for $m\ n\ q :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *nat-mult-min-right*: $m * \min n\ q = \min (m * n) (m * q)$
for $m\ n\ q :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *nat-add-max-left*: $\max m\ n + q = \max (m + q) (n + q)$
for $m\ n\ q :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *nat-add-max-right*: $m + \max n\ q = \max (m + n) (m + q)$
for $m\ n\ q :: \text{nat}$

$\langle proof \rangle$

lemma *nat-mult-max-left*: $max\ m\ n\ *\ q = max\ (m\ *\ q)\ (n\ *\ q)$
for $m\ n\ q :: nat$
 $\langle proof \rangle$

lemma *nat-mult-max-right*: $m\ *\ max\ n\ q = max\ (m\ *\ n)\ (m\ *\ q)$
for $m\ n\ q :: nat$
 $\langle proof \rangle$

17.4.7 Additional theorems about (\leq)

Complete induction, aka course-of-values induction

instance *nat* :: *wellorder*
 $\langle proof \rangle$

lemma *Least-eq-0[simp]*: $P\ 0 \implies Least\ P = 0$
for $P :: nat \Rightarrow bool$
 $\langle proof \rangle$

lemma *Least-Suc*:
assumes $P\ n \neg P\ 0$
shows $(LEAST\ n.\ P\ n) = Suc\ (LEAST\ m.\ P\ (Suc\ m))$
 $\langle proof \rangle$

lemma *Least-Suc2*: $P\ n \implies Q\ m \implies \neg P\ 0 \implies \forall k.\ P\ (Suc\ k) = Q\ k \implies Least\ P = Suc\ (Least\ Q)$
 $\langle proof \rangle$

lemma *ex-least-nat-le*:
fixes $P :: nat \Rightarrow bool$
assumes $P\ n$
shows $\exists k \leq n.\ (\forall i < k.\ \neg P\ i) \wedge P\ k$
 $\langle proof \rangle$

lemma *ex-least-nat-less*:
fixes $P :: nat \Rightarrow bool$
assumes $P\ n \neg P\ 0$
shows $\exists k < n.\ (\forall i \leq k.\ \neg P\ i) \wedge P\ (Suc\ k)$
 $\langle proof \rangle$

lemma *nat-less-induct*:
fixes $P :: nat \Rightarrow bool$
assumes $\bigwedge n.\ \forall m.\ m < n \longrightarrow P\ m \implies P\ n$
shows $P\ n$
 $\langle proof \rangle$

lemma *measure-induct-rule* [*case-names less*]:
fixes $f :: 'a \Rightarrow 'b::wellorder$
assumes $step: \bigwedge x. (\bigwedge y. f\ y < f\ x \implies P\ y) \implies P\ x$
shows $P\ a$
 $\langle proof \rangle$

old style induction rules:

lemma *measure-induct*:
fixes $f :: 'a \Rightarrow 'b::wellorder$
shows $(\bigwedge x. \forall y. f\ y < f\ x \longrightarrow P\ y \implies P\ x) \implies P\ a$
 $\langle proof \rangle$

lemma *full-nat-induct*:
assumes $step: \bigwedge n. (\forall m. Suc\ m \leq n \longrightarrow P\ m) \implies P\ n$
shows $P\ n$
 $\langle proof \rangle$

An induction rule for establishing binary relations

lemma *less-Suc-induct* [*consumes 1*]:
assumes $less: i < j$
and $step: \bigwedge i. P\ i\ (Suc\ i)$
and $trans: \bigwedge i\ j\ k. i < j \implies j < k \implies P\ i\ j \implies P\ j\ k \implies P\ i\ k$
shows $P\ i\ j$
 $\langle proof \rangle$

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 $\neg P\ n$ prove there exists a smaller natural number m such that $\neg P\ m$.

lemma *infinite-descent*: $(\bigwedge n. \neg P\ n \implies \exists m < n. \neg P\ m) \implies P\ n$ **for** $P :: nat \Rightarrow bool$
— compact version without explicit base case
 $\langle proof \rangle$

lemma *infinite-descent0* [*case-names 0 smaller*]:
fixes $P :: nat \Rightarrow bool$
assumes $P\ 0$
and $\bigwedge n. n > 0 \implies \neg P\ n \implies \exists m. m < n \wedge \neg P\ m$
shows $P\ n$
 $\langle proof \rangle$

Infinite descent using a mapping to *nat*: $P\ x$ is true for all $x \in D$ if there exists a $V \in D \Rightarrow nat$ and

- case “0”: given $V\ x = 0$ prove $P\ x$

- “smaller”: given $V x > 0$ and $\neg P x$ prove there exists a $y \in D$ such that $V y < V x$ and $\neg P y$.

corollary *infinite-descent0-measure* [case-names 0 smaller]:

fixes $V :: 'a \Rightarrow nat$
assumes 1: $\bigwedge x. V x = 0 \implies P x$
and 2: $\bigwedge x. V x > 0 \implies \neg P x \implies \exists y. V y < V x \wedge \neg P y$
shows $P x$
 $\langle proof \rangle$

Again, without explicit base case:

lemma *infinite-descent-measure*:

fixes $V :: 'a \Rightarrow nat$
assumes $\bigwedge x. \neg P x \implies \exists y. V y < V x \wedge \neg P y$
shows $P x$
 $\langle proof \rangle$

A (clumsy) way of lifting $<$ monotonicity to \leq monotonicity

lemma *less-mono-imp-le-mono*:

fixes $f :: nat \Rightarrow nat$
and $i j :: nat$
assumes $\bigwedge i j :: nat. i < j \implies f i < f j$
and $i \leq j$
shows $f i \leq f j$
 $\langle proof \rangle$

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j \implies i + k \leq j + k$
for $i j k :: nat$
 $\langle proof \rangle$

non-strict, in both arguments

lemma *add-le-mono*: $i \leq j \implies k \leq l \implies i + k \leq j + l$
for $i j k l :: nat$
 $\langle proof \rangle$

lemma *le-add2*: $n \leq m + n$
for $m n :: nat$
 $\langle proof \rangle$

lemma *le-add1*: $n \leq n + m$
for $m n :: nat$
 $\langle proof \rangle$

lemma *less-add-Suc1*: $i < Suc (i + m)$
 $\langle proof \rangle$

lemma *less-add-Suc2*: $i < Suc (m + i)$

$\langle proof \rangle$

lemma *less-iff-Suc-add*: $m < n \longleftrightarrow (\exists k. n = \text{Suc } (m + k))$
 $\langle proof \rangle$

lemma *trans-le-add1*: $i \leq j \implies i \leq j + m$
for $i\ j\ m :: \text{nat}$
 $\langle proof \rangle$

lemma *trans-le-add2*: $i \leq j \implies i \leq m + j$
for $i\ j\ m :: \text{nat}$
 $\langle proof \rangle$

lemma *trans-less-add1*: $i < j \implies i < j + m$
for $i\ j\ m :: \text{nat}$
 $\langle proof \rangle$

lemma *trans-less-add2*: $i < j \implies i < m + j$
for $i\ j\ m :: \text{nat}$
 $\langle proof \rangle$

lemma *add-lessD1*: $i + j < k \implies i < k$
for $i\ j\ k :: \text{nat}$
 $\langle proof \rangle$

lemma *not-add-less1* [*iff*]: $\neg i + j < i$
for $i\ j :: \text{nat}$
 $\langle proof \rangle$

lemma *not-add-less2* [*iff*]: $\neg j + i < i$
for $i\ j :: \text{nat}$
 $\langle proof \rangle$

lemma *add-leD1*: $m + k \leq n \implies m \leq n$
for $k\ m\ n :: \text{nat}$
 $\langle proof \rangle$

lemma *add-leD2*: $m + k \leq n \implies k \leq n$
for $k\ m\ n :: \text{nat}$
 $\langle proof \rangle$

lemma *add-leE*: $m + k \leq n \implies (m \leq n \implies k \leq n \implies R) \implies R$
for $k\ m\ n :: \text{nat}$
 $\langle proof \rangle$

needs $\bigwedge k$ for *ac-simps* to work

lemma *less-add-eq-less*: $\bigwedge k. k < l \implies m + l = k + n \implies m < n$
for $l\ m\ n :: \text{nat}$
 $\langle proof \rangle$

17.4.8 More results about difference

lemma *Suc-diff-le*: $n \leq m \implies \text{Suc } m - n = \text{Suc } (m - n)$
 ⟨proof⟩

lemma *diff-less-Suc*: $m - n < \text{Suc } m$
 ⟨proof⟩

lemma *diff-le-self* [simp]: $m - n \leq m$
 for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *less-imp-diff-less*: $j < k \implies j - n < k$
 for $j \ k \ n :: \text{nat}$
 ⟨proof⟩

lemma *diff-Suc-less* [simp]: $0 < n \implies n - \text{Suc } i < n$
 ⟨proof⟩

lemma *diff-add-assoc*: $k \leq j \implies (i + j) - k = i + (j - k)$
 for $i \ j \ k :: \text{nat}$
 ⟨proof⟩

lemma *add-diff-assoc* [simp]: $k \leq j \implies i + (j - k) = i + j - k$
 for $i \ j \ k :: \text{nat}$
 ⟨proof⟩

lemma *diff-add-assoc2*: $k \leq j \implies (j + i) - k = (j - k) + i$
 for $i \ j \ k :: \text{nat}$
 ⟨proof⟩

lemma *add-diff-assoc2* [simp]: $k \leq j \implies j - k + i = j + i - k$
 for $i \ j \ k :: \text{nat}$
 ⟨proof⟩

lemma *le-imp-diff-is-add*: $i \leq j \implies (j - i = k) = (j = k + i)$
 for $i \ j \ k :: \text{nat}$
 ⟨proof⟩

lemma *diff-is-0-eq* [simp]: $m - n = 0 \longleftrightarrow m \leq n$
 for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *diff-is-0-eq'* [simp]: $m \leq n \implies m - n = 0$
 for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *zero-less-diff* [simp]: $0 < n - m \longleftrightarrow m < n$
 for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *less-imp-add-positive*:

assumes $i < j$

shows $\exists k :: \text{nat}. 0 < k \wedge i + k = j$

$\langle \text{proof} \rangle$

a nice rewrite for bounded subtraction

lemma *nat-minus-add-max*: $n - m + m = \max n m$

for $m n :: \text{nat}$

$\langle \text{proof} \rangle$

lemma *nat-diff-split*: $P (a - b) \longleftrightarrow (a < b \longrightarrow P 0) \wedge (\forall d. a = b + d \longrightarrow P d)$

for $a b :: \text{nat}$

— elimination of $-$ on *nat*

$\langle \text{proof} \rangle$

lemma *nat-diff-split-asm*: $P (a - b) \longleftrightarrow \neg (a < b \wedge \neg P 0 \vee (\exists d. a = b + d \wedge \neg P d))$

for $a b :: \text{nat}$

— elimination of $-$ on *nat* in assumptions

$\langle \text{proof} \rangle$

lemmas *nat-diff-splits* = *nat-diff-split nat-diff-split-asm*

lemma *Suc-pred'*: $0 < n \implies n = \text{Suc}(n - 1)$

$\langle \text{proof} \rangle$

lemma *add-eq-if*: $m + n = (\text{if } m = 0 \text{ then } n \text{ else } \text{Suc} ((m - 1) + n))$

$\langle \text{proof} \rangle$

lemma *mult-eq-if*: $m * n = (\text{if } m = 0 \text{ then } 0 \text{ else } n + ((m - 1) * n))$

for $m n :: \text{nat}$

$\langle \text{proof} \rangle$

lemma *Suc-diff-eq-diff-pred*: $0 < n \implies \text{Suc } m - n = m - (n - 1)$

$\langle \text{proof} \rangle$

lemma *diff-Suc-eq-diff-pred*: $m - \text{Suc } n = (m - 1) - n$

$\langle \text{proof} \rangle$

lemma *Let-Suc [simp]*: *Let* $(\text{Suc } n) f \equiv f (\text{Suc } n)$

$\langle \text{proof} \rangle$

17.4.9 Monotonicity of multiplication

lemma *mult-le-mono1*: $i \leq j \implies i * k \leq j * k$

for $i j k :: \text{nat}$

$\langle \text{proof} \rangle$

lemma *mult-le-mono2*: $i \leq j \implies k * i \leq k * j$
for $i\ j\ k :: \text{nat}$
 $\langle \text{proof} \rangle$

\leq monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \leq j \implies k \leq l \implies i * k \leq j * l$
for $i\ j\ k\ l :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *mult-less-mono1*: $i < j \implies 0 < k \implies i * k < j * k$
for $i\ j\ k :: \text{nat}$
 $\langle \text{proof} \rangle$

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [simp]: $0 < m * n \longleftrightarrow 0 < m \wedge 0 < n$
for $m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *one-le-mult-iff* [simp]: $\text{Suc } 0 \leq m * n \longleftrightarrow \text{Suc } 0 \leq m \wedge \text{Suc } 0 \leq n$
 $\langle \text{proof} \rangle$

lemma *mult-less-cancel2* [simp]: $m * k < n * k \longleftrightarrow 0 < k \wedge m < n$
for $k\ m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *mult-less-cancel1* [simp]: $k * m < k * n \longleftrightarrow 0 < k \wedge m < n$
for $k\ m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *mult-le-cancel1* [simp]: $k * m \leq k * n \longleftrightarrow (0 < k \longrightarrow m \leq n)$
for $k\ m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *mult-le-cancel2* [simp]: $m * k \leq n * k \longleftrightarrow (0 < k \longrightarrow m \leq n)$
for $k\ m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *Suc-mult-less-cancel1*: $\text{Suc } k * m < \text{Suc } k * n \longleftrightarrow m < n$
 $\langle \text{proof} \rangle$

lemma *Suc-mult-le-cancel1*: $\text{Suc } k * m \leq \text{Suc } k * n \longleftrightarrow m \leq n$
 $\langle \text{proof} \rangle$

lemma *le-square*: $m \leq m * m$
for $m :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *le-cube*: $m \leq m * (m * m)$

for $m :: nat$
 $\langle proof \rangle$

Lemma for *gcd*

lemma *mult-eq-self-implies-10*:
fixes $m n :: nat$
assumes $m = m * n$ **shows** $n = 1 \vee m = 0$
 $\langle proof \rangle$

lemma *mono-times-nat*:
fixes $n :: nat$
assumes $n > 0$
shows *mono* (*times* n)
 $\langle proof \rangle$

The lattice order on *nat*.

instantiation $nat :: distrib_lattice$
begin

definition (*inf* :: $nat \Rightarrow nat \Rightarrow nat$) = *min*

definition (*sup* :: $nat \Rightarrow nat \Rightarrow nat$) = *max*

instance
 $\langle proof \rangle$

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 *compow* :: $nat \Rightarrow 'a \Rightarrow 'a$

abbreviation *compower* :: $'a \Rightarrow nat \Rightarrow 'a$ (**infixr** $\langle \overset{\sim}{\sim} \rangle$ 80)
where $f \overset{\sim}{\sim} n \equiv compow\ n\ f$

notation (*latex output*)
compower ($\langle (-)^\sim \rangle$ [1000] 1000)

$f \overset{\sim}{\sim} n = f \circ \dots \circ f$, the n -fold composition of f

overloading

funpow $\equiv compow :: nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$
begin

primrec *funpow* :: $nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
where
funpow 0 $f = id$

```

| funpow (Suc n) f = f ∘ funpow n f

end

lemma funpow-0 [simp]: (f  $\sim$  0) x = x
  <proof>

lemma funpow-Suc-right: f  $\sim$  Suc n = f  $\sim$  n ∘ f
  <proof>

lemmas funpow-simps-right = funpow.simps(1) funpow-Suc-right

For code generation.

context
begin

qualified definition funpow :: nat  $\Rightarrow$  ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$  'a
  where funpow-code-def [code-abbrev]: funpow = compow

lemma [code]:
  funpow 0 f = id
  funpow (Suc n) f = f ∘ funpow n f
  <proof>

end

lemma funpow-add: f  $\sim$  (m + n) = f  $\sim$  m ∘ f  $\sim$  n
  <proof>

lemma funpow-mult: (f  $\sim$  m)  $\sim$  n = f  $\sim$  (m * n)
  for f :: 'a  $\Rightarrow$  'a
  <proof>

lemma funpow-swap1: f ((f  $\sim$  n) x) = (f  $\sim$  n) (f x)
  <proof>

lemma comp-funpow: comp f  $\sim$  n = comp (f  $\sim$  n)
  for f :: 'a  $\Rightarrow$  'a
  <proof>

lemma Suc-funpow[simp]: Suc  $\sim$  n = ((+) n)
  <proof>

lemma id-funpow[simp]: id  $\sim$  n = id
  <proof>

lemma funpow-mono: mono f  $\implies$  A  $\leq$  B  $\implies$  (f  $\sim$  n) A  $\leq$  (f  $\sim$  n) B
  for f :: 'a  $\Rightarrow$  ('a::order)
  <proof>

```

lemma *funpow-mono2*:

assumes *mono f*
 and $i \leq j$
 and $x \leq y$
 and $x \leq f\ x$
 shows $(f \rightsquigarrow i)\ x \leq (f \rightsquigarrow j)\ y$
 $\langle proof \rangle$

lemma *inj-fn[simp]*:

fixes $f :: 'a \Rightarrow 'a$
 assumes *inj f*
 shows *inj* $(f \rightsquigarrow n)$
 $\langle proof \rangle$

lemma *surj-fn[simp]*:

fixes $f :: 'a \Rightarrow 'a$
 assumes *surj f*
 shows *surj* $(f \rightsquigarrow n)$
 $\langle proof \rangle$

lemma *bij-fn[simp]*:

fixes $f :: 'a \Rightarrow 'a$
 assumes *bij f*
 shows *bij* $(f \rightsquigarrow n)$
 $\langle proof \rangle$

lemma *bij-betw-funpow*:

assumes *bij-betw f S S* shows *bij-betw* $(f \rightsquigarrow n)\ S\ S$
 $\langle proof \rangle$

17.6 Kleene iteration

lemma *Kleene-iter-lfp*:

fixes $f :: 'a :: \text{order-bot} \Rightarrow 'a$
 assumes *mono f*
 and $f\ p \leq p$
 shows $(f \rightsquigarrow k)\ \text{bot} \leq p$
 $\langle proof \rangle$

lemma *lfp-Kleene-iter*:

assumes *mono f*
 and $(f \rightsquigarrow \text{Suc } k)\ \text{bot} = (f \rightsquigarrow k)\ \text{bot}$
 shows $\text{lfp } f = (f \rightsquigarrow k)\ \text{bot}$
 $\langle proof \rangle$

lemma *mono-pow*: $\text{mono } f \implies \text{mono } (f \rightsquigarrow n)$

for $f :: 'a \Rightarrow 'a :: \text{order}$
 $\langle proof \rangle$

lemma *lfp-funpow*:
 assumes *f*: *mono* *f*
 shows $\text{lfp } (f \text{ } \rightsquigarrow \text{ Suc } n) = \text{lfp } f$
 $\langle \text{proof} \rangle$

lemma *gfp-funpow*:
 assumes *f*: *mono* *f*
 shows $\text{gfp } (f \text{ } \rightsquigarrow \text{ Suc } n) = \text{gfp } f$
 $\langle \text{proof} \rangle$

lemma *Kleene-iter-gfp*:
 fixes *f* :: '*a*::*order-top* \Rightarrow '*a*
 assumes *mono* *f*
 and $p \leq f \text{ } p$
 shows $p \leq (f \text{ } \rightsquigarrow k) \text{ top}$
 $\langle \text{proof} \rangle$

lemma *gfp-Kleene-iter*:
 assumes *mono* *f*
 and $(f \text{ } \rightsquigarrow \text{ Suc } k) \text{ top} = (f \text{ } \rightsquigarrow k) \text{ top}$
 shows $\text{gfp } f = (f \text{ } \rightsquigarrow k) \text{ top}$
 (is *?lhs* = *?rhs*)
 $\langle \text{proof} \rangle$

17.7 Embedding of the naturals into any *semiring-1*: *of-nat*

context *semiring-1*
begin

definition *of-nat* :: *nat* \Rightarrow '*a*
 where *of-nat* *n* = (*plus* 1 \rightsquigarrow *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*
 $\langle \text{proof} \rangle$

lemma *of-nat-1* [*simp*]: *of-nat* 1 = 1
 $\langle \text{proof} \rangle$

lemma *of-nat-add* [*simp*]: *of-nat* (*m* + *n*) = *of-nat* *m* + *of-nat* *n*
 $\langle \text{proof} \rangle$

lemma *of-nat-mult* [*simp*]: *of-nat* (*m* * *n*) = *of-nat* *m* * *of-nat* *n*
 $\langle \text{proof} \rangle$

lemma *mult-of-nat-commute*: *of-nat* *x* * *y* = *y* * *of-nat* *x*
 $\langle \text{proof} \rangle$

```

primrec of-nat-aux :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  nat  $\Rightarrow$  'a  $\Rightarrow$  '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
   $\langle$ proof $\rangle$ 

```

```

lemma of-nat-of-bool [simp]:
  of-nat (of-bool P) = of-bool P
   $\langle$ proof $\rangle$ 

```

```

end

```

```

declare of-nat-code [code]

```

```

context semiring-1-cancel
begin

```

```

lemma of-nat-diff [simp]:
   $\langle$ of-nat (m - n) = of-nat m - of-nat n $\rangle$  if  $\langle$ n  $\leq$  m $\rangle$ 
   $\langle$ proof $\rangle$ 

```

```

lemma of-nat-diff-iff:  $\langle$ of-nat (m - n) = (if n  $\leq$  m then of-nat m - of-nat n else 0) $\rangle$ 
   $\langle$ proof $\rangle$ 

```

```

end

```

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

```

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  $\longleftrightarrow$  m = n
   $\langle$ proof $\rangle$ 

```

Special cases where either operand is zero

```

lemma of-nat-0-eq-iff [simp]: 0 = of-nat n  $\longleftrightarrow$  0 = n
   $\langle$ proof $\rangle$ 

```

```

lemma of-nat-eq-0-iff [simp]: of-nat m = 0  $\longleftrightarrow$  m = 0
   $\langle$ proof $\rangle$ 

```

```

lemma of-nat-1-eq-iff [simp]: 1 = of-nat n  $\longleftrightarrow$  n=1
   $\langle$ proof $\rangle$ 

```

lemma *of-nat-eq-1-iff* [*simp*]: $\text{of-nat } n = 1 \longleftrightarrow n=1$
 ⟨*proof*⟩

lemma *of-nat-neq-0* [*simp*]: $\text{of-nat } (\text{Suc } n) \neq 0$
 ⟨*proof*⟩

lemma *of-nat-0-neq* [*simp*]: $0 \neq \text{of-nat } (\text{Suc } n)$
 ⟨*proof*⟩

end

class *ring-char-0* = *ring-1* + *semiring-char-0*

lemma (in *ordered-semiring-1*) *of-nat-0-le-iff* [*simp*]: $0 \leq \text{of-nat } n$
 ⟨*proof*⟩

context *linordered-nonzero-semiring*
begin

lemma *of-nat-less-0-iff* [*simp*]: $\neg \text{of-nat } m < 0$
 ⟨*proof*⟩

lemma *of-nat-mono* [*simp*]: $i \leq j \implies \text{of-nat } i \leq \text{of-nat } j$
 ⟨*proof*⟩

lemma *of-nat-less-iff* [*simp*]: $\text{of-nat } m < \text{of-nat } n \longleftrightarrow m < n$
 ⟨*proof*⟩

lemma *of-nat-le-iff* [*simp*]: $\text{of-nat } m \leq \text{of-nat } n \longleftrightarrow m \leq n$
 ⟨*proof*⟩

lemma *less-imp-of-nat-less*: $m < n \implies \text{of-nat } m < \text{of-nat } n$
 ⟨*proof*⟩

lemma *of-nat-less-imp-less*: $\text{of-nat } m < \text{of-nat } n \implies m < n$
 ⟨*proof*⟩

Every *linordered-nonzero-semiring* has characteristic zero.

subclass *semiring-char-0*
 ⟨*proof*⟩

Special cases where either operand is zero

lemma *of-nat-le-0-iff* [*simp*]: $\text{of-nat } m \leq 0 \longleftrightarrow m = 0$
 ⟨*proof*⟩

lemma *of-nat-0-less-iff* [*simp*]: $0 < \text{of-nat } n \longleftrightarrow 0 < n$
 ⟨*proof*⟩

end

```

context linordered-nonzero-semiring
begin

lemma of-nat-max: of-nat (max x y) = max (of-nat x) (of-nat y)
  ⟨proof⟩

lemma of-nat-min: of-nat (min x y) = min (of-nat x) (of-nat y)
  ⟨proof⟩

end

context linordered-semidom
begin

subclass linordered-nonzero-semiring ⟨proof⟩

subclass semiring-char-0 ⟨proof⟩

end

context linordered-idom
begin

lemma abs-of-nat [simp]:
  |of-nat n| = of-nat n
  ⟨proof⟩

lemma sgn-of-nat [simp]:
  sgn (of-nat n) = of-bool (n > 0)
  ⟨proof⟩

end

lemma of-nat-id [simp]: of-nat n = n
  ⟨proof⟩

lemma of-nat-eq-id [simp]: of-nat = id
  ⟨proof⟩

```

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

```

$\langle proof \rangle$

lemma *Nats-0* [*simp*]: $0 \in \mathbb{N}$
 $\langle proof \rangle$

lemma *Nats-1* [*simp*]: $1 \in \mathbb{N}$
 $\langle proof \rangle$

lemma *Nats-add* [*simp*]: $a \in \mathbb{N} \implies b \in \mathbb{N} \implies a + b \in \mathbb{N}$
 $\langle proof \rangle$

lemma *Nats-mult* [*simp*]: $a \in \mathbb{N} \implies b \in \mathbb{N} \implies a * b \in \mathbb{N}$
 $\langle proof \rangle$

lemma *Nats-cases* [*cases set: Nats*]:
assumes $x \in \mathbb{N}$
obtains $(of\text{-}nat) \ n$ **where** $x = of\text{-}nat \ n$
 $\langle proof \rangle$

lemma *Nats-induct* [*case-names of-nat, induct set: Nats*]: $x \in \mathbb{N} \implies (\bigwedge n. P (of\text{-}nat \ n)) \implies P \ x$
 $\langle proof \rangle$

lemma *Nats-nonempty* [*simp*]: $\mathbb{N} \neq \{\}$
 $\langle proof \rangle$

end

lemma *Nats-diff* [*simp*]:
fixes $a::'a::linordered-idom$
assumes $a \in \mathbb{N} \ b \in \mathbb{N} \ b \leq a$ **shows** $a - b \in \mathbb{N}$
 $\langle proof \rangle$

17.9 Further arithmetic facts concerning the natural numbers

lemma *subst-equals*:
assumes $t = s$ **and** $u = t$
shows $u = s$
 $\langle proof \rangle$

locale *nat-arith*
begin

lemma *add1*: $(A::'a::comm-monoid-add) \equiv k + a \implies A + b \equiv k + (a + b)$
 $\langle proof \rangle$

lemma *add2*: $(B::'a::comm-monoid-add) \equiv k + b \implies a + B \equiv k + (a + b)$
 $\langle proof \rangle$

lemma *suc1*: $A == k + a \implies \text{Suc } A \equiv k + \text{Suc } a$
 $\langle \text{proof} \rangle$

lemma *rule0*: $(a::'a::\text{comm-monoid-add}) \equiv a + 0$
 $\langle \text{proof} \rangle$

end

$\langle ML \rangle$

context *preorder*
begin

lemma *lift-Suc-mono-le*:
 assumes *mono*: $\bigwedge n. f\ n \leq f\ (\text{Suc } n)$
 and $n \leq n'$
 shows $f\ n \leq f\ n'$
 $\langle \text{proof} \rangle$

lemma *lift-Suc-antimono-le*:
 assumes *mono*: $\bigwedge n. f\ n \geq f\ (\text{Suc } n)$
 and $n \leq n'$
 shows $f\ n \geq f\ n'$
 $\langle \text{proof} \rangle$

lemma *lift-Suc-mono-less*:
 assumes *mono*: $\bigwedge n. f\ n < f\ (\text{Suc } n)$
 and $n < n'$
 shows $f\ n < f\ n'$
 $\langle \text{proof} \rangle$

lemma *lift-Suc-mono-less-iff*: $(\bigwedge n. f\ n < f\ (\text{Suc } n)) \implies f\ n < f\ m \longleftrightarrow n < m$
 $\langle \text{proof} \rangle$

end

lemma *mono-iff-le-Suc*: $\text{mono } f \longleftrightarrow (\forall n. f\ n \leq f\ (\text{Suc } n))$
 $\langle \text{proof} \rangle$

lemma *antimono-iff-le-Suc*: $\text{antimono } f \longleftrightarrow (\forall n. f\ (\text{Suc } n) \leq f\ n)$
 $\langle \text{proof} \rangle$

lemma *strict-mono-Suc-iff*: $\text{strict-mono } f \longleftrightarrow (\forall n. f\ n < f\ (\text{Suc } n))$
 $\langle \text{proof} \rangle$

lemma *strict-mono-add*: $\text{strict-mono } (\lambda n::'a::\text{linordered-semidom}. n + k)$
 $\langle \text{proof} \rangle$

lemma *mono-nat-linear-lb*:

fixes $f :: \text{nat} \Rightarrow \text{nat}$

assumes $\bigwedge m n. m < n \implies f\ m < f\ n$

shows $f\ m + k \leq f\ (m + k)$

$\langle \text{proof} \rangle$

lemma *bex-const1-if-mono-below-diag*: **fixes** $f :: \text{nat} \Rightarrow \text{nat}$ **assumes** *mono f*

shows $f\ n < n \implies \exists i < n. f(\text{Suc } i) = f\ i$

$\langle \text{proof} \rangle$

lemma *bex-const1-if-mono-below-diag-Suc*:

fixes $f :: \text{nat} \Rightarrow \text{nat}$ **assumes** *mono f* $f(\text{Suc } m) \leq m$

shows $\exists i \leq m. f(\text{Suc } i) = f\ i$

$\langle \text{proof} \rangle$

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*:

fixes $a\ b\ c :: \text{nat}$

assumes $a < b$ **and** $c \leq a$

shows $a - c < b - c$

$\langle \text{proof} \rangle$

lemma *less-diff-conv*: $i < j - k \longleftrightarrow i + k < j$

for $i\ j\ k :: \text{nat}$

$\langle \text{proof} \rangle$

lemma *less-diff-conv2*: $k \leq j \implies j - k < i \longleftrightarrow j < i + k$

for $j\ k\ i :: \text{nat}$

$\langle \text{proof} \rangle$

lemma *le-diff-conv*: $j - k \leq i \longleftrightarrow j \leq i + k$

for $j\ k\ i :: \text{nat}$

$\langle \text{proof} \rangle$

lemma *diff-diff-cancel* [*simp*]: $i \leq n \implies n - (n - i) = i$

for $i\ n :: \text{nat}$

$\langle \text{proof} \rangle$

lemma *diff-less* [*simp*]: $0 < n \implies 0 < m \implies m - n < m$

for $i\ n :: \text{nat}$

$\langle \text{proof} \rangle$

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: $k \leq m \implies k \leq n \implies m - k - (n - k) = m - n$

for $m\ n\ k :: \text{nat}$

$\langle \text{proof} \rangle$

hide-fact (**open**) *diff-diff-eq*

lemma *eq-diff-iff*: $k \leq m \implies k \leq n \implies m - k = n - k \longleftrightarrow m = n$
for $m\ n\ k :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *less-diff-iff*: $k \leq m \implies k \leq n \implies m - k < n - k \longleftrightarrow m < n$
for $m\ n\ k :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *le-diff-iff*: $k \leq m \implies k \leq n \implies m - k \leq n - k \longleftrightarrow m \leq n$
for $m\ n\ k :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *le-diff-iff'*: $a \leq c \implies b \leq c \implies c - a \leq c - b \longleftrightarrow b \leq a$
for $a\ b\ c :: \text{nat}$
 $\langle \text{proof} \rangle$

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq n \implies m - l \leq n - l$
for $m\ n\ l :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-le-mono2*: $m \leq n \implies l - n \leq l - m$
for $m\ n\ l :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-less-mono2*: $m < n \implies m < l \implies l - n < l - m$
for $m\ n\ l :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diffs0-imp-equal*: $m - n = 0 \implies n - m = 0 \implies m = n$
for $m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *min-diff*: $\min (m - i) (n - i) = \min m\ n - i$
for $m\ n\ i :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *inj-on-diff-nat*:
fixes $k :: \text{nat}$
assumes $\bigwedge n. n \in N \implies k \leq n$
shows *inj-on* $(\lambda n. n - k)$ N
 $\langle \text{proof} \rangle$

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j \implies i - (j - k) = i + k - j$
for $i\ j\ k :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-Suc-diff-eq1* [*simp*]:

assumes $k \leq j$
shows $i - \text{Suc } (j - k) = i + k - \text{Suc } j$
 $\langle \text{proof} \rangle$

lemma *diff-Suc-diff-eq2* [simp]:
assumes $k \leq j$
shows $\text{Suc } (j - k) - i = \text{Suc } j - (k + i)$
 $\langle \text{proof} \rangle$

lemma *Suc-diff-Suc*:
assumes $n < m$
shows $\text{Suc } (m - \text{Suc } n) = m - n$
 $\langle \text{proof} \rangle$

lemma *one-less-mult*: $\text{Suc } 0 < n \implies \text{Suc } 0 < m \implies \text{Suc } 0 < m * n$
 $\langle \text{proof} \rangle$

lemma *n-less-m-mult-n*: $0 < n \implies \text{Suc } 0 < m \implies n < m * n$
 $\langle \text{proof} \rangle$

lemma *n-less-n-mult-m*: $0 < n \implies \text{Suc } 0 < m \implies n < n * m$
 $\langle \text{proof} \rangle$

Induction starting beyond zero

lemma *nat-induct-at-least* [consumes 1, case-names base Suc]:
 $P \ n \text{ if } n \geq m \ P \ m \ \bigwedge n. n \geq m \implies P \ n \implies P \ (\text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *nat-induct-non-zero* [consumes 1, case-names 1 Suc]:
 $P \ n \text{ if } n > 0 \ P \ 1 \ \bigwedge n. n > 0 \implies P \ n \implies P \ (\text{Suc } n)$
 $\langle \text{proof} \rangle$

Specialized induction principles that work "backwards":

lemma *inc-induct* [consumes 1, case-names base step]:
assumes *less*: $i \leq j$
and *base*: $P \ j$
and *step*: $\bigwedge n. i \leq n \implies n < j \implies P \ (\text{Suc } n) \implies P \ n$
shows $P \ i$
 $\langle \text{proof} \rangle$

lemma *strict-inc-induct* [consumes 1, case-names base step]:
assumes *less*: $i < j$
and *base*: $\bigwedge i. j = \text{Suc } i \implies P \ i$
and *step*: $\bigwedge i. \text{Suc } i < j \implies P \ (\text{Suc } i) \implies P \ i$
shows $P \ i$
 $\langle \text{proof} \rangle$

lemma *zero-induct-lemma*: $P \ k \implies (\bigwedge n. P \ (\text{Suc } n) \implies P \ n) \implies P \ (k - i)$
 $\langle \text{proof} \rangle$

lemma *zero-induct*: $P\ k \implies (\bigwedge n. P\ (Suc\ n) \implies P\ n) \implies P\ 0$
 ⟨proof⟩

Further induction rule similar to $\llbracket ?i \leq ?j; ?P\ ?j; \bigwedge n. \llbracket ?i \leq n; n < ?j; ?P\ (Suc\ n) \rrbracket \implies ?P\ n \rrbracket \implies ?P\ ?i$.

lemma *dec-induct* [consumes 1, case-names base step]:
 $i \leq j \implies P\ i \implies (\bigwedge n. i \leq n \implies n < j \implies P\ n \implies P\ (Suc\ n)) \implies P\ j$
 ⟨proof⟩

lemma *transitive-stepwise-le*:

assumes $m \leq n \bigwedge x. R\ x\ x \bigwedge x\ y\ z. R\ x\ y \implies R\ y\ z \implies R\ x\ z$ **and** $\bigwedge n. R\ n\ (Suc\ n)$
shows $R\ m\ n$
 ⟨proof⟩

17.9.1 Greatest operator

lemma *ex-has-greatest-nat*:
 $P\ (k::nat) \implies \forall y. P\ y \longrightarrow y \leq b \implies \exists x. P\ x \wedge (\forall y. P\ y \longrightarrow y \leq x)$
 ⟨proof⟩

lemma
fixes $k::nat$
assumes $P\ k$ **and** *minor*: $\bigwedge y. P\ y \implies y \leq b$
shows *GreatestI-nat*: $P\ (Greatest\ P)$
and *Greatest-le-nat*: $k \leq Greatest\ P$
 ⟨proof⟩

lemma *GreatestI-ex-nat*:
 $\llbracket \exists k::nat. P\ k; \bigwedge y. P\ y \implies y \leq b \rrbracket \implies P\ (Greatest\ P)$
 ⟨proof⟩

17.10 Monotonicity of funpow

lemma *funpow-increasing*: $m \leq n \implies mono\ f \implies (f \rightsquigarrow n) \top \leq (f \rightsquigarrow m) \top$
for $f :: 'a::order-top \Rightarrow 'a$
 ⟨proof⟩

lemma *funpow-decreasing*: $m \leq n \implies mono\ f \implies (f \rightsquigarrow m) \bot \leq (f \rightsquigarrow n) \bot$
for $f :: 'a::order-bot \Rightarrow 'a$
 ⟨proof⟩

lemma *mono-funpow*: $mono\ Q \implies mono\ (\lambda i. (Q \rightsquigarrow i) \bot)$
for $Q :: 'a::order-bot \Rightarrow 'a$
 ⟨proof⟩

lemma *antimono-funpow*: $mono\ Q \implies antimono\ (\lambda i. (Q \rightsquigarrow i) \top)$
for $Q :: 'a::order-top \Rightarrow 'a$
 ⟨proof⟩

17.11 Kleene’s fixed point theorem for continuous functions

Kleene’s fixed point theorem shows that the *lfp* of a omega-continuous function can be obtained as the supremum of an omega chain. It only requires an omega-complete partial order. We prove it here for complete lattices because the latter structures are not defined in Main but the theorem is also useful for complete lattices.

definition *omega-chain* :: (*nat* \Rightarrow (*'a::complete-lattice*) \Rightarrow *bool*) **where**
omega-chain *C* = ($\forall i. C\ i \leq C(Suc\ i)$)

definition *omega-cont* :: ((*'a::complete-lattice*) \Rightarrow (*'b::complete-lattice*) \Rightarrow *bool*) **where**
omega-cont *f* = ($\forall C. \text{omega-chain } C \longrightarrow f(SUP\ n. C\ n) = (SUP\ n. f(C\ n))$)

lemma *omega-chain-mono*: *omega-chain* *C* $\Longrightarrow i \leq j \Longrightarrow C\ i \leq C\ j$
 $\langle proof \rangle$

lemma *mono-if-omega-cont*: **fixes** *f* :: (*'a::complete-lattice*) \Rightarrow (*'b::complete-lattice*)
assumes *omega-cont* *f* **shows** *mono* *f*
 $\langle proof \rangle$

lemma *omega-chain-iterates*: **fixes** *f* :: (*'a::complete-lattice*) \Rightarrow *'a*
assumes *mono* *f* **shows** *omega-chain*($\lambda n. (f \sim n)$ *bot*)
 $\langle proof \rangle$

theorem *Kleene-lfp*:
assumes *omega-cont* *f* **shows** *lfp* *f* = (*SUP* *n*. (*f* \sim *n*) *bot*) (**is** - = ?*U*)
 $\langle proof \rangle$

17.12 The divides relation on *nat*

lemma *dvd-1-left* [*iff*]: *Suc* 0 *dvd* *k*
 $\langle proof \rangle$

lemma *dvd-1-iff-1* [*simp*]: *m* *dvd* *Suc* 0 $\longleftrightarrow m = Suc\ 0$
 $\langle proof \rangle$

lemma *nat-dvd-1-iff-1* [*simp*]: *m* *dvd* 1 $\longleftrightarrow m = 1$
for *m* :: *nat*
 $\langle proof \rangle$

lemma *dvd-antisym*: *m* *dvd* *n* $\Longrightarrow n$ *dvd* *m* $\Longrightarrow m = n$
for *m* *n* :: *nat*
 $\langle proof \rangle$

lemma *dvd-diff-nat* [*simp*]: *k* *dvd* *m* $\Longrightarrow k$ *dvd* *n* $\Longrightarrow k$ *dvd* (*m* - *n*)
for *k* *m* *n* :: *nat*
 $\langle proof \rangle$

lemma *dvd-diffD*:

fixes $k\ m\ n :: \text{nat}$

assumes $k\ \text{dvd}\ m - n\ k\ \text{dvd}\ n\ n \leq m$

shows $k\ \text{dvd}\ m$

<proof>

lemma *dvd-diffD1*: $k\ \text{dvd}\ m - n \implies k\ \text{dvd}\ m \implies n \leq m \implies k\ \text{dvd}\ n$

for $k\ m\ n :: \text{nat}$

<proof>

lemma *dvd-mult-cancel*:

fixes $m\ n\ k :: \text{nat}$

assumes $k * m\ \text{dvd}\ k * n$ **and** $0 < k$

shows $m\ \text{dvd}\ n$

<proof>

lemma *dvd-mult-cancel1*:

fixes $m\ n :: \text{nat}$

assumes $0 < m$

shows $m * n\ \text{dvd}\ m \longleftrightarrow n = 1$

<proof>

lemma *dvd-mult-cancel2*: $0 < m \implies n * m\ \text{dvd}\ m \longleftrightarrow n = 1$

for $m\ n :: \text{nat}$

<proof>

lemma *dvd-imp-le*: $k\ \text{dvd}\ n \implies 0 < n \implies k \leq n$

for $k\ n :: \text{nat}$

<proof>

lemma *nat-dvd-not-less*: $0 < m \implies m < n \implies \neg n\ \text{dvd}\ m$

for $m\ n :: \text{nat}$

<proof>

lemma *less-eq-dvd-minus*:

fixes $m\ n :: \text{nat}$

assumes $m \leq n$

shows $m\ \text{dvd}\ n \longleftrightarrow m\ \text{dvd}\ n - m$

<proof>

lemma *dvd-minus-self*: $m\ \text{dvd}\ n - m \longleftrightarrow n < m \vee m\ \text{dvd}\ n$

for $m\ n :: \text{nat}$

<proof>

lemma *dvd-minus-add*:

fixes $m\ n\ q\ r :: \text{nat}$

assumes $q \leq n\ q \leq r * m$

shows $m\ \text{dvd}\ n - q \longleftrightarrow m\ \text{dvd}\ n + (r * m - q)$

<proof>

17.13 Aliasses

lemma *nat-mult-1*: $1 * n = n$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *nat-mult-1-right*: $n * 1 = n$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-mult-distrib*: $(m - n) * k = (m * k) - (n * k)$
for $k \ m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-mult-distrib2*: $k * (m - n) = (k * m) - (k * n)$
for $k \ m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *le-diff-conv2*: $k \leq j \implies (i \leq j - k) = (i + k \leq j)$
for $i \ j \ k :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-self-eq-0* [*simp*]: $m - m = 0$
for $m :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-diff-left* [*simp*]: $i - j - k = i - (j + k)$
for $i \ j \ k :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-commute*: $i - j - k = i - k - j$
for $i \ j \ k :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-add-inverse*: $(n + m) - n = m$
for $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-add-inverse2*: $(m + n) - n = m$
for $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-cancel*: $(k + m) - (k + n) = m - n$
for $k \ m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *diff-cancel2*: $(m + k) - (n + k) = m - n$
for $k \ m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

```

lemma diff-add-0:  $n - (n + m) = 0$ 
  for  $m\ n :: \text{nat}$ 
   $\langle \text{proof} \rangle$ 

lemma add-mult-distrib2:  $k * (m + n) = (k * m) + (k * n)$ 
  for  $k\ m\ n :: \text{nat}$ 
   $\langle \text{proof} \rangle$ 

lemmas nat-distrib =
  add-mult-distrib distrib-left diff-mult-distrib diff-mult-distrib2

```

17.14 Size of a datatype value

```

class size =
  fixes size :: 'a  $\Rightarrow$  nat — see further theory Wellfounded

instantiation nat :: size
begin

definition size-nat where [simp, code]: size ( $n :: \text{nat}$ ) =  $n$ 

instance  $\langle \text{proof} \rangle$ 

end

lemmas size-nat = size-nat-def

lemma size-neq-size-imp-neq:  $\text{size } x \neq \text{size } y \implies x \neq y$ 
   $\langle \text{proof} \rangle$ 

```

17.15 Code module namespace

```

code-identifier
  code-module Nat  $\hookrightarrow$  (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  $\Rightarrow$  'a
begin

abbreviation inverse-divide :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl <'/'> 70)
where
  inverse-divide  $\equiv$  divide

end

```

Setup for linear arithmetic prover

<ML>

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]:  $a \neq 0 \implies \text{inverse } a * a = 1$ 
  assumes right-inverse [simp]:  $a \neq 0 \implies a * \text{inverse } a = 1$ 
  assumes divide-inverse:  $a / b = a * \text{inverse } b$ 
  assumes inverse-zero [simp]:  $\text{inverse } 0 = 0$ 
begin

```

```

subclass ring-1-no-zero-divisors
<proof>

```

```

lemma nonzero-imp-inverse-nonzero:
   $a \neq 0 \implies \text{inverse } a \neq 0$ 
<proof>

```

```

lemma inverse-zero-imp-zero:
  assumes  $\text{inverse } a = 0$  shows  $a = 0$ 
<proof>

```

```

lemma inverse-unique:
  assumes ab:  $a * b = 1$ 
  shows  $\text{inverse } a = b$ 
<proof>

```

```

lemma nonzero-inverse-minus-eq:
   $a \neq 0 \implies \text{inverse } (- a) = - \text{inverse } a$ 
<proof>

```

```

lemma nonzero-inverse-inverse-eq:
   $a \neq 0 \implies \text{inverse } (\text{inverse } a) = a$ 
<proof>

```

lemma *nonzero-inverse-eq-imp-eq*:

assumes $\text{inverse } a = \text{inverse } b$ **and** $a \neq 0$ **and** $b \neq 0$

shows $a = b$

$\langle \text{proof} \rangle$

lemma *inverse-1 [simp]*: $\text{inverse } 1 = 1$

$\langle \text{proof} \rangle$

subclass *divide-trivial*

$\langle \text{proof} \rangle$

lemma *nonzero-inverse-mult-distrib*:

assumes $a \neq 0$ **and** $b \neq 0$

shows $\text{inverse } (a * b) = \text{inverse } b * \text{inverse } a$

$\langle \text{proof} \rangle$

lemma *division-ring-inverse-add*:

$a \neq 0 \implies b \neq 0 \implies \text{inverse } a + \text{inverse } b = \text{inverse } a * (a + b) * \text{inverse } b$

$\langle \text{proof} \rangle$

lemma *division-ring-inverse-diff*:

$a \neq 0 \implies b \neq 0 \implies \text{inverse } a - \text{inverse } b = \text{inverse } a * (b - a) * \text{inverse } b$

$\langle \text{proof} \rangle$

lemma *right-inverse-eq*: $b \neq 0 \implies a / b = 1 \longleftrightarrow a = b$

$\langle \text{proof} \rangle$

lemma *nonzero-inverse-eq-divide*: $a \neq 0 \implies \text{inverse } a = 1 / a$

$\langle \text{proof} \rangle$

lemma *divide-self [simp]*: $a \neq 0 \implies a / a = 1$

$\langle \text{proof} \rangle$

lemma *inverse-eq-divide [field-simps, field-split-simps, divide-simps]*: $\text{inverse } a = 1 / a$

$\langle \text{proof} \rangle$

lemma *add-divide-distrib*: $(a+b) / c = a/c + b/c$

$\langle \text{proof} \rangle$

lemma *times-divide-eq-right [simp]*: $a * (b / c) = (a * b) / c$

$\langle \text{proof} \rangle$

lemma *minus-divide-left*: $-(a / b) = (-a) / b$

$\langle \text{proof} \rangle$

lemma *nonzero-minus-divide-right*: $b \neq 0 \implies -(a / b) = a / (-b)$

$\langle \text{proof} \rangle$

lemma *nonzero-minus-divide-divide*: $b \neq 0 \implies (-a) / (-b) = a / b$
 ⟨proof⟩

lemma *divide-minus-left* [simp]: $(-a) / b = -(a / b)$
 ⟨proof⟩

lemma *diff-divide-distrib*: $(a - b) / c = a / c - b / c$
 ⟨proof⟩

lemma *nonzero-eq-divide-eq* [field-simps]: $c \neq 0 \implies a = b / c \longleftrightarrow a * c = b$
 ⟨proof⟩

lemma *nonzero-divide-eq-eq* [field-simps]: $c \neq 0 \implies b / c = a \longleftrightarrow b = a * c$
 ⟨proof⟩

lemma *nonzero-neg-divide-eq-eq* [field-simps]: $b \neq 0 \implies -(a / b) = c \longleftrightarrow -a = c * b$
 ⟨proof⟩

lemma *nonzero-neg-divide-eq-eq2* [field-simps]: $b \neq 0 \implies c = -(a / b) \longleftrightarrow c * b = -a$
 ⟨proof⟩

lemma *divide-eq-imp*: $c \neq 0 \implies b = a * c \implies b / c = a$
 ⟨proof⟩

lemma *eq-divide-imp*: $c \neq 0 \implies a * c = b \implies a = b / c$
 ⟨proof⟩

lemma *add-divide-eq-iff* [field-simps]:
 $z \neq 0 \implies x + y / z = (x * z + y) / z$
 ⟨proof⟩

lemma *divide-add-eq-iff* [field-simps]:
 $z \neq 0 \implies x / z + y = (x + y * z) / z$
 ⟨proof⟩

lemma *diff-divide-eq-iff* [field-simps]:
 $z \neq 0 \implies x - y / z = (x * z - y) / z$
 ⟨proof⟩

lemma *minus-divide-add-eq-iff* [field-simps]:
 $z \neq 0 \implies -(x / z) + y = (-x + y * z) / z$
 ⟨proof⟩

lemma *divide-diff-eq-iff* [field-simps]:
 $z \neq 0 \implies x / z - y = (x - y * z) / z$
 ⟨proof⟩

lemma *minus-divide-diff-eq-iff* [*field-simps*]:
 $z \neq 0 \implies -(x / z) - y = (-x - y * z) / z$
 ⟨*proof*⟩

lemma *division-ring-divide-zero*:
 $a / 0 = 0$
 ⟨*proof*⟩

lemma *divide-self-if* [*simp*]:
 $a / a = (\text{if } a = 0 \text{ then } 0 \text{ else } 1)$
 ⟨*proof*⟩

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
 $\text{inverse } a = 0 \iff a = 0$
 ⟨*proof*⟩

lemma *inverse-minus-eq* [*simp*]:
 $\text{inverse } (-a) = -\text{inverse } a$
 ⟨*proof*⟩

lemma *inverse-inverse-eq* [*simp*]:
 $\text{inverse } (\text{inverse } a) = a$
 ⟨*proof*⟩

lemma *inverse-eq-imp-eq*:
 $\text{inverse } a = \text{inverse } b \implies a = b$
 ⟨*proof*⟩

lemma *inverse-eq-iff-eq* [*simp*]:
 $\text{inverse } a = \text{inverse } b \iff a = b$
 ⟨*proof*⟩

lemma *mult-commute-imp-mult-inverse-commute*:
 assumes $y * x = x * y$
 shows $\text{inverse } y * x = x * \text{inverse } y$
 ⟨*proof*⟩

lemmas *mult-inverse-of-nat-commute* =
 $\text{mult-commute-imp-mult-inverse-commute}[OF \text{ mult-of-nat-commute}]$

lemma *divide-divide-eq-left'*:
 $(a / b) / c = a / (c * b)$
 ⟨*proof*⟩

lemma *add-divide-eq-if-simps* [*field-split-simps*, *divide-simps*]:
 $a + b / z = (\text{if } z = 0 \text{ then } a \text{ else } (a * z + b) / z)$
 $a / z + b = (\text{if } z = 0 \text{ then } b \text{ else } (a + b * z) / z)$
 $-(a / z) + b = (\text{if } z = 0 \text{ then } b \text{ else } (-a + b * z) / z)$

$$\begin{aligned}
a - b / z &= (\text{if } z = 0 \text{ then } a \text{ else } (a * z - b) / z) \\
a / z - b &= (\text{if } z = 0 \text{ then } -b \text{ else } (a - b * z) / z) \\
-(a / z) - b &= (\text{if } z = 0 \text{ then } -b \text{ else } (-a - b * z) / z)
\end{aligned}$$

⟨proof⟩

lemma [*field-split-simps*, *divide-simps*]:

shows *divide-eq-eq*: $b / c = a \longleftrightarrow (\text{if } c \neq 0 \text{ then } b = a * c \text{ else } a = 0)$
and *eq-divide-eq*: $a = b / c \longleftrightarrow (\text{if } c \neq 0 \text{ then } a * c = b \text{ else } a = 0)$
and *minus-divide-eq-eq*: $-(b / c) = a \longleftrightarrow (\text{if } c \neq 0 \text{ then } -b = a * c \text{ else } a = 0)$
and *eq-minus-divide-eq*: $a = -(b / c) \longleftrightarrow (\text{if } c \neq 0 \text{ then } a * c = -b \text{ else } a = 0)$
 ⟨proof⟩

end

18.2 Fields

class *field* = *comm-ring-1* + *inverse* +
assumes *field-inverse*: $a \neq 0 \implies \text{inverse } a * a = 1$
assumes *field-divide-inverse*: $a / b = a * \text{inverse } b$
assumes *field-inverse-zero*: $\text{inverse } 0 = 0$
begin

subclass *division-ring*
 ⟨proof⟩

subclass *idom-divide*
 ⟨proof⟩

There is no slick version using division by zero.

lemma *inverse-add*:

$a \neq 0 \implies b \neq 0 \implies \text{inverse } a + \text{inverse } b = (a + b) * \text{inverse } a * \text{inverse } b$
 ⟨proof⟩

lemma *nonzero-mult-divide-mult-cancel-left* [*simp*]:

assumes [*simp*]: $c \neq 0$
shows $(c * a) / (c * b) = a / b$
 ⟨proof⟩

lemma *nonzero-mult-divide-mult-cancel-right* [*simp*]:

$c \neq 0 \implies (a * c) / (b * c) = a / b$
 ⟨proof⟩

lemma *times-divide-eq-left* [*simp*]: $(b / c) * a = (b * a) / c$

⟨proof⟩

lemma *divide-inverse-commute*: $a / b = \text{inverse } b * a$

⟨proof⟩

lemma *add-frac-eq*:

assumes $y \neq 0$ **and** $z \neq 0$

shows $x / y + w / z = (x * z + w * y) / (y * z)$

<proof>

Special Cancellation Simprules for Division

lemma *nonzero-divide-mult-cancel-right* [simp]:

$b \neq 0 \implies b / (a * b) = 1 / a$

<proof>

lemma *nonzero-divide-mult-cancel-left* [simp]:

$a \neq 0 \implies a / (a * b) = 1 / b$

<proof>

lemma *nonzero-mult-divide-mult-cancel-left2* [simp]:

$c \neq 0 \implies (c * a) / (b * c) = a / b$

<proof>

lemma *nonzero-mult-divide-mult-cancel-right2* [simp]:

$c \neq 0 \implies (a * c) / (c * b) = a / b$

<proof>

lemma *diff-frac-eq*:

$y \neq 0 \implies z \neq 0 \implies x / y - w / z = (x * z - w * y) / (y * z)$

<proof>

lemma *frac-eq-eq*:

$y \neq 0 \implies z \neq 0 \implies (x / y = w / z) = (x * z = w * y)$

<proof>

lemma *divide-minus1* [simp]: $x / - 1 = - x$

<proof>

This version builds in division by zero while also re-orienting the right-hand side.

lemma *inverse-mult-distrib* [simp]:

$\text{inverse } (a * b) = \text{inverse } a * \text{inverse } b$

<proof>

lemma *inverse-divide* [simp]:

$\text{inverse } (a / b) = b / a$

<proof>

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]:

$$a / (b / c) = (a * c) / b$$

<proof>

lemma *divide-divide-eq-left* [simp]:

$$(a / b) / c = a / (b * c)$$

<proof>

lemma *divide-divide-times-eq*:

$$(x / y) / (z / w) = (x * w) / (y * z)$$

<proof>

Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if* [simp]:

$$\text{shows } (c * a) / (c * b) = (\text{if } c = 0 \text{ then } 0 \text{ else } a / b)$$

<proof>

Division and Unary Minus

lemma *minus-divide-right*:

$$-(a / b) = a / -b$$

<proof>

lemma *divide-minus-right* [simp]:

$$a / -b = -(a / b)$$

<proof>

lemma *minus-divide-divide*:

$$(-a) / (-b) = a / b$$

<proof>

lemma *inverse-eq-1-iff* [simp]:

$$\text{inverse } x = 1 \longleftrightarrow x = 1$$

<proof>

lemma *divide-eq-0-iff* [simp]:

$$a / b = 0 \longleftrightarrow a = 0 \vee b = 0$$

<proof>

lemma *divide-cancel-right* [simp]:

$$a / c = b / c \longleftrightarrow c = 0 \vee a = b$$

<proof>

lemma *divide-cancel-left* [simp]:

$$c / a = c / b \longleftrightarrow c = 0 \vee a = b$$

<proof>

lemma *divide-eq-1-iff* [simp]:
 $a / b = 1 \longleftrightarrow b \neq 0 \wedge a = b$
 ⟨proof⟩

lemma *one-eq-divide-iff* [simp]:
 $1 = a / b \longleftrightarrow b \neq 0 \wedge a = b$
 ⟨proof⟩

lemma *divide-eq-minus-1-iff*:
 $(a / b = - 1) \longleftrightarrow b \neq 0 \wedge a = - b$
 ⟨proof⟩

lemma *times-divide-times-eq*:
 $(x / y) * (z / w) = (x * z) / (y * w)$
 ⟨proof⟩

lemma *add-frac-num*:
 $y \neq 0 \implies x / y + z = (x + z * y) / y$
 ⟨proof⟩

lemma *add-num-frac*:
 $y \neq 0 \implies z + x / y = (x + z * y) / y$
 ⟨proof⟩

lemma *dvd-field-iff*:
 $a \text{ dvd } b \longleftrightarrow (a = 0 \longrightarrow b = 0)$
 ⟨proof⟩

lemma *inj-divide-right* [simp]:
 $\text{inj } (\lambda b. b / a) \longleftrightarrow a \neq 0$
 ⟨proof⟩

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]:
 $\text{sgn } (\text{inverse } a) = \text{inverse } (\text{sgn } a)$
 ⟨proof⟩

lemma *abs-inverse* [simp]:
 $|\text{inverse } a| = \text{inverse } |a|$
 ⟨proof⟩

lemma *sgn-divide* [*simp*]:
 $\text{sgn } (a / b) = \text{sgn } a / \text{sgn } b$
 ⟨*proof*⟩

lemma *abs-divide* [*simp*]:
 $|a / b| = |a| / |b|$
 ⟨*proof*⟩

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*⟩

lemma *negative-imp-inverse-negative*:
 $a < 0 \implies \text{inverse } a < 0$
 ⟨*proof*⟩

lemma *inverse-le-imp-le*:
 assumes *invle*: $\text{inverse } a \leq \text{inverse } b$ and *apos*: $0 < a$
 shows $b \leq a$
 ⟨*proof*⟩

lemma *inverse-positive-imp-positive*:
 assumes *inv-gt-0*: $0 < \text{inverse } a$ and *nz*: $a \neq 0$
 shows $0 < a$
 ⟨*proof*⟩

lemma *inverse-negative-imp-negative*:
 assumes *inv-less-0*: $\text{inverse } a < 0$ and *nz*: $a \neq 0$
 shows $a < 0$
 ⟨*proof*⟩

lemma *linordered-field-no-lb*:
 $\forall x. \exists y. y < x$
 ⟨*proof*⟩

lemma *linordered-field-no-ub*:
 $\forall x. \exists y. y > x$
 ⟨*proof*⟩

lemma *less-imp-inverse-less*:
 assumes *less*: $a < b$ and *apos*: $0 < a$
 shows $\text{inverse } b < \text{inverse } a$
 ⟨*proof*⟩

lemma *inverse-less-imp-less*:

assumes $\text{inverse } a < \text{inverse } b \ 0 < a$

shows $b < a$

$\langle \text{proof} \rangle$

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [simp]:

$0 < a \implies 0 < b \implies \text{inverse } a < \text{inverse } b \longleftrightarrow b < a$

$\langle \text{proof} \rangle$

lemma *le-imp-inverse-le*:

$a \leq b \implies 0 < a \implies \text{inverse } b \leq \text{inverse } a$

$\langle \text{proof} \rangle$

lemma *inverse-le-iff-le* [simp]:

$0 < a \implies 0 < b \implies \text{inverse } a \leq \text{inverse } b \longleftrightarrow b \leq a$

$\langle \text{proof} \rangle$

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:

assumes $\text{inverse } a \leq \text{inverse } b \ b < 0$

shows $b \leq a$

$\langle \text{proof} \rangle$

lemma *less-imp-inverse-less-neg*:

assumes $a < b \ b < 0$

shows $\text{inverse } b < \text{inverse } a$

$\langle \text{proof} \rangle$

lemma *inverse-less-imp-less-neg*:

assumes $\text{inverse } a < \text{inverse } b \ b < 0$

shows $b < a$

$\langle \text{proof} \rangle$

lemma *inverse-less-iff-less-neg* [simp]:

$a < 0 \implies b < 0 \implies \text{inverse } a < \text{inverse } b \longleftrightarrow b < a$

$\langle \text{proof} \rangle$

lemma *le-imp-inverse-le-neg*:

$a \leq b \implies b < 0 \implies \text{inverse } b \leq \text{inverse } a$

$\langle \text{proof} \rangle$

lemma *inverse-le-iff-le-neg* [simp]:

$a < 0 \implies b < 0 \implies \text{inverse } a \leq \text{inverse } b \longleftrightarrow b \leq a$

$\langle \text{proof} \rangle$

lemma *one-less-inverse*:

$0 < a \implies a < 1 \implies 1 < \text{inverse } a$
 $\langle \text{proof} \rangle$

lemma *one-le-inverse*:

$0 < a \implies a \leq 1 \implies 1 \leq \text{inverse } a$
 $\langle \text{proof} \rangle$

lemma *pos-le-divide-eq* [*field-simps*]:

assumes $0 < c$
shows $a \leq b / c \longleftrightarrow a * c \leq b$
 $\langle \text{proof} \rangle$

lemma *pos-less-divide-eq* [*field-simps*]:

assumes $0 < c$
shows $a < b / c \longleftrightarrow a * c < b$
 $\langle \text{proof} \rangle$

lemma *neg-less-divide-eq* [*field-simps*]:

assumes $c < 0$
shows $a < b / c \longleftrightarrow b < a * c$
 $\langle \text{proof} \rangle$

lemma *neg-le-divide-eq* [*field-simps*]:

assumes $c < 0$
shows $a \leq b / c \longleftrightarrow b \leq a * c$
 $\langle \text{proof} \rangle$

lemma *pos-divide-le-eq* [*field-simps*]:

assumes $0 < c$
shows $b / c \leq a \longleftrightarrow b \leq a * c$
 $\langle \text{proof} \rangle$

lemma *pos-divide-less-eq* [*field-simps*]:

assumes $0 < c$
shows $b / c < a \longleftrightarrow b < a * c$
 $\langle \text{proof} \rangle$

lemma *neg-divide-le-eq* [*field-simps*]:

assumes $c < 0$
shows $b / c \leq a \longleftrightarrow a * c \leq b$
 $\langle \text{proof} \rangle$

lemma *neg-divide-less-eq* [*field-simps*]:

assumes $c < 0$
shows $b / c < a \longleftrightarrow a * c < b$
 $\langle \text{proof} \rangle$

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*]: $0 < c \implies a \leq - (b / c) \longleftrightarrow a * c \leq - b$
 ⟨*proof*⟩

lemma *neg-le-minus-divide-eq* [*field-simps*]: $c < 0 \implies a \leq - (b / c) \longleftrightarrow - b \leq a * c$
 ⟨*proof*⟩

lemma *pos-less-minus-divide-eq* [*field-simps*]: $0 < c \implies a < - (b / c) \longleftrightarrow a * c < - b$
 ⟨*proof*⟩

lemma *neg-less-minus-divide-eq* [*field-simps*]: $c < 0 \implies a < - (b / c) \longleftrightarrow - b < a * c$
 ⟨*proof*⟩

lemma *pos-minus-divide-less-eq* [*field-simps*]: $0 < c \implies - (b / c) < a \longleftrightarrow - b < a * c$
 ⟨*proof*⟩

lemma *neg-minus-divide-less-eq* [*field-simps*]: $c < 0 \implies - (b / c) < a \longleftrightarrow a * c < - b$
 ⟨*proof*⟩

lemma *pos-minus-divide-le-eq* [*field-simps*]: $0 < c \implies - (b / c) \leq a \longleftrightarrow - b \leq a * c$
 ⟨*proof*⟩

lemma *neg-minus-divide-le-eq* [*field-simps*]: $c < 0 \implies - (b / c) \leq a \longleftrightarrow a * c \leq - b$
 ⟨*proof*⟩

lemma *frac-less-eq*:
 $y \neq 0 \implies z \neq 0 \implies x / y < w / z \longleftrightarrow (x * z - w * y) / (y * z) < 0$
 ⟨*proof*⟩

lemma *frac-le-eq*:
 $y \neq 0 \implies z \neq 0 \implies x / y \leq w / z \longleftrightarrow (x * z - w * y) / (y * z) \leq 0$
 ⟨*proof*⟩

lemma *divide-pos-pos[simp]*:
 $0 < x \implies 0 < y \implies 0 < x / y$
 ⟨*proof*⟩

lemma *divide-nonneg-pos*:
 $0 \leq x \implies 0 < y \implies 0 \leq x / y$
 ⟨*proof*⟩

lemma *divide-neg-pos*:

$$x < 0 \implies 0 < y \implies x / y < 0$$

<proof>

lemma *divide-nonpos-pos:*

$$x \leq 0 \implies 0 < y \implies x / y \leq 0$$

<proof>

lemma *divide-pos-neg:*

$$0 < x \implies y < 0 \implies x / y < 0$$

<proof>

lemma *divide-nonneg-neg:*

$$0 \leq x \implies y < 0 \implies x / y \leq 0$$

<proof>

lemma *divide-neg-neg:*

$$x < 0 \implies y < 0 \implies 0 < x / y$$

<proof>

lemma *divide-nonpos-neg:*

$$x \leq 0 \implies y < 0 \implies 0 \leq x / y$$

<proof>

lemma *divide-strict-right-mono:*

$$\llbracket a < b; 0 < c \rrbracket \implies a / c < b / c$$

<proof>

lemma *divide-strict-right-mono-neg:*

assumes $b < a$ $c < 0$ **shows** $a / c < b / c$

<proof>

The last premise ensures that a and b have the same sign

lemma *divide-strict-left-mono:*

$$\llbracket b < a; 0 < c; 0 < a*b \rrbracket \implies c / a < c / b$$

<proof>

lemma *divide-left-mono:*

$$\llbracket b \leq a; 0 \leq c; 0 < a*b \rrbracket \implies c / a \leq c / b$$

<proof>

lemma *divide-strict-left-mono-neg:*

$$\llbracket a < b; c < 0; 0 < a*b \rrbracket \implies c / a < c / b$$

<proof>

lemma *mult-imp-div-pos-le:* $0 < y \implies x \leq z * y \implies x / y \leq z$

<proof>

lemma *mult-imp-le-div-pos:* $0 < y \implies z * y \leq x \implies z \leq x / y$

$\langle \text{proof} \rangle$

lemma *mult-imp-div-pos-less*: $0 < y \implies x < z * y \implies x / y < z$
 $\langle \text{proof} \rangle$

lemma *mult-imp-less-div-pos*: $0 < y \implies z * y < x \implies z < x / y$
 $\langle \text{proof} \rangle$

lemma *frac-le*:
 assumes $0 \leq y \ x \leq y \ 0 < w \ w \leq z$
 shows $x / z \leq y / w$
 $\langle \text{proof} \rangle$

lemma *frac-less*:
 assumes $0 \leq x \ x < y \ 0 < w \ w \leq z$
 shows $x / z < y / w$
 $\langle \text{proof} \rangle$

lemma *frac-less2*:
 assumes $0 < x \ x \leq y \ 0 < w \ w < z$
 shows $x / z < y / w$
 $\langle \text{proof} \rangle$

As above, with a better name

lemma *divide-mono*:
 $\llbracket b \leq a; c \leq d; 0 < b; 0 \leq c \rrbracket \implies c / a \leq d / b$
 $\langle \text{proof} \rangle$

lemma *less-half-sum*: $a < b \implies a < (a+b) / (1+1)$
 $\langle \text{proof} \rangle$

lemma *gt-half-sum*: $a < b \implies (a+b)/(1+1) < b$
 $\langle \text{proof} \rangle$

subclass *unbounded-dense-linorder*
 $\langle \text{proof} \rangle$

subclass *field-abs-sgn* $\langle \text{proof} \rangle$

lemma *inverse-sgn [simp]*:
 $\text{inverse } (\text{sgn } a) = \text{sgn } a$
 $\langle \text{proof} \rangle$

lemma *divide-sgn [simp]*:
 $a / \text{sgn } b = a * \text{sgn } b$
 $\langle \text{proof} \rangle$

lemma *nonzero-abs-inverse*:
 $a \neq 0 \implies |\text{inverse } a| = \text{inverse } |a|$

$\langle \text{proof} \rangle$

lemma *nonzero-abs-divide*:

$b \neq 0 \implies |a / b| = |a| / |b|$

$\langle \text{proof} \rangle$

lemma *field-le-epsilon*:

assumes $e: \bigwedge e. 0 < e \implies x \leq y + e$

shows $x \leq y$

$\langle \text{proof} \rangle$

lemma *inverse-positive-iff-positive* [simp]: $(0 < \text{inverse } a) = (0 < a)$

$\langle \text{proof} \rangle$

lemma *inverse-negative-iff-negative* [simp]: $(\text{inverse } a < 0) = (a < 0)$

$\langle \text{proof} \rangle$

lemma *inverse-nonnegative-iff-nonnegative* [simp]: $0 \leq \text{inverse } a \longleftrightarrow 0 \leq a$

$\langle \text{proof} \rangle$

lemma *inverse-nonpositive-iff-nonpositive* [simp]: $\text{inverse } a \leq 0 \longleftrightarrow a \leq 0$

$\langle \text{proof} \rangle$

lemma *one-less-inverse-iff*: $1 < \text{inverse } x \longleftrightarrow 0 < x \wedge x < 1$

$\langle \text{proof} \rangle$

lemma *one-le-inverse-iff*: $1 \leq \text{inverse } x \longleftrightarrow 0 < x \wedge x \leq 1$

$\langle \text{proof} \rangle$

lemma *inverse-less-1-iff*: $\text{inverse } x < 1 \longleftrightarrow x \leq 0 \vee 1 < x$

$\langle \text{proof} \rangle$

lemma *inverse-le-1-iff*: $\text{inverse } x \leq 1 \longleftrightarrow x \leq 0 \vee 1 \leq x$

$\langle \text{proof} \rangle$

lemma [field-split-simps, divide-simps]:

shows *le-divide-eq*: $a \leq b / c \longleftrightarrow (\text{if } 0 < c \text{ then } a * c \leq b \text{ else if } c < 0 \text{ then } b \leq a * c \text{ else } a \leq 0)$

and *divide-le-eq*: $b / c \leq a \longleftrightarrow (\text{if } 0 < c \text{ then } b \leq a * c \text{ else if } c < 0 \text{ then } a * c \leq b \text{ else } 0 \leq a)$

and *less-divide-eq*: $a < b / c \longleftrightarrow (\text{if } 0 < c \text{ then } a * c < b \text{ else if } c < 0 \text{ then } b < a * c \text{ else } a < 0)$

and *divide-less-eq*: $b / c < a \longleftrightarrow (\text{if } 0 < c \text{ then } b < a * c \text{ else if } c < 0 \text{ then } a * c < b \text{ else } 0 < a)$

and *le-minus-divide-eq*: $a \leq -(b / c) \longleftrightarrow (\text{if } 0 < c \text{ then } a * c \leq -b \text{ else if } c < 0 \text{ then } -b \leq a * c \text{ else } a \leq 0)$

and *minus-divide-le-eq*: $-(b / c) \leq a \longleftrightarrow (\text{if } 0 < c \text{ then } -b \leq a * c \text{ else if } c < 0 \text{ then } a * c \leq -b \text{ else } 0 \leq a)$

and *less-minus-divide-eq*: $a < -(b / c) \longleftrightarrow (\text{if } 0 < c \text{ then } a * c < -b \text{ else if } c < 0 \text{ then } -b < a * c \text{ else } a < 0)$

if $c < 0$ then $-b < a * c$ else $a < 0$)
and *minus-divide-less-eq*: $-(b / c) < a \longleftrightarrow (\text{if } 0 < c \text{ then } -b < a * c \text{ else}$
 if $c < 0$ then $a * c < -b$ else $0 < a$)
 $\langle \text{proof} \rangle$

Division and Signs

lemma

shows *zero-less-divide-iff*: $0 < a / b \longleftrightarrow 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$
and *divide-less-0-iff*: $a / b < 0 \longleftrightarrow 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$
and *zero-le-divide-iff*: $0 \leq a / b \longleftrightarrow 0 \leq a \wedge 0 \leq b \vee a \leq 0 \wedge b \leq 0$
and *divide-le-0-iff*: $a / b \leq 0 \longleftrightarrow 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$
 $\langle \text{proof} \rangle$

Division and the Number One

Simplify expressions equated with 1

lemma *zero-eq-1-divide-iff* [simp]: $0 = 1 / a \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *one-divide-eq-0-iff* [simp]: $1 / a = 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemma *zero-le-divide-1-iff* [simp]:
 $0 \leq 1 / a \longleftrightarrow 0 \leq a$
 $\langle \text{proof} \rangle$

lemma *zero-less-divide-1-iff* [simp]:
 $0 < 1 / a \longleftrightarrow 0 < a$
 $\langle \text{proof} \rangle$

lemma *divide-le-0-1-iff* [simp]:
 $1 / a \leq 0 \longleftrightarrow a \leq 0$
 $\langle \text{proof} \rangle$

lemma *divide-less-0-1-iff* [simp]:
 $1 / a < 0 \longleftrightarrow a < 0$
 $\langle \text{proof} \rangle$

lemma *divide-right-mono*:
 $\llbracket a \leq b; 0 \leq c \rrbracket \Longrightarrow a/c \leq b/c$
 $\langle \text{proof} \rangle$

lemma *divide-right-mono-neg*: $a \leq b \Longrightarrow c \leq 0 \Longrightarrow b / c \leq a / c$
 $\langle \text{proof} \rangle$

lemma *divide-left-mono-neg*: $a \leq b \Longrightarrow c \leq 0 \Longrightarrow 0 < a * b \Longrightarrow c / a \leq c / b$
 $\langle \text{proof} \rangle$

lemma *inverse-le-iff*: $\text{inverse } a \leq \text{inverse } b \longleftrightarrow (0 < a * b \longrightarrow b \leq a) \wedge (a * b \leq 0 \longrightarrow a \leq b)$
 ⟨proof⟩

lemma *inverse-less-iff*: $\text{inverse } a < \text{inverse } b \longleftrightarrow (0 < a * b \longrightarrow b < a) \wedge (a * b \leq 0 \longrightarrow a < b)$
 ⟨proof⟩

lemma *divide-le-cancel*: $a / c \leq b / c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
 ⟨proof⟩

lemma *divide-less-cancel*: $a / c < b / c \longleftrightarrow (0 < c \longrightarrow a < b) \wedge (c < 0 \longrightarrow b < a) \wedge c \neq 0$
 ⟨proof⟩

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1*:
 $(1 \leq b / a) = ((0 < a \wedge a \leq b) \vee (a < 0 \wedge b \leq a))$
 ⟨proof⟩

lemma *divide-le-eq-1*:
 $(b / a \leq 1) = ((0 < a \wedge b \leq a) \vee (a < 0 \wedge a \leq b) \vee a=0)$
 ⟨proof⟩

lemma *less-divide-eq-1*:
 $(1 < b / a) = ((0 < a \wedge a < b) \vee (a < 0 \wedge b < a))$
 ⟨proof⟩

lemma *divide-less-eq-1*:
 $(b / a < 1) = ((0 < a \wedge b < a) \vee (a < 0 \wedge a < b) \vee a=0)$
 ⟨proof⟩

lemma *divide-nonneg-nonneg* [simp]:
 $0 \leq x \implies 0 \leq y \implies 0 \leq x / y$
 ⟨proof⟩

lemma *divide-nonpos-nonpos*:
 $x \leq 0 \implies y \leq 0 \implies 0 \leq x / y$
 ⟨proof⟩

lemma *divide-nonneg-nonpos*:
 $0 \leq x \implies y \leq 0 \implies x / y \leq 0$
 ⟨proof⟩

lemma *divide-nonpos-nonneg*:
 $x \leq 0 \implies 0 \leq y \implies x / y \leq 0$
 ⟨proof⟩

Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [simp]:
 $0 < a \implies (1 \leq b/a) = (a \leq b)$
 ⟨proof⟩

lemma *le-divide-eq-1-neg* [simp]:
 $a < 0 \implies (1 \leq b/a) = (b \leq a)$
 ⟨proof⟩

lemma *divide-le-eq-1-pos* [simp]:
 $0 < a \implies (b/a \leq 1) = (b \leq a)$
 ⟨proof⟩

lemma *divide-le-eq-1-neg* [simp]:
 $a < 0 \implies (b/a \leq 1) = (a \leq b)$
 ⟨proof⟩

lemma *less-divide-eq-1-pos* [simp]:
 $0 < a \implies (1 < b/a) = (a < b)$
 ⟨proof⟩

lemma *less-divide-eq-1-neg* [simp]:
 $a < 0 \implies (1 < b/a) = (b < a)$
 ⟨proof⟩

lemma *divide-less-eq-1-pos* [simp]:
 $0 < a \implies (b/a < 1) = (b < a)$
 ⟨proof⟩

lemma *divide-less-eq-1-neg* [simp]:
 $a < 0 \implies b/a < 1 \longleftrightarrow a < b$
 ⟨proof⟩

lemma *eq-divide-eq-1* [simp]:
 $(1 = b/a) = ((a \neq 0 \wedge a = b))$
 ⟨proof⟩

lemma *divide-eq-eq-1* [simp]:
 $(b/a = 1) = ((a \neq 0 \wedge a = b))$
 ⟨proof⟩

lemma *abs-div-pos*: $0 < y \implies |x| / y = |x / y|$
 ⟨proof⟩

lemma *zero-le-divide-abs-iff* [simp]: $(0 \leq a / |b|) = (0 \leq a \vee b = 0)$
 ⟨proof⟩

lemma *divide-le-0-abs-iff* [simp]: $(a / |b| \leq 0) = (a \leq 0 \vee b = 0)$
 ⟨proof⟩

lemma *field-le-mult-one-interval*:
assumes *: $\bigwedge z. \llbracket 0 < z ; z < 1 \rrbracket \implies z * x \leq y$
shows $x \leq y$
 $\langle \text{proof} \rangle$

For creating values between u and v .

lemma *scaling-mono*:
assumes $u \leq v \ 0 \leq r \ r \leq s$
shows $u + r * (v - u) / s \leq v$
 $\langle \text{proof} \rangle$

end

Min/max Simplification Rules

lemma *min-mult-distrib-left*:
fixes $x::'a::\text{linordered-idom}$
shows $p * \min x y = (\text{if } 0 \leq p \text{ then } \min (p*x) (p*y) \text{ else } \max (p*x) (p*y))$
 $\langle \text{proof} \rangle$

lemma *min-mult-distrib-right*:
fixes $x::'a::\text{linordered-idom}$
shows $\min x y * p = (\text{if } 0 \leq p \text{ then } \min (x*p) (y*p) \text{ else } \max (x*p) (y*p))$
 $\langle \text{proof} \rangle$

lemma *min-divide-distrib-right*:
fixes $x::'a::\text{linordered-field}$
shows $\min x y / p = (\text{if } 0 \leq p \text{ then } \min (x/p) (y/p) \text{ else } \max (x/p) (y/p))$
 $\langle \text{proof} \rangle$

lemma *max-mult-distrib-left*:
fixes $x::'a::\text{linordered-idom}$
shows $p * \max x y = (\text{if } 0 \leq p \text{ then } \max (p*x) (p*y) \text{ else } \min (p*x) (p*y))$
 $\langle \text{proof} \rangle$

lemma *max-mult-distrib-right*:
fixes $x::'a::\text{linordered-idom}$
shows $\max x y * p = (\text{if } 0 \leq p \text{ then } \max (x*p) (y*p) \text{ else } \min (x*p) (y*p))$
 $\langle \text{proof} \rangle$

lemma *max-divide-distrib-right*:
fixes $x::'a::\text{linordered-field}$
shows $\max x y / p = (\text{if } 0 \leq p \text{ then } \max (x/p) (y/p) \text{ else } \min (x/p) (y/p))$
 $\langle \text{proof} \rangle$

hide-fact (**open**) *field-inverse field-divide-inverse field-inverse-zero*

code-identifier

code-module *Fields* \rightarrow (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

19 Relations – as sets of pairs, and binary predicates

```
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*: $(\bigwedge x y. (x, y) \in r \implies (x, y) \in s) \implies r \subseteq s$
 — Version of *subsetI* for binary relations
 $\langle \text{proof} \rangle$

lemma *lfp-induct2*:
 $(a, b) \in \text{lfp } f \implies \text{mono } f \implies$
 $(\bigwedge a b. (a, b) \in f (\text{lfp } f \cap \{(x, y). P x y\}) \implies P a b) \implies P a b$
 — Version of *lfp-induct* for binary relations
 $\langle \text{proof} \rangle$

19.1.2 Conversions between set and predicate relations

lemma *pred-equals-eq* [*pred-set-conv*]: $(\lambda x. x \in R) = (\lambda x. x \in S) \longleftrightarrow R = S$
 $\langle \text{proof} \rangle$

lemma *pred-equals-eq2* [*pred-set-conv*]: $(\lambda x y. (x, y) \in R) = (\lambda x y. (x, y) \in S) \longleftrightarrow R = S$
 $\langle \text{proof} \rangle$

lemma *pred-subset-eq* [*pred-set-conv*]: $(\lambda x. x \in R) \leq (\lambda x. x \in S) \longleftrightarrow R \subseteq S$
 $\langle \text{proof} \rangle$

lemma *pred-subset-eq2* [*pred-set-conv*]: $(\lambda x y. (x, y) \in R) \leq (\lambda x y. (x, y) \in S) \longleftrightarrow R \subseteq S$
 $\langle \text{proof} \rangle$

lemma *bot-empty-eq* [*pred-set-conv*]: $\perp = (\lambda x. x \in \{\})$
 $\langle \text{proof} \rangle$

lemma *bot-empty-eq2* [*pred-set-conv*]: $\perp = (\lambda x y. (x, y) \in \{\})$
 $\langle \text{proof} \rangle$

lemma *top-empty-eq*: $\top = (\lambda x. x \in \text{UNIV})$
 $\langle \text{proof} \rangle$

lemma *top-empty-eq2*: $\top = (\lambda x y. (x, y) \in \text{UNIV})$
 $\langle \text{proof} \rangle$

lemma *inf-Int-eq* [*pred-set-conv*]: $(\lambda x. x \in R) \sqcap (\lambda x. x \in S) = (\lambda x. x \in R \cap S)$
 $\langle \text{proof} \rangle$

lemma *inf-Int-eq2* [*pred-set-conv*]: $(\lambda x y. (x, y) \in R) \sqcap (\lambda x y. (x, y) \in S) = (\lambda x y. (x, y) \in R \cap S)$
 $\langle proof \rangle$

lemma *sup-Un-eq* [*pred-set-conv*]: $(\lambda x. x \in R) \sqcup (\lambda x. x \in S) = (\lambda x. x \in R \cup S)$
 $\langle proof \rangle$

lemma *sup-Un-eq2* [*pred-set-conv*]: $(\lambda x y. (x, y) \in R) \sqcup (\lambda x y. (x, y) \in S) = (\lambda x y. (x, y) \in R \cup S)$
 $\langle proof \rangle$

lemma *INF-INT-eq* [*pred-set-conv*]: $(\bigcap_{i \in S}. (\lambda x. x \in r i)) = (\lambda x. x \in (\bigcap_{i \in S}. r i))$
 $\langle proof \rangle$

lemma *INF-INT-eq2* [*pred-set-conv*]: $(\bigcap_{i \in S}. (\lambda x y. (x, y) \in r i)) = (\lambda x y. (x, y) \in (\bigcap_{i \in S}. r i))$
 $\langle proof \rangle$

lemma *SUP-UN-eq* [*pred-set-conv*]: $(\bigcup_{i \in S}. (\lambda x. x \in r i)) = (\lambda x. x \in (\bigcup_{i \in S}. r i))$
 $\langle proof \rangle$

lemma *SUP-UN-eq2* [*pred-set-conv*]: $(\bigcup_{i \in S}. (\lambda x y. (x, y) \in r i)) = (\lambda x y. (x, y) \in (\bigcup_{i \in S}. r i))$
 $\langle proof \rangle$

lemma *Inf-INT-eq* [*pred-set-conv*]: $\bigcap S = (\lambda x. x \in (\bigcap (Collect \text{ ‘ } S)))$
 $\langle proof \rangle$

lemma *INF-Int-eq* [*pred-set-conv*]: $(\bigcap_{i \in S}. (\lambda x. x \in i)) = (\lambda x. x \in \bigcap S)$
 $\langle proof \rangle$

lemma *Inf-INT-eq2* [*pred-set-conv*]: $\bigcap S = (\lambda x y. (x, y) \in (\bigcap (Collect \text{ ‘ } case-prod \text{ ‘ } S)))$
 $\langle proof \rangle$

lemma *INF-Int-eq2* [*pred-set-conv*]: $(\bigcap_{i \in S}. (\lambda x y. (x, y) \in i)) = (\lambda x y. (x, y) \in \bigcap S)$
 $\langle proof \rangle$

lemma *Sup-SUP-eq* [*pred-set-conv*]: $\bigcup S = (\lambda x. x \in \bigcup (Collect \text{ ‘ } S))$
 $\langle proof \rangle$

lemma *SUP-Sup-eq* [*pred-set-conv*]: $(\bigcup_{i \in S}. (\lambda x. x \in i)) = (\lambda x. x \in \bigcup S)$
 $\langle proof \rangle$

lemma *Sup-SUP-eq2* [*pred-set-conv*]: $\bigcup S = (\lambda x y. (x, y) \in (\bigcup (Collect \text{ ‘ } case-prod \text{ ‘ } S)))$

$\langle proof \rangle$

lemma *SUP-Sup-eq2* [*pred-set-conv*]: $(\bigsqcup i \in S. (\lambda x y. (x, y) \in i)) = (\lambda x y. (x, y) \in \bigcup S)$
 $\langle proof \rangle$

19.2 Properties of relations

19.2.1 Reflexivity

definition *refl-on* :: 'a set \Rightarrow 'a rel \Rightarrow bool
where *refl-on* A r $\longleftrightarrow (\forall x \in A. (x, x) \in r)$

abbreviation *refl* :: 'a rel \Rightarrow bool — reflexivity over a type
where *refl* \equiv *refl-on* UNIV

definition *reflp-on* :: 'a set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool
where *reflp-on* A R $\longleftrightarrow (\forall x \in A. R x x)$

abbreviation *reflp* :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool
where *reflp* \equiv *reflp-on* UNIV

lemma *reflp-def*[*no-atp*]: *reflp* R $\longleftrightarrow (\forall x. R x x)$
 $\langle proof \rangle$

reflp-def is for backward compatibility.

lemma *reflp-on-refl-on-eq*[*pred-set-conv*]: *reflp-on* A ($\lambda a b. (a, b) \in r$) \longleftrightarrow *refl-on* A r
 $\langle proof \rangle$

lemmas *reflp-refl-eq* = *reflp-on-refl-on-eq*[*of UNIV*]

lemma *refl-onI* [*intro?*]: $(\bigwedge x. x \in A \Longrightarrow (x, x) \in r) \Longrightarrow$ *refl-on* A r
 $\langle proof \rangle$

lemma *reflI*: $(\bigwedge x. (x, x) \in r) \Longrightarrow$ *refl* r
 $\langle proof \rangle$

lemma *reflp-onI*:
 $(\bigwedge x. x \in A \Longrightarrow R x x) \Longrightarrow$ *reflp-on* A R
 $\langle proof \rangle$

lemma *reflpI*[*intro?*]: $(\bigwedge x. R x x) \Longrightarrow$ *reflp* R
 $\langle proof \rangle$

lemma *refl-onD*: *refl-on* A r $\Longrightarrow a \in A \Longrightarrow (a, a) \in r$
 $\langle proof \rangle$

lemma *reflD*: *refl* r $\Longrightarrow (a, a) \in r$
 $\langle proof \rangle$

lemma *reflp-onD*:

reflp-on A $R \implies x \in A \implies R\ x\ x$
 $\langle proof \rangle$

lemma *reflpD[dest?]*: *reflp* $R \implies R\ x\ x$

$\langle proof \rangle$

lemma *reflpE*:

assumes *reflp* r

obtains $r\ x\ x$

$\langle proof \rangle$

lemma *refl-on-top[simp]*: *refl-on* $A\ \top$

$\langle proof \rangle$

lemma *reflp-on-top[simp]*: *reflp-on* $A\ \top$

$\langle proof \rangle$

lemma *reflp-on-mono-strong*:

reflp-on $B\ R \implies A \subseteq B \implies (\bigwedge x\ y. x \in A \implies y \in A \implies R\ x\ y \implies Q\ x\ y) \implies$
reflp-on $A\ Q$
 $\langle proof \rangle$

lemma *reflp-on-mono[mono]*: $A \subseteq B \implies R \leq Q \implies \text{reflp-on } B\ R \leq \text{reflp-on } A\ Q$

$\langle proof \rangle$

lemma *reflp-on-subset*: *reflp-on* $B\ R \implies A \subseteq B \implies \text{reflp-on } A\ R$

$\langle proof \rangle$

lemma *reflp-on-image*: *reflp-on* $(f\ 'A)\ R \longleftrightarrow \text{reflp-on } A\ (\lambda a\ b. R\ (f\ a)\ (f\ b))$

$\langle proof \rangle$

lemma *refl-on-Int*: *refl-on* $A\ r \implies \text{refl-on } B\ s \implies \text{refl-on } (A \cap B)\ (r \cap s)$

$\langle proof \rangle$

lemma *reflp-on-inf*: *reflp-on* $A\ R \implies \text{reflp-on } B\ S \implies \text{reflp-on } (A \cap B)\ (R \sqcap S)$

$\langle proof \rangle$

lemma *reflp-inf*: *reflp* $r \implies \text{reflp } s \implies \text{reflp } (r \sqcap s)$

$\langle proof \rangle$

lemma *refl-on-Un*: *refl-on* $A\ r \implies \text{refl-on } B\ s \implies \text{refl-on } (A \cup B)\ (r \cup s)$

$\langle proof \rangle$

lemma *reflp-on-sup*: *reflp-on* $A\ R \implies \text{reflp-on } B\ S \implies \text{reflp-on } (A \cup B)\ (R \sqcup S)$

$\langle proof \rangle$

lemma *reflp-sup*: $\text{reflp } r \implies \text{reflp } s \implies \text{reflp } (r \sqcup s)$
 $\langle \text{proof} \rangle$

lemma *refl-on-INTER*: $\forall x \in S. \text{refl-on } (A \ x) \ (r \ x) \implies \text{refl-on } (\bigcap (A \ ' S)) \ (\bigcap (r \ ' S))$
 $\langle \text{proof} \rangle$

lemma *reflp-on-Inf*: $\forall x \in S. \text{reflp-on } (A \ x) \ (R \ x) \implies \text{reflp-on } (\bigcap (A \ ' S)) \ (\bigcap (R \ ' S))$
 $\langle \text{proof} \rangle$

lemma *refl-on-UNION*: $\forall x \in S. \text{refl-on } (A \ x) \ (r \ x) \implies \text{refl-on } (\bigcup (A \ ' S)) \ (\bigcup (r \ ' S))$
 $\langle \text{proof} \rangle$

lemma *reflp-on-Sup*: $\forall x \in S. \text{reflp-on } (A \ x) \ (R \ x) \implies \text{reflp-on } (\bigcup (A \ ' S)) \ (\bigcup (R \ ' S))$
 $\langle \text{proof} \rangle$

lemma *refl-on-empty* [simp]: $\text{refl-on } \{\} \ r$
 $\langle \text{proof} \rangle$

lemma *reflp-on-empty* [simp]: $\text{reflp-on } \{\} \ R$
 $\langle \text{proof} \rangle$

lemma *refl-on-singleton* [simp]: $\text{refl-on } \{x\} \ \{(x, x)\}$
 $\langle \text{proof} \rangle$

lemma *reflp-on-equality* [simp]: $\text{reflp-on } A \ (=)$
 $\langle \text{proof} \rangle$

lemma (in *preorder*) *reflp-on-le*[simp]: $\text{reflp-on } A \ (\leq)$
 $\langle \text{proof} \rangle$

lemma (in *preorder*) *reflp-on-ge*[simp]: $\text{reflp-on } A \ (\geq)$
 $\langle \text{proof} \rangle$

19.2.2 Irreflexivity

definition *irrefl-on* :: $'a \text{ set} \Rightarrow 'a \text{ rel} \Rightarrow \text{bool}$ **where**
 $\text{irrefl-on } A \ r \longleftrightarrow (\forall a \in A. (a, a) \notin r)$

abbreviation *irrefl* :: $'a \text{ rel} \Rightarrow \text{bool}$ **where**
 $\text{irrefl} \equiv \text{irrefl-on } \text{UNIV}$

definition *irreflp-on* :: $'a \text{ set} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{irreflp-on } A \ R \longleftrightarrow (\forall a \in A. \neg R \ a \ a)$

abbreviation *irreflp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**

$$\text{irreflp} \equiv \text{irreflp-on UNIV}$$

lemma *irrefl-def[no-atp]*: $\text{irrefl } r \longleftrightarrow (\forall a. (a, a) \notin r)$
 $\langle \text{proof} \rangle$

lemma *irreflp-def[no-atp]*: $\text{irreflp } R \longleftrightarrow (\forall a. \neg R a a)$
 $\langle \text{proof} \rangle$

irrefl-def and *irreflp-def* are for backward compatibility.

lemma *irreflp-on-irrefl-on-eq [pred-set-conv]*: $\text{irreflp-on } A (\lambda a b. (a, b) \in r) \longleftrightarrow \text{irrefl-on } A r$
 $\langle \text{proof} \rangle$

lemmas *irreflp-irrefl-eq* = *irreflp-on-irrefl-on-eq*[of UNIV]

lemma *irrefl-onI*: $(\bigwedge a. a \in A \implies (a, a) \notin r) \implies \text{irrefl-on } A r$
 $\langle \text{proof} \rangle$

lemma *irreflI[intro?]*: $(\bigwedge a. (a, a) \notin r) \implies \text{irrefl } r$
 $\langle \text{proof} \rangle$

lemma *irreflp-onI*: $(\bigwedge a. a \in A \implies \neg R a a) \implies \text{irreflp-on } A R$
 $\langle \text{proof} \rangle$

lemma *irreflpI[intro?]*: $(\bigwedge a. \neg R a a) \implies \text{irreflp } R$
 $\langle \text{proof} \rangle$

lemma *irrefl-onD*: $\text{irrefl-on } A r \implies a \in A \implies (a, a) \notin r$
 $\langle \text{proof} \rangle$

lemma *irreflD*: $\text{irrefl } r \implies (x, x) \notin r$
 $\langle \text{proof} \rangle$

lemma *irreflp-onD*: $\text{irreflp-on } A R \implies a \in A \implies \neg R a a$
 $\langle \text{proof} \rangle$

lemma *irreflpD*: $\text{irreflp } R \implies \neg R x x$
 $\langle \text{proof} \rangle$

lemma *irrefl-on-bot[simp]*: $\text{irrefl-on } A \perp$
 $\langle \text{proof} \rangle$

lemma *irreflp-on-bot[simp]*: $\text{irreflp-on } A \perp$
 $\langle \text{proof} \rangle$

lemma *irrefl-on-distinct [code]*: $\text{irrefl-on } A r \longleftrightarrow (\forall (a, b) \in r. a \in A \longrightarrow b \in A \longrightarrow a \neq b)$
 $\langle \text{proof} \rangle$

lemmas *irrefl-distinct* = *irrefl-on-distinct* — For backward compatibility

lemma *irreflp-on-mono-strong*:

$\text{irreflp-on } B \ Q \implies A \subseteq B \implies (\bigwedge x \ y. x \in A \implies y \in A \implies R \ x \ y \implies Q \ x \ y) \implies \text{irreflp-on } A \ R$
 $\langle \text{proof} \rangle$

lemma *irreflp-on-mono[mono]*: $A \subseteq B \implies R \leq Q \implies \text{irreflp-on } B \ Q \leq \text{irreflp-on } A \ R$

$\langle \text{proof} \rangle$

lemma *irrefl-on-subset*: $\text{irrefl-on } B \ r \implies A \subseteq B \implies \text{irrefl-on } A \ r$

$\langle \text{proof} \rangle$

lemma *irreflp-on-subset*: $\text{irreflp-on } B \ R \implies A \subseteq B \implies \text{irreflp-on } A \ R$

$\langle \text{proof} \rangle$

lemma *irreflp-on-image*: $\text{irreflp-on } (f \text{ ‘ } A) \ R \longleftrightarrow \text{irreflp-on } A \ (\lambda a \ b. R \ (f \ a) \ (f \ b))$

$\langle \text{proof} \rangle$

lemma (*in preorder*) *irreflp-on-less[simp]*: $\text{irreflp-on } A \ (<)$

$\langle \text{proof} \rangle$

lemma (*in preorder*) *irreflp-on-greater[simp]*: $\text{irreflp-on } A \ (>)$

$\langle \text{proof} \rangle$

19.2.3 Asymmetry

definition *asym-on* :: ‘a set \Rightarrow ‘a rel \Rightarrow bool **where**

$\text{asym-on } A \ r \longleftrightarrow (\forall x \in A. \forall y \in A. (x, y) \in r \longrightarrow (y, x) \notin r)$

abbreviation *asym* :: ‘a rel \Rightarrow bool **where**

$\text{asym} \equiv \text{asym-on } \text{UNIV}$

definition *asym-on* :: ‘a set \Rightarrow (‘a \Rightarrow ‘a \Rightarrow bool) \Rightarrow bool **where**

$\text{asym-on } A \ R \longleftrightarrow (\forall x \in A. \forall y \in A. R \ x \ y \longrightarrow \neg R \ y \ x)$

abbreviation *asym* :: (‘a \Rightarrow ‘a \Rightarrow bool) \Rightarrow bool **where**

$\text{asym} \equiv \text{asym-on } \text{UNIV}$

lemma *asym-on-asym-on-eq[pred-set-conv]*: $\text{asym-on } A \ (\lambda x \ y. (x, y) \in r) \longleftrightarrow \text{asym-on } A \ r$

$\langle \text{proof} \rangle$

lemmas *asym-asym-eq* = *asym-on-asym-on-eq[of UNIV]* — For backward compatibility

lemma *asym-onI[intro]*:

$(\bigwedge x \ y. x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \notin r) \implies \text{asym-on } A \ r$

$\langle proof \rangle$

lemma *asymI[intro]*: $(\bigwedge x y. (x, y) \in r \implies (y, x) \notin r) \implies asym\ r$
 $\langle proof \rangle$

lemma *asym-onI[intro]*:
 $(\bigwedge x y. x \in A \implies y \in A \implies R\ x\ y \implies \neg R\ y\ x) \implies asym-on\ A\ R$
 $\langle proof \rangle$

lemma *asymplI[intro]*: $(\bigwedge x y. R\ x\ y \implies \neg R\ y\ x) \implies asympl\ R$
 $\langle proof \rangle$

lemma *asym-onD*: $asym-on\ A\ r \implies x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \notin r$
 $\langle proof \rangle$

lemma *asymD*: $asym\ r \implies (x, y) \in r \implies (y, x) \notin r$
 $\langle proof \rangle$

lemma *asympl-onD*: $asympl-on\ A\ R \implies x \in A \implies y \in A \implies R\ x\ y \implies \neg R\ y\ x$
 $\langle proof \rangle$

lemma *asymplD*: $asympl\ R \implies R\ x\ y \implies \neg R\ y\ x$
 $\langle proof \rangle$

lemma *asym-on-bot[simp]*: $asym-on\ A\ \perp$
 $\langle proof \rangle$

lemma *asympl-on-bot[simp]*: $asympl-on\ A\ \perp$
 $\langle proof \rangle$

lemma *asym-iff*: $asym\ r \longleftrightarrow (\forall x y. (x, y) \in r \longrightarrow (y, x) \notin r)$
 $\langle proof \rangle$

lemma *asympl-on-mono-strong*:
 $asympl-on\ B\ Q \implies A \subseteq B \implies (\bigwedge x y. x \in A \implies y \in A \implies R\ x\ y \implies Q\ x\ y)$
 $\implies asympl-on\ A\ R$
 $\langle proof \rangle$

lemma *asympl-on-mono[mono]*: $A \subseteq B \implies R \leq Q \implies asympl-on\ B\ Q \leq asympl-on\ A\ R$
 $\langle proof \rangle$

lemma *asym-on-subset*: $asym-on\ B\ r \implies A \subseteq B \implies asym-on\ A\ r$
 $\langle proof \rangle$

lemma *asympl-on-subset*: $asympl-on\ B\ R \implies A \subseteq B \implies asympl-on\ A\ R$
 $\langle proof \rangle$

lemma *asym-on-image*: $\text{asym-on } (f \text{ ‘ } A) R \longleftrightarrow \text{asym-on } A (\lambda a b. R (f a) (f b))$
 ⟨proof⟩

lemma *irrefl-on-if-asym-on[simp]*: $\text{asym-on } A r \implies \text{irrefl-on } A r$
 ⟨proof⟩

lemma *irreflp-on-if-asym-on[simp]*: $\text{asym-on } A r \implies \text{irreflp-on } A r$
 ⟨proof⟩

lemma (in *preorder*) *asym-on-less[simp]*: $\text{asym-on } A (<)$
 ⟨proof⟩

lemma (in *preorder*) *asym-on-greater[simp]*: $\text{asym-on } A (>)$
 ⟨proof⟩

19.2.4 Symmetry

definition *sym-on* :: $'a \text{ set} \Rightarrow 'a \text{ rel} \Rightarrow \text{bool}$ **where**
 $\text{sym-on } A r \longleftrightarrow (\forall x \in A. \forall y \in A. (x, y) \in r \longrightarrow (y, x) \in r)$

abbreviation *sym* :: $'a \text{ rel} \Rightarrow \text{bool}$ **where**
 $\text{sym} \equiv \text{sym-on } \text{UNIV}$

definition *symp-on* :: $'a \text{ set} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{symp-on } A R \longleftrightarrow (\forall x \in A. \forall y \in A. R x y \longrightarrow R y x)$

abbreviation *symp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{symp} \equiv \text{symp-on } \text{UNIV}$

lemma *sym-def[no-atp]*: $\text{sym } r \longleftrightarrow (\forall x y. (x, y) \in r \longrightarrow (y, x) \in r)$
 ⟨proof⟩

lemma *symp-def[no-atp]*: $\text{symp } R \longleftrightarrow (\forall x y. R x y \longrightarrow R y x)$
 ⟨proof⟩

sym-def and *symp-def* are for backward compatibility.

lemma *symp-on-sym-on-eq[pred-set-conv]*: $\text{symp-on } A (\lambda x y. (x, y) \in r) \longleftrightarrow \text{sym-on } A r$
 ⟨proof⟩

lemmas *symp-sym-eq* = *symp-on-sym-on-eq[of UNIV]* — For backward compatibility

lemma *sym-onI*: $(\bigwedge x y. x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \in r) \implies \text{sym-on } A r$
 ⟨proof⟩

lemma *symI* [intro?]: $(\bigwedge x y. (x, y) \in r \implies (y, x) \in r) \implies \text{sym } r$

$\langle proof \rangle$

lemma *symp-onI*: $(\bigwedge x y. x \in A \implies y \in A \implies R x y \implies R y x) \implies \text{symp-on } A$
 R
 $\langle proof \rangle$

lemma *sympI* [intro?]: $(\bigwedge x y. R x y \implies R y x) \implies \text{symp } R$
 $\langle proof \rangle$

lemma *symE*:
 assumes *sym r* and $(b, a) \in r$
 obtains $(a, b) \in r$
 $\langle proof \rangle$

lemma *sympE*:
 assumes *symp r* and $r b a$
 obtains $r a b$
 $\langle proof \rangle$

lemma *sym-onD*: $\text{sym-on } A r \implies x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \in r$
 $\langle proof \rangle$

lemma *symD* [dest?]: $\text{sym } r \implies (x, y) \in r \implies (y, x) \in r$
 $\langle proof \rangle$

lemma *symp-onD*: $\text{symp-on } A R \implies x \in A \implies y \in A \implies R x y \implies R y x$
 $\langle proof \rangle$

lemma *sympD* [dest?]: $\text{symp } R \implies R x y \implies R y x$
 $\langle proof \rangle$

lemma *sym-on-bot[simp]*: $\text{sym-on } A \perp$
 $\langle proof \rangle$

lemma *symp-on-bot[simp]*: $\text{symp-on } A \perp$
 $\langle proof \rangle$

lemma *sym-on-top[simp]*: $\text{sym-on } A \top$
 $\langle proof \rangle$

lemma *symp-on-top[simp]*: $\text{symp-on } A \top$
 $\langle proof \rangle$

lemma *sym-on-subset*: $\text{sym-on } B r \implies A \subseteq B \implies \text{sym-on } A r$
 $\langle proof \rangle$

lemma *symp-on-subset*: $\text{symp-on } B R \implies A \subseteq B \implies \text{symp-on } A R$
 $\langle proof \rangle$

lemma *symp-on-image*: $\text{symp-on } (f \text{ ‘ } A) R \longleftrightarrow \text{symp-on } A (\lambda a b. R (f a) (f b))$
 $\langle \text{proof} \rangle$

lemma *symp-on-equality[simp]*: $\text{symp-on } A (=)$
 $\langle \text{proof} \rangle$

lemma *sym-Int*: $\text{sym } r \implies \text{sym } s \implies \text{sym } (r \cap s)$
 $\langle \text{proof} \rangle$

lemma *symp-inf*: $\text{symp } r \implies \text{symp } s \implies \text{symp } (r \sqcap s)$
 $\langle \text{proof} \rangle$

lemma *sym-Un*: $\text{sym } r \implies \text{sym } s \implies \text{sym } (r \cup s)$
 $\langle \text{proof} \rangle$

lemma *symp-sup*: $\text{symp } r \implies \text{symp } s \implies \text{symp } (r \sqcup s)$
 $\langle \text{proof} \rangle$

lemma *sym-INTER*: $\forall x \in S. \text{sym } (r x) \implies \text{sym } (\bigcap (r \text{ ‘ } S))$
 $\langle \text{proof} \rangle$

lemma *symp-INF*: $\forall x \in S. \text{symp } (r x) \implies \text{symp } (\bigcap (r \text{ ‘ } S))$
 $\langle \text{proof} \rangle$

lemma *sym-UNION*: $\forall x \in S. \text{sym } (r x) \implies \text{sym } (\bigcup (r \text{ ‘ } S))$
 $\langle \text{proof} \rangle$

lemma *symp-SUP*: $\forall x \in S. \text{symp } (r x) \implies \text{symp } (\bigcup (r \text{ ‘ } S))$
 $\langle \text{proof} \rangle$

19.2.5 Antisymmetry

definition *antisym-on* :: $'a \text{ set} \Rightarrow 'a \text{ rel} \Rightarrow \text{bool}$ **where**
 $\text{antisym-on } A r \longleftrightarrow (\forall x \in A. \forall y \in A. (x, y) \in r \longrightarrow (y, x) \in r \longrightarrow x = y)$

abbreviation *antisym* :: $'a \text{ rel} \Rightarrow \text{bool}$ **where**
 $\text{antisym} \equiv \text{antisym-on UNIV}$

definition *antisymp-on* :: $'a \text{ set} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{antisymp-on } A R \longleftrightarrow (\forall x \in A. \forall y \in A. R x y \longrightarrow R y x \longrightarrow x = y)$

abbreviation *antisymp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{antisymp} \equiv \text{antisymp-on UNIV}$

lemma *antisym-def[no-atp]*: $\text{antisym } r \longleftrightarrow (\forall x y. (x, y) \in r \longrightarrow (y, x) \in r \longrightarrow x = y)$
 $\langle \text{proof} \rangle$

lemma *antisymp-def[no-atp]*: $\text{antisymp } R \longleftrightarrow (\forall x y. R x y \longrightarrow R y x \longrightarrow x = y)$

$y)$
 $\langle proof \rangle$

antisym-def and *antisym-def* are for backward compatibility.

lemma *antisym-on-antisym-on-eq[pred-set-conv]*:
 $antisym-on\ A\ (\lambda x\ y. (x, y) \in r) \longleftrightarrow antisym-on\ A\ r$
 $\langle proof \rangle$

lemmas *antisym-antisym-eq = antisym-on-antisym-on-eq[of UNIV]* — For backward compatibility

lemma *antisym-onI*:
 $(\bigwedge x\ y. x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \in r \implies x = y) \implies antisym-on\ A\ r$
 $\langle proof \rangle$

lemma *antisymI [intro?]*:
 $(\bigwedge x\ y. (x, y) \in r \implies (y, x) \in r \implies x = y) \implies antisym\ r$
 $\langle proof \rangle$

lemma *antisym-onI*:
 $(\bigwedge x\ y. x \in A \implies y \in A \implies R\ x\ y \implies R\ y\ x \implies x = y) \implies antisym-on\ A\ R$
 $\langle proof \rangle$

lemma *antisymI [intro?]*:
 $(\bigwedge x\ y. R\ x\ y \implies R\ y\ x \implies x = y) \implies antisym\ R$
 $\langle proof \rangle$

lemma *antisym-onD*:
 $antisym-on\ A\ r \implies x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \in r \implies x = y$
 $\langle proof \rangle$

lemma *antisymD [dest?]*:
 $antisym\ r \implies (x, y) \in r \implies (y, x) \in r \implies x = y$
 $\langle proof \rangle$

lemma *antisym-onD*:
 $antisym-on\ A\ R \implies x \in A \implies y \in A \implies R\ x\ y \implies R\ y\ x \implies x = y$
 $\langle proof \rangle$

lemma *antisymD [dest?]*:
 $antisym\ R \implies R\ x\ y \implies R\ y\ x \implies x = y$
 $\langle proof \rangle$

lemma *antisym-on-bot[simp]*: $antisym-on\ A\ \perp$
 $\langle proof \rangle$

lemma *antisym-on-bot[simp]*: $antisym-on\ A\ \perp$
 $\langle proof \rangle$

lemma *antisym-on-mono-stronger*:

fixes

$A :: 'a \text{ set}$ **and** $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ **and**

$B :: 'b \text{ set}$ **and** $Q :: 'b \Rightarrow 'b \Rightarrow \text{bool}$ **and**

$f :: 'a \Rightarrow 'b$

assumes *antisym-on B Q* **and** $f \text{ ' } A \subseteq B$ **and**

Q-imp-R: $\bigwedge x y. x \in A \Longrightarrow y \in A \Longrightarrow R x y \Longrightarrow Q (f x) (f y)$ **and**

inj-f: *inj-on f A*

shows *antisym-on A R*

<proof>

lemma *antisym-on-mono-strong*:

antisym-on B Q $\Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x y. x \in A \Longrightarrow y \in A \Longrightarrow R x y \Longrightarrow Q x y) \Longrightarrow \text{antisym-on } A R$

<proof>

lemma *antisym-on-mono[mono]*: $A \subseteq B \Longrightarrow R \leq Q \Longrightarrow \text{antisym-on } B Q \leq \text{antisym-on } A R$

<proof>

lemma *antisym-on-subset*: *antisym-on B r* $\Longrightarrow A \subseteq B \Longrightarrow \text{antisym-on } A r$

<proof>

lemma *antisym-on-subset*: *antisym-on B R* $\Longrightarrow A \subseteq B \Longrightarrow \text{antisym-on } A R$

<proof>

lemma *antisym-on-image*:

assumes *inj-on f A*

shows *antisym-on (f ' A) R* $\longleftrightarrow \text{antisym-on } A (\lambda a b. R (f a) (f b))$

<proof>

lemma *antisym-subset*:

$r \subseteq s \Longrightarrow \text{antisym } s \Longrightarrow \text{antisym } r$

<proof>

lemma *antisym-less-eq*:

$r \leq s \Longrightarrow \text{antisym } s \Longrightarrow \text{antisym } r$

<proof>

lemma *antisym-on-equality[simp]*: *antisym-on A (=)*

<proof>

lemma *antisym-singleton [simp]*:

antisym {x}

<proof>

lemma *antisym-on-if-asym-on*: *asym-on A r* $\Longrightarrow \text{antisym-on } A r$

<proof>

lemma *antisymp-on-if-asymp-on*: $\text{asymp-on } A \ R \implies \text{antisymp-on } A \ R$
 $\langle \text{proof} \rangle$

lemma (*in preorder*) *antisymp-on-less[simp]*: $\text{antisymp-on } A \ (<)$
 $\langle \text{proof} \rangle$

lemma (*in preorder*) *antisymp-on-greater[simp]*: $\text{antisymp-on } A \ (>)$
 $\langle \text{proof} \rangle$

lemma (*in order*) *antisymp-on-le[simp]*: $\text{antisymp-on } A \ (\leq)$
 $\langle \text{proof} \rangle$

lemma (*in order*) *antisymp-on-ge[simp]*: $\text{antisymp-on } A \ (\geq)$
 $\langle \text{proof} \rangle$

19.2.6 Transitivity

definition *trans-on* :: $'a \ \text{set} \Rightarrow 'a \ \text{rel} \Rightarrow \text{bool}$ **where**
 $\text{trans-on } A \ r \longleftrightarrow (\forall x \in A. \forall y \in A. \forall z \in A. (x, y) \in r \longrightarrow (y, z) \in r \longrightarrow (x, z) \in r)$

abbreviation *trans* :: $'a \ \text{rel} \Rightarrow \text{bool}$ **where**
 $\text{trans} \equiv \text{trans-on } \text{UNIV}$

definition *transp-on* :: $'a \ \text{set} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{transp-on } A \ R \longleftrightarrow (\forall x \in A. \forall y \in A. \forall z \in A. R \ x \ y \longrightarrow R \ y \ z \longrightarrow R \ x \ z)$

abbreviation *transp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{transp} \equiv \text{transp-on } \text{UNIV}$

lemma *trans-def[no-atp]*: $\text{trans } r \longleftrightarrow (\forall x \ y \ z. (x, y) \in r \longrightarrow (y, z) \in r \longrightarrow (x, z) \in r)$
 $\langle \text{proof} \rangle$

lemma *transp-def*: $\text{transp } R \longleftrightarrow (\forall x \ y \ z. R \ x \ y \longrightarrow R \ y \ z \longrightarrow R \ x \ z)$
 $\langle \text{proof} \rangle$

trans-def and *transp-def* are for backward compatibility.

lemma *transp-on-trans-on-eq[pred-set-conv]*: $\text{transp-on } A \ (\lambda x \ y. (x, y) \in r) \longleftrightarrow \text{trans-on } A \ r$
 $\langle \text{proof} \rangle$

lemmas *transp-trans-eq* = *transp-on-trans-on-eq[of UNIV]* — For backward compatibility

lemma *trans-onI*:
 $(\bigwedge x \ y \ z. x \in A \implies y \in A \implies z \in A \implies (x, y) \in r \implies (y, z) \in r \implies (x, z) \in r) \implies$

trans-on A r
 $\langle \text{proof} \rangle$

lemma *transI* [*intro?*]: $(\bigwedge x y z. (x, y) \in r \implies (y, z) \in r \implies (x, z) \in r) \implies$
trans r
 $\langle \text{proof} \rangle$

lemma *transp-onI*:
 $(\bigwedge x y z. x \in A \implies y \in A \implies z \in A \implies R x y \implies R y z \implies R x z) \implies$
transp-on A R
 $\langle \text{proof} \rangle$

lemma *transpI* [*intro?*]: $(\bigwedge x y z. R x y \implies R y z \implies R x z) \implies \text{transp } R$
 $\langle \text{proof} \rangle$

lemma *transE*:
assumes *trans r* **and** $(x, y) \in r$ **and** $(y, z) \in r$
obtains $(x, z) \in r$
 $\langle \text{proof} \rangle$

lemma *transpE*:
assumes *transp r* **and** $r x y$ **and** $r y z$
obtains $r x z$
 $\langle \text{proof} \rangle$

lemma *trans-onD*:
 $\text{trans-on } A r \implies x \in A \implies y \in A \implies z \in A \implies (x, y) \in r \implies (y, z) \in r \implies$
 $(x, z) \in r$
 $\langle \text{proof} \rangle$

lemma *transD*[*dest?*]: $\text{trans } r \implies (x, y) \in r \implies (y, z) \in r \implies (x, z) \in r$
 $\langle \text{proof} \rangle$

lemma *transp-onD*: $\text{transp-on } A R \implies x \in A \implies y \in A \implies z \in A \implies R x y$
 $\implies R y z \implies R x z$
 $\langle \text{proof} \rangle$

lemma *transpD*[*dest?*]: $\text{transp } R \implies R x y \implies R y z \implies R x z$
 $\langle \text{proof} \rangle$

lemma *trans-on-subset*: $\text{trans-on } B r \implies A \subseteq B \implies \text{trans-on } A r$
 $\langle \text{proof} \rangle$

lemma *transp-on-subset*: $\text{transp-on } B R \implies A \subseteq B \implies \text{transp-on } A R$
 $\langle \text{proof} \rangle$

lemma *transp-on-image*: $\text{transp-on } (f \text{ ‘ } A) R \longleftrightarrow \text{transp-on } A (\lambda a b. R (f a) (f b))$
 $\langle \text{proof} \rangle$

lemma *trans-Int*: $\text{trans } r \implies \text{trans } s \implies \text{trans } (r \cap s)$
 $\langle \text{proof} \rangle$

lemma *transp-inf*: $\text{transp } r \implies \text{transp } s \implies \text{transp } (r \sqcap s)$
 $\langle \text{proof} \rangle$

lemma *trans-INTER*: $\forall x \in S. \text{trans } (r \ x) \implies \text{trans } (\bigcap (r \text{ ‘ } S))$
 $\langle \text{proof} \rangle$

lemma *transp-INF*: $\forall x \in S. \text{transp } (r \ x) \implies \text{transp } (\bigcap (r \text{ ‘ } S))$
 $\langle \text{proof} \rangle$

lemma *trans-on-join* [code]:
 $\text{trans-on } A \ r \longleftrightarrow (\forall (x, y1) \in r. x \in A \longrightarrow y1 \in A \longrightarrow$
 $(\forall (y2, z) \in r. y1 = y2 \longrightarrow z \in A \longrightarrow (x, z) \in r))$
 $\langle \text{proof} \rangle$

lemma *trans-join*: $\text{trans } r \longleftrightarrow (\forall (x, y1) \in r. \forall (y2, z) \in r. y1 = y2 \longrightarrow (x, z) \in r)$
 $\langle \text{proof} \rangle$

lemma *transp-trans*: $\text{transp } r \longleftrightarrow \text{trans } \{(x, y). r \ x \ y\}$
 $\langle \text{proof} \rangle$

lemma *transp-on-equality*[simp]: $\text{transp-on } A \ (=)$
 $\langle \text{proof} \rangle$

lemma *trans-on-bot*[simp]: $\text{trans-on } A \ \perp$
 $\langle \text{proof} \rangle$

lemma *transp-on-bot*[simp]: $\text{transp-on } A \ \perp$
 $\langle \text{proof} \rangle$

lemma *trans-on-top*[simp]: $\text{trans-on } A \ \top$
 $\langle \text{proof} \rangle$

lemma *transp-on-top*[simp]: $\text{transp-on } A \ \top$
 $\langle \text{proof} \rangle$

lemma *transp-empty* [simp]: $\text{transp } (\lambda x \ y. \text{False})$
 $\langle \text{proof} \rangle$

lemma *trans-singleton* [simp]: $\text{trans } \{(a, a)\}$
 $\langle \text{proof} \rangle$

lemma *transp-singleton* [simp]: $\text{transp } (\lambda x \ y. x = a \wedge y = a)$
 $\langle \text{proof} \rangle$

lemma *asym-on-iff-irrefl-on-if-trans-on*: $\text{trans-on } A \ r \implies \text{asym-on } A \ r \longleftrightarrow \text{ir-refl-on } A \ r$

<proof>

lemma *asym-on-iff-irrefl-on-if-trans-on*: $\text{trans-on } A \ R \implies \text{asym-on } A \ R \longleftrightarrow \text{irrefl-on } A \ R$

<proof>

lemma (*in preorder*) *transp-on-le[simp]*: $\text{transp-on } A \ (\leq)$

<proof>

lemma (*in preorder*) *transp-on-less[simp]*: $\text{transp-on } A \ (<)$

<proof>

lemma (*in preorder*) *transp-on-ge[simp]*: $\text{transp-on } A \ (\geq)$

<proof>

lemma (*in preorder*) *transp-on-greater[simp]*: $\text{transp-on } A \ (>)$

<proof>

19.2.7 Totality

definition *total-on* :: $'a \text{ set} \Rightarrow 'a \text{ rel} \Rightarrow \text{bool}$ **where**

$\text{total-on } A \ r \longleftrightarrow (\forall x \in A. \forall y \in A. x \neq y \longrightarrow (x, y) \in r \vee (y, x) \in r)$

abbreviation *total* :: $'a \text{ rel} \Rightarrow \text{bool}$ **where**

$\text{total} \equiv \text{total-on } \text{UNIV}$

definition *totalp-on* :: $'a \text{ set} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**

$\text{totalp-on } A \ R \longleftrightarrow (\forall x \in A. \forall y \in A. x \neq y \longrightarrow R \ x \ y \vee R \ y \ x)$

abbreviation *totalp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**

$\text{totalp} \equiv \text{totalp-on } \text{UNIV}$

lemma *totalp-on-total-on-eq[pred-set-conv]*: $\text{totalp-on } A \ (\lambda x \ y. (x, y) \in r) \longleftrightarrow \text{total-on } A \ r$

<proof>

lemma *total-onI* [*intro?*]:

$(\bigwedge x \ y. x \in A \implies y \in A \implies x \neq y \implies (x, y) \in r \vee (y, x) \in r) \implies \text{total-on } A \ r$

<proof>

lemma *totalI*: $(\bigwedge x \ y. x \neq y \implies (x, y) \in r \vee (y, x) \in r) \implies \text{total } r$

<proof>

lemma *totalp-onI*: $(\bigwedge x \ y. x \in A \implies y \in A \implies x \neq y \implies R \ x \ y \vee R \ y \ x) \implies \text{totalp-on } A \ R$

<proof>

lemma *totalpI*: $(\bigwedge x y. x \neq y \implies R x y \vee R y x) \implies \text{totalp } R$
 $\langle \text{proof} \rangle$

lemma *totalp-onD*:
 $\text{totalp-on } A R \implies x \in A \implies y \in A \implies x \neq y \implies R x y \vee R y x$
 $\langle \text{proof} \rangle$

lemma *totalpD*: $\text{totalp } R \implies x \neq y \implies R x y \vee R y x$
 $\langle \text{proof} \rangle$

lemma *total-on-top[simp]*: $\text{total-on } A \top$
 $\langle \text{proof} \rangle$

lemma *totalp-on-top[simp]*: $\text{totalp-on } A \top$
 $\langle \text{proof} \rangle$

lemma *totalp-on-mono-stronger*:
fixes
 $A :: 'a \text{ set}$ **and** $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ **and**
 $B :: 'b \text{ set}$ **and** $Q :: 'b \Rightarrow 'b \Rightarrow \text{bool}$ **and**
 $f :: 'a \Rightarrow 'b$
assumes $\text{totalp-on } B Q$ **and** $f ' A \subseteq B$ **and**
 $Q\text{-imp-}R: \bigwedge x y. x \in A \implies y \in A \implies Q (f x) (f y) \implies R x y$ **and**
 $\text{inj-}f: \text{inj-on } f A$
shows $\text{totalp-on } A R$
 $\langle \text{proof} \rangle$

lemma *totalp-on-mono-stronger-alt*:
fixes
 $A :: 'a \text{ set}$ **and** $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ **and**
 $B :: 'b \text{ set}$ **and** $Q :: 'b \Rightarrow 'b \Rightarrow \text{bool}$ **and**
 $f :: 'b \Rightarrow 'a$
assumes $\text{totalp-on } B Q$ **and** $A \subseteq f ' B$ **and**
 $Q\text{-imp-}R: \bigwedge x y. x \in B \implies y \in B \implies Q x y \implies R (f x) (f y)$
shows $\text{totalp-on } A R$
 $\langle \text{proof} \rangle$

lemma *totalp-on-mono-strong*:
 $\text{totalp-on } B Q \implies A \subseteq B \implies (\bigwedge x y. x \in A \implies y \in A \implies Q x y \implies R x y)$
 $\implies \text{totalp-on } A R$
 $\langle \text{proof} \rangle$

lemma *totalp-on-mono[mono]*: $A \subseteq B \implies Q \leq R \implies \text{totalp-on } B Q \leq \text{totalp-on } A R$
 $\langle \text{proof} \rangle$

lemma *total-on-subset*: $\text{total-on } B r \implies A \subseteq B \implies \text{total-on } A r$
 $\langle \text{proof} \rangle$

lemma *totalp-on-subset*: $\text{totalp-on } B \ R \implies A \subseteq B \implies \text{totalp-on } A \ R$
 $\langle \text{proof} \rangle$

lemma *totalp-on-image*:
assumes *inj-on* $f \ A$
shows $\text{totalp-on } (f \cdot A) \ R \longleftrightarrow \text{totalp-on } A \ (\lambda a \ b. \ R \ (f \ a) \ (f \ b))$
 $\langle \text{proof} \rangle$

lemma *total-on-empty* [*simp*]: $\text{total-on } \{\} \ r$
 $\langle \text{proof} \rangle$

lemma *totalp-on-empty* [*simp*]: $\text{totalp-on } \{\} \ R$
 $\langle \text{proof} \rangle$

lemma *total-on-singleton* [*simp*]: $\text{total-on } \{x\} \ r$
 $\langle \text{proof} \rangle$

lemma *totalp-on-singleton* [*simp*]: $\text{totalp-on } \{x\} \ R$
 $\langle \text{proof} \rangle$

lemma (**in** *linorder*) *totalp-on-less* [*simp*]: $\text{totalp-on } A \ (<)$
 $\langle \text{proof} \rangle$

lemma (**in** *linorder*) *totalp-on-greater* [*simp*]: $\text{totalp-on } A \ (>)$
 $\langle \text{proof} \rangle$

lemma (**in** *linorder*) *totalp-on-le* [*simp*]: $\text{totalp-on } A \ (\leq)$
 $\langle \text{proof} \rangle$

lemma (**in** *linorder*) *totalp-on-ge* [*simp*]: $\text{totalp-on } A \ (\geq)$
 $\langle \text{proof} \rangle$

19.2.8 Left uniqueness

definition *left-unique* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{left-unique } R \longleftrightarrow (\forall x \ y \ z. \ R \ x \ z \longrightarrow R \ y \ z \longrightarrow x = y)$

lemma *left-uniqueI*: $(\bigwedge x \ y \ z. \ A \ x \ z \implies A \ y \ z \implies x = y) \implies \text{left-unique } A$
 $\langle \text{proof} \rangle$

lemma *left-uniqueD*: $\text{left-unique } A \implies A \ x \ z \implies A \ y \ z \implies x = y$
 $\langle \text{proof} \rangle$

lemma *left-unique-iff-Uniq*: $\text{left-unique } r \longleftrightarrow (\forall y. \ \exists_{\leq 1} x. \ r \ x \ y)$
 $\langle \text{proof} \rangle$

lemma *left-unique-bot* [*simp*]: $\text{left-unique } \perp$
 $\langle \text{proof} \rangle$

lemma *left-unique-mono-strong*:

left-unique $Q \implies (\bigwedge x y. R x y \implies Q x y) \implies \text{left-unique } R$

$\langle \text{proof} \rangle$

lemma *left-unique-mono[mono]*: $R \leq Q \implies \text{left-unique } Q \leq \text{left-unique } R$

$\langle \text{proof} \rangle$

19.2.9 Right uniqueness

definition *single-valued* :: $('a \times 'b) \text{ set} \Rightarrow \text{bool}$

where *single-valued* $r \longleftrightarrow (\forall x y. (x, y) \in r \longrightarrow (\forall z. (x, z) \in r \longrightarrow y = z))$

definition *right-unique* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**

right-unique $R \longleftrightarrow (\forall x y z. R x y \longrightarrow R x z \longrightarrow y = z)$

lemma *right-unique-single-valued-eq* [*pred-set-conv*]:

right-unique $(\lambda x y. (x, y) \in r) \longleftrightarrow \text{single-valued } r$

$\langle \text{proof} \rangle$

lemma *right-unique-iff-Uniq*:

right-unique $r \longleftrightarrow (\forall x. \exists_{\leq 1} y. r x y)$

$\langle \text{proof} \rangle$

lemma *single-valuedI*:

$(\bigwedge x y. (x, y) \in r \implies (\bigwedge z. (x, z) \in r \implies y = z)) \implies \text{single-valued } r$

$\langle \text{proof} \rangle$

lemma *right-uniqueI*: $(\bigwedge x y z. R x y \implies R x z \implies y = z) \implies \text{right-unique } R$

$\langle \text{proof} \rangle$

lemma *single-valuedD*:

single-valued $r \implies (x, y) \in r \implies (x, z) \in r \implies y = z$

$\langle \text{proof} \rangle$

lemma *right-uniqueD*: *right-unique* $R \implies R x y \implies R x z \implies y = z$

$\langle \text{proof} \rangle$

lemma *single-valued-empty* [*simp*]:

single-valued $\{\}$

$\langle \text{proof} \rangle$

lemma *right-unique-bot*[*simp*]: *right-unique* \perp

$\langle \text{proof} \rangle$

lemma *right-unique-mono-strong*:

right-unique $Q \implies (\bigwedge x y. R x y \implies Q x y) \implies \text{right-unique } R$

$\langle \text{proof} \rangle$

lemma *right-unique-mono[mono]*: $R \leq Q \implies \text{right-unique } Q \leq \text{right-unique } R$

$\langle proof \rangle$

lemma *single-valued-subset*:

$r \subseteq s \implies \text{single-valued } s \implies \text{single-valued } r$

$\langle proof \rangle$

lemma *right-unique-less-eq*: $r \leq s \implies \text{right-unique } s \implies \text{right-unique } r$

$\langle proof \rangle$

19.3 Relation operations

19.3.1 The identity relation

definition *Id* :: 'a rel

where $Id = \{p. \exists x. p = (x, x)\}$

lemma *IdI* [intro]: $(a, a) \in Id$

$\langle proof \rangle$

lemma *IdE* [elim!]: $p \in Id \implies (\bigwedge x. p = (x, x) \implies P) \implies P$

$\langle proof \rangle$

lemma *pair-in-Id-conv* [iff]: $(a, b) \in Id \longleftrightarrow a = b$

$\langle proof \rangle$

lemma *refl-Id*: *refl* *Id*

$\langle proof \rangle$

lemma *antisym-Id*: *antisym* *Id*

— A strange result, since *Id* is also symmetric.

$\langle proof \rangle$

lemma *sym-Id*: *sym* *Id*

$\langle proof \rangle$

lemma *trans-Id*: *trans* *Id*

$\langle proof \rangle$

lemma *single-valued-Id* [simp]: *single-valued* *Id*

$\langle proof \rangle$

lemma *irrefl-diff-Id* [simp]: *irrefl* (*r* − *Id*)

$\langle proof \rangle$

lemma *trans-on-diff-Id*: *trans-on* *A* *r* \implies *antisym-on* *A* *r* \implies *trans-on* *A* (*r* − *Id*)

$\langle proof \rangle$

lemma *trans-diff-Id*[no-atp]: *trans* *r* \implies *antisym* *r* \implies *trans* (*r* − *Id*)

$\langle proof \rangle$

lemma *total-on-diff-Id* [simp]: *total-on* A $(r - Id) = \text{total-on } A \ r$
 $\langle \text{proof} \rangle$

lemma *Id-fstsnd-eq*: $Id = \{x. \text{fst } x = \text{snd } x\}$
 $\langle \text{proof} \rangle$

19.3.2 Diagonal: identity over a set

definition *Id-on* :: 'a set \Rightarrow 'a rel
where *Id-on* $A = (\bigcup x \in A. \{(x, x)\})$

lemma *Id-on-empty* [simp]: *Id-on* $\{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *Id-on-eqI*: $a = b \Longrightarrow a \in A \Longrightarrow (a, b) \in \text{Id-on } A$
 $\langle \text{proof} \rangle$

lemma *Id-onI* [intro!]: $a \in A \Longrightarrow (a, a) \in \text{Id-on } A$
 $\langle \text{proof} \rangle$

lemma *Id-onE* [elim!]: $c \in \text{Id-on } A \Longrightarrow (\bigwedge x. x \in A \Longrightarrow c = (x, x) \Longrightarrow P) \Longrightarrow P$
 — The general elimination rule.
 $\langle \text{proof} \rangle$

lemma *Id-on-iff*: $(x, y) \in \text{Id-on } A \longleftrightarrow x = y \wedge x \in A$
 $\langle \text{proof} \rangle$

lemma *Id-on-def'* [nitpick-unfold]: *Id-on* $\{x. A \ x\} = \text{Collect } (\lambda(x, y). x = y \wedge A \ x)$
 $\langle \text{proof} \rangle$

lemma *Id-on-subset-Times*: *Id-on* $A \subseteq A \times A$
 $\langle \text{proof} \rangle$

lemma *refl-on-Id-on*: *refl-on* A $(\text{Id-on } A)$
 $\langle \text{proof} \rangle$

lemma *antisym-Id-on* [simp]: *antisym* $(\text{Id-on } A)$
 $\langle \text{proof} \rangle$

lemma *sym-Id-on* [simp]: *sym* $(\text{Id-on } A)$
 $\langle \text{proof} \rangle$

lemma *trans-Id-on* [simp]: *trans* $(\text{Id-on } A)$
 $\langle \text{proof} \rangle$

lemma *single-valued-Id-on* [simp]: *single-valued* $(\text{Id-on } A)$
 $\langle \text{proof} \rangle$

19.3.3 Composition

inductive-set *relcomp* :: $('a \times 'b) \text{ set} \Rightarrow ('b \times 'c) \text{ set} \Rightarrow ('a \times 'c) \text{ set}$
for $r :: ('a \times 'b) \text{ set}$ **and** $s :: ('b \times 'c) \text{ set}$
where *relcompI* [intro]: $(a, b) \in r \Longrightarrow (b, c) \in s \Longrightarrow (a, c) \in \text{relcomp } r \ s$

open-bundle *relcomp-syntax*

begin

notation *relcomp* (infixr $\langle O \rangle$ 75) **and** *relcompp* (infixr $\langle OO \rangle$ 75)

end

lemmas *relcomppI* = *relcompp.intros*

For historic reasons, the elimination rules are not wholly corresponding. Feel free to consolidate this.

inductive-cases *relcompEpair*: $(a, c) \in r \ O \ s$

inductive-cases *relcomppE* [elim!]: $(r \ OO \ s) \ a \ c$

lemma *relcompE* [elim!]: $xz \in r \ O \ s \Longrightarrow$
 $(\bigwedge x \ y \ z. xz = (x, z) \Longrightarrow (x, y) \in r \Longrightarrow (y, z) \in s \Longrightarrow P) \Longrightarrow P$
 $\langle \text{proof} \rangle$

lemma *R-O-Id* [simp]: $R \ O \ \text{Id} = R$
 $\langle \text{proof} \rangle$

lemma *Id-O-R* [simp]: $\text{Id} \ O \ R = R$
 $\langle \text{proof} \rangle$

lemma *relcomp-empty1* [simp]: $\{\} \ O \ R = \{\}$
 $\langle \text{proof} \rangle$

lemma *relcompp-bot1* [simp]: $\perp \ OO \ R = \perp$
 $\langle \text{proof} \rangle$

lemma *relcomp-empty2* [simp]: $R \ O \ \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *relcompp-bot2* [simp]: $R \ OO \ \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *O-assoc*: $(R \ O \ S) \ O \ T = R \ O \ (S \ O \ T)$
 $\langle \text{proof} \rangle$

lemma *relcompp-assoc*: $(r \ OO \ s) \ OO \ t = r \ OO \ (s \ OO \ t)$
 $\langle \text{proof} \rangle$

lemma *trans-O-subset*: $\text{trans } r \Longrightarrow r \ O \ r \subseteq r$
 $\langle \text{proof} \rangle$

lemma *transp-relcompp-less-eq*: $\text{transp } r \Longrightarrow r \ OO \ r \leq r$

$\langle \text{proof} \rangle$

lemma *relcomp-mono*: $r' \subseteq r \implies s' \subseteq s \implies r' \circ s' \subseteq r \circ s$
 $\langle \text{proof} \rangle$

lemma *relcompp-mono*: $r' \leq r \implies s' \leq s \implies r' \circ\circ s' \leq r \circ\circ s$
 $\langle \text{proof} \rangle$

lemma *relcomp-subset-Sigma*: $r \subseteq A \times B \implies s \subseteq B \times C \implies r \circ s \subseteq A \times C$
 $\langle \text{proof} \rangle$

lemma *relcomp-distrib* [simp]: $R \circ (S \cup T) = (R \circ S) \cup (R \circ T)$
 $\langle \text{proof} \rangle$

lemma *relcompp-distrib* [simp]: $R \circ\circ (S \sqcup T) = R \circ\circ S \sqcup R \circ\circ T$
 $\langle \text{proof} \rangle$

lemma *relcomp-distrib2* [simp]: $(S \cup T) \circ R = (S \circ R) \cup (T \circ R)$
 $\langle \text{proof} \rangle$

lemma *relcompp-distrib2* [simp]: $(S \sqcup T) \circ\circ R = S \circ\circ R \sqcup T \circ\circ R$
 $\langle \text{proof} \rangle$

lemma *relcomp-UNION-distrib*: $s \circ \bigcup (r \text{ ' } I) = \bigcup_{i \in I} s \circ r \ i$
 $\langle \text{proof} \rangle$

lemma *relcompp-SUP-distrib*: $s \circ\circ \bigsqcup (r \text{ ' } I) = \bigsqcup_{i \in I} s \circ\circ r \ i$
 $\langle \text{proof} \rangle$

lemma *relcomp-UNION-distrib2*: $\bigcup (r \text{ ' } I) \circ s = \bigcup_{i \in I} r \ i \circ s$
 $\langle \text{proof} \rangle$

lemma *relcompp-SUP-distrib2*: $\bigsqcup (r \text{ ' } I) \circ\circ s = \bigsqcup_{i \in I} r \ i \circ\circ s$
 $\langle \text{proof} \rangle$

lemma *single-valued-relcomp*: *single-valued* $r \implies \text{single-valued } s \implies \text{single-valued } (r \circ s)$
 $\langle \text{proof} \rangle$

lemma *relcomp-unfold*: $r \circ s = \{(x, z). \exists y. (x, y) \in r \wedge (y, z) \in s\}$
 $\langle \text{proof} \rangle$

lemma *relcompp-apply*: $(R \circ\circ S) \ a \ c \longleftrightarrow (\exists b. R \ a \ b \wedge S \ b \ c)$
 $\langle \text{proof} \rangle$

lemma *eq-OO*: $(=) \circ\circ R = R$
 $\langle \text{proof} \rangle$

lemma *OO-eq*: $R \circ\circ (=) = R$

<proof>

19.3.4 Converse

inductive-set *converse* :: (*'a* × *'b*) set \Rightarrow (*'b* × *'a*) set
for *r* :: (*'a* × *'b*) set
where (*a*, *b*) ∈ *r* \Longrightarrow (*b*, *a*) ∈ *converse r*

open-bundle *converse-syntax*

begin

notation

converse (*<(notation=<postfix -1>>⁻¹) [1000] 999>*) **and**
conversep (*<(notation=<postfix -1-1>>⁻¹⁻¹) [1000] 1000>*)

notation (*ASCII*)

converse (*<(notation=<postfix -1>>⁻1) [1000] 999>*) **and**
conversep (*<(notation=<postfix -1-1>>⁻-1) [1000] 1000>*)

end

lemma *converseI* [*sym*]: (*a*, *b*) ∈ *r* \Longrightarrow (*b*, *a*) ∈ *r*⁻¹
<proof>

lemma *conversepI* : *r a b* \Longrightarrow *r*⁻¹⁻¹ *b a*
<proof>

lemma *converseD* [*sym*]: (*a*, *b*) ∈ *r*⁻¹ \Longrightarrow (*b*, *a*) ∈ *r*
<proof>

lemma *conversepD* : *r*⁻¹⁻¹ *b a* \Longrightarrow *r a b*
<proof>

lemma *converseE* [*elim!*]: *yx* ∈ *r*⁻¹ \Longrightarrow ($\bigwedge x y. yx = (y, x) \Longrightarrow (x, y) \in r \Longrightarrow P$) $\Longrightarrow P$
 — More general than *converseD*, as it “splits” the member of the relation.
<proof>

lemmas *conversepE* [*elim!*] = *conversep.cases*

lemma *converse-iff* [*iff*]: (*a*, *b*) ∈ *r*⁻¹ \longleftrightarrow (*b*, *a*) ∈ *r*
<proof>

lemma *conversep-iff* [*iff*]: *r*⁻¹⁻¹ *a b* = *r b a*
<proof>

lemma *converse-converse* [*simp*]: (*r*⁻¹)⁻¹ = *r*
<proof>

lemma *conversep-conversep* [*simp*]: (*r*⁻¹⁻¹)⁻¹⁻¹ = *r*
<proof>

lemma *converse-empty[simp]*: $\{\}^{-1} = \{\}$
 $\langle proof \rangle$

lemma *converse-UNIV[simp]*: $UNIV^{-1} = UNIV$
 $\langle proof \rangle$

lemma *converse-relcomp*: $(r \ O \ s)^{-1} = s^{-1} \ O \ r^{-1}$
 $\langle proof \rangle$

lemma *converse-relcompp*: $(r \ OO \ s)^{-1-1} = s^{-1-1} \ OO \ r^{-1-1}$
 $\langle proof \rangle$

lemma *converse-Int*: $(r \cap s)^{-1} = r^{-1} \cap s^{-1}$
 $\langle proof \rangle$

lemma *converse-meet*: $(r \sqcap s)^{-1-1} = r^{-1-1} \sqcap s^{-1-1}$
 $\langle proof \rangle$

lemma *converse-Un*: $(r \cup s)^{-1} = r^{-1} \cup s^{-1}$
 $\langle proof \rangle$

lemma *converse-join*: $(r \sqcup s)^{-1-1} = r^{-1-1} \sqcup s^{-1-1}$
 $\langle proof \rangle$

lemma *converse-INTER*: $(\bigcap (r \ ' \ S))^{-1} = (\bigcap x \in S. (r \ x)^{-1})$
 $\langle proof \rangle$

lemma *converse-UNION*: $(\bigcup (r \ ' \ S))^{-1} = (\bigcup x \in S. (r \ x)^{-1})$
 $\langle proof \rangle$

lemma *converse-mono[simp]*: $r^{-1} \subseteq s^{-1} \longleftrightarrow r \subseteq s$
 $\langle proof \rangle$

lemma *conversep-mono[simp]*: $r^{-1-1} \leq s^{-1-1} \longleftrightarrow r \leq s$
 $\langle proof \rangle$

lemma *converse-inject[simp]*: $r^{-1} = s^{-1} \longleftrightarrow r = s$
 $\langle proof \rangle$

lemma *conversep-inject[simp]*: $r^{-1-1} = s^{-1-1} \longleftrightarrow r = s$
 $\langle proof \rangle$

lemma *converse-subset-swap*: $r \subseteq s^{-1} \longleftrightarrow r^{-1} \subseteq s$
 $\langle proof \rangle$

lemma *conversep-le-swap*: $r \leq s^{-1-1} \longleftrightarrow r^{-1-1} \leq s$
 $\langle proof \rangle$

lemma *converse-Id [simp]*: $Id^{-1} = Id$

$\langle proof \rangle$

lemma *converse-Id-on* [simp]: $(Id-on\ A)^{-1} = Id-on\ A$
 $\langle proof \rangle$

lemma *refl-on-converse* [simp]: $refl-on\ A\ (r^{-1}) = refl-on\ A\ r$
 $\langle proof \rangle$

lemma *reflp-on-conversp* [simp]: $reflp-on\ A\ R^{-1-1} \longleftrightarrow reflp-on\ A\ R$
 $\langle proof \rangle$

lemma *irrefl-on-converse* [simp]: $irrefl-on\ A\ (r^{-1}) = irrefl-on\ A\ r$
 $\langle proof \rangle$

lemma *irreflp-on-converse* [simp]: $irreflp-on\ A\ (r^{-1-1}) = irreflp-on\ A\ r$
 $\langle proof \rangle$

lemma *sym-on-converse* [simp]: $sym-on\ A\ (r^{-1}) = sym-on\ A\ r$
 $\langle proof \rangle$

lemma *symp-on-conversep* [simp]: $symp-on\ A\ R^{-1-1} = symp-on\ A\ R$
 $\langle proof \rangle$

lemma *asym-on-converse* [simp]: $asym-on\ A\ (r^{-1}) = asym-on\ A\ r$
 $\langle proof \rangle$

lemma *asymp-on-conversep* [simp]: $asymp-on\ A\ R^{-1-1} = asymp-on\ A\ R$
 $\langle proof \rangle$

lemma *antisym-on-converse* [simp]: $antisym-on\ A\ (r^{-1}) = antisym-on\ A\ r$
 $\langle proof \rangle$

lemma *antisymp-on-conversep* [simp]: $antisymp-on\ A\ R^{-1-1} = antisymp-on\ A\ R$
 $\langle proof \rangle$

lemma *trans-on-converse* [simp]: $trans-on\ A\ (r^{-1}) = trans-on\ A\ r$
 $\langle proof \rangle$

lemma *transp-on-conversep* [simp]: $transp-on\ A\ R^{-1-1} = transp-on\ A\ R$
 $\langle proof \rangle$

lemma *sym-conv-converse-eq*: $sym\ r \longleftrightarrow r^{-1} = r$
 $\langle proof \rangle$

lemma *sym-Un-converse*: $sym\ (r \cup r^{-1})$
 $\langle proof \rangle$

lemma *sym-Int-converse*: $sym\ (r \cap r^{-1})$
 $\langle proof \rangle$

lemma *total-on-converse* [simp]: $\text{total-on } A \ (r^{-1}) = \text{total-on } A \ r$
 ⟨proof⟩

lemma *totalp-on-converse* [simp]: $\text{totalp-on } A \ R^{-1-1} = \text{totalp-on } A \ R$
 ⟨proof⟩

lemma *left-unique-conversep*[simp]: $\text{left-unique } A^{-1-1} \longleftrightarrow \text{right-unique } A$
 ⟨proof⟩

lemma *right-unique-conversep*[simp]: $\text{right-unique } A^{-1-1} \longleftrightarrow \text{left-unique } A$
 ⟨proof⟩

lemma *conversep-noteq* [simp]: $(\neq)^{-1-1} = (\neq)$
 ⟨proof⟩

lemma *conversep-eq* [simp]: $(=)^{-1-1} = (=)$
 ⟨proof⟩

lemma *converse-unfold* [code]: $r^{-1} = \{(y, x). (x, y) \in r\}$
 ⟨proof⟩

19.3.5 Domain, range and field

inductive-set *Domain* :: $('a \times 'b) \text{ set} \Rightarrow 'a \text{ set}$ **for** $r :: ('a \times 'b) \text{ set}$
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!]: $\text{Domainp } r \ a$

inductive-set *Range* :: $('a \times 'b) \text{ set} \Rightarrow 'b \text{ set}$ **for** $r :: ('a \times 'b) \text{ set}$
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!]: $\text{Rangep } r \ b$

definition *Field* :: $'a \text{ rel} \Rightarrow 'a \text{ set}$
where *Field* $r = \text{Domain } r \cup \text{Range } r$

lemma *Field-iff*: $x \in \text{Field } r \longleftrightarrow (\exists y. (x, y) \in r \vee (y, x) \in r)$
 ⟨proof⟩

lemma *FieldI1*: $(i, j) \in R \Longrightarrow i \in \text{Field } R$
 ⟨proof⟩

lemma *FieldI2*: $(i, j) \in R \implies j \in \text{Field } R$
 $\langle \text{proof} \rangle$

lemma *Domain-fst* [code]: $\text{Domain } r = \text{fst } ' r$
 $\langle \text{proof} \rangle$

lemma *Range-snd* [code]: $\text{Range } r = \text{snd } ' r$
 $\langle \text{proof} \rangle$

lemma *fst-eq-Domain*: $\text{fst } ' R = \text{Domain } R$
 $\langle \text{proof} \rangle$

lemma *snd-eq-Range*: $\text{snd } ' R = \text{Range } R$
 $\langle \text{proof} \rangle$

lemma *range-fst* [simp]: $\text{range fst} = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *range-snd* [simp]: $\text{range snd} = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *Domain-empty* [simp]: $\text{Domain } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *Range-empty* [simp]: $\text{Range } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *Field-empty* [simp]: $\text{Field } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *Domain-empty-iff*: $\text{Domain } r = \{\} \longleftrightarrow r = \{\}$
 $\langle \text{proof} \rangle$

lemma *Range-empty-iff*: $\text{Range } r = \{\} \longleftrightarrow r = \{\}$
 $\langle \text{proof} \rangle$

lemma *Domain-insert* [simp]: $\text{Domain } (\text{insert } (a, b) r) = \text{insert } a (\text{Domain } r)$
 $\langle \text{proof} \rangle$

lemma *Range-insert* [simp]: $\text{Range } (\text{insert } (a, b) r) = \text{insert } b (\text{Range } r)$
 $\langle \text{proof} \rangle$

lemma *Field-insert* [simp]: $\text{Field } (\text{insert } (a, b) r) = \{a, b\} \cup \text{Field } r$
 $\langle \text{proof} \rangle$

lemma *Domain-iff*: $a \in \text{Domain } r \longleftrightarrow (\exists y. (a, y) \in r)$
 $\langle \text{proof} \rangle$

lemma *Range-iff*: $a \in \text{Range } r \longleftrightarrow (\exists y. (y, a) \in r)$

$\langle proof \rangle$

lemma *Domain-Id* [simp]: $Domain\ Id = UNIV$
 $\langle proof \rangle$

lemma *Range-Id* [simp]: $Range\ Id = UNIV$
 $\langle proof \rangle$

lemma *Domain-Id-on* [simp]: $Domain\ (Id-on\ A) = A$
 $\langle proof \rangle$

lemma *Range-Id-on* [simp]: $Range\ (Id-on\ A) = A$
 $\langle proof \rangle$

lemma *Domain-Un-eq*: $Domain\ (A \cup B) = Domain\ A \cup Domain\ B$
 $\langle proof \rangle$

lemma *Range-Un-eq*: $Range\ (A \cup B) = Range\ A \cup Range\ B$
 $\langle proof \rangle$

lemma *Field-Un* [simp]: $Field\ (r \cup s) = Field\ r \cup Field\ s$
 $\langle proof \rangle$

lemma *Domain-Int-subset*: $Domain\ (A \cap B) \subseteq Domain\ A \cap Domain\ B$
 $\langle proof \rangle$

lemma *Range-Int-subset*: $Range\ (A \cap B) \subseteq Range\ A \cap Range\ B$
 $\langle proof \rangle$

lemma *Domain-Diff-subset*: $Domain\ A - Domain\ B \subseteq Domain\ (A - B)$
 $\langle proof \rangle$

lemma *Range-Diff-subset*: $Range\ A - Range\ B \subseteq Range\ (A - B)$
 $\langle proof \rangle$

lemma *Domain-Union*: $Domain\ (\bigcup S) = (\bigcup A \in S. Domain\ A)$
 $\langle proof \rangle$

lemma *Range-Union*: $Range\ (\bigcup S) = (\bigcup A \in S. Range\ A)$
 $\langle proof \rangle$

lemma *Field-Union* [simp]: $Field\ (\bigcup R) = \bigcup (Field\ ` R)$
 $\langle proof \rangle$

lemma *Domain-converse* [simp]: $Domain\ (r^{-1}) = Range\ r$
 $\langle proof \rangle$

lemma *Range-converse* [simp]: $Range\ (r^{-1}) = Domain\ r$
 $\langle proof \rangle$

lemma *Field-converse* [simp]: $\text{Field } (r^{-1}) = \text{Field } r$
 ⟨proof⟩

lemma *Domain-Collect-case-prod* [simp]: $\text{Domain } \{(x, y). P \ x \ y\} = \{x. \exists y. P \ x \ y\}$
 ⟨proof⟩

lemma *Range-Collect-case-prod* [simp]: $\text{Range } \{(x, y). P \ x \ y\} = \{y. \exists x. P \ x \ y\}$
 ⟨proof⟩

lemma *Domain-mono*: $r \subseteq s \implies \text{Domain } r \subseteq \text{Domain } s$
 ⟨proof⟩

lemma *Range-mono*: $r \subseteq s \implies \text{Range } r \subseteq \text{Range } s$
 ⟨proof⟩

lemma *mono-Field*: $r \subseteq s \implies \text{Field } r \subseteq \text{Field } s$
 ⟨proof⟩

lemma *Domain-unfold*: $\text{Domain } r = \{x. \exists y. (x, y) \in r\}$
 ⟨proof⟩

lemma *Field-square* [simp]: $\text{Field } (x \times x) = x$
 ⟨proof⟩

19.3.6 Image of a set under a relation

definition *Image* :: $('a \times 'b) \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set}$ (**infixr** ‘‘ \circ ’ 90)
 where $r \circ s = \{y. \exists x \in s. (x, y) \in r\}$

lemma *Image-iff*: $b \in r \circ A \longleftrightarrow (\exists x \in A. (x, b) \in r)$
 ⟨proof⟩

lemma *Image-singleton*: $r \circ \{a\} = \{b. (a, b) \in r\}$
 ⟨proof⟩

lemma *Image-singleton-iff* [iff]: $b \in r \circ \{a\} \longleftrightarrow (a, b) \in r$
 ⟨proof⟩

lemma *ImageI* [intro]: $(a, b) \in r \implies a \in A \implies b \in r \circ A$
 ⟨proof⟩

lemma *ImageE* [elim!]: $b \in r \circ A \implies (\bigwedge x. (x, b) \in r \implies x \in A \implies P) \implies P$
 ⟨proof⟩

lemma *rev-ImageI*: $a \in A \implies (a, b) \in r \implies b \in r \circ A$
 — This version’s more effective when we already have the required a
 ⟨proof⟩

lemma *Image-empty1* [simp]: $\{\} \text{ “ } X = \{\}$
 $\langle \text{proof} \rangle$

lemma *Image-empty2* [simp]: $R \text{ “ } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *Image-Id* [simp]: $\text{Id} \text{ “ } A = A$
 $\langle \text{proof} \rangle$

lemma *Image-Id-on* [simp]: $\text{Id-on } A \text{ “ } B = A \cap B$
 $\langle \text{proof} \rangle$

lemma *Image-Int-subset*: $R \text{ “ } (A \cap B) \subseteq R \text{ “ } A \cap R \text{ “ } B$
 $\langle \text{proof} \rangle$

lemma *Image-Int-eq*: *single-valued* (*converse* R) $\implies R \text{ “ } (A \cap B) = R \text{ “ } A \cap R \text{ “ } B$
 $\langle \text{proof} \rangle$

lemma *Image-Un*: $R \text{ “ } (A \cup B) = R \text{ “ } A \cup R \text{ “ } B$
 $\langle \text{proof} \rangle$

lemma *Un-Image*: $(R \cup S) \text{ “ } A = R \text{ “ } A \cup S \text{ “ } A$
 $\langle \text{proof} \rangle$

lemma *Image-subset*: $r \subseteq A \times B \implies r \text{ “ } C \subseteq B$
 $\langle \text{proof} \rangle$

lemma *Image-eq-UN*: $r \text{ “ } B = (\bigcup_{y \in B. r \text{ “ } \{y\}}$
 — NOT suitable for rewriting
 $\langle \text{proof} \rangle$

lemma *Image-mono*: $r' \subseteq r \implies A' \subseteq A \implies (r' \text{ “ } A') \subseteq (r \text{ “ } A)$
 $\langle \text{proof} \rangle$

lemma *Image-UN*: $r \text{ “ } (\bigcup (B \text{ ‘ } A)) = (\bigcup_{x \in A. r \text{ “ } (B \ x))$
 $\langle \text{proof} \rangle$

lemma *UN-Image*: $(\bigcup_{i \in I. X \ i) \text{ “ } S = (\bigcup_{i \in I. X \ i \text{ “ } S)$
 $\langle \text{proof} \rangle$

lemma *Image-INT-subset*: $(r \text{ “ } (\bigcap (B \text{ ‘ } A))) \subseteq (\bigcap_{x \in A. r \text{ “ } (B \ x))$
 $\langle \text{proof} \rangle$

Converse inclusion requires some assumptions

lemma *Image-INT-eq*:
 assumes *single-valued* (r^{-1})
 and $A \neq \{\}$

shows $r \text{ “ } (\bigcap (B \text{ ‘ } A)) = (\bigcap_{x \in A} r \text{ “ } B \ x)$
 $\langle \text{proof} \rangle$

lemma *Image-subset-eq*: $r \text{ “ } A \subseteq B \longleftrightarrow A \subseteq - ((r^{-1}) \text{ “ } (- B))$
 $\langle \text{proof} \rangle$

lemma *Image-Collect-case-prod* [simp]: $\{(x, y). P \ x \ y\} \text{ “ } A = \{y. \exists x \in A. P \ x \ y\}$
 $\langle \text{proof} \rangle$

lemma *Sigma-Image*: $(\text{SIGMA } x:A. B \ x) \text{ “ } X = (\bigcup_{x \in X \cap A} B \ x)$
 $\langle \text{proof} \rangle$

lemma *relcomp-Image*: $(X \ O \ Y) \text{ “ } Z = Y \text{ “ } (X \text{ “ } Z)$
 $\langle \text{proof} \rangle$

19.3.7 Inverse image

definition *inv-image* :: $'b \text{ rel} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ rel}$
where $\text{inv-image } r \ f = \{(x, y). (f \ x, f \ y) \in r\}$

definition *inv-imagep* :: $('b \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$
where $\text{inv-imagep } r \ f = (\lambda x \ y. r \ (f \ x) \ (f \ y))$

lemma [pred-set-conv]: $\text{inv-imagep } (\lambda x \ y. (x, y) \in r) \ f = (\lambda x \ y. (x, y) \in \text{inv-image } r \ f)$
 $\langle \text{proof} \rangle$

lemma *sym-inv-image*: $\text{sym } r \Longrightarrow \text{sym } (\text{inv-image } r \ f)$
 $\langle \text{proof} \rangle$

lemma *trans-inv-image*: $\text{trans } r \Longrightarrow \text{trans } (\text{inv-image } r \ f)$
 $\langle \text{proof} \rangle$

lemma *total-inv-image*: $\llbracket \text{inj } f; \text{total } r \rrbracket \Longrightarrow \text{total } (\text{inv-image } r \ f)$
 $\langle \text{proof} \rangle$

lemma *asym-inv-image*: $\text{asym } R \Longrightarrow \text{asym } (\text{inv-image } R \ f)$
 $\langle \text{proof} \rangle$

lemma *in-inv-image*[simp]: $(x, y) \in \text{inv-image } r \ f \longleftrightarrow (f \ x, f \ y) \in r$
 $\langle \text{proof} \rangle$

lemma *converse-inv-image*[simp]: $(\text{inv-image } R \ f)^{-1} = \text{inv-image } (R^{-1}) \ f$
 $\langle \text{proof} \rangle$

lemma *in-inv-imagep* [simp]: $\text{inv-imagep } r \ f \ x \ y = r \ (f \ x) \ (f \ y)$
 $\langle \text{proof} \rangle$

19.3.8 Powerset

definition $Powp :: ('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow bool$
where $Powp\ A = (\lambda B. \forall x \in B. A\ x)$

lemma $Powp\text{-}Pow\text{-}eq\ [pred\text{-}set\text{-}conv]: Powp\ (\lambda x. x \in A) = (\lambda x. x \in Pow\ A)$
 $\langle proof \rangle$

lemmas $Powp\text{-}mono\ [mono] = Pow\text{-}mono\ [to\text{-}pred]$

end

20 Finite sets

theory $Finite\text{-}Set$
imports $Product\text{-}Type\ Sum\text{-}Type\ Fields\ Relation$
begin

20.1 Predicate for finite sets

context notes $[[inductive\text{-}internals]]$
begin

inductive $finite :: 'a\ set \Rightarrow bool$
where
 $emptyI\ [simp, intro!]: finite\ \{\}$
 $| insertI\ [simp, intro!]: finite\ A \Longrightarrow finite\ (insert\ a\ A)$

end

$\langle ML \rangle$

declare $[[simproc\ del: finite\text{-}Collect]]$

lemma $finite\text{-}induct\ [case\text{-}names\ empty\ insert, induct\ set: finite]:$
 $\text{--- Discharging } x \notin F \text{ entails extra work.}$
assumes $finite\ F$
assumes $P\ \{\}$
and $insert: \bigwedge x\ F. finite\ F \Longrightarrow x \notin F \Longrightarrow P\ F \Longrightarrow P\ (insert\ x\ F)$
shows $P\ F$
 $\langle proof \rangle$

lemma $infinite\text{-}finite\text{-}induct\ [case\text{-}names\ infinite\ empty\ insert]:$
assumes $infinite: \bigwedge A. \neg finite\ A \Longrightarrow P\ A$
and $empty: P\ \{\}$
and $insert: \bigwedge x\ F. finite\ F \Longrightarrow x \notin F \Longrightarrow P\ F \Longrightarrow P\ (insert\ x\ F)$
shows $P\ A$
 $\langle proof \rangle$

20.1.1 Choice principles

lemma *ex-new-if-finite*: — does not depend on def of finite at all

assumes $\neg \text{finite } (\text{UNIV} :: 'a \text{ set})$ **and** *finite A*

shows $\exists a :: 'a. a \notin A$

<proof>

A finite choice principle. Does not need the SOME choice operator.

lemma *finite-set-choice*: *finite A* $\implies \forall x \in A. \exists y. P x y \implies \exists f. \forall x \in A. P x (f x)$

<proof>

20.1.2 Finite sets are the images of initial segments of natural numbers

lemma *finite-imp-nat-seg-image-inj-on*:

assumes *finite A*

shows $\exists (n :: \text{nat}) f. A = f \text{ ` } \{i. i < n\} \wedge \text{inj-on } f \text{ ` } \{i. i < n\}$

<proof>

lemma *nat-seg-image-imp-finite*: $A = f \text{ ` } \{i :: \text{nat}. i < n\} \implies \text{finite } A$

<proof>

lemma *finite-conv-nat-seg-image*: *finite A* $\longleftrightarrow (\exists n f. A = f \text{ ` } \{i :: \text{nat}. i < n\})$

<proof>

lemma *finite-imp-inj-to-nat-seg*:

assumes *finite A*

shows $\exists f n. f \text{ ` } A = \{i :: \text{nat}. i < n\} \wedge \text{inj-on } f A$

<proof>

lemma *finite-Collect-less-nat [iff]*: *finite* $\{n :: \text{nat}. n < k\}$

<proof>

lemma *finite-Collect-le-nat [iff]*: *finite* $\{n :: \text{nat}. n \leq k\}$

<proof>

20.2 Finiteness and common set operations

lemma *rev-finite-subset*: *finite B* $\implies A \subseteq B \implies \text{finite } A$

<proof>

lemma *finite-subset*: $A \subseteq B \implies \text{finite } B \implies \text{finite } A$

<proof>

<ML>

declare $[[\text{simproc del: finite}]]$

lemma *finite-UnI*:

assumes *finite F* **and** *finite G*
shows *finite (F ∪ G)*
 ⟨proof⟩

lemma *finite-Un [iff]*: *finite (F ∪ G) ⟷ finite F ∧ finite G*
 ⟨proof⟩

lemma *finite-insert [simp]*: *finite (insert a A) ⟷ finite A*
 ⟨proof⟩

lemma *finite-Int [simp, intro]*: *finite F ∨ finite G ⟹ finite (F ∩ G)*
 ⟨proof⟩

lemma *finite-Collect-conjI [simp, intro]*:
finite {x. P x} ∨ finite {x. Q x} ⟹ finite {x. P x ∧ Q x}
 ⟨proof⟩

lemma *finite-Collect-disjI [simp]*:
finite {x. P x ∨ Q x} ⟷ finite {x. P x} ∧ finite {x. Q x}
 ⟨proof⟩

lemma *finite-Diff [simp, intro]*: *finite A ⟹ finite (A − B)*
 ⟨proof⟩

lemma *finite-Diff2 [simp]*:
assumes *finite B*
shows *finite (A − B) ⟷ finite A*
 ⟨proof⟩

lemma *finite-Diff-insert [iff]*: *finite (A − insert a B) ⟷ finite (A − B)*
 ⟨proof⟩

lemma *finite-compl [simp]*:
finite (A :: 'a set) ⟹ finite (− A) ⟷ finite (UNIV :: 'a set)
 ⟨proof⟩

lemma *finite-Collect-not [simp]*:
finite {x :: 'a. P x} ⟹ finite {x. ¬ P x} ⟷ finite (UNIV :: 'a set)
 ⟨proof⟩

lemma *finite-Union [simp, intro]*:
finite A ⟹ (⋀ M. M ∈ A ⟹ finite M) ⟹ finite (⋃ A)
 ⟨proof⟩

lemma *finite-UN-I [intro]*:
finite A ⟹ (⋀ a. a ∈ A ⟹ finite (B a)) ⟹ finite (⋃ a ∈ A. B a)
 ⟨proof⟩

lemma *finite-UN [simp]*: *finite A ⟹ finite (⋃ (B ` A)) ⟷ (∀ x ∈ A. finite (B x))*

$\langle \text{proof} \rangle$

lemma *finite-Inter* [intro]: $\exists A \in M. \text{finite } A \implies \text{finite } (\bigcap M)$
 $\langle \text{proof} \rangle$

lemma *finite-INT* [intro]: $\exists x \in I. \text{finite } (A \ x) \implies \text{finite } (\bigcap_{x \in I} A \ x)$
 $\langle \text{proof} \rangle$

lemma *finite-imageI* [simp, intro]: $\text{finite } F \implies \text{finite } (h \ ' F)$
 $\langle \text{proof} \rangle$

lemma *finite-image-set* [simp]: $\text{finite } \{x. P \ x\} \implies \text{finite } \{f \ x \mid x. P \ x\}$
 $\langle \text{proof} \rangle$

lemma *finite-image-set2*:
 $\text{finite } \{x. P \ x\} \implies \text{finite } \{y. Q \ y\} \implies \text{finite } \{f \ x \ y \mid x \ y. P \ x \wedge Q \ y\}$
 $\langle \text{proof} \rangle$

lemma *finite-imageD*:
assumes $\text{finite } (f \ ' A)$ **and** $\text{inj-on } f \ A$
shows $\text{finite } A$
 $\langle \text{proof} \rangle$

lemma *finite-image-iff*: $\text{inj-on } f \ A \implies \text{finite } (f \ ' A) \longleftrightarrow \text{finite } A$
 $\langle \text{proof} \rangle$

lemma *finite-surj*: $\text{finite } A \implies B \subseteq f \ ' A \implies \text{finite } B$
 $\langle \text{proof} \rangle$

lemma *finite-range-imageI*: $\text{finite } (\text{range } g) \implies \text{finite } (\text{range } (\lambda x. f \ (g \ x)))$
 $\langle \text{proof} \rangle$

lemma *finite-subset-image*:
assumes $\text{finite } B$
shows $B \subseteq f \ ' A \implies \exists C \subseteq A. \text{finite } C \wedge B = f \ ' C$
 $\langle \text{proof} \rangle$

lemma *all-subset-image*: $(\forall B. B \subseteq f \ ' A \longrightarrow P \ B) \longleftrightarrow (\forall B. B \subseteq A \longrightarrow P(f \ ' B))$
 $\langle \text{proof} \rangle$

lemma *all-finite-subset-image*:
 $(\forall B. \text{finite } B \wedge B \subseteq f \ ' A \longrightarrow P \ B) \longleftrightarrow (\forall B. \text{finite } B \wedge B \subseteq A \longrightarrow P(f \ ' B))$
 $\langle \text{proof} \rangle$

lemma *ex-finite-subset-image*:
 $(\exists B. \text{finite } B \wedge B \subseteq f \ ' A \wedge P \ B) \longleftrightarrow (\exists B. \text{finite } B \wedge B \subseteq A \wedge P(f \ ' B))$
 $\langle \text{proof} \rangle$

lemma *finite-vimage-IntI*: $\text{finite } F \implies \text{inj-on } h \ A \implies \text{finite } (h -' F \cap A)$
 ⟨proof⟩

lemma *finite-finite-vimage-IntI*:
 assumes $\text{finite } F$
 and $\bigwedge y. y \in F \implies \text{finite } ((h -' \{y\}) \cap A)$
 shows $\text{finite } (h -' F \cap A)$
 ⟨proof⟩

lemma *finite-vimageI*: $\text{finite } F \implies \text{inj } h \implies \text{finite } (h -' F)$
 ⟨proof⟩

lemma *finite-vimageD'*: $\text{finite } (f -' A) \implies A \subseteq \text{range } f \implies \text{finite } A$
 ⟨proof⟩

lemma *finite-vimageD*: $\text{finite } (h -' F) \implies \text{surj } h \implies \text{finite } F$
 ⟨proof⟩

lemma *finite-vimage-iff*: $\text{bij } h \implies \text{finite } (h -' F) \longleftrightarrow \text{finite } F$
 ⟨proof⟩

lemma *finite-inverse-image-gen*:
 assumes $\text{finite } A \text{ inj-on } f \ D$
 shows $\text{finite } \{j \in D. f \ j \in A\}$
 ⟨proof⟩

lemma *finite-inverse-image*:
 assumes $\text{finite } A \text{ inj } f$
 shows $\text{finite } \{j. f \ j \in A\}$
 ⟨proof⟩

lemma *finite-Collect-bex [simp]*:
 assumes $\text{finite } A$
 shows $\text{finite } \{x. \exists y \in A. Q \ x \ y\} \longleftrightarrow (\forall y \in A. \text{finite } \{x. Q \ x \ y\})$
 ⟨proof⟩

lemma *finite-Collect-bounded-ex [simp]*:
 assumes $\text{finite } \{y. P \ y\}$
 shows $\text{finite } \{x. \exists y. P \ y \wedge Q \ x \ y\} \longleftrightarrow (\forall y. P \ y \longrightarrow \text{finite } \{x. Q \ x \ y\})$
 ⟨proof⟩

lemma *finite-Plus*: $\text{finite } A \implies \text{finite } B \implies \text{finite } (A <+> B)$
 ⟨proof⟩

lemma *finite-PlusD*:
 fixes $A :: 'a \text{ set}$ and $B :: 'b \text{ set}$
 assumes $\text{fin: finite } (A <+> B)$
 shows $\text{finite } A \text{ finite } B$
 ⟨proof⟩

lemma *finite-Plus-iff* [*simp*]: $\text{finite } (A <+> B) \longleftrightarrow \text{finite } A \wedge \text{finite } B$
 ⟨*proof*⟩

lemma *finite-Plus-UNIV-iff* [*simp*]:
 $\text{finite } (\text{UNIV} :: ('a + 'b) \text{ set}) \longleftrightarrow \text{finite } (\text{UNIV} :: 'a \text{ set}) \wedge \text{finite } (\text{UNIV} :: 'b \text{ set})$
 ⟨*proof*⟩

lemma *finite-SigmaI* [*simp, intro*]:
 $\text{finite } A \implies (\bigwedge a. a \in A \implies \text{finite } (B \ a)) \implies \text{finite } (\text{SIGMA } a:A. B \ a)$
 ⟨*proof*⟩

lemma *finite-SigmaI2*:
 assumes $\text{finite } \{x \in A. B \ x \neq \{\}\}$
 and $\bigwedge a. a \in A \implies \text{finite } (B \ a)$
 shows $\text{finite } (\text{Sigma } A \ B)$
 ⟨*proof*⟩

lemma *finite-cartesian-product*: $\text{finite } A \implies \text{finite } B \implies \text{finite } (A \times B)$
 ⟨*proof*⟩

lemma *finite-Prod-UNIV*:
 $\text{finite } (\text{UNIV} :: 'a \text{ set}) \implies \text{finite } (\text{UNIV} :: 'b \text{ set}) \implies \text{finite } (\text{UNIV} :: ('a \times 'b) \text{ set})$
 ⟨*proof*⟩

lemma *finite-cartesian-productD1*:
 assumes $\text{finite } (A \times B)$ and $B \neq \{\}$
 shows $\text{finite } A$
 ⟨*proof*⟩

lemma *finite-cartesian-productD2*:
 assumes $\text{finite } (A \times B)$ and $A \neq \{\}$
 shows $\text{finite } B$
 ⟨*proof*⟩

lemma *finite-cartesian-product-iff*:
 $\text{finite } (A \times B) \longleftrightarrow (A = \{\} \vee B = \{\} \vee (\text{finite } A \wedge \text{finite } B))$
 ⟨*proof*⟩

lemma *finite-prod*:
 $\text{finite } (\text{UNIV} :: ('a \times 'b) \text{ set}) \longleftrightarrow \text{finite } (\text{UNIV} :: 'a \text{ set}) \wedge \text{finite } (\text{UNIV} :: 'b \text{ set})$
 ⟨*proof*⟩

lemma *finite-Pow-iff* [*iff*]: $\text{finite } (\text{Pow } A) \longleftrightarrow \text{finite } A$
 ⟨*proof*⟩

corollary *finite-Collect-subsets* [*simp, intro*]: $\text{finite } A \implies \text{finite } \{B. B \subseteq A\}$
 ⟨*proof*⟩

lemma *finite-set*: $\text{finite } (\text{UNIV} :: 'a \text{ set set}) \longleftrightarrow \text{finite } (\text{UNIV} :: 'a \text{ set})$
 ⟨*proof*⟩

lemma *finite-UnionD*: $\text{finite } (\bigcup A) \implies \text{finite } A$
 ⟨*proof*⟩

lemma *finite-bind*:
 assumes *finite S*
 assumes $\forall x \in S. \text{finite } (f\ x)$
 shows $\text{finite } (\text{Set.bind } S\ f)$
 ⟨*proof*⟩

lemma *finite-filter* [*simp*]: $\text{finite } S \implies \text{finite } (\text{Set.filter } P\ S)$
 ⟨*proof*⟩

lemma *finite-set-of-finite-funs*:
 assumes *finite A finite B*
 shows $\text{finite } \{f. \forall x. (x \in A \longrightarrow f\ x \in B) \wedge (x \notin A \longrightarrow f\ x = d)\}$ (*is finite ?S*)
 ⟨*proof*⟩

lemma *not-finite-existsD*:
 assumes $\neg \text{finite } \{a. P\ a\}$
 shows $\exists a. P\ a$
 ⟨*proof*⟩

lemma *finite-converse* [*iff*]: $\text{finite } (r^{-1}) \longleftrightarrow \text{finite } r$
 ⟨*proof*⟩

lemma *finite-Domain*: $\text{finite } r \implies \text{finite } (\text{Domain } r)$
 ⟨*proof*⟩

lemma *finite-Range*: $\text{finite } r \implies \text{finite } (\text{Range } r)$
 ⟨*proof*⟩

lemma *finite-Field*: $\text{finite } r \implies \text{finite } (\text{Field } r)$
 ⟨*proof*⟩

lemma *finite-Image*[*simp*]: $\text{finite } R \implies \text{finite } (R \text{ `` } A)$
 ⟨*proof*⟩

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 $\bigwedge x. P\ \{x\}$
 and $\bigwedge x\ F. \text{finite } F \implies F \neq \{\} \implies x \notin F \implies P\ F \implies P\ (\text{insert } x\ F)$

shows $P\ F$
 $\langle proof \rangle$

lemma *finite-subset-induct* [*consumes 2, case-names empty insert*]:
assumes *finite* F **and** $F \subseteq A$
and *empty*: $P\ \{\}$
and *insert*: $\bigwedge a\ F. \text{finite } F \implies a \in A \implies a \notin F \implies P\ F \implies P\ (\text{insert } a\ F)$
shows $P\ F$
 $\langle proof \rangle$

lemma *finite-empty-induct*:
assumes *finite* A
and $P\ A$
and *remove*: $\bigwedge a\ A. \text{finite } A \implies a \in A \implies P\ A \implies P\ (A - \{a\})$
shows $P\ \{\}$
 $\langle proof \rangle$

lemma *finite-update-induct* [*consumes 1, case-names const update*]:
assumes *finite*: *finite* $\{a. f\ a \neq c\}$
and *const*: $P\ (\lambda a. c)$
and *update*: $\bigwedge a\ b\ f. \text{finite } \{a. f\ a \neq c\} \implies f\ a = c \implies b \neq c \implies P\ f \implies P\ (f(a := b))$
shows $P\ f$
 $\langle proof \rangle$

lemma *finite-subset-induct'* [*consumes 2, case-names empty insert*]:
assumes *finite* F **and** $F \subseteq A$
and *empty*: $P\ \{\}$
and *insert*: $\bigwedge a\ F. [\text{finite } F; a \in A; F \subseteq A; a \notin F; P\ F] \implies P\ (\text{insert } a\ F)$
shows $P\ F$
 $\langle proof \rangle$

20.4 Class *finite*

class *finite* =
assumes *finite-UNIV*: *finite* (*UNIV* :: 'a set)
begin

lemma *finite* [*simp*]: *finite* ($A :: 'a\ set$)
 $\langle proof \rangle$

lemma *finite-code* [*code*]: *finite* ($A :: 'a\ set$) $\longleftrightarrow True$
 $\langle proof \rangle$

end

instance *prod* :: (*finite*, *finite*) *finite*
 $\langle proof \rangle$

lemma *inj-graph*: $\text{inj } (\lambda f. \{(x, y). y = f x\})$
 $\langle \text{proof} \rangle$

instance *fun* :: (*finite*, *finite*) *finite*
 $\langle \text{proof} \rangle$

instance *bool* :: *finite*
 $\langle \text{proof} \rangle$

instance *set* :: (*finite*) *finite*
 $\langle \text{proof} \rangle$

instance *unit* :: *finite*
 $\langle \text{proof} \rangle$

instance *sum* :: (*finite*, *finite*) *finite*
 $\langle \text{proof} \rangle$

20.5 A basic fold functional for finite sets

The intended behaviour is $\text{fold } f \ z \ \{x_1, \dots, x_n\} = f \ x_1 \ (\dots (f \ x_n \ z) \dots)$ if f is “left-commutative”. The commutativity requirement is relativised to the carrier set S :

locale *comp-fun-commute-on* =
fixes $S :: 'a \text{ set}$
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b$
assumes *comp-fun-commute-on*: $x \in S \Longrightarrow y \in S \Longrightarrow f \ y \circ f \ x = f \ x \circ f \ y$
begin

lemma *fun-left-comm*: $x \in S \Longrightarrow y \in S \Longrightarrow f \ y \ (f \ x \ z) = f \ x \ (f \ y \ z)$
 $\langle \text{proof} \rangle$

lemma *commute-left-comp*: $x \in S \Longrightarrow y \in S \Longrightarrow f \ y \circ (f \ x \circ g) = f \ x \circ (f \ y \circ g)$
 $\langle \text{proof} \rangle$

end

inductive *fold-graph* :: ($'a \Rightarrow 'b \Rightarrow 'b$) $\Rightarrow 'b \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow \text{bool}$
for $f :: 'a \Rightarrow 'b \Rightarrow 'b$ **and** $z :: 'b$
where
 $\text{emptyI} \ [\text{intro}]: \text{fold-graph } f \ z \ \{\} \ z$
 $|\ \text{insertI} \ [\text{intro}]: x \notin A \Longrightarrow \text{fold-graph } f \ z \ A \ y \Longrightarrow \text{fold-graph } f \ z \ (\text{insert } x \ A) \ (f \ x \ y)$

inductive-cases *empty-fold-graphE* [*elim!*]: $\text{fold-graph } f \ z \ \{\} \ x$

lemma *fold-graph-closed-lemma*:
 $\text{fold-graph } f \ z \ A \ x \wedge x \in B$
if $\text{fold-graph } g \ z \ A \ x$

$\bigwedge a\ b. a \in A \implies b \in B \implies f\ a\ b = g\ a\ b$
 $\bigwedge a\ b. a \in A \implies b \in B \implies g\ a\ b \in B$
 $z \in B$
 <proof>

lemma *fold-graph-closed-eq*:

fold-graph $f\ z\ A = \text{fold-graph } g\ z\ A$
if $\bigwedge a\ b. a \in A \implies b \in B \implies f\ a\ b = g\ a\ b$
 $\bigwedge a\ b. a \in A \implies b \in B \implies g\ a\ b \in B$
 $z \in B$
 <proof>

definition *fold* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ \text{set} \Rightarrow 'b$
where *fold* $f\ z\ A = (\text{if finite } A \text{ then } (THE\ y. \text{fold-graph } f\ z\ A\ y) \text{ else } z)$

lemma *fold-closed-eq*: *fold* $f\ z\ A = \text{fold } g\ z\ A$

if $\bigwedge a\ b. a \in A \implies b \in B \implies f\ a\ b = g\ a\ b$
 $\bigwedge a\ b. a \in A \implies b \in B \implies g\ a\ b \in B$
 $z \in B$
 <proof>

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* $A \implies \exists x. \text{fold-graph } f\ z\ A\ x$
 <proof>

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
 <proof>

lemma *fold-graph-insertE-aux*:

assumes $A \subseteq S$
assumes *fold-graph* $f\ z\ A\ y\ a \in A$
shows $\exists y'. y = f\ a\ y' \wedge \text{fold-graph } f\ z\ (A - \{a\})\ y'$
 <proof>

lemma *fold-graph-insertE*:

assumes *insert* $x\ A \subseteq S$
assumes *fold-graph* $f\ z\ (\text{insert } x\ A)\ v$ **and** $x \notin A$
obtains y **where** $v = f\ x\ y$ **and** *fold-graph* $f\ z\ A\ y$
 <proof>

lemma *fold-graph-determ*:

assumes $A \subseteq S$

assumes *fold-graph* $f\ z\ A\ x$ *fold-graph* $f\ z\ A\ y$

shows $y = x$

<proof>

lemma *fold-equality*: $A \subseteq S \implies \text{fold-graph } f\ z\ A\ y \implies \text{fold } f\ z\ A = y$

<proof>

lemma *fold-graph-fold*:

assumes $A \subseteq S$

assumes *finite* A

shows *fold-graph* $f\ z\ A$ (*fold* $f\ z\ A$)

<proof>

The base case for *fold*:

lemma (**in** $-$) *fold-infinite* [*simp*]: $\neg \text{finite } A \implies \text{fold } f\ z\ A = z$

<proof>

lemma (**in** $-$) *fold-empty* [*simp*]: $\text{fold } f\ z\ \{\} = z$

<proof>

The various recursion equations for *fold*:

lemma *fold-insert* [*simp*]:

assumes *insert* $x\ A \subseteq S$

assumes *finite* A **and** $x \notin A$

shows *fold* $f\ z$ (*insert* $x\ A$) = $f\ x$ (*fold* $f\ z\ A$)

<proof>

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 $f\ x$ (*fold* $f\ z\ A$) = *fold* f ($f\ x\ z$) A

<proof>

lemma *fold-insert2*:

$\text{insert } x\ A \subseteq S \implies \text{finite } A \implies x \notin A \implies \text{fold } f\ z$ (*insert* $x\ A$) = *fold* f ($f\ x\ z$)

A

<proof>

lemma *fold-rec*:

assumes $A \subseteq S$

assumes *finite* A **and** $x \in A$

shows *fold* $f\ z\ A$ = $f\ x$ (*fold* $f\ z$ ($A - \{x\}$))

<proof>

lemma *fold-insert-remove*:

assumes *insert* $x \ A \subseteq S$

assumes *finite* A

shows $\text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ (A - \{x\}))$

<proof>

lemma *fold-set-union-disj*:

assumes $A \subseteq S \ B \subseteq S$

assumes *finite* A *finite* B $A \cap B = \{\}$

shows $\text{Finite-Set.fold } f \ z \ (A \cup B) = \text{Finite-Set.fold } f \ (\text{Finite-Set.fold } f \ z \ A) \ B$

<proof>

end

Other properties of *fold*:

lemma *finite-set-fold-single [simp]*: $\text{Finite-Set.fold } f \ z \ \{x\} = f \ x \ z$

<proof>

lemma *fold-graph-image*:

assumes *inj-on* $g \ A$

shows $\text{fold-graph } f \ z \ (g \ ` \ A) = \text{fold-graph } (f \circ g) \ z \ A$

<proof>

lemma *fold-image*:

assumes *inj-on* $g \ A$

shows $\text{fold } f \ z \ (g \ ` \ A) = \text{fold } (f \circ g) \ z \ A$

<proof>

lemma *fold-cong*:

assumes *comp-fun-commute-on* $S \ f \ \text{comp-fun-commute-on } S \ g$

and $A \subseteq S$ *finite* A

and *cong*: $\bigwedge x. x \in A \implies f \ x = g \ x$

and $s = t$ **and** $A = B$

shows $\text{fold } f \ s \ A = \text{fold } g \ t \ B$

<proof>

A simplified version for idempotent functions:

locale *comp-fun-idem-on* = *comp-fun-commute-on* +

assumes *comp-fun-idem-on*: $x \in S \implies f \ x \circ f \ x = f \ x$

begin

lemma *fun-left-idem*: $x \in S \implies f \ x \ (f \ x \ z) = f \ x \ z$

<proof>

lemma *fold-insert-idem*:

assumes *insert* $x \ A \subseteq S$

assumes *fin*: *finite* A

shows $\text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ A)$

$\langle \text{proof} \rangle$

declare *fold-insert* [*simp del*] *fold-insert-idem* [*simp*]

lemma *fold-insert-idem2*: $\text{insert } x \ A \subseteq S \implies \text{finite } A \implies \text{fold } f \ z \ (\text{insert } x \ A) = \text{fold } f \ (f \ x \ z) \ A$
 $\langle \text{proof} \rangle$

end

20.5.2 Liftings to *comp-fun-commute-on* etc.

lemma (**in** *comp-fun-commute-on*) *comp-comp-fun-commute-on*:
 $\text{range } g \subseteq S \implies \text{comp-fun-commute-on } R \ (f \circ g)$
 $\langle \text{proof} \rangle$

lemma (**in** *comp-fun-idem-on*) *comp-comp-fun-idem-on*:
assumes $\text{range } g \subseteq S$
shows $\text{comp-fun-idem-on } R \ (f \circ g)$
 $\langle \text{proof} \rangle$

lemma (**in** *comp-fun-commute-on*) *comp-fun-commute-on-funpow*:
 $\text{comp-fun-commute-on } S \ (\lambda x. f \ x \ \widehat{\sim} g \ x)$
 $\langle \text{proof} \rangle$

20.5.3 *UNIV* as carrier set

locale *comp-fun-commute* =
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b$
assumes *comp-fun-commute*: $f \ y \circ f \ x = f \ x \circ f \ y$
begin

lemma (**in** $-$) *comp-fun-commute-def'*: $\text{comp-fun-commute } f = \text{comp-fun-commute-on } UNIV \ f$
 $\langle \text{proof} \rangle$

We abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

sublocale *comp-fun-commute-on UNIV f*
rewrites $\bigwedge X. (X \subseteq UNIV) \equiv \text{True}$
and $\bigwedge x. x \in UNIV \equiv \text{True}$
and $\bigwedge P. (\text{True} \implies P) \equiv \text{Trueprop } P$
and $\bigwedge P \ Q. (\text{True} \implies \text{PROP } P \implies \text{PROP } Q) \equiv (\text{PROP } P \implies \text{True} \implies \text{PROP } Q)$
 $\langle \text{proof} \rangle$

end

lemma (**in** *comp-fun-commute*) *comp-comp-fun-commute*: $\text{comp-fun-commute } (f \circ g)$

<proof>

lemma (in *comp-fun-commute*) *comp-fun-commute-funpow*: *comp-fun-commute* ($\lambda x. f\ x \frown g\ x$)
<proof>

locale *comp-fun-idem* = *comp-fun-commute* +
assumes *comp-fun-idem*: $f\ x\ o\ f\ x = f\ x$
begin

lemma (in $-$) *comp-fun-idem-def'*: *comp-fun-idem* $f = \text{comp-fun-idem-on } UNIV\ f$
<proof>

Again, we abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

sublocale *comp-fun-idem-on UNIV f*
rewrites $\bigwedge X. (X \subseteq UNIV) \equiv \text{True}$
and $\bigwedge x. x \in UNIV \equiv \text{True}$
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>

end

lemma (in *comp-fun-idem*) *comp-comp-fun-idem*: *comp-fun-idem* ($f\ o\ g$)
<proof>

20.5.4 Expressing set operations via *fold*

lemma *comp-fun-commute-const*: *comp-fun-commute* ($\lambda-. f$)
<proof>

lemma *comp-fun-idem-insert*: *comp-fun-idem insert*
<proof>

lemma *comp-fun-idem-remove*: *comp-fun-idem Set.remove*
<proof>

lemma (in *semilattice-inf*) *comp-fun-idem-inf*: *comp-fun-idem inf*
<proof>

lemma (in *semilattice-sup*) *comp-fun-idem-sup*: *comp-fun-idem sup*
<proof>

lemma *union-fold-insert*:
assumes *finite A*
shows $A \cup B = \text{fold insert } B\ A$

$\langle proof \rangle$

lemma *minus-fold-remove*:

assumes *finite A*

shows $B - A = fold\ Set.remove\ B\ A$

$\langle proof \rangle$

lemma *comp-fun-commute-filter-fold*:

comp-fun-commute $(\lambda x\ A'.\ if\ P\ x\ then\ Set.insert\ x\ A'\ else\ A')$

$\langle proof \rangle$

lemma *Set-filter-fold*:

assumes *finite A*

shows $Set.filter\ P\ A = fold\ (\lambda x\ A'.\ if\ P\ x\ then\ Set.insert\ x\ A'\ else\ A')\ \{\}\ A$

$\langle proof \rangle$

lemma *inter-Set-filter*:

assumes *finite B*

shows $A \cap B = Set.filter\ (\lambda x.\ x \in A)\ B$

$\langle proof \rangle$

lemma *image-fold-insert*:

assumes *finite A*

shows $image\ f\ A = fold\ (\lambda k\ A.\ Set.insert\ (f\ k)\ A)\ \{\}\ A$

$\langle proof \rangle$

lemma *Ball-fold*:

assumes *finite A*

shows $Ball\ A\ P = fold\ (\lambda k\ s.\ s \wedge P\ k)\ True\ A$

$\langle proof \rangle$

lemma *Bex-fold*:

assumes *finite A*

shows $Bex\ A\ P = fold\ (\lambda k\ s.\ s \vee P\ k)\ False\ A$

$\langle proof \rangle$

lemma *comp-fun-commute-Pow-fold*: *comp-fun-commute* $(\lambda x\ A.\ A \cup Set.insert\ x\ A)$

$\langle proof \rangle$

lemma *Pow-fold*:

assumes *finite A*

shows $Pow\ A = fold\ (\lambda x\ A.\ A \cup Set.insert\ x\ A)\ \{\{\}\}\ A$

$\langle proof \rangle$

lemma *fold-union-pair*:

assumes *finite B*

shows $(\bigcup y \in B.\ \{(x, y)\}) \cup A = fold\ (\lambda y.\ Set.insert\ (x, y))\ A\ B$

$\langle proof \rangle$

lemma *comp-fun-commute-product-fold*:

finite B \implies *comp-fun-commute* ($\lambda x z. \text{fold } (\lambda y. \text{Set.insert } (x, y)) z B$)
 $\langle \text{proof} \rangle$

lemma *product-fold*:

assumes *finite A finite B*
shows $A \times B = \text{fold } (\lambda x z. \text{fold } (\lambda y. \text{Set.insert } (x, y)) z B) \{\} A$
 $\langle \text{proof} \rangle$

context *complete-lattice*
begin

lemma *inf-Inf-fold-inf*:

assumes *finite A*
shows $\text{inf } (\text{Inf } A) B = \text{fold inf } B A$
 $\langle \text{proof} \rangle$

lemma *sup-Sup-fold-sup*:

assumes *finite A*
shows $\text{sup } (\text{Sup } A) B = \text{fold sup } B A$
 $\langle \text{proof} \rangle$

lemma *Inf-fold-inf*: *finite A* \implies $\text{Inf } A = \text{fold inf top } A$
 $\langle \text{proof} \rangle$

lemma *Sup-fold-sup*: *finite A* \implies $\text{Sup } A = \text{fold sup bot } A$
 $\langle \text{proof} \rangle$

lemma *inf-INF-fold-inf*:

assumes *finite A*
shows $\text{inf } B (\bigsqcap (f \text{ ` } A)) = \text{fold } (\text{inf } \circ f) B A$ (**is** $?inf = ?fold$)
 $\langle \text{proof} \rangle$

lemma *sup-SUP-fold-sup*:

assumes *finite A*
shows $\text{sup } B (\bigsqcup (f \text{ ` } A)) = \text{fold } (\text{sup } \circ f) B A$ (**is** $?sup = ?fold$)
 $\langle \text{proof} \rangle$

lemma *INF-fold-inf*: *finite A* \implies $\bigsqcap (f \text{ ` } A) = \text{fold } (\text{inf } \circ f) \text{ top } A$
 $\langle \text{proof} \rangle$

lemma *SUP-fold-sup*: *finite A* \implies $\bigsqcup (f \text{ ` } A) = \text{fold } (\text{sup } \circ f) \text{ bot } A$
 $\langle \text{proof} \rangle$

lemma *finite-Inf-in*:

assumes *finite A* $A \neq \{\}$ **and** *inf*: $\bigwedge x y. \llbracket x \in A; y \in A \rrbracket \implies \text{inf } x y \in A$
shows $\text{Inf } A \in A$
 $\langle \text{proof} \rangle$

lemma *finite-Sup-in*:

assumes *finite A* $A \neq \{\}$ **and** *sup*: $\bigwedge x y. \llbracket x \in A; y \in A \rrbracket \implies \text{sup } x y \in A$

shows $\text{Sup } A \in A$

$\langle \text{proof} \rangle$

end

20.5.5 Expressing relation operations via *fold*

lemma *Id-on-fold*:

assumes *finite A*

shows $\text{Id-on } A = \text{Finite-Set.fold } (\lambda x. \text{Set.insert } (\text{Pair } x x)) \{\} A$

$\langle \text{proof} \rangle$

lemma *comp-fun-commute-Image-fold*:

comp-fun-commute $(\lambda(x,y) A. \text{if } x \in S \text{ then } \text{Set.insert } y A \text{ else } A)$

$\langle \text{proof} \rangle$

lemma *Image-fold*:

assumes *finite R*

shows $R \text{ “ } S = \text{Finite-Set.fold } (\lambda(x,y) A. \text{if } x \in S \text{ then } \text{Set.insert } y A \text{ else } A) \{\} R$

$\langle \text{proof} \rangle$

lemma *insert-relcomp-union-fold*:

assumes *finite S*

shows $\{x\} O S \cup X = \text{Finite-Set.fold } (\lambda(w,z) A'. \text{if } \text{snd } x = w \text{ then } \text{Set.insert } (\text{fst } x, z) A' \text{ else } A') X S$

$\langle \text{proof} \rangle$

lemma *insert-relcomp-fold*:

assumes *finite S*

shows $\text{Set.insert } x R O S =$

$\text{Finite-Set.fold } (\lambda(w,z) A'. \text{if } \text{snd } x = w \text{ then } \text{Set.insert } (\text{fst } x, z) A' \text{ else } A') (R O S) S$

$\langle \text{proof} \rangle$

lemma *comp-fun-commute-relcomp-fold*:

assumes *finite S*

shows *comp-fun-commute* $(\lambda(x,y) A.$

$\text{Finite-Set.fold } (\lambda(w,z) A'. \text{if } y = w \text{ then } \text{Set.insert } (x, z) A' \text{ else } A') A S)$

$\langle \text{proof} \rangle$

lemma *relcomp-fold*:

assumes *finite R finite S*

shows $R O S = \text{Finite-Set.fold}$

$(\lambda(x,y) A. \text{Finite-Set.fold } (\lambda(w,z) A'. \text{if } y = w \text{ then } \text{Set.insert } (x, z) A' \text{ else } A') A S) \{\} R$

$\langle proof \rangle$

20.6 Locales as mini-packages for fold operations

20.6.1 The natural case

locale *folding-on* =
 fixes $S :: 'a \text{ set}$
 fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b$ and $z :: 'b$
 assumes *comp-fun-commute-on*: $x \in S \Longrightarrow y \in S \Longrightarrow f\ y \circ f\ x = f\ x \circ f\ y$
begin

interpretation *fold?*: *comp-fun-commute-on* $S\ f$
 $\langle proof \rangle$

definition $F :: 'a \text{ set} \Rightarrow 'b$
 where *eq-fold*: $F\ A = \text{Finite-Set.fold}\ f\ z\ A$

lemma *empty* [*simp*]: $F\ \{\} = z$
 $\langle proof \rangle$

lemma *infinite* [*simp*]: $\neg \text{finite}\ A \Longrightarrow F\ A = z$
 $\langle proof \rangle$

lemma *insert* [*simp*]:
 assumes $\text{insert}\ x\ A \subseteq S$ and *finite* A and $x \notin A$
 shows $F\ (\text{insert}\ x\ A) = f\ x\ (F\ A)$
 $\langle proof \rangle$

lemma *remove*:
 assumes $A \subseteq S$ and *finite* A and $x \in A$
 shows $F\ A = f\ x\ (F\ (A - \{x\}))$
 $\langle proof \rangle$

lemma *insert-remove*:
 assumes $\text{insert}\ x\ A \subseteq S$ and *finite* A
 shows $F\ (\text{insert}\ x\ A) = f\ x\ (F\ (A - \{x\}))$
 $\langle proof \rangle$

end

20.6.2 With idempotency

locale *folding-idem-on* = *folding-on* +
 assumes *comp-fun-idem-on*: $x \in S \Longrightarrow y \in S \Longrightarrow f\ x \circ f\ x = f\ x$
begin

declare *insert* [*simp del*]

interpretation *fold?*: *comp-fun-idem-on* $S\ f$

$\langle proof \rangle$

lemma *insert-idem* [*simp*]:
assumes $insert\ x\ A \subseteq S$ **and** *finite* A
shows $F\ (insert\ x\ A) = f\ x\ (F\ A)$
 $\langle proof \rangle$

end

20.6.3 UNIV as the carrier set

locale *folding* =
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b$ **and** $z :: 'b$
assumes *comp-fun-commute*: $f\ y \circ f\ x = f\ x \circ f\ y$
begin

lemma (**in** $-$) *folding-def'*: $folding\ f = folding-on\ UNIV\ f$
 $\langle proof \rangle$

Again, we abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

sublocale *folding-on UNIV f*
rewrites $\bigwedge X. (X \subseteq UNIV) \equiv True$
and $\bigwedge x. x \in UNIV \equiv True$
and $\bigwedge P. (True \Longrightarrow P) \equiv Trueprop\ P$
and $\bigwedge P\ Q. (True \Longrightarrow PROP\ P \Longrightarrow PROP\ Q) \equiv (PROP\ P \Longrightarrow True \Longrightarrow PROP\ Q)$
 $\langle proof \rangle$

end

locale *folding-idem* = *folding* +
assumes *comp-fun-idem*: $f\ x \circ f\ x = f\ x$
begin

lemma (**in** $-$) *folding-idem-def'*: $folding-idem\ f = folding-idem-on\ UNIV\ f$
 $\langle proof \rangle$

Again, we abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

sublocale *folding-idem-on UNIV f*
rewrites $\bigwedge X. (X \subseteq UNIV) \equiv True$
and $\bigwedge x. x \in UNIV \equiv True$
and $\bigwedge P. (True \Longrightarrow P) \equiv Trueprop\ P$
and $\bigwedge P\ Q. (True \Longrightarrow PROP\ P \Longrightarrow PROP\ Q) \equiv (PROP\ P \Longrightarrow True \Longrightarrow PROP\ Q)$
 $\langle proof \rangle$

end

20.7 Finite cardinality

The traditional definition $\text{card } A \equiv \text{LEAST } n. \exists f. A = \{f \ i \mid i. i < n\}$ is ugly to work with. But now that we have *fold* things are easy:

global-interpretation *card*: *folding* $\lambda-. \text{Suc } 0$

defines $\text{card} = \text{folding-on.F } (\lambda-. \text{Suc}) \ 0$

<proof>

lemma *card-insert-disjoint*: $\text{finite } A \implies x \notin A \implies \text{card } (\text{insert } x \ A) = \text{Suc } (\text{card } A)$

<proof>

lemma *card-insert-if*: $\text{finite } A \implies \text{card } (\text{insert } x \ A) = (\text{if } x \in A \text{ then } \text{card } A \text{ else } \text{Suc } (\text{card } A))$

<proof>

lemma *card-ge-0-finite*: $\text{card } A > 0 \implies \text{finite } A$

<proof>

lemma *card-0-eq [simp]*: $\text{finite } A \implies \text{card } A = 0 \longleftrightarrow A = \{\}$

<proof>

lemma *finite-UNIV-card-ge-0*: $\text{finite } (\text{UNIV} :: 'a \text{ set}) \implies \text{card } (\text{UNIV} :: 'a \text{ set}) > 0$

<proof>

lemma *card-eq-0-iff*: $\text{card } A = 0 \longleftrightarrow A = \{\} \vee \neg \text{finite } A$

<proof>

lemma *card-range-greater-zero*: $\text{finite } (\text{range } f) \implies \text{card } (\text{range } f) > 0$

<proof>

lemma *card-gt-0-iff*: $0 < \text{card } A \longleftrightarrow A \neq \{\} \wedge \text{finite } A$

<proof>

lemma *card-Suc-Diff1*:

assumes $\text{finite } A \ x \in A$ **shows** $\text{Suc } (\text{card } (A - \{x\})) = \text{card } A$

<proof>

lemma *card-insert-le-m1*:

assumes $n > 0 \ \text{card } y \leq n - 1$ **shows** $\text{card } (\text{insert } x \ y) \leq n$

<proof>

lemma *card-Diff-singleton*:

assumes $x \in A$ **shows** $\text{card } (A - \{x\}) = \text{card } A - 1$

<proof>

lemma *card-Diff-singleton-if*:

$\text{card } (A - \{x\}) = (\text{if } x \in A \text{ then } \text{card } A - 1 \text{ else } \text{card } A)$

$\langle \text{proof} \rangle$

lemma *card-Diff-insert[simp]*:
 assumes $a \in A$ and $a \notin B$
 shows $\text{card } (A - \text{insert } a \ B) = \text{card } (A - B) - 1$
 $\langle \text{proof} \rangle$

lemma *card-insert-le*: $\text{card } A \leq \text{card } (\text{insert } x \ A)$
 $\langle \text{proof} \rangle$

lemma *card-Collect-less-nat[simp]*: $\text{card } \{i::\text{nat}. i < n\} = n$
 $\langle \text{proof} \rangle$

lemma *card-Collect-le-nat[simp]*: $\text{card } \{i::\text{nat}. i \leq n\} = \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *card-mono*:
 assumes *finite* B and $A \subseteq B$
 shows $\text{card } A \leq \text{card } B$
 $\langle \text{proof} \rangle$

lemma *card-seteq*:
 assumes *finite* B and $A: A \subseteq B \ \text{card } B \leq \text{card } A$
 shows $A = B$
 $\langle \text{proof} \rangle$

lemma *psubset-card-mono*: $\text{finite } B \implies A < B \implies \text{card } A < \text{card } B$
 $\langle \text{proof} \rangle$

lemma *card-Un-Int*:
 assumes *finite* A *finite* B
 shows $\text{card } A + \text{card } B = \text{card } (A \cup B) + \text{card } (A \cap B)$
 $\langle \text{proof} \rangle$

lemma *card-Un-disjoint*: $\text{finite } A \implies \text{finite } B \implies A \cap B = \{\} \implies \text{card } (A \cup B) = \text{card } A + \text{card } B$
 $\langle \text{proof} \rangle$

lemma *card-Un-disjnt*: $\llbracket \text{finite } A; \text{finite } B; \text{disjnt } A \ B \rrbracket \implies \text{card } (A \cup B) = \text{card } A + \text{card } B$
 $\langle \text{proof} \rangle$

lemma *card-Un-le*: $\text{card } (A \cup B) \leq \text{card } A + \text{card } B$
 $\langle \text{proof} \rangle$

lemma *card-Diff-subset*:
 assumes *finite* B
 and $B \subseteq A$
 shows $\text{card } (A - B) = \text{card } A - \text{card } B$

$\langle \text{proof} \rangle$

lemma *card-Diff-subset-Int*:

assumes *finite* $(A \cap B)$

shows $\text{card } (A - B) = \text{card } A - \text{card } (A \cap B)$

$\langle \text{proof} \rangle$

lemma *card-Int-Diff*:

assumes *finite* A

shows $\text{card } A = \text{card } (A \cap B) + \text{card } (A - B)$

$\langle \text{proof} \rangle$

lemma *diff-card-le-card-Diff*:

assumes *finite* B

shows $\text{card } A - \text{card } B \leq \text{card } (A - B)$

$\langle \text{proof} \rangle$

lemma *card-le-sym-Diff*:

assumes *finite* A *finite* B $\text{card } A \leq \text{card } B$

shows $\text{card}(A - B) \leq \text{card}(B - A)$

$\langle \text{proof} \rangle$

lemma *card-less-sym-Diff*:

assumes *finite* A *finite* B $\text{card } A < \text{card } B$

shows $\text{card}(A - B) < \text{card}(B - A)$

$\langle \text{proof} \rangle$

lemma *card-Diff1-less-iff*: $\text{card } (A - \{x\}) < \text{card } A \longleftrightarrow \text{finite } A \wedge x \in A$

$\langle \text{proof} \rangle$

lemma *card-Diff1-less*: $\text{finite } A \implies x \in A \implies \text{card } (A - \{x\}) < \text{card } A$

$\langle \text{proof} \rangle$

lemma *card-Diff2-less*:

assumes *finite* A $x \in A$ $y \in A$ **shows** $\text{card } (A - \{x\} - \{y\}) < \text{card } A$

$\langle \text{proof} \rangle$

lemma *card-Diff1-le*: $\text{card } (A - \{x\}) \leq \text{card } A$

$\langle \text{proof} \rangle$

lemma *card-psubset*: $\text{finite } B \implies A \subseteq B \implies \text{card } A < \text{card } B \implies A < B$

$\langle \text{proof} \rangle$

lemma *card-le-inj*:

assumes fA : *finite* A

and fB : *finite* B

and c : $\text{card } A \leq \text{card } B$

shows $\exists f. f ' A \subseteq B \wedge \text{inj-on } f A$

$\langle \text{proof} \rangle$

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>

lemma *insert-partition*:

$x \notin F \implies \forall c1 \in \text{insert } x F. \forall c2 \in \text{insert } x F. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\} \implies$
 $x \cap \bigcup F = \{\}$

<proof>

lemma *finite-psubset-induct* [*consumes 1, case-names psubset*]:

assumes *finite*: *finite A*

and *major*: $\bigwedge A. \text{finite } A \implies (\bigwedge B. B \subset A \implies P B) \implies P A$

shows $P A$

<proof>

lemma *finite-induct-select* [*consumes 1, case-names empty select*]:

assumes *finite S*

and $P \{\}$

and *select*: $\bigwedge T. T \subset S \implies P T \implies \exists s \in S - T. P (\text{insert } s T)$

shows $P S$

<proof>

lemma *remove-induct* [*case-names empty infinite remove*]:

assumes *empty*: $P (\{\} :: 'a \text{ set})$

and *infinite*: $\neg \text{finite } B \implies P B$

and *remove*: $\bigwedge A. \text{finite } A \implies A \neq \{\} \implies A \subseteq B \implies (\bigwedge x. x \in A \implies P (A - \{x\})) \implies P A$

shows $P B$

<proof>

lemma *finite-remove-induct* [*consumes 1, case-names empty remove*]:

fixes $P :: 'a \text{ set} \Rightarrow \text{bool}$

assumes *finite B*

and $P \{\}$

and $\bigwedge A. \text{finite } A \implies A \neq \{\} \implies A \subseteq B \implies (\bigwedge x. x \in A \implies P (A - \{x\})) \implies P A$

defines $B' \equiv B$

shows $P B'$

<proof>

Main cardinality theorem.

lemma *card-partition* [*rule-format*]:

$\text{finite } C \implies \text{finite } (\bigcup C) \implies (\forall c \in C. \text{card } c = k) \implies$
 $(\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}) \implies$
 $k * \text{card } C = \text{card } (\bigcup C)$

⟨proof⟩

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⟩

The form of a finite set of given cardinality

lemma *card-eq-SucD*:
assumes *card* *A* = *Suc* *k*
shows $\exists b \ B. \ A = \text{insert } b \ B \wedge b \notin B \wedge \text{card } B = k \wedge (k = 0 \longrightarrow B = \{\})$
 ⟨proof⟩

lemma *card-Suc-eq*:
 $\text{card } A = \text{Suc } k \longleftrightarrow$
 $(\exists b \ B. \ A = \text{insert } b \ B \wedge b \notin B \wedge \text{card } B = k \wedge (k = 0 \longrightarrow B = \{\}))$
 ⟨proof⟩

lemma *card-Suc-eq-finite*:
 $\text{card } A = \text{Suc } k \longleftrightarrow (\exists b \ B. \ A = \text{insert } b \ B \wedge b \notin B \wedge \text{card } B = k \wedge \text{finite } B)$
 ⟨proof⟩

lemma *card-1-singletonE*:
assumes *card* *A* = 1
obtains *x* **where** *A* = {*x*}
 ⟨proof⟩

lemma *is-singleton-iff-card-eq-Suc-0* [code]:
 $\langle \text{is-singleton } A \longleftrightarrow \text{card } A = \text{Suc } 0 \rangle$
 ⟨proof⟩

lemma *is-singleton-altdef*:
 $\langle \text{is-singleton } A \longleftrightarrow \text{card } A = 1 \rangle$
 ⟨proof⟩

lemma *card-eq-Suc-0-iff-is-singleton*:
 $\langle \text{card } A = \text{Suc } 0 \longleftrightarrow \text{is-singleton } A \rangle$
 ⟨proof⟩

lemma *card-1-singleton-iff*:
 $\langle \text{card } A = \text{Suc } 0 \longleftrightarrow (\exists x. \ A = \{x\}) \rangle$
 ⟨proof⟩

lemma *card-le-Suc0-iff-eq*:
assumes *finite* *A*
shows $\text{card } A \leq \text{Suc } 0 \longleftrightarrow (\forall a1 \in A. \ \forall a2 \in A. \ a1 = a2) \text{ (is } ?C = ?A)$
 ⟨proof⟩

lemma *card-le-Suc-iff*:

$Suc\ n \leq card\ A = (\exists a\ B. A = insert\ a\ B \wedge a \notin B \wedge n \leq card\ B \wedge finite\ B)$
 $\langle proof \rangle$

lemma *finite-fun-UNIVD2*:

assumes *fin*: $finite\ (UNIV :: ('a \Rightarrow 'b)\ set)$
shows $finite\ (UNIV :: 'b\ set)$
 $\langle proof \rangle$

lemma *card-UNIV-unit [simp]*: $card\ (UNIV :: unit\ set) = 1$
 $\langle proof \rangle$

lemma *infinite-arbitrarily-large*:

assumes $\neg finite\ A$
shows $\exists B. finite\ B \wedge card\ B = n \wedge B \subseteq A$
 $\langle proof \rangle$

corollary *finite-arbitrarily-large-disj*:

$\llbracket \neg finite\ (UNIV :: 'a\ set); finite\ (A :: 'a\ set) \rrbracket \implies \exists B. finite\ B \wedge card\ B = n \wedge A \cap B = \{\}$
 $\langle proof \rangle$

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 $\bigwedge G. G \subseteq F \implies finite\ G \implies card\ G \leq C$
shows $finite\ F \wedge card\ F \leq C$
 $\langle proof \rangle$

lemma *obtain-subset-with-card-n*:

assumes $n \leq card\ S$
obtains T **where** $T \subseteq S \wedge card\ T = n \wedge finite\ T$
 $\langle proof \rangle$

lemma *exists-subset-between*:

assumes
 $card\ A \leq n$
 $n \leq card\ C$
 $A \subseteq C$
 $finite\ C$
shows $\exists B. A \subseteq B \wedge B \subseteq C \wedge card\ B = n$
 $\langle proof \rangle$

20.7.1 Cardinality of image

lemma *card-image-le*: $finite\ A \implies card\ (f\ ` A) \leq card\ A$
 $\langle proof \rangle$

lemma *card-image*: $\text{inj-on } f \ A \implies \text{card } (f \text{ ‘ } A) = \text{card } A$
 ⟨proof⟩

lemma *bij-betw-same-card*: $\text{bij-betw } f \ A \ B \implies \text{card } A = \text{card } B$
 ⟨proof⟩

lemma *endo-inj-surj*: $\text{finite } A \implies f \text{ ‘ } A \subseteq A \implies \text{inj-on } f \ A \implies f \text{ ‘ } A = A$
 ⟨proof⟩

lemma *eq-card-imp-inj-on*:
 assumes $\text{finite } A \ \text{card}(f \text{ ‘ } A) = \text{card } A$
 shows $\text{inj-on } f \ A$
 ⟨proof⟩

lemma *inj-on-iff-eq-card*: $\text{finite } A \implies \text{inj-on } f \ A \longleftrightarrow \text{card } (f \text{ ‘ } A) = \text{card } A$
 ⟨proof⟩

lemma *card-inj-on-le*:
 assumes $\text{inj-on } f \ A \ f \text{ ‘ } A \subseteq B \ \text{finite } B$
 shows $\text{card } A \leq \text{card } B$
 ⟨proof⟩

lemma *inj-on-iff-card-le*:
 $\llbracket \text{finite } A; \text{finite } B \rrbracket \implies (\exists f. \text{inj-on } f \ A \wedge f \text{ ‘ } A \subseteq B) = (\text{card } A \leq \text{card } B)$
 ⟨proof⟩

lemma *surj-card-le*: $\text{finite } A \implies B \subseteq f \text{ ‘ } A \implies \text{card } B \leq \text{card } A$
 ⟨proof⟩

lemma *card-bij-eq*:
 $\text{inj-on } f \ A \implies f \text{ ‘ } A \subseteq B \implies \text{inj-on } g \ B \implies g \text{ ‘ } B \subseteq A \implies \text{finite } A \implies \text{finite } B$
 $\implies \text{card } A = \text{card } B$
 ⟨proof⟩

lemma *bij-betw-finite*: $\text{bij-betw } f \ A \ B \implies \text{finite } A \longleftrightarrow \text{finite } B$
 ⟨proof⟩

lemma *inj-on-finite*: $\text{inj-on } f \ A \implies f \text{ ‘ } A \subseteq B \implies \text{finite } B \implies \text{finite } A$
 ⟨proof⟩

lemma *card-vimage-inj-on-le*:
 assumes $\text{inj-on } f \ D \ \text{finite } A$
 shows $\text{card } (f \text{ ‘ } A \cap D) \leq \text{card } A$
 ⟨proof⟩

lemma *card-vimage-inj*: $\text{inj } f \implies A \subseteq \text{range } f \implies \text{card } (f \text{ ‘ } A) = \text{card } A$
 ⟨proof⟩

lemma *card-inverse[simp]*: $\text{card } (R^{-1}) = \text{card } R$

<proof>

20.7.2 Pigeonhole Principles

lemma *pigeonhole*: $\text{card } A > \text{card } (f \cdot A) \implies \neg \text{inj-on } f \ A$
<proof>

lemma *pigeonhole-infinite*:
assumes $\neg \text{finite } A$ **and** $\text{finite } (f \cdot A)$
shows $\exists a0 \in A. \neg \text{finite } \{a \in A. f \ a = f \ a0\}$
<proof>

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 $\exists b \in B. \neg \text{finite } \{a \in A. R \ a \ b\}$
<proof>

20.7.3 Cardinality of sums

lemma *card-Plus*:
assumes $\text{finite } A \ \text{finite } B$
shows $\text{card } (A <+> B) = \text{card } A + \text{card } B$
<proof>

lemma *card-Plus-conv-if*:
 $\text{card } (A <+> B) = (\text{if } \text{finite } A \wedge \text{finite } B \text{ then } \text{card } A + \text{card } B \text{ else } 0)$
<proof>

Relates to equivalence classes. Based on a theorem of F. Kammüller.

lemma *dvd-partition*:
assumes $f: \text{finite } (\bigcup C)$
and $\forall c \in C. k \ \text{dvd} \ \text{card } c \ \forall c1 \in C. \forall c2 \in C. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}$
shows $k \ \text{dvd} \ \text{card } (\bigcup C)$
<proof>

20.8 Minimal and maximal elements of finite sets

context begin

qualified lemma

assumes $\text{finite } A$ **and** $\text{asympt-on } A \ R$ **and** $\text{transp-on } A \ R$ **and** $\exists x \in A. P \ x$
shows
bex-min-element-with-property: $\exists x \in A. P \ x \wedge (\forall y \in A. R \ y \ x \longrightarrow \neg P \ y)$ **and**
bex-max-element-with-property: $\exists x \in A. P \ x \wedge (\forall y \in A. R \ x \ y \longrightarrow \neg P \ y)$
<proof> **lemma**
assumes $\text{finite } A$ **and** $\text{asympt-on } A \ R$ **and** $\text{transp-on } A \ R$ **and** $A \neq \{\}$
shows
bex-min-element: $\exists m \in A. \forall x \in A. x \neq m \longrightarrow \neg R \ x \ m$ **and**

bex-max-element: $\exists m \in A. \forall x \in A. x \neq m \longrightarrow \neg R\ m\ x$
 $\langle proof \rangle$

end

The following alternative form might sometimes be easier to work with.

lemma *is-min-element-in-set-iff*:

asympt-on $A\ R \implies (\forall y \in A. y \neq x \longrightarrow \neg R\ y\ x) \longleftrightarrow (\forall y. R\ y\ x \longrightarrow y \notin A)$
 $\langle proof \rangle$

lemma *is-max-element-in-set-iff*:

asympt-on $A\ R \implies (\forall y \in A. y \neq x \longrightarrow \neg R\ x\ y) \longleftrightarrow (\forall y. R\ x\ y \longrightarrow y \notin A)$
 $\langle proof \rangle$

context begin

qualified lemma

assumes *finite* A **and** $A \neq \{\}$ **and** *transp-on* $A\ R$ **and** *totalp-on* $A\ R$
shows

bex-least-element: $\exists l \in A. \forall x \in A. x \neq l \longrightarrow R\ l\ x$ **and**
bex-greatest-element: $\exists g \in A. \forall x \in A. x \neq g \longrightarrow R\ x\ g$
 $\langle proof \rangle$

end

20.8.1 Finite orders

context *order*

begin

lemma *finite-has-maximal*:

assumes *finite* A **and** $A \neq \{\}$
shows $\exists m \in A. \forall b \in A. m \leq b \longrightarrow m = b$
 $\langle proof \rangle$

lemma *finite-has-maximal2*:

$\llbracket \text{finite } A; a \in A \rrbracket \implies \exists m \in A. a \leq m \wedge (\forall b \in A. m \leq b \longrightarrow m = b)$
 $\langle proof \rangle$

lemma *finite-has-minimal*:

assumes *finite* A **and** $A \neq \{\}$
shows $\exists m \in A. \forall b \in A. b \leq m \longrightarrow m = b$
 $\langle proof \rangle$

lemma *finite-has-minimal2*:

$\llbracket \text{finite } A; a \in A \rrbracket \implies \exists m \in A. m \leq a \wedge (\forall b \in A. b \leq m \longrightarrow m = b)$
 $\langle proof \rangle$

end

20.8.2 Relating injectivity and surjectivity

lemma *finite-surj-inj*:

assumes *finite* $A \ A \subseteq f \text{ ' } A$

shows *inj-on* $f \ A$

$\langle \text{proof} \rangle$

lemma *finite-UNIV-surj-inj*: *finite*(*UNIV*:: 'a set) \implies *surj* $f \implies$ *inj* f

for $f :: 'a \Rightarrow 'a$

$\langle \text{proof} \rangle$

lemma *finite-UNIV-inj-surj*: *finite*(*UNIV*:: 'a set) \implies *inj* $f \implies$ *surj* f

for $f :: 'a \Rightarrow 'a$

$\langle \text{proof} \rangle$

lemma *surjective-iff-injective-gen*:

assumes fS : *finite* S

and fT : *finite* T

and c : *card* $S = \text{card } T$

and ST : $f \text{ ' } S \subseteq T$

shows $(\forall y \in T. \exists x \in S. f \ x = y) \longleftrightarrow \text{inj-on } f \ S$

(is ?lhs \longleftrightarrow ?rhs)

$\langle \text{proof} \rangle$

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 \Rightarrow bool

where *infinite* $S \equiv \neg \text{finite } S$

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)

$\langle \text{proof} \rangle$

lemma *infinite-UNIV-char-0*: *infinite* (*UNIV* :: 'a::semiring-char-0 set)

$\langle \text{proof} \rangle$

lemma *infinite-imp-nonempty*: *infinite* $S \implies S \neq \{\}$

$\langle \text{proof} \rangle$

lemma *infinite-remove*: *infinite* $S \implies \text{infinite } (S - \{a\})$

$\langle \text{proof} \rangle$

lemma *Diff-infinite-finite*:

assumes *finite* T *infinite* S
shows *infinite* $(S - T)$
 $\langle \text{proof} \rangle$

lemma *Un-infinite*: *infinite* $S \implies \text{infinite } (S \cup T)$
 $\langle \text{proof} \rangle$

lemma *infinite-Un*: *infinite* $(S \cup T) \longleftrightarrow \text{infinite } S \vee \text{infinite } T$
 $\langle \text{proof} \rangle$

lemma *infinite-super*:
assumes $S \subseteq T$
and *infinite* S
shows *infinite* T
 $\langle \text{proof} \rangle$

proposition *infinite-coinduct* [*consumes 1, case-names infinite*]:
assumes $X\ A$
and step: $\bigwedge A. X\ A \implies \exists x \in A. X\ (A - \{x\}) \vee \text{infinite } (A - \{x\})$
shows *infinite* A
 $\langle \text{proof} \rangle$

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 *inf-img-fin-dom'*:
assumes *img*: *finite* $(f \text{ ' } A)$
and *dom*: *infinite* A
shows $\exists y \in f \text{ ' } A. \text{infinite } (f - \text{' } \{y\} \cap A)$
 $\langle \text{proof} \rangle$

lemma *inf-img-fin-domE'*:
assumes *finite* $(f \text{ ' } A)$ **and** *infinite* A
obtains y **where** $y \in f \text{ ' } A$ **and** *infinite* $(f - \text{' } \{y\} \cap A)$
 $\langle \text{proof} \rangle$

lemma *inf-img-fin-dom*:
assumes *img*: *finite* $(f \text{ ' } A)$ **and** *dom*: *infinite* A
shows $\exists y \in f \text{ ' } A. \text{infinite } (f - \text{' } \{y\})$
 $\langle \text{proof} \rangle$

lemma *inf-img-fin-domE*:
assumes *finite* $(f \text{ ' } A)$ **and** *infinite* A
obtains y **where** $y \in f \text{ ' } A$ **and** *infinite* $(f - \text{' } \{y\})$
 $\langle \text{proof} \rangle$

proposition *finite-image-absD*: *finite* $(\text{abs } \text{' } S) \implies \text{finite } S$
for $S :: \text{' } a :: \text{linordered-ring set}$

$\langle proof \rangle$

20.10 The finite powerset operator

definition $Fpow :: 'a\ set \Rightarrow 'a\ set\ set$
where $Fpow\ A \equiv \{X. X \subseteq A \wedge finite\ X\}$

lemma $Fpow\ mono: A \subseteq B \Longrightarrow Fpow\ A \subseteq Fpow\ B$
 $\langle proof \rangle$

lemma $empty\ in\ Fpow: \{\} \in Fpow\ A$
 $\langle proof \rangle$

lemma $Fpow\ not\ empty: Fpow\ A \neq \{\}$
 $\langle proof \rangle$

lemma $Fpow\ subset\ Pow: Fpow\ A \subseteq Pow\ A$
 $\langle proof \rangle$

lemma $Fpow\ Pow\ finite: Fpow\ A = Pow\ A\ Int\ \{A. finite\ A\}$
 $\langle proof \rangle$

lemma $inj\ on\ image\ Fpow:$
assumes $inj\ on\ f\ A$
shows $inj\ on\ (image\ f)\ (Fpow\ A)$
 $\langle proof \rangle$

lemma $image\ Fpow\ mono:$
assumes $f\ 'A \subseteq B$
shows $(image\ f)\ ' (Fpow\ A) \subseteq Fpow\ B$
 $\langle proof \rangle$

end

21 Reflexive and Transitive closure of a relation

theory *Transitive-Closure*
imports *Finite-Set*
abbrevs $\hat{*} = * **$
and $\hat{+} = + ++$
and $\hat{=} = = ==$
begin

$\langle ML \rangle$

$rtrancl$ 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 $[[\text{inductive-internals}]]$
begin

inductive-set $\text{rtranc1} :: ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set} \ (\langle \langle \text{notation} = \langle \text{postfix } * \rangle \rangle -^* \rangle$
 $[1000] \ 999)$
for $r :: ('a \times 'a) \text{ set}$
where
 $\text{rtranc1-refl} [\text{intro!}, \text{Pure.intro!}, \text{simp}]: (a, a) \in r^*$
 $| \text{rtranc1-into-rtranc1} [\text{Pure.intro}]: (a, b) \in r^* \Longrightarrow (b, c) \in r \Longrightarrow (a, c) \in r^*$

inductive-set $\text{tranc1} :: ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set} \ (\langle \langle \text{notation} = \langle \text{postfix } + \rangle \rangle -^+ \rangle$
 $[1000] \ 999)$
for $r :: ('a \times 'a) \text{ set}$
where
 $\text{r-into-tranc1} [\text{intro}, \text{Pure.intro}]: (a, b) \in r \Longrightarrow (a, b) \in r^+$
 $| \text{tranc1-into-tranc1} [\text{Pure.intro}]: (a, b) \in r^+ \Longrightarrow (b, c) \in r \Longrightarrow (a, c) \in r^+$

notation
 $\text{rtranc1p} \ (\langle \langle \text{notation} = \langle \text{postfix } * \rangle \rangle -^{**} \rangle [1000] \ 1000) \text{ and}$
 $\text{tranc1p} \ (\langle \langle \text{notation} = \langle \text{postfix } + \rangle \rangle -^{++} \rangle [1000] \ 1000)$

declare
 $\text{rtranc1-def} [\text{nitpick-unfold del}]$
 $\text{rtranc1p-def} [\text{nitpick-unfold del}]$
 $\text{tranc1-def} [\text{nitpick-unfold del}]$
 $\text{tranc1p-def} [\text{nitpick-unfold del}]$

end

lemma $\text{tranc1-incr}: r \subseteq r^+$
 $\langle \text{proof} \rangle$

abbreviation $\text{reflcl} :: ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set} \ (\langle \langle \text{notation} = \langle \text{postfix } = \rangle \rangle -^= \rangle$
 $[1000] \ 999)$
where $r^= \equiv r \cup \text{Id}$

abbreviation $\text{reflclp} :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool} \ (\langle \langle \text{notation} = \langle \text{postfix } == \rangle \rangle -^{==} \rangle$
 $[1000] \ 1000)$
where $r^{==} \equiv \text{sup } r \ (=)$

notation (*ASCII*)
 $\text{rtranc1} \ (\langle \langle \text{notation} = \langle \text{postfix } * \rangle \rangle -^{\wedge *} \rangle [1000] \ 999) \text{ and}$
 $\text{tranc1} \ (\langle \langle \text{notation} = \langle \text{postfix } + \rangle \rangle -^{\wedge +} \rangle [1000] \ 999) \text{ and}$
 $\text{reflcl} \ (\langle \langle \text{notation} = \langle \text{postfix } = \rangle \rangle -^{\wedge =} \rangle [1000] \ 999) \text{ and}$
 $\text{rtranc1p} \ (\langle \langle \text{notation} = \langle \text{postfix } * \rangle \rangle -^{\wedge **} \rangle [1000] \ 1000) \text{ and}$
 $\text{tranc1p} \ (\langle \langle \text{notation} = \langle \text{postfix } + \rangle \rangle -^{\wedge ++} \rangle [1000] \ 1000) \text{ and}$
 $\text{reflclp} \ (\langle \langle \text{notation} = \langle \text{postfix } == \rangle \rangle -^{\wedge ==} \rangle [1000] \ 1000)$

bundle rtranc1-syntax

```

begin
notation
  rtranc1 (⟨(⟨notation=⟨postfix *⟩⟩-*)⟩ [1000] 999) and
  rtranc1p (⟨(⟨notation=⟨postfix **⟩⟩-**)⟩ [1000] 1000)
notation (ASCII)
  rtranc1 (⟨(⟨notation=⟨postfix *⟩⟩-^*)⟩ [1000] 999) and
  rtranc1p (⟨(⟨notation=⟨postfix **⟩⟩-^**)⟩ [1000] 1000)
end

bundle tranc1-syntax
begin
notation
  tranc1 (⟨(⟨notation=⟨postfix +⟩⟩-+)⟩ [1000] 999) and
  tranc1p (⟨(⟨notation=⟨postfix ++⟩⟩-++)⟩ [1000] 1000)
notation (ASCII)
  tranc1 (⟨(⟨notation=⟨postfix +⟩⟩-^+)⟩ [1000] 999) and
  tranc1p (⟨(⟨notation=⟨postfix ++⟩⟩-^++)⟩ [1000] 1000)
end

bundle reflcl-syntax
begin
notation
  reflcl (⟨(⟨notation=⟨postfix ==⟩⟩-)=⟩ [1000] 999) and
  reflclp (⟨(⟨notation=⟨postfix ==⟩⟩-^==)⟩ [1000] 1000)
notation (ASCII)
  reflcl (⟨(⟨notation=⟨postfix ==⟩⟩-^=)⟩ [1000] 999) and
  reflclp (⟨(⟨notation=⟨postfix ==⟩⟩-^==)⟩ [1000] 1000)
end

```

21.1 Reflexive closure

lemma *reflcl-set-eq* [*pred-set-conv*]: $(\sup (\lambda x y. (x, y) \in r) (=)) = (\lambda x y. (x, y) \in r \cup Id)$
 ⟨*proof*⟩

lemma *refl-reflcl*[*simp*]: $refl (r^=)$
 ⟨*proof*⟩

lemma *reflp-on-reflclp*[*simp*]: $reflp\text{-}on\ A\ R^{==}$
 ⟨*proof*⟩

lemma *antisym-on-reflcl*[*simp*]: $antisym\text{-}on\ A\ (r^=) \longleftrightarrow antisym\text{-}on\ A\ r$
 ⟨*proof*⟩

lemma *antisym-on-reflclp*[*simp*]: $antisym\text{-}on\ A\ R^{==} \longleftrightarrow antisym\text{-}on\ A\ R$
 ⟨*proof*⟩

lemma *trans-on-reflcl*[*simp*]: $trans\text{-}on\ A\ r \implies trans\text{-}on\ A\ (r^=)$
 ⟨*proof*⟩

lemma *transp-on-reflclp[simp]*: $\text{transp-on } A \ R \implies \text{transp-on } A \ R^{==}$
 ⟨proof⟩

lemma *antisymp-on-reflclp-if-asymp-on*:
 assumes *asymp-on* $A \ R$
 shows *antisymp-on* $A \ R^{==}$
 ⟨proof⟩

lemma *antisym-on-reflcl-if-asymp-on*: $\text{asym-on } A \ R \implies \text{antisym-on } A \ (R^=)$
 ⟨proof⟩

lemma *reflclp-idemp [simp]*: $(P^{==})^{==} = P^{==}$
 ⟨proof⟩

lemma *reflclp-ident-if-reflp[simp]*: $\text{reflp } R \implies R^{==} = R$
 ⟨proof⟩

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]*: $(\leq)^{==} = (\leq)$
 ⟨proof⟩

lemma (in *preorder*) *reflclp-greater-eq[simp]*: $(\geq)^{==} = (\geq)$
 ⟨proof⟩

lemma *order-reflclp-if-transp-and-asymp*:
 assumes *transp* R and *asymp* R
 shows *class.order* $R^{==} \ R$
 ⟨proof⟩

21.2 Reflexive-transitive closure

lemma *r-into-rtrancl [intro]*: $\bigwedge p. p \in r \implies p \in r^*$
 — *rtrancl* of r contains r
 ⟨proof⟩

lemma *r-into-rtranclp [intro]*: $r \ x \ y \implies r^{**} \ x \ y$
 — *rtrancl* of r contains r
 ⟨proof⟩

lemma *rtranclp-mono*: $r \leq s \implies r^{**} \leq s^{**}$
 — monotonicity of *rtrancl*
 ⟨proof⟩

lemma *mono-rtranclp[mono]*: $(\bigwedge a \ b. x \ a \ b \longrightarrow y \ a \ b) \implies x^{**} \ a \ b \longrightarrow y^{**} \ a \ b$
 ⟨proof⟩

lemmas *rtrancl-mono = rtranclp-mono [to-set]*

theorem *rtrancp-induct* [*consumes 1, case-names base step, induct set: rtrancp*]:
assumes $a: r^{**} a b$
and cases: $P a \bigwedge y z. r^{**} a y \implies r y z \implies P y \implies P z$
shows $P b$
 $\langle \text{proof} \rangle$

lemmas *rtranc-induct* [*induct set: rtranc*] = *rtrancp-induct* [*to-set*]

lemmas *rtrancp-induct2* =
rtrancp-induct[*of - (ax,ay) (bx,by), split-rule, consumes 1, case-names refl step*]

lemmas *rtranc-induct2* =
rtranc-induct[*of (ax,ay) (bx,by), split-format (complete), consumes 1, case-names refl step*]

lemma *refl-rtranc*: *refl* (r^*)
 $\langle \text{proof} \rangle$

Transitivity of transitive closure.

lemma *trans-rtranc*: *trans* (r^*)
 $\langle \text{proof} \rangle$

lemmas *rtranc-trans* = *trans-rtranc* [*THEN transD*]

lemma *rtrancp-trans*:
assumes $r^{**} x y$
and $r^{**} y z$
shows $r^{**} x z$
 $\langle \text{proof} \rangle$

lemma *rtrancIE* [*cases set: rtranc*]:
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 *rtranc* – by induction on a special formula
 $\langle \text{proof} \rangle$

lemma *rtranc-Int-subset*: $Id \subseteq s \implies (r^* \cap s) \circ r \subseteq s \implies r^* \subseteq s$
 $\langle \text{proof} \rangle$

lemma *converse-rtrancp-into-rtrancp*: $r a b \implies r^{**} b c \implies r^{**} a c$
 $\langle \text{proof} \rangle$

lemmas *converse-rtranc-into-rtranc* = *converse-rtrancp-into-rtrancp* [*to-set*]

More r^* equations and inclusions.

lemma *rtranclp-idemp* [simp]: $(r^{**})^{**} = r^{**}$
 ⟨proof⟩

lemmas *rtrancl-idemp* [simp] = *rtranclp-idemp* [to-set]

lemma *rtrancl-idemp-self-comp* [simp]: $R^* \circ R^* = R^*$
 ⟨proof⟩

lemma *rtrancl-subset-rtrancl*: $r \subseteq s^* \implies r^* \subseteq s^*$
 ⟨proof⟩

lemma *rtranclp-subset*: $R \leq S \implies S \leq R^{**} \implies S^{**} = R^{**}$
 ⟨proof⟩

lemmas *rtrancl-subset* = *rtranclp-subset* [to-set]

lemma *rtranclp-sup-rtranclp*: $(\sup (R^{**}) (S^{**}))^{**} = (\sup R S)^{**}$
 ⟨proof⟩

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [to-set]

lemma *rtranclp-reflclp* [simp]: $(R^=)^{**} = R^{**}$
 ⟨proof⟩

lemmas *rtrancl-reflcl* [simp] = *rtranclp-reflclp* [to-set]

lemma *rtrancl-r-diff-Id*: $(r - Id)^* = r^*$
 ⟨proof⟩

lemma *rtranclp-r-diff-Id*: $(\inf r (\neq))^{**} = r^{**}$
 ⟨proof⟩

theorem *rtranclp-converseD*:
 assumes $(r^{-1-1})^{**} x y$
 shows $r^{**} y x$
 ⟨proof⟩

lemmas *rtrancl-converseD* = *rtranclp-converseD* [to-set]

theorem *rtranclp-converseI*:
 assumes $r^{**} y x$
 shows $(r^{-1-1})^{**} x y$
 ⟨proof⟩

lemmas *rtrancl-converseI* = *rtranclp-converseI* [to-set]

lemma *rtrancl-converse*: $(r^{-1})^* = (r^*)^{-1}$
 ⟨proof⟩

lemma *sym-rtrancl*: $\text{sym } r \implies \text{sym } (r^*)$
 $\langle \text{proof} \rangle$

theorem *converse-rtranclp-induct* [consumes 1, case-names base step]:
 assumes major: $r^{**} a b$
 and cases: $P b \bigwedge y z. r y z \implies r^{**} z b \implies P z \implies P y$
 shows $P a$
 $\langle \text{proof} \rangle$

lemmas *converse-rtrancl-induct* = *converse-rtranclp-induct* [to-set]

lemmas *converse-rtranclp-induct2* =
converse-rtranclp-induct [of - (ax, ay) (bx, by), split-rule, consumes 1, case-names
 refl step]

lemmas *converse-rtrancl-induct2* =
converse-rtrancl-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: $x = z \implies P \bigwedge y. r x y \implies r^{**} y z \implies P$
 shows P
 $\langle \text{proof} \rangle$

lemmas *converse-rtranclE* = *converse-rtranclpE* [to-set]

lemmas *converse-rtranclpE2* = *converse-rtranclpE* [of - (xa,xb) (za,zb), split-rule]

lemmas *converse-rtranclE2* = *converse-rtranclE* [of (xa,xb) (za,zb), split-rule]

lemma *r-comp-rtrancl-eq*: $r \circ r^* = r^* \circ r$
 $\langle \text{proof} \rangle$

lemma *rtrancl-unfold*: $r^* = \text{Id} \cup r^* \circ r$
 $\langle \text{proof} \rangle$

lemma *rtrancl-Un-separatorE*:
 $(a, b) \in (P \cup Q)^* \implies \forall x y. (a, x) \in P^* \longrightarrow (x, y) \in Q \longrightarrow x = y \implies (a, b) \in P^*$
 $\langle \text{proof} \rangle$

lemma *rtrancl-Un-separator-converseE*:
 $(a, b) \in (P \cup Q)^* \implies \forall x y. (x, b) \in P^* \longrightarrow (y, x) \in Q \longrightarrow y = x \implies (a, b) \in P^*$
 $\langle \text{proof} \rangle$

lemma *Image-closed-trancl*:
 assumes $r \text{ `` } X \subseteq X$

shows r^* “ $X = X$ ”
 $\langle \text{proof} \rangle$

lemma *rtranclp-ident-if-reflp-and-transp*:
assumes *reflp* R **and** *transp* R
shows $R^{**} = R$
 $\langle \text{proof} \rangle$

The following are special cases of *rtranclp-ident-if-reflp-and-transp*, but they appear duplicated in multiple, independent theories, which causes name clashes.

lemma (**in** *preorder*) *rtranclp-less-eq[simp]*: $(\leq)^{**} = (\leq)$
 $\langle \text{proof} \rangle$

lemma (**in** *preorder*) *rtranclp-greater-eq[simp]*: $(\geq)^{**} = (\geq)$
 $\langle \text{proof} \rangle$

21.3 Transitive closure

lemma *totalp-on-tranclp*: *totalp-on* A $R \implies \text{totalp-on } A \text{ (tranclp } R)$
 $\langle \text{proof} \rangle$

lemma *total-on-trancl*: *total-on* A $r \implies \text{total-on } A \text{ (trancl } r)$
 $\langle \text{proof} \rangle$

lemma *trancl-mono*:
assumes $p \in r^+$ $r \subseteq s$
shows $p \in s^+$
 $\langle \text{proof} \rangle$

lemma *trancl-mono-subset*: $A \subseteq B \implies A^{\wedge+} \subseteq B^{\wedge+}$
 $\langle \text{proof} \rangle$

lemma *r-into-trancl'*: $\bigwedge p. p \in r \implies p \in r^+$
 $\langle \text{proof} \rangle$

Conversions between *trancl* and *rtrancl*.

lemma *tranclp-into-rtranclp*: $r^{++} a b \implies r^{**} a b$
 $\langle \text{proof} \rangle$

lemmas *trancl-into-rtrancl* = *tranclp-into-rtranclp* [to-set]

lemma *rtranclp-into-tranclp1*:
assumes $r^{**} a b$
shows $r b c \implies r^{++} a c$
 $\langle \text{proof} \rangle$

lemmas *rtrancl-into-trancl1* = *rtranclp-into-tranclp1* [to-set]

lemma *rtrancp-into-trancp2*:

assumes $r\ a\ b\ r^{**}\ b\ c$ **shows** $r^{++}\ a\ c$

— intro rule from r and *rtranc*

<proof>

lemmas *rtranc-into-tranc2* = *rtrancp-into-trancp2* [*to-set*]

Nice induction rule for *tranc*

lemma *trancp-induct* [*consumes 1, case-names base step, induct pred: trancp*]:

assumes $a: r^{++}\ a\ b$

and cases: $\bigwedge y. r\ a\ y \implies P\ y \bigwedge y\ z. r^{++}\ a\ y \implies r\ y\ z \implies P\ y \implies P\ z$

shows $P\ b$

<proof>

lemmas *tranc-induct* [*induct set: tranc*] = *trancp-induct* [*to-set*]

lemmas *trancp-induct2* =

trancp-induct [*of - (ax, ay) (bx, by), split-rule, consumes 1, case-names base step*]

lemmas *tranc-induct2* =

tranc-induct [*of (ax, ay) (bx, by), split-format (complete), consumes 1, case-names base step*]

lemma *trancp-trans-induct*:

assumes *major:* $r^{++}\ x\ y$

and cases: $\bigwedge x\ y. r\ x\ y \implies P\ x\ y \bigwedge x\ y\ z. r^{++}\ x\ y \implies P\ x\ y \implies r^{++}\ y\ z \implies P\ y\ z \implies P\ x\ z$

shows $P\ x\ y$

— Another induction rule for *tranc*, incorporating transitivity

<proof>

lemmas *tranc-trans-induct* = *trancp-trans-induct* [*to-set*]

lemma *trancE* [*cases set: tranc*]:

assumes $(a, b) \in r^+$

obtains

(*base*) $(a, b) \in r$

| (*step*) c **where** $(a, c) \in r^+$ **and** $(c, b) \in r$

<proof>

lemma *tranc-Int-subset*: $r \subseteq s \implies (r^+ \cap s) \subseteq r \subseteq s \implies r^+ \subseteq s$

<proof>

lemma *tranc-unfold*: $r^+ = r \cup r^+ \subseteq r$

<proof>

Transitivity of r^+

lemma *trans-trancl* [*simp*]: $\text{trans } (r^+)$
 $\langle \text{proof} \rangle$

lemmas *trancl-trans* = *trans-trancl* [*THEN transD*]

lemma *tranclp-trans*:
 assumes $r^{++} x y$
 and $r^{++} y z$
 shows $r^{++} x z$
 $\langle \text{proof} \rangle$

lemma *trancl-id* [*simp*]: $\text{trans } r \implies r^+ = r$
 $\langle \text{proof} \rangle$

lemma *rtranclp-tranclp-tranclp*:
 assumes $r^{**} x y$
 shows $\bigwedge z. r^{++} y z \implies r^{++} x z$
 $\langle \text{proof} \rangle$

lemmas *rtrancl-trancl-trancl* = *rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: $r a b \implies r^{++} b c \implies r^{++} a c$
 $\langle \text{proof} \rangle$

lemmas *trancl-into-trancl2* = *tranclp-into-tranclp2* [*to-set*]

lemma *trancl-trancl-Un*: $(A^{\hat{+}} \cup B)^{\hat{+}} = (A \cup B)^{\hat{+}}$
 $\langle \text{proof} \rangle$

lemma *trancl-absorb-subset-trancl*: $B \subseteq A^{\hat{+}} \implies (A \cup B)^{\hat{+}} = A^{\hat{+}}$
 $\langle \text{proof} \rangle$

lemma *tranclp-converseI*:
 assumes $(r^{++})^{-1-1} x y$ shows $(r^{-1-1})^{++} x y$
 $\langle \text{proof} \rangle$

lemmas *trancl-converseI* = *tranclp-converseI* [*to-set*]

lemma *tranclp-converseD*:
 assumes $(r^{-1-1})^{++} x y$ shows $(r^{++})^{-1-1} x y$
 $\langle \text{proof} \rangle$

lemmas *trancl-converseD* = *tranclp-converseD* [*to-set*]

lemma *tranclp-converse*: $(r^{-1-1})^{++} = (r^{++})^{-1-1}$
 $\langle \text{proof} \rangle$

lemmas *trancl-converse* = *tranclp-converse* [*to-set*]

lemma *sym-trancl*: $\text{sym } r \implies \text{sym } (r^+)$
 $\langle \text{proof} \rangle$

lemma *converse-tranclp-induct* [*consumes 1, case-names base step*]:
assumes *major*: $r^{++} a b$
and cases: $\bigwedge y. r y b \implies P y \bigwedge y z. r y z \implies r^{++} z b \implies P z \implies P y$
shows $P a$
 $\langle \text{proof} \rangle$

lemmas *converse-trancl-induct* = *converse-tranclp-induct* [*to-set*]

lemma *tranclpD*: $R^{++} x y \implies \exists z. R x z \wedge R^{**} z y$
 $\langle \text{proof} \rangle$

lemmas *tranclD* = *tranclpD* [*to-set*]

lemma *converse-tranclpE*:
assumes *major*: *tranclp* $r x z$
and base: $r x z \implies P$
and step: $\bigwedge y. r x y \implies \text{tranclp } r y z \implies P$
shows P
 $\langle \text{proof} \rangle$

lemmas *converse-tranclE* = *converse-tranclpE* [*to-set*]

lemma *tranclD2*: $(x, y) \in R^+ \implies \exists z. (x, z) \in R^* \wedge (z, y) \in R$
 $\langle \text{proof} \rangle$

lemma *irrefl-tranclI*: $r^{-1} \cap r^* = \{\}$ $\implies (x, x) \notin r^+$
 $\langle \text{proof} \rangle$

lemma *irrefl-trancl-rD*: $\forall x. (x, x) \notin r^+ \implies (x, y) \in r \implies x \neq y$
 $\langle \text{proof} \rangle$

lemma *trancl-subset-Sigma-aux*: $(a, b) \in r^* \implies r \subseteq A \times A \implies a = b \vee a \in A$
 $\langle \text{proof} \rangle$

lemma *trancl-subset-Sigma*:
assumes $r \subseteq A \times A$ **shows** $r^+ \subseteq A \times A$
 $\langle \text{proof} \rangle$

lemma *reflclp-tranclp* [*simp*]: $(r^{++})^{==} = r^{**}$
 $\langle \text{proof} \rangle$

lemmas *reflcl-trancl* [*simp*] = *reflclp-tranclp* [*to-set*]

lemma *trancl-reflcl* [*simp*]: $(r^=)^+ = r^*$
 $\langle \text{proof} \rangle$

lemma *rtrancl-trancl-reflcl* [code]: $r^* = (r^+)^=$
 ⟨proof⟩

lemma *trancl-empty* [simp]: $\{\}^+ = \{\}$
 ⟨proof⟩

lemma *rtrancl-empty* [simp]: $\{\}^* = Id$
 ⟨proof⟩

lemma *rtrancl--Id*[simp]: $Id^* = Id$
 ⟨proof⟩

lemma *rtranclpD*: $R^{**} a b \implies a = b \vee a \neq b \wedge R^{++} a b$
 ⟨proof⟩

lemmas *rtranclD* = *rtranclpD* [to-set]

lemma *rtrancl-eq-or-trancl*: $(x, y) \in R^* \longleftrightarrow x = y \vee x \neq y \wedge (x, y) \in R^+$
 ⟨proof⟩

lemma *trancl-unfold-right*: $r^+ = r^* O r$
 ⟨proof⟩

lemma *trancl-unfold-left*: $r^+ = r O r^*$
 ⟨proof⟩

lemma *tranclp-unfold-left*: $r^{\wedge++} = r OO r^{\wedge**}$
 ⟨proof⟩

lemma *trancl-insert*: $(insert (y, x) r)^+ = r^+ \cup \{(a, b). (a, y) \in r^* \wedge (x, b) \in r^*\}$
 — primitive recursion for *trancl* over finite relations
 ⟨proof⟩

lemma *trancl-insert2*:
 $(insert (a, b) r)^+ = r^+ \cup \{(x, y). ((x, a) \in r^+ \vee x = a) \wedge ((b, y) \in r^+ \vee y = b)\}$
 ⟨proof⟩

lemma *rtrancl-insert*: $(insert (a, b) r)^* = r^* \cup \{(x, y). (x, a) \in r^* \wedge (b, y) \in r^*\}$
 ⟨proof⟩

Simplifying nested closures

lemma *rtrancl-trancl-absorb*[simp]: $(R^*)^+ = R^*$
 ⟨proof⟩

lemma *trancl-rtrancl-absorb*[simp]: $(R^+)^* = R^*$
 ⟨proof⟩

lemma *rtrancl-reflcl-absorb*[simp]: $(R^*)^= = R^*$

$\langle \text{proof} \rangle$

Domain and Range

lemma *Domain-rtranc1* [simp]: $\text{Domain } (R^*) = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *Range-rtranc1* [simp]: $\text{Range } (R^*) = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *rtranc1-Un-subset*: $(R^* \cup S^*) \subseteq (R \cup S)^*$
 $\langle \text{proof} \rangle$

lemma *in-rtranc1-UnI*: $x \in R^* \vee x \in S^* \implies x \in (R \cup S)^*$
 $\langle \text{proof} \rangle$

lemma *tranc1-domain* [simp]: $\text{Domain } (r^+) = \text{Domain } r$
 $\langle \text{proof} \rangle$

lemma *tranc1-range* [simp]: $\text{Range } (r^+) = \text{Range } r$
 $\langle \text{proof} \rangle$

lemma *Not-Domain-rtranc1*:
assumes $x \notin \text{Domain } R$ **shows** $(x, y) \in R^* \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *tranc1-subset-Field2*: $r^+ \subseteq \text{Field } r \times \text{Field } r$
 $\langle \text{proof} \rangle$

lemma *finite-tranc1*[simp]: $\text{finite } (r^+) = \text{finite } r$
 $\langle \text{proof} \rangle$

lemma *finite-rtranc1-Image*[simp]: **assumes** $\text{finite } R$ $\text{finite } A$ **shows** $\text{finite } (R^* \text{ `` } A)$
 $\langle \text{proof} \rangle$

More about converse *rtranc1* and *tranc1*, should be merged with main body.

lemma *single-valued-confluent*:
assumes *single-valued* r **and** $xy: (x, y) \in r^*$ **and** $xz: (x, z) \in r^*$
shows $(y, z) \in r^* \vee (z, y) \in r^*$
 $\langle \text{proof} \rangle$

lemma *r-r-into-tranc1*: $(a, b) \in R \implies (b, c) \in R \implies (a, c) \in R^+$
 $\langle \text{proof} \rangle$

lemma *tranc1-into-tranc1*: $(a, b) \in r^+ \implies (b, c) \in r \implies (a, c) \in r^+$
 $\langle \text{proof} \rangle$

lemma *tranc1p-rtranc1p-tranc1p*:
assumes $r^{++} a b$ $r^{**} b c$ **shows** $r^{++} a c$

<proof>

lemma *rtranclp-conversep*: $r^{-1-1**} = r^{*-1-1}$
<proof>

lemmas *symp-rtranclp* = *sym-rtrancl*[*to-pred*]

lemmas *symp-conv-conversep-eq* = *sym-conv-converse-eq*[*to-pred*]

lemmas *rtranclp-tranclp-absorb* [*simp*] = *rtrancl-trancl-absorb*[*to-pred*]

lemmas *tranclp-rtranclp-absorb* [*simp*] = *trancl-rtrancl-absorb*[*to-pred*]

lemmas *rtranclp-reflclp-absorb* [*simp*] = *rtrancl-reflcl-absorb*[*to-pred*]

lemmas *trancl-rtrancl-trancl* = *tranclp-rtranclp-tranclp* [*to-set*]

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-rtrancl-trancl

lemmas *transitive-closurep-trans'* [*trans*] =
tranclp-trans rtranclp-trans
tranclp.trancl-into-trancl tranclp-into-tranclp2
rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare *trancl-into-rtrancl* [*elim*]

lemma *tranclp-ident-if-transp*:
assumes *transp R*
shows $R^{++} = R$
<proof>

The following are special cases of *tranclp-ident-if-transp*, but they appear duplicated in multiple, independent theories, which causes name clashes.

lemma (**in** *preorder*) *tranclp-less*[*simp*]: $(<)^{++} = (<)$
<proof>

lemma (**in** *preorder*) *tranclp-less-eq*[*simp*]: $(\leq)^{++} = (\leq)$
<proof>

lemma (**in** *preorder*) *tranclp-greater*[*simp*]: $(>)^{++} = (>)$
<proof>

lemma (**in** *preorder*) *tranclp-greater-eq*[*simp*]: $(\geq)^{++} = (\geq)$
<proof>

21.4 Symmetric closure

definition $\text{symclp} :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$
where $\text{symclp } r \ x \ y \longleftrightarrow r \ x \ y \vee r \ y \ x$

lemma symclpI [*simp, intro?*]:
shows $\text{symclpI1}: r \ x \ y \Longrightarrow \text{symclp } r \ x \ y$
and $\text{symclpI2}: r \ y \ x \Longrightarrow \text{symclp } r \ x \ y$
 $\langle \text{proof} \rangle$

lemma symclpE [*consumes 1, cases pred*]:
assumes $\text{symclp } r \ x \ y$
obtains $(\text{base}) \ r \ x \ y \mid (\text{sym}) \ r \ y \ x$
 $\langle \text{proof} \rangle$

lemma $\text{symclp-pointfree}: \text{symclp } r = \sup r \ r^{-1-1}$
 $\langle \text{proof} \rangle$

lemma $\text{symclp-greater}: r \leq \text{symclp } r$
 $\langle \text{proof} \rangle$

lemma symclp-conversep [*simp*]: $\text{symclp } r^{-1-1} = \text{symclp } r$
 $\langle \text{proof} \rangle$

lemma symp-on-symclp [*simp*]: $\text{symp-on } A \ (\text{symclp } R)$
 $\langle \text{proof} \rangle$

lemma $\text{symp-symclp-eq}: \text{symp } r \Longrightarrow \text{symclp } r = r$
 $\langle \text{proof} \rangle$

lemma $\text{symp-rtranclp-symclp}$ [*simp*]: $\text{symp } (\text{symclp } r)^{**}$
 $\langle \text{proof} \rangle$

lemma $\text{rtranclp-symclp-sym}$ [*sym*]: $(\text{symclp } r)^{**} \ x \ y \Longrightarrow (\text{symclp } r)^{**} \ y \ x$
 $\langle \text{proof} \rangle$

lemma symclp-idem [*simp*]: $\text{symclp } (\text{symclp } r) = \text{symclp } r$
 $\langle \text{proof} \rangle$

lemma reflp-on-rtranclp [*simp*]: $\text{reflp-on } A \ R^{**}$
 $\langle \text{proof} \rangle$

21.5 The power operation on relations

$R \rightsquigarrow n = R \ O \ \dots \ O \ R$, the n-fold composition of R

overloading

$\text{relpow} \equiv \text{compow} :: \text{nat} \Rightarrow ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set}$

$\text{relpow} \equiv \text{compow} :: \text{nat} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool})$

begin


```

primrec relpow :: nat  $\Rightarrow$  ('a  $\times$  'a) set  $\Rightarrow$  ('a  $\times$  'a) set
  where
    relpow 0 R = Id
  | relpow (Suc n) R = (R  $\sim$  n) O R

primrec relpowp :: nat  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)
  where
    relpowp 0 R = HOL.eq
  | relpowp (Suc n) R = (R  $\sim$  n) OO R

end

lemmas relpowp-Suc-right = relpowp.simps(2)

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 :: 'a rel
  <proof>

For code generation:

definition relpow :: nat  $\Rightarrow$  ('a  $\times$  'a) set  $\Rightarrow$  ('a  $\times$  'a) set
  where relpow-code-def [code-abbrev]: relpow = compow

definition relpowp :: nat  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)
  where relpowp-code-def [code-abbrev]: relpowp = compow

lemma [code]:
  relpow 0 R = Id
  relpow (Suc n) R = relpow n R O R
  <proof>

lemma [code]:
  relpowp 0 R = HOL.eq
  relpowp (Suc n) R = relpowp n R OO R
  <proof>

hide-const (open) relpow
hide-const (open) relpowp

lemma relpow-1 [simp]: R  $\sim$  1 = R
  for R :: ('a  $\times$  'a) set
  <proof>

lemma relpowp-1 [simp]: P  $\sim$  1 = P
  for P :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  <proof>

lemma relpowp-Suc-0 [simp]: P  $\sim$  (Suc 0) = P
  for P :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool

```

$\langle \text{proof} \rangle$

lemma *relpow-0-I*: $(x, x) \in R \rightsquigarrow 0$
 $\langle \text{proof} \rangle$

lemma *relpowp-0-I*: $(P \rightsquigarrow 0) x x$
 $\langle \text{proof} \rangle$

lemma *relpow-Suc-I*: $(x, y) \in R \rightsquigarrow n \implies (y, z) \in R \implies (x, z) \in R \rightsquigarrow \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *relpowp-Suc-I[trans]*: $(P \rightsquigarrow n) x y \implies P y z \implies (P \rightsquigarrow \text{Suc } n) x z$
 $\langle \text{proof} \rangle$

lemma *relpow-Suc-I2*: $(x, y) \in R \implies (y, z) \in R \rightsquigarrow n \implies (x, z) \in R \rightsquigarrow \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *relpowp-Suc-I2[trans]*: $P x y \implies (P \rightsquigarrow n) y z \implies (P \rightsquigarrow \text{Suc } n) x z$
 $\langle \text{proof} \rangle$

lemma *relpow-0-E*: $(x, y) \in R \rightsquigarrow 0 \implies (x = y \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *relpowp-0-E*: $(P \rightsquigarrow 0) x y \implies (x = y \implies Q) \implies Q$
 $\langle \text{proof} \rangle$

lemma *relpow-Suc-E*: $(x, z) \in R \rightsquigarrow \text{Suc } n \implies (\bigwedge y. (x, y) \in R \rightsquigarrow n \implies (y, z) \in R \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *relpowp-Suc-E*: $(P \rightsquigarrow \text{Suc } n) x z \implies (\bigwedge y. (P \rightsquigarrow n) x y \implies P y z \implies Q) \implies Q$
 $\langle \text{proof} \rangle$

lemma *relpow-E*:
 $(x, z) \in R \rightsquigarrow n \implies$
 $(n = 0 \implies x = z \implies P) \implies$
 $(\bigwedge y m. n = \text{Suc } m \implies (x, y) \in R \rightsquigarrow m \implies (y, z) \in R \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *relpowp-E*:
 $(P \rightsquigarrow n) x z \implies$
 $(n = 0 \implies x = z \implies Q) \implies$
 $(\bigwedge y m. n = \text{Suc } m \implies (P \rightsquigarrow m) x y \implies P y z \implies Q) \implies Q$
 $\langle \text{proof} \rangle$

lemma *relpow-Suc-D2*: $(x, z) \in R \rightsquigarrow \text{Suc } n \implies (\exists y. (x, y) \in R \wedge (y, z) \in R \rightsquigarrow n)$
 $\langle \text{proof} \rangle$

lemma *relpowp-Suc-D2*: $(P \sim\!\!\sim Suc\ n)\ x\ z \implies \exists y. P\ x\ y \wedge (P \sim\!\!\sim n)\ y\ z$
 ⟨proof⟩

lemma *relpow-Suc-E2*: $(x, z) \in R \sim\!\!\sim Suc\ n \implies (\bigwedge y. (x, y) \in R \implies (y, z) \in R \sim\!\!\sim n \implies P) \implies P$
 ⟨proof⟩

lemma *relpowp-Suc-E2*: $(P \sim\!\!\sim Suc\ n)\ x\ z \implies (\bigwedge y. P\ x\ y \implies (P \sim\!\!\sim n)\ y\ z \implies Q) \implies Q$
 ⟨proof⟩

lemma *relpow-Suc-D2'*: $\forall x\ y\ z. (x, y) \in R \sim\!\!\sim n \wedge (y, z) \in R \longrightarrow (\exists w. (x, w) \in R \wedge (w, z) \in R \sim\!\!\sim n)$
 ⟨proof⟩

lemma *relpowp-Suc-D2'*: $\forall x\ y\ z. (P \sim\!\!\sim n)\ x\ y \wedge P\ y\ z \longrightarrow (\exists w. P\ x\ w \wedge (P \sim\!\!\sim n)\ w\ z)$
 ⟨proof⟩

lemma *relpow-E2*:
 assumes $(x, z) \in R \sim\!\!\sim n\ n = 0 \implies x = z \implies P$
 $\bigwedge y\ m. n = Suc\ m \implies (x, y) \in R \implies (y, z) \in R \sim\!\!\sim m \implies P$
 shows P
 ⟨proof⟩

lemma *relpowp-E2*:
 $(P \sim\!\!\sim n)\ x\ z \implies$
 $(n = 0 \implies x = z \implies Q) \implies$
 $(\bigwedge y\ m. n = Suc\ m \implies P\ x\ y \implies (P \sim\!\!\sim m)\ y\ z \implies Q) \implies Q$
 ⟨proof⟩

lemma *relpowp-trans[trans]*: $(R \sim\!\!\sim i)\ x\ y \implies (R \sim\!\!\sim j)\ y\ z \implies (R \sim\!\!\sim (i + j))\ x\ z$
 ⟨proof⟩

lemma *relpowp-mono*:
 fixes $x\ y :: 'a$
 shows $(\bigwedge x\ y. R\ x\ y \implies S\ x\ y) \implies (R \sim\!\!\sim n)\ x\ y \implies (S \sim\!\!\sim n)\ x\ y$
 ⟨proof⟩

lemma *relpow-trans[trans]*: $(x, y) \in R \sim\!\!\sim i \implies (y, z) \in R \sim\!\!\sim j \implies (x, z) \in R \sim\!\!\sim (i + j)$
 ⟨proof⟩

lemma *relpowp-left-unique*:
 fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$ and $n :: nat$ and $x\ y\ z :: 'a$
 assumes *lunique*: $\bigwedge x\ y\ z. R\ x\ z \implies R\ y\ z \implies x = y$
 shows $(R \sim\!\!\sim n)\ x\ z \implies (R \sim\!\!\sim n)\ y\ z \implies x = y$
 ⟨proof⟩

lemma *relpow-left-unique*:

fixes $R :: ('a \times 'a) \text{ set}$ **and** $n :: \text{nat}$ **and** $x\ y\ z :: 'a$
shows $(\bigwedge x\ y\ z. (x, z) \in R \implies (y, z) \in R \implies x = y) \implies$
 $(x, z) \in R^{\sim n} \implies (y, z) \in R^{\sim n} \implies x = y$
 $\langle \text{proof} \rangle$

lemma *relpowp-right-unique*:

fixes $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ **and** $n :: \text{nat}$ **and** $x\ y\ z :: 'a$
assumes *runique*: $\bigwedge x\ y\ z. R\ x\ y \implies R\ x\ z \implies y = z$
shows $(R^{\sim n})\ x\ y \implies (R^{\sim n})\ x\ z \implies y = z$
 $\langle \text{proof} \rangle$

lemma *relpow-right-unique*:

fixes $R :: ('a \times 'a) \text{ set}$ **and** $n :: \text{nat}$ **and** $x\ y\ z :: 'a$
shows $(\bigwedge x\ y\ z. (x, y) \in R \implies (x, z) \in R \implies y = z) \implies$
 $(x, y) \in (R^{\sim n}) \implies (x, z) \in (R^{\sim n}) \implies y = z$
 $\langle \text{proof} \rangle$

lemma *relpow-add*: $R^{\sim (m + n)} = R^{\sim m} \circ R^{\sim n}$
 $\langle \text{proof} \rangle$

lemma *relpowp-add*: $P^{\sim (m + n)} = P^{\sim m} \circ \circ P^{\sim n}$
 $\langle \text{proof} \rangle$

lemma *relpow-commute*: $R \circ R^{\sim n} = R^{\sim n} \circ R$
 $\langle \text{proof} \rangle$

lemma *relpowp-commute*: $P \circ \circ P^{\sim n} = P^{\sim n} \circ \circ P$
 $\langle \text{proof} \rangle$

lemma *relpowp-Suc-left*: $R^{\sim \text{Suc } n} = R \circ \circ (R^{\sim n})$
 $\langle \text{proof} \rangle$

lemma *relpow-empty*: $0 < n \implies (\{\} :: ('a \times 'a) \text{ set})^{\sim n} = \{\}$
 $\langle \text{proof} \rangle$

lemma *relpowp-bot*: $0 < n \implies (\perp :: 'a \Rightarrow 'a \Rightarrow \text{bool})^{\sim n} = \perp$
 $\langle \text{proof} \rangle$

lemma *rtranc1-imp-UN-relpow*:

assumes $p \in R^*$
shows $p \in (\bigcup n. R^{\sim n})$
 $\langle \text{proof} \rangle$

lemma *rtranc1p-imp-Sup-relpowp*:

assumes $(P^{**})\ x\ y$
shows $(\bigsqcup n. P^{\sim n})\ x\ y$
 $\langle \text{proof} \rangle$

lemma *relpow-imp-rtrancl*:

assumes $p \in R \rightsquigarrow n$

shows $p \in R^*$

<proof>

lemma *relpowp-imp-rtranclp*: $(P \rightsquigarrow n) x y \implies (P^{**}) x y$

<proof>

lemma *rtrancl-is-UN-relpow*: $R^* = (\bigcup n. R \rightsquigarrow n)$

<proof>

lemma *rtranclp-is-Sup-relpowp*: $P^{**} = (\bigsqcup n. P \rightsquigarrow n)$

<proof>

lemma *rtrancl-power*: $p \in R^* \longleftrightarrow (\exists n. p \in R \rightsquigarrow n)$

<proof>

lemma *rtranclp-power*: $(P^{**}) x y \longleftrightarrow (\exists n. (P \rightsquigarrow n) x y)$

<proof>

lemma *trancl-power*: $p \in R^+ \longleftrightarrow (\exists n > 0. p \in R \rightsquigarrow n)$

<proof>

lemma *tranclp-power*: $(P^{++}) x y \longleftrightarrow (\exists n > 0. (P \rightsquigarrow n) x y)$

<proof>

lemma *rtrancl-imp-relpow*: $p \in R^* \implies \exists n. p \in R \rightsquigarrow n$

<proof>

lemma *rtranclp-imp-relpowp*: $(P^{**}) x y \implies \exists n. (P \rightsquigarrow n) x y$

<proof>

By Sternagel/Thiemann:

lemma *relpow-fun-conv*: $(a, b) \in R \rightsquigarrow n \longleftrightarrow (\exists f. f\ 0 = a \wedge f\ n = b \wedge (\forall i < n. (f\ i, f\ (Suc\ i)) \in R))$

<proof>

lemma *relpowp-fun-conv*: $(P \rightsquigarrow n) x y \longleftrightarrow (\exists f. f\ 0 = x \wedge f\ n = y \wedge (\forall i < n. P\ (f\ i)\ (f\ (Suc\ i))))$

<proof>

lemma *relpow-finite-bounded1*:

fixes $R :: ('a \times 'a)\ set$

assumes *finite* R **and** $k > 0$

shows $R \rightsquigarrow k \subseteq (\bigcup n \in \{n. 0 < n \wedge n \leq \text{card } R\}. R \rightsquigarrow n)$

(is - \subseteq ?r)

<proof>

lemma *relpow-finite-bounded*:

fixes $R :: ('a \times 'a) \text{ set}$

assumes *finite R*

shows $R^{\sim k} \subseteq (\bigcup_{n \in \{n. n \leq \text{card } R\}} R^{\sim n})$

<proof>

lemma *rtrancl-finite-eq-relpow*: $\text{finite } R \implies R^* = (\bigcup_{n \in \{n. n \leq \text{card } R\}} R^{\sim n})$

<proof>

lemma *trancl-finite-eq-relpow*:

assumes *finite R* **shows** $R^+ = (\bigcup_{n \in \{n. 0 < n \wedge n \leq \text{card } R\}} R^{\sim n})$

<proof>

lemma *finite-relcomp[simp,intro]*:

assumes *finite R* **and** *finite S*

shows *finite (R O S)*

<proof>

lemma *finite-relpow [simp, intro]*:

fixes $R :: ('a \times 'a) \text{ set}$

assumes *finite R*

shows $n > 0 \implies \text{finite } (R^{\sim n})$

<proof>

lemma *single-valued-relpow*:

fixes $R :: ('a \times 'a) \text{ set}$

shows *single-valued R* \implies *single-valued (R \sim n)*

<proof>

21.6 Bounded transitive closure

definition *ntrancl* :: $\text{nat} \Rightarrow ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set}$

where $\text{ntrancl } n \ R = (\bigcup_{i \in \{i. 0 < i \wedge i \leq \text{Suc } n\}} R^{\sim i})$

lemma *ntrancl-Zero [simp, code]*: $\text{ntrancl } 0 \ R = R$

<proof>

lemma *ntrancl-Suc [simp]*: $\text{ntrancl } (\text{Suc } n) \ R = \text{ntrancl } n \ R \ O \ (\text{Id} \cup R)$

<proof>

lemma [code]: $\text{ntrancl } (\text{Suc } n) \ r = (\text{let } r' = \text{ntrancl } n \ r \text{ in } r' \cup r' \ O \ r)$

<proof>

lemma *finite-trancl-ntrancl*: $\text{finite } R \implies \text{trancl } R = \text{ntrancl } (\text{card } R - 1) \ R$

<proof>

21.7 Acyclic relations

definition *acyclic* :: $('a \times 'a) \text{ set} \Rightarrow \text{bool}$

where $\text{acyclic } r \longleftrightarrow (\forall x. (x, x) \notin r^+)$

abbreviation $acyclicP :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where $acyclicP\ r \equiv acyclic\ \{(x, y). r\ x\ y\}$

lemma $acyclic\text{-}irrefl$ [code]: $acyclic\ r \longleftrightarrow irrefl\ (r^+)$
 $\langle proof \rangle$

lemma $acyclicI$: $\forall x. (x, x) \notin r^+ \implies acyclic\ r$
 $\langle proof \rangle$

lemma (in $preorder$) $acyclicI\text{-}order$:
assumes *: $\bigwedge a\ b. (a, b) \in r \implies f\ b < f\ a$
shows $acyclic\ r$
 $\langle proof \rangle$

lemma $acyclic\text{-}insert$ [iff]: $acyclic\ (insert\ (y, x)\ r) \longleftrightarrow acyclic\ r \wedge (x, y) \notin r^*$
 $\langle proof \rangle$

lemma $acyclic\text{-}converse$ [iff]: $acyclic\ (r^{-1}) \longleftrightarrow acyclic\ r$
 $\langle proof \rangle$

lemmas $acyclicP\text{-}converse$ [iff] = $acyclic\text{-}converse$ [to-pred]

lemma $acyclic\text{-}impl\text{-}antisym\text{-}rtrancl$: $acyclic\ r \implies antisym\ (r^*)$
 $\langle proof \rangle$

lemma $acyclic\text{-}subset$: $acyclic\ s \implies r \subseteq s \implies acyclic\ r$
 $\langle proof \rangle$

21.8 Setup of transitivity reasoner

$\langle ML \rangle$

lemma $transp\text{-}rtranclp$ [simp]: $transp\ R^{**}$
 $\langle proof \rangle$

Optional methods.

$\langle ML \rangle$

end

22 Well-founded Recursion

theory *Wellfounded*
imports *Transitive-Closure*
begin

22.1 Basic Definitions

definition $wf\text{-}on :: 'a \text{ set} \Rightarrow 'a \text{ rel} \Rightarrow bool$ **where**

$wf\text{-}on \ A \ r \longleftrightarrow (\forall P. (\forall x \in A. (\forall y \in A. (y, x) \in r \longrightarrow P \ y) \longrightarrow P \ x) \longrightarrow (\forall x \in A. P \ x))$

abbreviation $wf :: ('a \times 'a) \text{ set} \Rightarrow bool$ **where**

$wf \equiv wf\text{-}on \ UNIV$

definition $wfp\text{-}on :: 'a \text{ set} \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**

$wfp\text{-}on \ A \ R \longleftrightarrow (\forall P. (\forall x \in A. (\forall y \in A. R \ y \ x \longrightarrow P \ y) \longrightarrow P \ x) \longrightarrow (\forall x \in A. P \ x))$

abbreviation $wfP :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**

$wfP \equiv wfp\text{-}on \ UNIV$

alias $wfp = wfP$

We keep old name wfp for backward compatibility, but offer new name wfp to be consistent with similar predicates, e.g., *asympt*, *transp*, *totalp*.

22.2 Equivalence of Definitions

lemma $wfp\text{-}on\text{-}wf\text{-}on\text{-}eq[pred\text{-}set\text{-}conv]$: $wfp\text{-}on \ A \ (\lambda x \ y. (x, y) \in r) \longleftrightarrow wf\text{-}on \ A \ r$
 $\langle proof \rangle$

lemma $wf\text{-}def$: $wf \ r \longleftrightarrow (\forall P. (\forall x. (\forall y. (y, x) \in r \longrightarrow P \ y) \longrightarrow P \ x) \longrightarrow (\forall x. P \ x))$
 $\langle proof \rangle$

lemma $wfp\text{-}def$: $wfp \ r \longleftrightarrow wf \ \{(x, y). r \ x \ y\}$
 $\langle proof \rangle$

lemma $wfp\text{-}wf\text{-}eq$: $wfp \ (\lambda x \ y. (x, y) \in r) = wf \ r$
 $\langle proof \rangle$

22.3 Induction Principles

lemma $wf\text{-}on\text{-}induct[consumes \ 1, \text{ case-names in-set less, induct set: } wf\text{-}on]$:

assumes $wf\text{-}on \ A \ r$ **and** $x \in A$ **and** $\bigwedge x. x \in A \Longrightarrow (\bigwedge y. y \in A \Longrightarrow (y, x) \in r \Longrightarrow P \ y) \Longrightarrow P \ x$

shows $P \ x$

$\langle proof \rangle$

lemma $wfp\text{-}on\text{-}induct[consumes \ 1, \text{ case-names in-set less, induct pred: } wfp\text{-}on]$:

assumes $wfp\text{-}on \ A \ r$ **and** $x \in A$ **and** $\bigwedge x. x \in A \Longrightarrow (\bigwedge y. y \in A \Longrightarrow r \ y \ x \Longrightarrow P \ y) \Longrightarrow P \ x$

shows $P \ x$

$\langle proof \rangle$

lemma *wf-induct*:

assumes *wf r*

and $\bigwedge x. \forall y. (y, x) \in r \longrightarrow P y \Longrightarrow P x$

shows $P a$

<proof>

lemmas *wfp-induct* = *wf-induct* [*to-pred*]

lemmas *wf-induct-rule* = *wf-induct* [*rule-format*, *consumes 1*, *case-names less*, *induct set: wf*]

lemmas *wfp-induct-rule* = *wf-induct-rule* [*to-pred*, *induct set: wfp*]

lemma *wf-on-iff-wf*: $wf\text{-on } A \ r \longleftrightarrow wf \ \{(x, y) \in r. x \in A \wedge y \in A\}$

<proof>

22.4 Introduction Rules

lemma *wfUNIVI*: $(\bigwedge P x. (\forall x. (\forall y. (y, x) \in r \longrightarrow P y) \longrightarrow P x) \Longrightarrow P x) \Longrightarrow$

wf r

<proof>

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*:

assumes $r \subseteq A \times B$

and $\bigwedge x P. [\forall x. (\forall y. (y, x) \in r \longrightarrow P y) \longrightarrow P x; \ x \in A; \ x \in B] \Longrightarrow P x$

shows *wf r*

<proof>

22.5 Ordering Properties

lemma *wf-not-sym*: $wf \ r \Longrightarrow (a, x) \in r \Longrightarrow (x, a) \notin r$

<proof>

lemma *wf-asymp*:

assumes $wf \ r \ (a, x) \in r$

obtains $(x, a) \notin r$

<proof>

lemma *wf-imp-asymp*: $wf \ r \Longrightarrow asymp \ r$

<proof>

lemma *wfp-imp-asymp*: $wfp \ r \Longrightarrow asymp \ r$

<proof>

lemma *wf-not-refl* [*simp*]: $wf\ r \implies (a, a) \notin r$
 $\langle proof \rangle$

lemma *wf-irrefl*:
assumes $wf\ r$
obtains $(a, a) \notin r$
 $\langle proof \rangle$

lemma *wf-imp-irrefl*:
assumes $wf\ r$ **shows** $irrefl\ r$
 $\langle proof \rangle$

lemma *wfp-imp-irreflp*: $wfp\ r \implies irreflp\ r$
 $\langle proof \rangle$

lemma *wf-wellorderI*:
assumes $wf: wf\ \{(x::'a::ord, y). x < y\}$
and $lin: OFCLASS('a::ord, linorder-class)$
shows $OFCLASS('a::ord, wellorder-class)$
 $\langle proof \rangle$

lemma (**in** *wellorder*) $wf: wf\ \{(x, y). x < y\}$
 $\langle proof \rangle$

lemma (**in** *wellorder*) *wfp-on-less*[*simp*]: $wfp-on\ A\ (<)$
 $\langle proof \rangle$

22.6 Basic Results

Point-free characterization of well-foundedness

lemma *wf-onE-pf*:
assumes $wf: wf-on\ A\ r$ **and** $B \subseteq A$ **and** $B \subseteq r$ “ B
shows $B = \{\}$
 $\langle proof \rangle$

lemma *wfE-pf*: $wf\ R \implies A \subseteq R$ “ $A \implies A = \{\}$
 $\langle proof \rangle$

lemma *wf-onI-pf*:
assumes $\bigwedge B. B \subseteq A \implies B \subseteq R$ “ $B \implies B = \{\}$
shows $wf-on\ A\ R$
 $\langle proof \rangle$

lemma *wfI-pf*: $(\bigwedge A. A \subseteq R \text{ “ } A \implies A = \{\}) \implies wf\ R$
 $\langle proof \rangle$

22.6.1 Minimal-element characterization of well-foundedness

lemma *wf-on-iff-ex-minimal*: $wf\text{-}on\ A\ R \longleftrightarrow (\forall B \subseteq A. B \neq \{\} \longrightarrow (\exists z \in B. \forall y. (y, z) \in R \longrightarrow y \notin B))$
 ⟨proof⟩

lemma *wf-iff-ex-minimal*: $wf\ R \longleftrightarrow (\forall B. B \neq \{\} \longrightarrow (\exists z \in B. \forall y. (y, z) \in R \longrightarrow y \notin B))$
 ⟨proof⟩

lemma *wfp-on-iff-ex-minimal*: $wfp\text{-}on\ A\ R \longleftrightarrow (\forall B \subseteq A. B \neq \{\} \longrightarrow (\exists z \in B. \forall y. R\ y\ z \longrightarrow y \notin B))$
 ⟨proof⟩

lemma *wfp-iff-ex-minimal*: $wfp\ R \longleftrightarrow (\forall B. B \neq \{\} \longrightarrow (\exists z \in B. \forall y. R\ y\ z \longrightarrow y \notin B))$
 ⟨proof⟩

lemma *wfE-min*:
 assumes $wf: wf\ R$ and $Q: x \in Q$
 obtains z where $z \in Q \wedge y. (y, z) \in R \implies y \notin Q$
 ⟨proof⟩

lemma *wfE-min'*:
 $wf\ R \implies Q \neq \{\} \implies (\bigwedge z. z \in Q \implies (\bigwedge y. (y, z) \in R \implies y \notin Q) \implies thesis)$
 $\implies thesis$
 ⟨proof⟩

lemma *wfI-min*:
 assumes $a: \bigwedge x\ Q. x \in Q \implies \exists z \in Q. \forall y. (y, z) \in R \longrightarrow y \notin Q$
 shows $wf\ R$
 ⟨proof⟩

lemma *wf-eq-minimal*: $wf\ r \longleftrightarrow (\forall Q\ x. x \in Q \longrightarrow (\exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q))$
 ⟨proof⟩

lemmas $wfp\text{-}eq\text{-}minimal = wf\text{-}eq\text{-}minimal\ [to\text{-}pred]$

22.6.2 Finite characterization of well-foundedness

lemma *strict-partial-order-wfp-on-finite-set*:
 assumes $transp\text{-}on\ \mathcal{X}\ R$ and $asyp\text{-}on\ \mathcal{X}\ R$ and $finite\ \mathcal{X}$
 shows $wfp\text{-}on\ \mathcal{X}\ R$
 ⟨proof⟩

22.6.3 Antimonotonicity

lemma *wfp-on-mono-stronger*:
 fixes

$A :: 'a \text{ set}$ and $B :: 'b \text{ set}$ and
 $f :: 'a \Rightarrow 'b$ and
 $R :: 'b \Rightarrow 'b \Rightarrow \text{bool}$ and $Q :: 'a \Rightarrow 'a \Rightarrow \text{bool}$
assumes
 $wf: wfp\text{-on } B \ R$ and
 $sub: f ' A \subseteq B$ and
 $mono: \bigwedge x y. x \in A \Longrightarrow y \in A \Longrightarrow Q \ x \ y \Longrightarrow R \ (f \ x) \ (f \ y)$
shows $wfp\text{-on } A \ Q$
 $\langle proof \rangle$

lemma *wf-on-mono-stronger*:

assumes
 $wf\text{-on } B \ r$ and
 $f ' A \subseteq B$ and
 $(\bigwedge x y. x \in A \Longrightarrow y \in A \Longrightarrow (x, y) \in q \Longrightarrow (f \ x, f \ y) \in r)$
shows $wf\text{-on } A \ q$
 $\langle proof \rangle$

lemma *wf-on-mono-strong*:

assumes $wf\text{-on } B \ r$ and $A \subseteq B$ and $(\bigwedge x y. x \in A \Longrightarrow y \in A \Longrightarrow (x, y) \in q \Longrightarrow (x, y) \in r)$
shows $wf\text{-on } A \ q$
 $\langle proof \rangle$

lemma *wfp-on-mono-strong*:

$wfp\text{-on } B \ R \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x y. x \in A \Longrightarrow y \in A \Longrightarrow Q \ x \ y \Longrightarrow R \ x \ y) \Longrightarrow$
 $wfp\text{-on } A \ Q$
 $\langle proof \rangle$

lemma *wf-on-mono*: $A \subseteq B \Longrightarrow q \subseteq r \Longrightarrow wf\text{-on } B \ r \leq wf\text{-on } A \ q$
 $\langle proof \rangle$

lemma *wfp-on-mono*: $A \subseteq B \Longrightarrow Q \leq R \Longrightarrow wfp\text{-on } B \ R \leq wfp\text{-on } A \ Q$
 $\langle proof \rangle$

lemma *wf-on-subset*: $wf\text{-on } B \ r \Longrightarrow A \subseteq B \Longrightarrow wf\text{-on } A \ r$
 $\langle proof \rangle$

lemma *wfp-on-subset*: $wfp\text{-on } B \ R \Longrightarrow A \subseteq B \Longrightarrow wfp\text{-on } A \ R$
 $\langle proof \rangle$

22.6.4 Equivalence between *wfp-on* and *wfp*

lemma *wfp-on-iff-wfp*: $wfp\text{-on } A \ R \longleftrightarrow wfp \ (\lambda x y. R \ x \ y \wedge x \in A \wedge y \in A)$
(is ?LHS \longleftrightarrow ?RHS)
 $\langle proof \rangle$

22.6.5 Well-foundedness of transitive closure

lemma *bex-rtrancl-min-element-if-wf-on*:

assumes $wf: wf\text{-on } A \text{ } r$ **and** $x\text{-in}: x \in A$
shows $\exists y \in A. (y, x) \in r^* \wedge \neg(\exists z \in A. (z, y) \in r)$
 $\langle proof \rangle$

lemma *be x -rtransclp-min-element-if-wfp-on*: $wfp\text{-on } A \text{ } R \implies x \in A \implies \exists y \in A. R^{**} y x \wedge \neg(\exists z \in A. R z y)$
 $\langle proof \rangle$

lemma *ex-terminating-rtranclp-strong*:
assumes $wf: wfp\text{-on } \{x'. R^{**} x x'\} R^{-1-1}$
shows $\exists y. R^{**} x y \wedge (\nexists z. R y z)$
 $\langle proof \rangle$

lemma *ex-terminating-rtranclp*:
assumes $wf: wfp R^{-1-1}$
shows $\exists y. R^{**} x y \wedge (\nexists z. R y z)$
 $\langle proof \rangle$

lemma *wf-trancl*:
assumes $wf \text{ } r$
shows $wf (r^+)$
 $\langle proof \rangle$

lemmas $wfp\text{-tranclp} = wf\text{-trancl } [to\text{-pred}]$

lemma *wf-converse-trancl*: $wf (r^{-1}) \implies wf ((r^+)^{-1})$
 $\langle proof \rangle$

Well-foundedness of subsets

lemma *wf-subset*: $wf \text{ } r \implies p \subseteq r \implies wf \text{ } p$
 $\langle proof \rangle$

lemmas $wfp\text{-subset} = wf\text{-subset } [to\text{-pred}]$

Well-foundedness of the empty relation

lemma *wf-on-bot[iff]*: $wf\text{-on } A \perp$
 $\langle proof \rangle$

lemma *wfp-on-bot[iff]*: $wfp\text{-on } A \perp$
 $\langle proof \rangle$

lemma *wfp-empty [iff]*: $wfp (\lambda x y. False)$
 $\langle proof \rangle$

lemma *wf-Int1*: $wf \text{ } r \implies wf (r \cap r')$
 $\langle proof \rangle$

lemma *wf-Int2*: $wf \text{ } r \implies wf (r' \cap r)$
 $\langle proof \rangle$

Exponentiation.

lemma *wf-exp*:
 assumes $wf (R \rightsquigarrow n)$
 shows $wf R$
 $\langle proof \rangle$

Well-foundedness of *insert*.

lemma *wf-insert [iff]*: $wf (insert (y,x) r) \longleftrightarrow wf r \wedge (x,y) \notin r^* \text{ (is ?lhs = ?rhs)}$
 $\langle proof \rangle$

22.6.6 Well-foundedness of image

lemma *wf-map-prod-image-Dom-Ran*:
 fixes $r :: ('a \times 'a) \text{ set}$
 and $f :: 'a \Rightarrow 'b$
 assumes $wf\text{-}r: wf\ r$
 and *inj*: $\bigwedge a\ a'. a \in \text{Domain } r \implies a' \in \text{Range } r \implies f\ a = f\ a' \implies a = a'$
 shows $wf (map\text{-}prod\ f\ f\text{' } r)$
 $\langle proof \rangle$

lemma *wf-map-prod-image*: $wf\ r \implies \text{inj } f \implies wf (map\text{-}prod\ f\ f\text{' } r)$
 $\langle proof \rangle$

lemma *wfp-on-image*: $wfp\text{-}on\ (f\text{' } A)\ R \longleftrightarrow wfp\text{-}on\ A\ (\lambda a\ b. R\ (f\ a)\ (f\ b))$
 $\langle proof \rangle$

22.7 Well-Foundedness Results for Unions

lemma *wf-union-compatible*:
 assumes $wf\ R\ wf\ S$
 assumes $R\ O\ S \subseteq R$
 shows $wf (R \cup S)$
 $\langle proof \rangle$

Well-foundedness of indexed union with disjoint domains and ranges.

lemma *wf-UN*:
 assumes $r: \bigwedge i. i \in I \implies wf\ (r\ i)$
 and *disj*: $\bigwedge i\ j. \llbracket i \in I; j \in I; r\ i \neq r\ j \rrbracket \implies \text{Domain } (r\ i) \cap \text{Range } (r\ j) = \{\}$
 shows $wf (\bigcup_{i \in I}. r\ i)$
 $\langle proof \rangle$

lemma *wfp-SUP*:
 $\forall i. wfp\ (r\ i) \implies \forall i\ j. r\ i \neq r\ j \longrightarrow \inf (\text{Domainp } (r\ i)) (\text{Rangep } (r\ j)) = \text{bot}$
 \implies
 $wfp (\bigsqcup (range\ r))$
 $\langle proof \rangle$

lemma *wf-Union*:
 assumes $\forall r \in R. wf\ r$

and $\forall r \in R. \forall s \in R. r \neq s \longrightarrow \text{Domain } r \cap \text{Range } s = \{\}$
shows $\text{wf } (\bigcup R)$
 $\langle \text{proof} \rangle$

Intuition: We find an $R \cup S$ -min element of a nonempty subset A by case distinction.

1. There is a step $a -R\rightarrow b$ with $a, b \in A$. Pick an R -min element z of the (nonempty) set $\{a \in A \mid \exists b \in A. a -R\rightarrow b\}$. By definition, there is $z' \in A$ s.t. $z -R\rightarrow z'$. 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 *wf-Un*: $\text{wf } r \implies \text{wf } s \implies \text{Domain } r \cap \text{Range } s = \{\} \implies \text{wf } (r \cup s)$
 $\langle \text{proof} \rangle$

lemma *wf-union-merge*: $\text{wf } (R \cup S) = \text{wf } (R \circ R \cup S \circ R \cup S)$
 (is $\text{wf } ?A = \text{wf } ?B$)
 $\langle \text{proof} \rangle$

lemma *wf-comp-self*: $\text{wf } R \longleftrightarrow \text{wf } (R \circ R)$ — special case
 $\langle \text{proof} \rangle$

22.8 Well-Foundedness of Composition

Bachmair and Dershowitz 1986, Lemma 2. [Provided by Tjark Weber]

lemma *qc-wf-relto-iff*:
assumes $R \circ S \subseteq (R \cup S)^* \circ R$ — R quasi-commutes over S
shows $\text{wf } (S^* \circ R \circ S^*) \longleftrightarrow \text{wf } R$
 (is $\text{wf } ?S \longleftrightarrow -$)
 $\langle \text{proof} \rangle$

corollary *wf-relcomp-compatible*:
assumes $\text{wf } R$ **and** $R \circ S \subseteq S \circ R$
shows $\text{wf } (S \circ R)$
 $\langle \text{proof} \rangle$

22.9 Acyclic relations

lemma *wf-acyclic*: $\text{wf } r \implies \text{acyclic } r$
 $\langle \text{proof} \rangle$

lemmas $\text{wfp-acyclic } P = \text{wf-acyclic } [\text{to-pred}]$

22.9.1 Wellfoundedness of finite acyclic relations

lemma *finite-acyclic-wf*:
 assumes *finite r acyclic r* shows *wf r*
 $\langle \text{proof} \rangle$

lemma *finite-acyclic-wf-converse*: *finite r \implies acyclic r \implies wf (r^{-1})*
 $\langle \text{proof} \rangle$

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 *wf-converse*:
 assumes *irrefl r and trans r and finite r*
 shows *wf (r^{-1})*
 $\langle \text{proof} \rangle$

lemma *wf-iff-acyclic-if-finite*: *finite r \implies wf r = acyclic r*
 $\langle \text{proof} \rangle$

22.10 nat is well-founded

lemma *less-nat-rel*: *($<$) = ($\lambda m n. n = \text{Suc } m$) $^{++}$*
 $\langle \text{proof} \rangle$

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 $^+$ \longleftrightarrow m < n*
 $\langle \text{proof} \rangle$

lemma *pred-nat-trancl-eq-le*: *(m, n) \in pred-nat* \longleftrightarrow m \leq n*
 $\langle \text{proof} \rangle$

lemma *wf-pred-nat*: *wf pred-nat*
 $\langle \text{proof} \rangle$

lemma *wf-less-than [iff]*: *wf less-than*
 $\langle \text{proof} \rangle$

lemma *trans-less-than [iff]*: *trans less-than*
 $\langle \text{proof} \rangle$

lemma *less-than-iff [iff]*: *((x,y) \in less-than) = (x < y)*
 $\langle \text{proof} \rangle$

lemma *irrefl-less-than*: *irrefl less-than*
 $\langle \text{proof} \rangle$

lemma *asym-less-than*: *asym less-than*
 $\langle \text{proof} \rangle$

lemma *total-less-than*: *total less-than* **and** *total-on-less-than* [*simp*]: *total-on A less-than*
 $\langle \text{proof} \rangle$

lemma *wf-less*: *wf* $\{(x, y::\text{nat}). x < y\}$
 $\langle \text{proof} \rangle$

22.11 Accessible Part

Inductive definition of the accessible part *acc r* of a relation; see also [6].

inductive-set *acc* :: $('a \times 'a) \text{ set} \Rightarrow 'a \text{ set}$ **for** *r* :: $('a \times 'a) \text{ set}$
where *accI*: $(\bigwedge y. (y, x) \in r \implies y \in \text{acc } r) \implies x \in \text{acc } r$

abbreviation *termip* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow \text{bool}$
where *termip* *r* $\equiv \text{accp } (r^{-1-1})$

abbreviation *termi* :: $('a \times 'a) \text{ set} \Rightarrow 'a \text{ set}$
where *termi* *r* $\equiv \text{acc } (r^{-1})$

lemmas *accpI* = *accp.accI*

lemma *accp-eq-acc* [*code*]: *accp* *r* = $(\lambda x. x \in \text{Wellfounded}.\text{acc } \{(x, y). r \ x \ y\})$
 $\langle \text{proof} \rangle$

Induction rules

theorem *accp-induct*:
assumes *major*: *accp* *r* *a*
assumes *hyp*: $\bigwedge x. \text{accp } r \ x \implies \forall y. r \ y \ x \longrightarrow P \ y \implies P \ x$
shows *P a*
 $\langle \text{proof} \rangle$

lemmas *accp-induct-rule* = *accp-induct* [*rule-format*, *induct set*: *accp*]

theorem *accp-downward*: *accp* *r* *b* $\implies r \ a \ b \implies \text{accp } r \ a$
 $\langle \text{proof} \rangle$

lemma *not-accp-down*:
assumes *na*: $\neg \text{accp } R \ x$
obtains *z* **where** *R z x* **and** $\neg \text{accp } R \ z$
 $\langle \text{proof} \rangle$

lemma *accp-downwards-aux*: $r^{**} \ b \ a \implies \text{accp } r \ a \longrightarrow \text{accp } r \ b$
 $\langle \text{proof} \rangle$

theorem *accp-downwards*: *accp* *r* *a* $\implies r^{**} \ b \ a \implies \text{accp } r \ b$

$\langle proof \rangle$

theorem *accp-wfpI*: $\forall x. \text{accp } r \ x \implies \text{wfp } r$
 $\langle proof \rangle$

theorem *accp-wfpD*: $\text{wfp } r \implies \text{accp } r \ x$
 $\langle proof \rangle$

theorem *wfp-iff-accp*: $\text{wfp } r = (\forall x. \text{accp } r \ x)$
 $\langle proof \rangle$

Smaller relations have bigger accessible parts:

lemma *accp-subset*:
assumes $R1 \leq R2$
shows $\text{accp } R2 \leq \text{accp } R1$
 $\langle proof \rangle$

This is a generalized induction theorem that works on subsets of the accessible part.

lemma *accp-subset-induct*:
assumes *subset*: $D \leq \text{accp } R$
and *dcl*: $\bigwedge x \ z. D \ x \implies R \ z \ x \implies D \ z$
and $D \ x$
and *istep*: $\bigwedge x. D \ x \implies (\bigwedge z. R \ z \ x \implies P \ z) \implies P \ x$
shows $P \ x$
 $\langle proof \rangle$

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-iff-acc* = *wfp-iff-accp* [*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,introl*]:
fixes $f :: 'a \Rightarrow 'b$
assumes $\text{wf } r$
shows $\text{wf } (\text{inv-image } r \ f)$
 $\langle proof \rangle$

lemma *wfp-on-inv-imagep*:
assumes $wf: wfp\text{-}on\ (f \text{ ' } A)\ R$
shows $wfp\text{-}on\ A\ (inv\text{-}imagep\ R\ f)$
 $\langle proof \rangle$

22.12.1 Conversion to a known well-founded relation

lemma *wfp-on-if-convertible-to-wfp-on*:
assumes
 $wf: wfp\text{-}on\ (f \text{ ' } A)\ Q$ **and**
 $convertible: (\bigwedge x\ y. x \in A \implies y \in A \implies R\ x\ y \implies Q\ (f\ x)\ (f\ y))$
shows $wfp\text{-}on\ A\ R$
 $\langle proof \rangle$

lemma *wf-on-if-convertible-to-wf-on*: $wf\text{-}on\ (f \text{ ' } A)\ Q \implies (\bigwedge x\ y. x \in A \implies y \in A \implies (x, y) \in R \implies (f\ x, f\ y) \in Q) \implies wf\text{-}on\ A\ R$
 $\langle proof \rangle$

lemma *wf-if-convertible-to-wf*:
fixes $r :: 'a\ rel$ **and** $s :: 'b\ rel$ **and** $f :: 'a \Rightarrow 'b$
assumes $wf\ s$ **and** $convertible: \bigwedge x\ y. (x, y) \in r \implies (f\ x, f\ y) \in s$
shows $wf\ r$
 $\langle proof \rangle$

lemma *wfp-if-convertible-to-wfp*: $wfp\ S \implies (\bigwedge x\ y. R\ x\ y \implies S\ (f\ x)\ (f\ y)) \implies wfp\ R$
 $\langle proof \rangle$

Converting to *nat* is a very common special case that might be found more easily by Sledgehammer.

lemma *wfp-if-convertible-to-nat*:
fixes $f :: - \Rightarrow nat$
shows $(\bigwedge x\ y. R\ x\ y \implies f\ x < f\ y) \implies wfp\ R$
 $\langle proof \rangle$

22.12.2 Measure functions into *nat*

definition *measure* :: $('a \Rightarrow nat) \Rightarrow ('a \times 'a)\ set$
where $measure = inv\text{-}image\ less\text{-}than$

lemma *in-measure*[*simp, code-unfold*]: $(x, y) \in measure\ f \iff f\ x < f\ y$
 $\langle proof \rangle$

lemma *wf-measure* [*iff*]: $wf\ (measure\ f)$
 $\langle proof \rangle$

lemma *wf-if-measure*: $(\bigwedge x. P\ x \implies f(g\ x) < f\ x) \implies wf\ \{(y, x). P\ x \wedge y = g\ x\}$
for $f :: 'a \Rightarrow nat$
 $\langle proof \rangle$

22.12.3 Lexicographic combinations

definition $lex\text{-}prod :: ('a \times 'a) \text{ set} \Rightarrow ('b \times 'b) \text{ set} \Rightarrow (('a \times 'b) \times ('a \times 'b)) \text{ set}$
 (infixr $\langle *lex* \rangle$ 80)
 where $ra \langle *lex* \rangle rb = \{((a, b), (a', b')). (a, a') \in ra \vee a = a' \wedge (b, b') \in rb\}$

lemma $in\text{-}lex\text{-}prod[simp]$: $((a, b), (a', b')) \in r \langle *lex* \rangle s \longleftrightarrow (a, a') \in r \vee a = a' \wedge (b, b') \in s$
 $\langle proof \rangle$

lemma $wf\text{-}on\text{-}lex\text{-}prod[intro]$:
 assumes wfA : $wf\text{-}on\ A\ r_A$ and wfB : $wf\text{-}on\ B\ r_B$
 shows $wf\text{-}on\ (A \times B)\ (r_A \langle *lex* \rangle r_B)$
 $\langle proof \rangle$

lemma $wf\text{-}lex\text{-}prod\ [intro!]$:
 assumes $wf\ ra\ wf\ rb$
 shows $wf\ (ra \langle *lex* \rangle rb)$
 $\langle proof \rangle$

lemma $refl\text{-}lex\text{-}prod[simp]$: $refl\ r_B \Longrightarrow refl\ (r_A \langle *lex* \rangle r_B)$
 $\langle proof \rangle$

lemma $irrefl\text{-}on\text{-}lex\text{-}prod[simp]$:
 $irrefl\text{-}on\ A\ r_A \Longrightarrow irrefl\text{-}on\ B\ r_B \Longrightarrow irrefl\text{-}on\ (A \times B)\ (r_A \langle *lex* \rangle r_B)$
 $\langle proof \rangle$

lemma $irrefl\text{-}lex\text{-}prod[simp]$: $irrefl\ r_A \Longrightarrow irrefl\ r_B \Longrightarrow irrefl\ (r_A \langle *lex* \rangle r_B)$
 $\langle proof \rangle$

lemma $sym\text{-}on\text{-}lex\text{-}prod[simp]$:
 $sym\text{-}on\ A\ r_A \Longrightarrow sym\text{-}on\ B\ r_B \Longrightarrow sym\text{-}on\ (A \times B)\ (r_A \langle *lex* \rangle r_B)$
 $\langle proof \rangle$

lemma $sym\text{-}lex\text{-}prod[simp]$:
 $sym\ r_A \Longrightarrow sym\ r_B \Longrightarrow sym\ (r_A \langle *lex* \rangle r_B)$
 $\langle proof \rangle$

lemma $asym\text{-}on\text{-}lex\text{-}prod[simp]$:
 $asym\text{-}on\ A\ r_A \Longrightarrow asym\text{-}on\ B\ r_B \Longrightarrow asym\text{-}on\ (A \times B)\ (r_A \langle *lex* \rangle r_B)$
 $\langle proof \rangle$

lemma $asym\text{-}lex\text{-}prod[simp]$:
 $asym\ r_A \Longrightarrow asym\ r_B \Longrightarrow asym\ (r_A \langle *lex* \rangle r_B)$
 $\langle proof \rangle$

lemma $trans\text{-}on\text{-}lex\text{-}prod[simp]$:
 assumes $trans\text{-}on\ A\ r_A$ and $trans\text{-}on\ B\ r_B$
 shows $trans\text{-}on\ (A \times B)\ (r_A \langle *lex* \rangle r_B)$
 $\langle proof \rangle$

lemma *trans-lex-prod* [*simp,intro!*]: $\text{trans } r_A \implies \text{trans } r_B \implies \text{trans } (r_A <*\text{lex}*> r_B)$
 ⟨*proof*⟩

lemma *total-on-lex-prod*[*simp*]:
 $\text{total-on } A \ r_A \implies \text{total-on } B \ r_B \implies \text{total-on } (A \times B) \ (r_A <*\text{lex}*> r_B)$
 ⟨*proof*⟩

lemma *total-lex-prod*[*simp*]: $\text{total } r_A \implies \text{total } r_B \implies \text{total } (r_A <*\text{lex}*> r_B)$
 ⟨*proof*⟩

lexicographic combinations with measure functions

definition *mlex-prod* :: $('a \Rightarrow \text{nat}) \Rightarrow ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set}$ (**infixr** $<*\text{mlex}*>$ 80)
where $f <*\text{mlex}*> R = \text{inv-image } (\text{less-than } <*\text{lex}*> R) \ (\lambda x. (f \ x, x))$

lemma
wf-mlex: $\text{wf } R \implies \text{wf } (f <*\text{mlex}*> R)$ **and**
mlex-less: $f \ x < f \ y \implies (x, y) \in f <*\text{mlex}*> R$ **and**
mlex-leq: $f \ x \leq f \ y \implies (x, y) \in R \implies (x, y) \in f <*\text{mlex}*> R$ **and**
mlex-iff: $(x, y) \in f <*\text{mlex}*> R \iff f \ x < f \ y \vee f \ x = f \ y \wedge (x, y) \in R$
 ⟨*proof*⟩

Proper subset relation on finite sets.

definition *finite-psubset* :: $('a \text{ set} \times 'a \text{ set}) \text{ set}$
where $\text{finite-psubset} = \{(A, B). A \subset B \wedge \text{finite } B\}$

lemma *wf-finite-psubset*[*simp*]: $\text{wf } \text{finite-psubset}$
 ⟨*proof*⟩

lemma *trans-finite-psubset*: $\text{trans } \text{finite-psubset}$
 ⟨*proof*⟩

lemma *in-finite-psubset*[*simp*]: $(A, B) \in \text{finite-psubset} \iff A \subset B \wedge \text{finite } B$
 ⟨*proof*⟩

max- and min-extension of order to finite sets

inductive-set *max-ext* :: $('a \times 'a) \text{ set} \Rightarrow ('a \text{ set} \times 'a \text{ set}) \text{ set}$
for $R :: ('a \times 'a) \text{ set}$
where *max-extI*[*intro*]:
 $\text{finite } X \implies \text{finite } Y \implies Y \neq \{\} \implies (\bigwedge x. x \in X \implies \exists y \in Y. (x, y) \in R) \implies (X, Y) \in \text{max-ext } R$

lemma *max-ext-wf*:
assumes *wf*: $\text{wf } r$
shows $\text{wf } (\text{max-ext } r)$
 ⟨*proof*⟩

lemma *max-ext-additive*: $(A, B) \in \text{max-ext } R \implies (C, D) \in \text{max-ext } R \implies (A \cup C, B \cup D) \in \text{max-ext } R$
 $\langle \text{proof} \rangle$

definition *min-ext* :: $('a \times 'a) \text{ set} \Rightarrow ('a \text{ set} \times 'a \text{ set}) \text{ set}$
where *min-ext* $r = \{(X, Y) \mid X \text{ } Y. X \neq \{\} \wedge (\forall y \in Y. (\exists x \in X. (x, y) \in r))\}$

lemma *min-ext-wf*:
assumes *wf* r
shows *wf* $(\text{min-ext } r)$
 $\langle \text{proof} \rangle$

22.12.4 Bounded increase must terminate

lemma *wf-bounded-measure*:
fixes *ub* :: $'a \Rightarrow \text{nat}$
and *f* :: $'a \Rightarrow \text{nat}$
assumes $\bigwedge a \ b. (b, a) \in r \implies \text{ub } b \leq \text{ub } a \wedge \text{ub } a \geq f \ b \wedge f \ b > f \ a$
shows *wf* r
 $\langle \text{proof} \rangle$

lemma *wf-bounded-set*:
fixes *ub* :: $'a \Rightarrow 'b \text{ set}$
and *f* :: $'a \Rightarrow 'b \text{ set}$
assumes $\bigwedge a \ b. (b, a) \in r \implies \text{finite } (\text{ub } a) \wedge \text{ub } b \subseteq \text{ub } a \wedge \text{ub } a \supseteq f \ b \wedge f \ b \supset f \ a$
shows *wf* r
 $\langle \text{proof} \rangle$

lemma *finite-subset-wf*:
assumes *finite* A
shows *wf* $\{(X, Y). X \subset Y \wedge Y \subseteq A\}$
 $\langle \text{proof} \rangle$

hide-const (**open**) *acc accp*

22.13 Code Generation Setup

Code equations with *wf* or *wfp* on the left-hand side are not supported by the code generation module because of the *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* :: $('a \times 'a) \text{ set} \Rightarrow \text{bool}$ **where**
 $[\text{code-abbrev}]: \text{wf-code } r \longleftrightarrow \text{wf } r$

definition *wfp-code* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $[\text{code-abbrev}]: \text{wfp-code } R \longleftrightarrow \text{wfp } R$

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 = (λy. if (y, x) ∈ R then f y else undefined)

definition adm-wf :: ('a × 'a) set ⇒ (('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 = (λx. THE y. wfrec-rel R (λf 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)
 ⟨proof⟩

lemma cut-apply: (x, a) ∈ R ⇒ cut f R a x = f x
 ⟨proof⟩

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
 ⟨proof⟩

lemma wfrec-unique:
assumes adm-wf R F wf R
shows ∃!y. wfrec-rel R F x y
 ⟨proof⟩

lemma adm-lemma: adm-wf R (λf x. F (cut f R x) x)
 ⟨proof⟩

lemma wfrec: wf R ⇒ wfrec R F a = F (cut (wfrec R F) R a) a
 ⟨proof⟩

This form avoids giant explosions in proofs. NOTE USE OF ≡.

lemma def-wfrec: f ≡ wfrec R F ⇒ wf R ⇒ f a = F (cut f R a) a
 ⟨proof⟩

23.0.1 Well-founded recursion via genuine fixpoints

lemma *wfrec-fixpoint*:

assumes *wf*: $wf\ R$

and *adm*: $adm\text{-}wf\ R\ F$

shows $wfrec\ R\ F = F\ (wfrec\ R\ F)$

<proof>

lemma *wfrec-def-adm*: $f \equiv wfrec\ R\ F \implies wf\ R \implies adm\text{-}wf\ R\ F \implies f = F\ f$

<proof>

23.1 Wellfoundedness of *same-fst*

definition *same-fst* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow ('b \times 'b)\ set) \Rightarrow (('a \times 'b) \times ('a \times 'b))\ set$

where $same\text{-}fst\ P\ R = \{((x', y'), (x, y)) \mid x' = x \wedge P\ x \wedge (y', y) \in R\ x\}$

— For *wfrec* declarations where the first *n* parameters stay unchanged in the recursive call.

lemma *same-fstI* [*intro!*]: $P\ x \implies (y', y) \in R\ x \implies ((x, y'), (x, y)) \in same\text{-}fst\ P\ R$

<proof>

lemma *wf-same-fst*:

assumes $\bigwedge x. P\ x \implies wf\ (R\ x)$

shows $wf\ (same\text{-}fst\ P\ R)$

<proof>

end

24 Orders as Relations

theory *Order-Relation*

imports *Wfrec*

begin

24.1 Orders on a set

definition *preorder-on* $A\ r \equiv r \subseteq A \times A \wedge refl\text{-}on\ A\ r \wedge trans\ r$

definition *partial-order-on* $A\ r \equiv preorder\text{-}on\ A\ r \wedge antisym\ r$

definition *linear-order-on* $A\ r \equiv partial\text{-}order\text{-}on\ A\ r \wedge total\text{-}on\ A\ r$

definition *strict-linear-order-on* $A\ r \equiv trans\ r \wedge irrefl\ r \wedge total\text{-}on\ A\ r$

definition *well-order-on* $A\ r \equiv linear\text{-}order\text{-}on\ A\ r \wedge wf\ (r - Id)$

lemmas *order-on-defs* =

preorder-on-def partial-order-on-def linear-order-on-def

strict-linear-order-on-def well-order-on-def

lemma *partial-order-onD*:

assumes *partial-order-on A r* **shows** *refl-on A r* **and** *trans r* **and** *antisym r*
and $r \subseteq A \times A$
 $\langle \text{proof} \rangle$

lemma *preorder-on-empty[simp]*: *preorder-on* $\{\}$ $\{\}$
 $\langle \text{proof} \rangle$

lemma *partial-order-on-empty[simp]*: *partial-order-on* $\{\}$ $\{\}$
 $\langle \text{proof} \rangle$

lemma *linear-order-on-empty[simp]*: *linear-order-on* $\{\}$ $\{\}$
 $\langle \text{proof} \rangle$

lemma *well-order-on-empty[simp]*: *well-order-on* $\{\}$ $\{\}$
 $\langle \text{proof} \rangle$

lemma *preorder-on-converse[simp]*: *preorder-on A* $(r^{-1}) = \text{preorder-on A } r$
 $\langle \text{proof} \rangle$

lemma *partial-order-on-converse[simp]*: *partial-order-on A* $(r^{-1}) = \text{partial-order-on A } r$
 $\langle \text{proof} \rangle$

lemma *linear-order-on-converse[simp]*: *linear-order-on A* $(r^{-1}) = \text{linear-order-on A } r$
 $\langle \text{proof} \rangle$

lemma *partial-order-on-acyclic*:
partial-order-on A r $\implies \text{acyclic } (r - \text{Id})$
 $\langle \text{proof} \rangle$

lemma *partial-order-on-well-order-on*:
 $\text{finite } r \implies \text{partial-order-on A } r \implies \text{wf } (r - \text{Id})$
 $\langle \text{proof} \rangle$

lemma *strict-linear-order-on-diff-Id*: *linear-order-on A r* $\implies \text{strict-linear-order-on A } (r - \text{Id})$
 $\langle \text{proof} \rangle$

lemma *linear-order-on-singleton [simp]*: *linear-order-on* $\{x\}$ $\{(x, x)\}$
 $\langle \text{proof} \rangle$

lemma *linear-order-on-acyclic*:
assumes *linear-order-on A r*

shows *acyclic* ($r - Id$)
 $\langle proof \rangle$

lemma *linear-order-on-well-order-on*:
assumes *finite* r
shows *linear-order-on* A $r \longleftrightarrow$ *well-order-on* A r
 $\langle proof \rangle$

24.2 Orders on the field

abbreviation *Refl* $r \equiv$ *refl-on* (*Field* r) r

abbreviation *Preorder* $r \equiv$ *preorder-on* (*Field* r) r

abbreviation *Partial-order* $r \equiv$ *partial-order-on* (*Field* r) r

abbreviation *Total* $r \equiv$ *total-on* (*Field* r) r

abbreviation *Linear-order* $r \equiv$ *linear-order-on* (*Field* r) r

abbreviation *Well-order* $r \equiv$ *well-order-on* (*Field* r) r

lemma *subset-Image-Image-iff*:
 $Preorder\ r \implies A \subseteq Field\ r \implies B \subseteq Field\ r \implies$
 $r \text{ “ } A \subseteq r \text{ “ } B \longleftrightarrow (\forall a \in A. \exists b \in B. (b, a) \in r)$
 $\langle proof \rangle$

lemma *subset-Image1-Image1-iff*:
 $Preorder\ r \implies a \in Field\ r \implies b \in Field\ r \implies r \text{ “ } \{a\} \subseteq r \text{ “ } \{b\} \longleftrightarrow (b, a) \in$
 r
 $\langle proof \rangle$

lemma *Refl-antisym-eq-Image1-Image1-iff*:
assumes *Refl* r
and *as*: *antisym* r
and *abf*: $a \in Field\ r \ b \in Field\ r$
shows $r \text{ “ } \{a\} = r \text{ “ } \{b\} \longleftrightarrow a = b$
 $(is\ ?lhs \longleftrightarrow ?rhs)$
 $\langle proof \rangle$

lemma *Partial-order-eq-Image1-Image1-iff*:
 $Partial-order\ r \implies a \in Field\ r \implies b \in Field\ r \implies r \text{ “ } \{a\} = r \text{ “ } \{b\} \longleftrightarrow a =$
 b
 $\langle proof \rangle$

lemma *Total-Id-Field*:
assumes *Total* r
and *not-Id*: $\neg r \subseteq Id$

shows $\text{Field } r = \text{Field } (r - \text{Id})$
 $\langle \text{proof} \rangle$

24.3 Relations given by a predicate and the field

definition $\text{relation-of} :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ set} \Rightarrow ('a \times 'a) \text{ set}$
where $\text{relation-of } P \ A \equiv \{ (a, b) \in A \times A. P \ a \ b \}$

lemma $\text{refl-relation-ofD}: \text{refl } (\text{relation-of } R \ S) \Longrightarrow \text{reflp-on } S \ R$
 $\langle \text{proof} \rangle$

lemma $\text{irrefl-relation-ofD}: \text{irrefl } (\text{relation-of } R \ S) \Longrightarrow \text{irreflp-on } S \ R$
 $\langle \text{proof} \rangle$

lemma $\text{sym-relation-of[simp]}: \text{sym } (\text{relation-of } R \ S) \longleftrightarrow \text{symp-on } S \ R$
 $\langle \text{proof} \rangle$

lemma $\text{asym-relation-of[simp]}: \text{asym } (\text{relation-of } R \ S) \longleftrightarrow \text{asymp-on } S \ R$
 $\langle \text{proof} \rangle$

lemma $\text{antisym-relation-of[simp]}: \text{antisym } (\text{relation-of } R \ S) \longleftrightarrow \text{antisymp-on } S \ R$
 $\langle \text{proof} \rangle$

lemma $\text{trans-relation-of[simp]}: \text{trans } (\text{relation-of } R \ S) \longleftrightarrow \text{transp-on } S \ R$
 $\langle \text{proof} \rangle$

lemma $\text{total-relation-ofD}: \text{total } (\text{relation-of } R \ S) \Longrightarrow \text{totalp-on } S \ R$
 $\langle \text{proof} \rangle$

lemma $\text{Field-relation-of}:$
assumes $\text{relation-of } P \ A \subseteq A \times A$ **and** $\text{refl-on } A \ (\text{relation-of } P \ A)$
shows $\text{Field } (\text{relation-of } P \ A) = A$
 $\langle \text{proof} \rangle$

lemma $\text{partial-order-on-relation-ofI}:$
assumes $\text{refl}: \bigwedge a. a \in A \Longrightarrow P \ a \ a$
and $\text{trans}: \bigwedge a \ b \ c. \llbracket a \in A; b \in A; c \in A \rrbracket \Longrightarrow P \ a \ b \Longrightarrow P \ b \ c \Longrightarrow P \ a \ c$
and $\text{antisym}: \bigwedge a \ b. \llbracket a \in A; b \in A \rrbracket \Longrightarrow P \ a \ b \Longrightarrow P \ b \ a \Longrightarrow a = b$
shows $\text{partial-order-on } A \ (\text{relation-of } P \ A)$
 $\langle \text{proof} \rangle$

lemma $\text{Partial-order-relation-ofI}:$
assumes $\text{partial-order-on } A \ (\text{relation-of } P \ A)$
shows $\text{Partial-order } (\text{relation-of } P \ A)$
 $\langle \text{proof} \rangle$

24.4 Orders on a type

abbreviation $\text{strict-linear-order} \equiv \text{strict-linear-order-on } \text{UNIV}$

abbreviation $linear\text{-}order \equiv linear\text{-}order\text{-}on\ UNIV$

abbreviation $well\text{-}order \equiv well\text{-}order\text{-}on\ UNIV$

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

corollary $well\text{-}order\text{-}on\text{-}domain$: $well\text{-}order\text{-}on\ A\ r \implies (a, b) \in r \implies a \in A \wedge b \in A$
 $\langle proof \rangle$

lemma $well\text{-}order\text{-}on\text{-}Field$: $well\text{-}order\text{-}on\ A\ r \implies A = Field\ r$
 $\langle proof \rangle$

lemma $well\text{-}order\text{-}on\text{-}Well\text{-}order$: $well\text{-}order\text{-}on\ A\ r \implies A = Field\ r \wedge Well\text{-}order\ r$
 $\langle proof \rangle$

lemma $Total\text{-}subset\text{-}Id$:
assumes $Total\ r$
and $r \subseteq Id$
shows $r = \{\} \vee (\exists a. r = \{(a, a)\})$
 $\langle proof \rangle$

lemma $Linear\text{-}order\text{-}in\text{-}diff\text{-}Id$:
assumes $Linear\text{-}order\ r$
and $a \in Field\ r$
and $b \in Field\ r$
shows $(a, b) \in r \longleftrightarrow (b, a) \notin r - Id$
 $\langle proof \rangle$

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., $underS\ a$ 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 $under\ r\ a \equiv \{b. (b, a) \in r\}$

definition $underS :: 'a \text{ rel} \Rightarrow 'a \Rightarrow 'a \text{ set}$
where $underS \ r \ a \equiv \{b. \ b \neq a \wedge (b, a) \in r\}$

definition $Under :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$
where $Under \ r \ A \equiv \{b \in Field \ r. \ \forall a \in A. \ (b, a) \in r\}$

definition $UnderS :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$
where $UnderS \ r \ A \equiv \{b \in Field \ r. \ \forall a \in A. \ b \neq a \wedge (b, a) \in r\}$

definition $above :: 'a \text{ rel} \Rightarrow 'a \Rightarrow 'a \text{ set}$
where $above \ r \ a \equiv \{b. \ (a, b) \in r\}$

definition $aboveS :: 'a \text{ rel} \Rightarrow 'a \Rightarrow 'a \text{ set}$
where $aboveS \ r \ a \equiv \{b. \ b \neq a \wedge (a, b) \in r\}$

definition $Above :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$
where $Above \ r \ A \equiv \{b \in Field \ r. \ \forall a \in A. \ (a, b) \in r\}$

definition $AboveS :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$
where $AboveS \ r \ A \equiv \{b \in Field \ r. \ \forall a \in A. \ b \neq a \wedge (a, b) \in r\}$

definition $ofilter :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow bool$
where $ofilter \ r \ A \equiv A \subseteq Field \ r \wedge (\forall a \in A. \ under \ r \ a \subseteq A)$

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 $underS\text{-subset-under}$: $underS \ r \ a \subseteq under \ r \ a$
 $\langle proof \rangle$

lemma $underS\text{-notIn}$: $a \notin underS \ r \ a$
 $\langle proof \rangle$

lemma $Refl\text{-under-in}$: $Refl \ r \Longrightarrow a \in Field \ r \Longrightarrow a \in under \ r \ a$
 $\langle proof \rangle$

lemma $AboveS\text{-disjoint}$: $A \cap (AboveS \ r \ A) = \{\}$
 $\langle proof \rangle$

lemma $in\text{-AboveS-underS}$: $a \in Field \ r \Longrightarrow a \in AboveS \ r \ (underS \ r \ a)$
 $\langle proof \rangle$

lemma $Refl\text{-under-underS}$: $Refl \ r \Longrightarrow a \in Field \ r \Longrightarrow under \ r \ a = underS \ r \ a \cup \{a\}$
 $\langle proof \rangle$

lemma $underS\text{-empty}$: $a \notin Field \ r \Longrightarrow underS \ r \ a = \{\}$
 $\langle proof \rangle$

lemma *under-Field*: $\text{under } r \ a \subseteq \text{Field } r$
 $\langle \text{proof} \rangle$

lemma *underS-Field*: $\text{underS } r \ a \subseteq \text{Field } r$
 $\langle \text{proof} \rangle$

lemma *underS-Field2*: $a \in \text{Field } r \implies \text{underS } r \ a \subset \text{Field } r$
 $\langle \text{proof} \rangle$

lemma *underS-Field3*: $\text{Field } r \neq \{\} \implies \text{underS } r \ a \subset \text{Field } r$
 $\langle \text{proof} \rangle$

lemma *AboveS-Field*: $\text{AboveS } r \ A \subseteq \text{Field } r$
 $\langle \text{proof} \rangle$

lemma *under-incr*:
assumes *trans* r
and $(a, b) \in r$
shows $\text{under } r \ a \subseteq \text{under } r \ b$
 $\langle \text{proof} \rangle$

lemma *underS-incr*:
assumes *trans* r
and *antisym* r
and $ab: (a, b) \in r$
shows $\text{underS } r \ a \subseteq \text{underS } r \ b$
 $\langle \text{proof} \rangle$

lemma *underS-incl-iff*:
assumes *LO*: *Linear-order* r
and *INa*: $a \in \text{Field } r$
and *INb*: $b \in \text{Field } r$
shows $\text{underS } r \ a \subseteq \text{underS } r \ b \longleftrightarrow (a, b) \in r$
(is ?lhs \longleftrightarrow ?rhs)
 $\langle \text{proof} \rangle$

lemma *finite-Partial-order-induct*[*consumes* \mathcal{I} , *case-names* *step*]:
assumes *Partial-order* r
and $x \in \text{Field } r$
and *finite* r
and *step*: $\bigwedge x. x \in \text{Field } r \implies (\bigwedge y. y \in \text{aboveS } r \ x \implies P \ y) \implies P \ x$
shows $P \ x$
 $\langle \text{proof} \rangle$

lemma *finite-Linear-order-induct*[*consumes* \mathcal{I} , *case-names* *step*]:
assumes *Linear-order* r
and $x \in \text{Field } r$
and *finite* r
and *step*: $\bigwedge x. x \in \text{Field } r \implies (\bigwedge y. y \in \text{aboveS } r \ x \implies P \ y) \implies P \ x$

shows $P\ x$
 $\langle proof \rangle$

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 any element is well-founded.

lemma *trans-wf-iff*:

assumes *trans* r

shows $wf\ r \longleftrightarrow (\forall a. wf\ (r \cap (r^{-1} \text{“}\{a\} \times r^{-1} \text{“}\{a\})))$

$\langle proof \rangle$

A transitive relation is well-founded if all initial segments are finite.

corollary *wf-finite-segments*:

assumes *irrefl* r **and** *trans* r **and** $\bigwedge x. finite\ \{y. (y, x) \in r\}$

shows $wf\ r$

$\langle proof \rangle$

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*: $wf\ r \longleftrightarrow (\forall A. A \subseteq Field\ r \wedge A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a', a) \notin r))$

$\langle proof \rangle$

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 $wf\ (r - Id) \longleftrightarrow (\forall A \subseteq Field\ r. A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a, a') \in r))$

$\langle proof \rangle$

corollary *Linear-order-Well-order-iff:*

Linear-order $r \implies$

Well-order $r \iff (\forall A \subseteq \text{Field } r. A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a, a') \in r))$

<proof>

end

25 Hilbert’s Epsilon-Operator and the Axiom of Choice

theory *Hilbert-Choice*

imports *Wellfounded*

keywords *specification :: thy-goal-defn*

begin

25.1 Hilbert’s epsilon

axiomatization *Eps* :: ($'a \Rightarrow \text{bool}$) $\Rightarrow 'a$

where *someI*: $P\ x \implies P\ (\text{Eps } P)$

syntax (*epsilon*)

-Eps :: $\text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \ (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } \epsilon \rangle \rangle \epsilon \text{ -./ -} \rangle) [0, 10] 10)$

syntax (*input*)

-Eps :: $\text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \ (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } @ \rangle \rangle @ \text{ -./ -} \rangle) [0, 10] 10)$

syntax

-Eps :: $\text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \ (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } \text{SOME} \rangle \rangle \text{SOME -./ -} \rangle) [0, 10] 10)$

syntax-consts *-Eps* \equiv *Eps*

translations

SOME $x. P \equiv \text{CONST } \text{Eps } (\lambda x. P)$

<ML>

definition *inv-into* :: ($'a \text{ set} \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$) **where**

inv-into $A\ f = (\lambda x. \text{SOME } y. y \in A \wedge f\ y = x)$

lemma *inv-into-def2*: *inv-into* $A\ f\ x = (\text{SOME } y. y \in A \wedge f\ y = x)$

<proof>

abbreviation *inv* :: ($'a \Rightarrow 'b$) $\Rightarrow ('b \Rightarrow 'a)$ **where**

inv $\equiv \text{inv-into } \text{UNIV}$

25.2 Hilbert’s Epsilon-operator

lemma *Eps-cong*:

assumes $\bigwedge x. P\ x = Q\ x$

shows $Eps\ P = Eps\ Q$
 $\langle proof \rangle$

Easier to use than *someI* if the witness comes from an existential formula.

lemma *someI-ex* [*elim?*]: $\exists x. P\ x \implies P\ (SOME\ x. P\ x)$
 $\langle proof \rangle$

lemma *some-eq-imp*:
assumes $Eps\ P = a\ P\ b$ **shows** $P\ a$
 $\langle proof \rangle$

Easier to use than *someI* because the conclusion has only one occurrence of P .

lemma *someI2*: $P\ a \implies (\bigwedge x. P\ x \implies Q\ x) \implies Q\ (SOME\ x. P\ x)$
 $\langle proof \rangle$

Easier to use than *someI2* if the witness comes from an existential formula.

lemma *someI2-ex*: $\exists a. P\ a \implies (\bigwedge x. P\ x \implies Q\ x) \implies Q\ (SOME\ x. P\ x)$
 $\langle proof \rangle$

lemma *someI2-bex*: $\exists a \in A. P\ a \implies (\bigwedge x. x \in A \wedge P\ x \implies Q\ x) \implies Q\ (SOME\ x. x \in A \wedge P\ x)$
 $\langle proof \rangle$

lemma *some-equality* [*intro*]: $P\ a \implies (\bigwedge x. P\ x \implies x = a) \implies (SOME\ x. P\ x) = a$
 $\langle proof \rangle$

lemma *some1-equality*: $\exists! x. P\ x \implies P\ a \implies (SOME\ x. P\ x) = a$
 $\langle proof \rangle$

lemma *some-eq-ex*: $P\ (SOME\ x. P\ x) \longleftrightarrow (\exists x. P\ x)$
 $\langle proof \rangle$

lemma *some-in-eq*: $(SOME\ x. x \in A) \in A \longleftrightarrow A \neq \{\}$
 $\langle proof \rangle$

lemma *some-eq-trivial* [*simp*]: $(SOME\ y. y = x) = x$
 $\langle proof \rangle$

lemma *some-sym-eq-trivial* [*simp*]: $(SOME\ y. x = y) = x$
 $\langle proof \rangle$

25.3 Axiom of Choice, Proved Using the Description Operator

lemma *choice*: $\forall x. \exists y. Q\ x\ y \implies \exists f. \forall x. Q\ x\ (f\ x)$
 $\langle proof \rangle$

lemma *bchoice*: $\forall x \in S. \exists y. Q\ x\ y \implies \exists f. \forall x \in S. Q\ x\ (f\ x)$
 $\langle proof \rangle$

lemma *choice-iff*: $(\forall x. \exists y. Q\ x\ y) \longleftrightarrow (\exists f. \forall x. Q\ x\ (f\ x))$
 $\langle proof \rangle$

lemma *choice-iff'*: $(\forall x. P\ x \longrightarrow (\exists y. Q\ x\ y)) \longleftrightarrow (\exists f. \forall x. P\ x \longrightarrow Q\ x\ (f\ x))$
 $\langle proof \rangle$

lemma *bchoice-iff*: $(\forall x \in S. \exists y. Q\ x\ y) \longleftrightarrow (\exists f. \forall x \in S. Q\ x\ (f\ x))$
 $\langle proof \rangle$

lemma *bchoice-iff'*: $(\forall x \in S. P\ x \longrightarrow (\exists y. Q\ x\ y)) \longleftrightarrow (\exists f. \forall x \in S. P\ x \longrightarrow Q\ x\ (f\ x))$
 $\langle proof \rangle$

lemma *dependent-nat-choice*:
assumes 1: $\exists x. P\ 0\ x$
and 2: $\bigwedge x\ n. P\ n\ x \implies \exists y. P\ (Suc\ n)\ y \wedge Q\ n\ x\ y$
shows $\exists f. \forall n. P\ n\ (f\ n) \wedge Q\ n\ (f\ n)\ (f\ (Suc\ n))$
 $\langle proof \rangle$

lemma *finite-subset-Union*:
assumes *finite* $A \subseteq \bigcup \mathcal{B}$
obtains \mathcal{F} **where** *finite* \mathcal{F} $\mathcal{F} \subseteq \mathcal{B}$ $A \subseteq \bigcup \mathcal{F}$
 $\langle proof \rangle$

25.4 Getting an element of a nonempty set

definition *some-elem* :: $'a\ set \Rightarrow 'a$
where *some-elem* $A = (SOME\ x. x \in A)$

lemma *some-elem-eq* [*simp*]: *some-elem* $\{x\} = x$
 $\langle proof \rangle$

lemma *some-elem-nonempty*: $A \neq \{\} \implies \text{some-elem}\ A \in A$
 $\langle proof \rangle$

lemma *is-singleton-some-elem*: $\text{is-singleton}\ A \longleftrightarrow A = \{\text{some-elem}\ A\}$
 $\langle proof \rangle$

lemma *some-elem-image-unique*:
assumes $A \neq \{\}$
and *: $\bigwedge y. y \in A \implies f\ y = a$
shows $\text{some-elem}\ (f\ ` A) = a$
 $\langle proof \rangle$

25.5 Function Inverse

lemma *inv-def*: $\text{inv}\ f = (\lambda y. SOME\ x. f\ x = y)$

$\langle \text{proof} \rangle$

lemma *inv-into-into*: $x \in f \text{ ‘ } A \implies \text{inv-into } A f x \in A$
 $\langle \text{proof} \rangle$

lemma *inv-identity* [simp]: $\text{inv } (\lambda a. a) = (\lambda a. a)$
 $\langle \text{proof} \rangle$

lemma *inv-id* [simp]: $\text{inv id} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *inv-into-f-f* [simp]: $\text{inj-on } f A \implies x \in A \implies \text{inv-into } A f (f x) = x$
 $\langle \text{proof} \rangle$

lemma *inv-f-f*: $\text{inj } f \implies \text{inv } f (f x) = x$
 $\langle \text{proof} \rangle$

lemma *f-inv-into-f*: $y \in f \text{ ‘ } A \implies f (\text{inv-into } A f y) = y$
 $\langle \text{proof} \rangle$

lemma *inv-into-f-eq*: $\text{inj-on } f A \implies x \in A \implies f x = y \implies \text{inv-into } A f y = x$
 $\langle \text{proof} \rangle$

lemma *inv-f-eq*: $\text{inj } f \implies f x = y \implies \text{inv } f y = x$
 $\langle \text{proof} \rangle$

lemma *inj-imp-inv-eq*: $\text{inj } f \implies \forall x. f (g x) = x \implies \text{inv } f = g$
 $\langle \text{proof} \rangle$

But is it useful?

lemma *inj-transfer*:
assumes *inj*: $\text{inj } f$
and *minor*: $\bigwedge y. y \in \text{range } f \implies P (\text{inv } f y)$
shows $P x$
 $\langle \text{proof} \rangle$

lemma *inj-iff*: $\text{inj } f \longleftrightarrow \text{inv } f \circ f = \text{id}$
 $\langle \text{proof} \rangle$

lemma *inv-o-cancel*[simp]: $\text{inj } f \implies \text{inv } f \circ f = \text{id}$
 $\langle \text{proof} \rangle$

lemma *o-inv-o-cancel*[simp]: $\text{inj } f \implies g \circ \text{inv } f \circ f = g$
 $\langle \text{proof} \rangle$

lemma *inv-into-image-cancel*[simp]: $\text{inj-on } f A \implies S \subseteq A \implies \text{inv-into } A f \text{ ‘ } f \text{ ‘ } S = S$
 $\langle \text{proof} \rangle$

lemma *inj-imp-surj-inv*: $\text{inj } f \implies \text{surj } (\text{inv } f)$
 $\langle \text{proof} \rangle$

lemma *surj-f-inv-f*: $\text{surj } f \implies f (\text{inv } f y) = y$
 $\langle \text{proof} \rangle$

lemma *bij-inv-eq-iff*: $\text{bij } p \implies x = \text{inv } p y \longleftrightarrow p x = y$
 $\langle \text{proof} \rangle$

lemma *inv-into-injective*:
 assumes *eq*: $\text{inv-into } A f x = \text{inv-into } A f y$
 and $x: x \in f'A$
 and $y: y \in f'A$
 shows $x = y$
 $\langle \text{proof} \rangle$

lemma *inj-on-inv-into*: $B \subseteq f'A \implies \text{inj-on } (\text{inv-into } A f) B$
 $\langle \text{proof} \rangle$

lemma *inj-imp-bij-betw-inv*: $\text{inj } f \implies \text{bij-betw } (\text{inv } f) (f ' M) M$
 $\langle \text{proof} \rangle$

lemma *bij-betw-inv-into*: $\text{bij-betw } f A B \implies \text{bij-betw } (\text{inv-into } A f) B A$
 $\langle \text{proof} \rangle$

lemma *surj-imp-inj-inv*: $\text{surj } f \implies \text{inj } (\text{inv } f)$
 $\langle \text{proof} \rangle$

lemma *surj-iff*: $\text{surj } f \longleftrightarrow f \circ \text{inv } f = \text{id}$
 $\langle \text{proof} \rangle$

lemma *surj-iff-all*: $\text{surj } f \longleftrightarrow (\forall x. f (\text{inv } f x) = x)$
 $\langle \text{proof} \rangle$

lemma *surj-imp-inv-eq*:
 assumes *surj* *f* and *gf*: $\bigwedge x. g (f x) = x$
 shows $\text{inv } f = g$
 $\langle \text{proof} \rangle$

lemma *bij-imp-bij-inv*: $\text{bij } f \implies \text{bij } (\text{inv } f)$
 $\langle \text{proof} \rangle$

lemma *inv-equality*: $(\bigwedge x. g (f x) = x) \implies (\bigwedge y. f (g y) = y) \implies \text{inv } f = g$
 $\langle \text{proof} \rangle$

lemma *inv-inv-eq*: $\text{bij } f \implies \text{inv } (\text{inv } f) = f$
 $\langle \text{proof} \rangle$

bij (*inv* *f*) implies little about *f*. Consider $f :: \text{bool} \Rightarrow \text{bool}$ such that *f* *True*

$= f \text{ False} = \text{True}$. Then it is consistent with axiom *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 $\text{inj } f$, $\text{surj } f$ and $\text{inv } (\text{inv } f) = f$ all fail.

lemma *inv-into-comp*:

$\text{inj-on } f \ (g \text{ ‘ } A) \implies \text{inj-on } g \ A \implies x \in f \text{ ‘ } g \text{ ‘ } A \implies$
 $\text{inv-into } A \ (f \circ g) \ x = (\text{inv-into } A \ g \circ \text{inv-into } (g \text{ ‘ } A) \ f) \ x$
 $\langle \text{proof} \rangle$

lemma *o-inv-distrib*: $\text{bij } f \implies \text{bij } g \implies \text{inv } (f \circ g) = \text{inv } g \circ \text{inv } f$
 $\langle \text{proof} \rangle$

lemma *image-f-inv-f*: $\text{surj } f \implies f \text{ ‘ } (\text{inv } f \text{ ‘ } A) = A$
 $\langle \text{proof} \rangle$

lemma *image-inv-f-f*: $\text{inj } f \implies \text{inv } f \text{ ‘ } (f \text{ ‘ } A) = A$
 $\langle \text{proof} \rangle$

lemma *bij-image-Collect-eq*:

assumes $\text{bij } f$
shows $f \text{ ‘ } \text{Collect } P = \{y. P \ (\text{inv } f \ y)\}$
 $\langle \text{proof} \rangle$

lemma *bij-vimage-eq-inv-image*:

assumes $\text{bij } f$
shows $f \text{ - ‘ } A = \text{inv } f \text{ ‘ } A$
 $\langle \text{proof} \rangle$

lemma *inv-fn-o-fn-is-id*:

fixes $f::'a \Rightarrow 'a$
assumes $\text{bij } f$
shows $((\text{inv } f) \text{ } \sim n) \circ (f \text{ } \sim n) = (\lambda x. x)$
 $\langle \text{proof} \rangle$

lemma *fn-o-inv-fn-is-id*:

fixes $f::'a \Rightarrow 'a$
assumes $\text{bij } f$
shows $(f \text{ } \sim n) \circ ((\text{inv } f) \text{ } \sim n) = (\lambda x. x)$
 $\langle \text{proof} \rangle$

lemma *inv-fn*:

fixes $f::'a \Rightarrow 'a$
assumes $\text{bij } f$
shows $\text{inv } (f \text{ } \sim n) = ((\text{inv } f) \text{ } \sim n)$
 $\langle \text{proof} \rangle$

lemma *funpow-inj-finite*:

assumes $\langle \text{inj } p \rangle \ \langle \text{finite } \{y. \exists n. y = (p \text{ } \sim n) \ x\} \rangle$
obtains n **where** $\langle n > 0 \rangle \ \langle (p \text{ } \sim n) \ x = x \rangle$
 $\langle \text{proof} \rangle$

lemma *mono-inv*:
fixes $f :: 'a::\text{linorder} \Rightarrow 'b::\text{linorder}$
assumes $\text{mono } f \text{ bij } f$
shows $\text{mono } (\text{inv } f)$
 $\langle \text{proof} \rangle$

lemma *strict-mono-inv-on-range*:
fixes $f :: 'a::\text{linorder} \Rightarrow 'b::\text{order}$
assumes $\text{strict-mono } f$
shows $\text{strict-mono-on } (\text{range } f) (\text{inv } f)$
 $\langle \text{proof} \rangle$

lemma *mono-bij-Inf*:
fixes $f :: 'a::\text{complete-linorder} \Rightarrow 'b::\text{complete-linorder}$
assumes $\text{mono } f \text{ bij } f$
shows $f (\text{Inf } A) = \text{Inf } (f'A)$
 $\langle \text{proof} \rangle$

lemma *finite-fun-UNIVD1*:
assumes $\text{fin}: \text{finite } (\text{UNIV} :: ('a \Rightarrow 'b) \text{ set})$
and $\text{card}: \text{card } (\text{UNIV} :: 'b \text{ set}) \neq \text{Suc } 0$
shows $\text{finite } (\text{UNIV} :: 'a \text{ set})$
 $\langle \text{proof} \rangle$

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 $\exists f::\text{nat} \Rightarrow 'a. \text{inj } f \wedge \text{range } f \subseteq S$
— Courtesy of Stephan Merz
 $\langle \text{proof} \rangle$

lemma *infinite-iff-countable-subset*: $\neg \text{finite } S \longleftrightarrow (\exists f::\text{nat} \Rightarrow 'a. \text{inj } f \wedge \text{range } f \subseteq S)$
— Courtesy of Stephan Merz
 $\langle \text{proof} \rangle$

lemma *image-inv-into-cancel*:
assumes $\text{surj}: f'A = A'$
and $\text{sub}: B' \subseteq A'$
shows $f'((\text{inv-into } A f)'B') = B'$

$\langle \text{proof} \rangle$

lemma *inv-into-inv-into-eq*:
assumes *bij-betw* f A A'
and $a: a \in A$
shows *inv-into* A' (*inv-into* A f) $a = f a$
 $\langle \text{proof} \rangle$

lemma *inj-on-iff-surj*:
assumes $A \neq \{\}$
shows $(\exists f. \text{inj-on } f \ A \wedge f' \ A \subseteq A') \longleftrightarrow (\exists g. g' \ A' = A)$
 $\langle \text{proof} \rangle$

lemma *Ex-inj-on-UNION-Sigma*:
 $\exists f. (\text{inj-on } f \ (\bigcup i \in I. A \ i) \wedge f' \ (\bigcup i \in I. A \ i) \subseteq (\text{SIGMA } i : I. A \ i))$
 $\langle \text{proof} \rangle$

lemma *inv-unique-comp*:
assumes *fg*: $f \circ g = \text{id}$
and *gf*: $g \circ f = \text{id}$
shows *inv* $f = g$
 $\langle \text{proof} \rangle$

lemma *subset-image-inj*:
 $S \subseteq f' \ T \longleftrightarrow (\exists U. U \subseteq T \wedge \text{inj-on } f \ U \wedge S = f' \ U)$
 $\langle \text{proof} \rangle$

25.6 Other Consequences of Hilbert’s Epsilon

Hilbert’s Epsilon and the *split* Operator

Looping simprule!

lemma *split-paired-Eps*: $(\text{SOME } x. P \ x) = (\text{SOME } (a, b). P \ (a, b))$
 $\langle \text{proof} \rangle$

lemma *Eps-case-prod*: $\text{Eps } (\text{case-prod } P) = (\text{SOME } xy. P \ (\text{fst } xy) \ (\text{snd } xy))$
 $\langle \text{proof} \rangle$

lemma *Eps-case-prod-eq [simp]*: $(\text{SOME } (x', y'). x = x' \wedge y = y') = (x, y)$
 $\langle \text{proof} \rangle$

A relation is wellfounded iff it has no infinite descending chain.

lemma *wf-iff-no-infinite-down-chain*: $wf \ r \longleftrightarrow (\nexists f. \forall i. (f \ (\text{Suc } i), f \ i) \in r)$
(is - $\longleftrightarrow \neg ?ex$)
 $\langle \text{proof} \rangle$

lemma *wf-no-infinite-down-chainE*:
assumes *wf* r
obtains k **where** $(f \ (\text{Suc } k), f \ k) \notin r$

$\langle proof \rangle$

A dynamically-scoped fact for TFL

lemma *tfl-some*: $\forall P x. P x \longrightarrow P (Eps P)$
 $\langle proof \rangle$

25.7 An aside: bounded accessible part

Finite monotone eventually stable sequences

lemma *finite-mono-remains-stable-implies-strict-prefix*:
fixes $f :: nat \Rightarrow 'a::order$
assumes $S: finite (range f) \ mono f$
and $eq: \forall n. f n = f (Suc n) \longrightarrow f (Suc n) = f (Suc (Suc n))$
shows $\exists N. (\forall n \leq N. \forall m \leq N. m < n \longrightarrow f m < f n) \wedge (\forall n \geq N. f N = f n)$
 $\langle proof \rangle$

lemma *finite-mono-strict-prefix-implies-finite-fixpoint*:
fixes $f :: nat \Rightarrow 'a \ set$
assumes $S: \bigwedge i. f i \subseteq S \ finite S$
and $ex: \exists N. (\forall n \leq N. \forall m \leq N. m < n \longrightarrow f m \subset f n) \wedge (\forall n \geq N. f N = f n)$
shows $f (card S) = (\bigcup n. f n)$
 $\langle proof \rangle$

25.8 More on injections, bijections, and inverses

locale *bijection* =
fixes $f :: 'a \Rightarrow 'a$
assumes *bij*: $bij f$
begin

lemma *bij-inv*: $bij (inv f)$
 $\langle proof \rangle$

lemma *surj [simp]*: $surj f$
 $\langle proof \rangle$

lemma *inj*: $inj f$
 $\langle proof \rangle$

lemma *surj-inv [simp]*: $surj (inv f)$
 $\langle proof \rangle$

lemma *inj-inv*: $inj (inv f)$
 $\langle proof \rangle$

lemma *eqI*: $f a = f b \Longrightarrow a = b$
 $\langle proof \rangle$

lemma *eq-iff [simp]*: $f a = f b \longleftrightarrow a = b$

$\langle \text{proof} \rangle$

lemma *eq-invI*: $\text{inv } f \ a = \text{inv } f \ b \implies a = b$
 $\langle \text{proof} \rangle$

lemma *eq-inv-iff [simp]*: $\text{inv } f \ a = \text{inv } f \ b \longleftrightarrow a = b$
 $\langle \text{proof} \rangle$

lemma *inv-left [simp]*: $\text{inv } f \ (f \ a) = a$
 $\langle \text{proof} \rangle$

lemma *inv-comp-left [simp]*: $\text{inv } f \circ f = \text{id}$
 $\langle \text{proof} \rangle$

lemma *inv-right [simp]*: $f \ (\text{inv } f \ a) = a$
 $\langle \text{proof} \rangle$

lemma *inv-comp-right [simp]*: $f \circ \text{inv } f = \text{id}$
 $\langle \text{proof} \rangle$

lemma *inv-left-eq-iff [simp]*: $\text{inv } f \ a = b \longleftrightarrow f \ b = a$
 $\langle \text{proof} \rangle$

lemma *inv-right-eq-iff [simp]*: $b = \text{inv } f \ a \longleftrightarrow f \ b = a$
 $\langle \text{proof} \rangle$

end

lemma *infinite-imp-bij-betw*:
assumes *infinite*: $\neg \text{finite } A$
shows $\exists h. \text{bij-betw } h \ A \ (A - \{a\})$
 $\langle \text{proof} \rangle$

lemma *infinite-imp-bij-betw2*:
assumes $\neg \text{finite } A$
shows $\exists h. \text{bij-betw } h \ A \ (A \cup \{a\})$
 $\langle \text{proof} \rangle$

lemma *bij-betw-inv-into-left*: $\text{bij-betw } f \ A \ A' \implies a \in A \implies \text{inv-into } A \ f \ (f \ a) = a$
 $\langle \text{proof} \rangle$

lemma *bij-betw-inv-into-right*: $\text{bij-betw } f \ A \ A' \implies a' \in A' \implies f \ (\text{inv-into } A \ f \ a') = a'$
 $\langle \text{proof} \rangle$

lemma *bij-betw-inv-into-subset*:
 $\text{bij-betw } f \ A \ A' \implies B \subseteq A \implies f \ ` \ B = B' \implies \text{bij-betw } (\text{inv-into } A \ f) \ B' \ B$
 $\langle \text{proof} \rangle$

25.9 Specification package – Hilbertized version

lemma *exE-some*: $Ex\ P \implies c \equiv Eps\ P \implies P\ c$
 $\langle proof \rangle$

$\langle ML \rangle$

25.10 Complete Distributive Lattices – Properties depending on Hilbert Choice

context *complete-distrib-lattice*
begin

lemma *Sup-Inf*: $\bigsqcup (Inf\ 'A) = \bigsqcap (Sup\ ' \{f\ 'A\} f. \forall B \in A. f\ B \in B)$
 $\langle proof \rangle$

lemma *dual-complete-distrib-lattice*:
class.*complete-distrib-lattice* *Sup Inf sup* (\geq) ($>$) *inf* $\top \perp$
 $\langle proof \rangle$

lemma *sup-Inf*: $a \sqcup \bigsqcap B = \bigsqcap ((\sqcup)\ a\ 'B)$
 $\langle proof \rangle$

lemma *inf-Sup*: $a \sqcap \bigsqcup B = \bigsqcup ((\sqcap)\ a\ 'B)$
 $\langle proof \rangle$

lemma *INF-SUP*: $(\bigsqcap y. \bigsqcup x. P\ x\ y) = (\bigsqcup f. \bigsqcap x. P\ (f\ x)\ x)$
 $\langle proof \rangle$

lemma *INF-SUP-set*: $(\bigsqcap B \in A. \bigsqcup (g\ 'B)) = (\bigsqcup B \in \{f\ 'A\} f. \forall C \in A. f\ C \in C). \bigsqcap (g\ 'B)$
 $(is\ - = (\bigsqcup B \in ?F. -))$
 $\langle proof \rangle$

lemma *SUP-INF*: $(\bigsqcup y. \bigsqcap x. P\ x\ y) = (\bigsqcap x. \bigsqcup y. P\ (x\ y)\ y)$
 $\langle proof \rangle$

lemma *SUP-INF-set*: $(\bigsqcup x \in A. \bigsqcap (g\ 'x)) = (\bigsqcap x \in \{f\ 'A\} f. \forall Y \in A. f\ Y \in Y). \bigsqcup (g\ 'x)$
 $\langle proof \rangle$

end

context *complete-distrib-lattice*
begin

lemma *sup-INF*: $a \sqcup (\bigsqcap b \in B. f\ b) = (\bigsqcap b \in B. a \sqcup f\ b)$
 $\langle proof \rangle$

lemma *inf-SUP*: $a \sqcap (\bigsqcup b \in B. f\ b) = (\bigsqcup b \in B. a \sqcap f\ b)$
 $\langle proof \rangle$

lemma *Inf-sup*: $\bigcap B \sqcup a = (\bigcap b \in B. b \sqcup a)$
 $\langle proof \rangle$

lemma *Sup-inf*: $\bigsqcup B \sqcap a = (\bigsqcup b \in B. b \sqcap a)$
 $\langle proof \rangle$

lemma *INF-sup*: $(\bigcap b \in B. f\ b) \sqcup a = (\bigcap b \in B. f\ b \sqcup a)$
 $\langle proof \rangle$

lemma *SUP-inf*: $(\bigsqcup b \in B. f\ b) \sqcap a = (\bigsqcup b \in B. f\ b \sqcap a)$
 $\langle proof \rangle$

lemma *Inf-sup-eq-top-iff*: $(\bigcap B \sqcup a = \top) \longleftrightarrow (\forall b \in B. b \sqcup a = \top)$
 $\langle proof \rangle$

lemma *Sup-inf-eq-bot-iff*: $(\bigsqcup B \sqcap a = \perp) \longleftrightarrow (\forall b \in B. b \sqcap a = \perp)$
 $\langle proof \rangle$

lemma *INF-sup-distrib2*: $(\bigcap a \in A. f\ a) \sqcup (\bigcap b \in B. g\ b) = (\bigcap a \in A. \bigcap b \in B. f\ a \sqcup g\ b)$
 $\langle proof \rangle$

lemma *SUP-inf-distrib2*: $(\bigsqcup a \in A. f\ a) \sqcap (\bigsqcup b \in B. g\ b) = (\bigsqcup a \in A. \bigsqcup b \in B. f\ a \sqcap g\ b)$
 $\langle proof \rangle$

end

instantiation *set* :: (type) complete-distrib-lattice

begin

instance $\langle proof \rangle$

end

instance *set* :: (type) complete-boolean-algebra $\langle proof \rangle$

instantiation *fun* :: (type, complete-distrib-lattice) complete-distrib-lattice

begin

instance $\langle proof \rangle$

end

instance *fun* :: (type, complete-boolean-algebra) complete-boolean-algebra $\langle proof \rangle$

context *complete-linorder*

begin

subclass *complete-distrib-lattice*

$\langle proof \rangle$
end

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 $A :: 'a\ set$
and $P :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle \sqsubseteq \rangle$ 50)
begin

abbreviation $Peq :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle \sqsubseteq \rangle$ 50)
where $x \sqsubseteq y \equiv P^{==} x\ y$

A chain is a totally ordered subset of A .

definition $chain :: 'a\ set \Rightarrow bool$
where $chain\ C \longleftrightarrow C \subseteq A \wedge (\forall x \in C. \forall y \in C. x \sqsubseteq y \vee y \sqsubseteq x)$

We call a chain that is a proper superset of some set X , but not necessarily a chain itself, a superchain of X .

abbreviation $superchain :: 'a\ set \Rightarrow 'a\ set \Rightarrow bool$ (**infix** $\langle <_c \rangle$ 50)
where $X <_c C \equiv chain\ C \wedge X \subset C$

A maximal chain is a chain that does not have a superchain.

definition $maxchain :: 'a\ set \Rightarrow bool$
where $maxchain\ C \longleftrightarrow chain\ C \wedge (\nexists S. C <_c S)$

We define the successor of a set to be an arbitrary superchain, if such exists, or the set itself, otherwise.

definition $suc :: 'a\ set \Rightarrow 'a\ set$
where $suc\ C = (if\ \neg chain\ C \vee maxchain\ C\ then\ C\ else\ (SOME\ D. C <_c D))$

lemma $chainI$ [*Pure.intro?*]: $C \subseteq A \Longrightarrow (\bigwedge x\ y. x \in C \Longrightarrow y \in C \Longrightarrow x \sqsubseteq y \vee y \sqsubseteq x) \Longrightarrow chain\ C$
 $\langle proof \rangle$

lemma $chain-total$: $chain\ C \Longrightarrow x \in C \Longrightarrow y \in C \Longrightarrow x \sqsubseteq y \vee y \sqsubseteq x$

$\langle \text{proof} \rangle$

lemma *not-chain-suc* [*simp*]: $\neg \text{chain } X \implies \text{suc } X = X$
 $\langle \text{proof} \rangle$

lemma *maxchain-suc* [*simp*]: $\text{maxchain } X \implies \text{suc } X = X$
 $\langle \text{proof} \rangle$

lemma *suc-subset*: $X \subseteq \text{suc } X$
 $\langle \text{proof} \rangle$

lemma *chain-empty* [*simp*]: $\text{chain } \{\}$
 $\langle \text{proof} \rangle$

lemma *not-maxchain-Some*: $\text{chain } C \implies \neg \text{maxchain } C \implies C <_c (\text{SOME } D. C <_c D)$
 $\langle \text{proof} \rangle$

lemma *suc-not-equals*: $\text{chain } C \implies \neg \text{maxchain } C \implies \text{suc } C \neq C$
 $\langle \text{proof} \rangle$

lemma *subset-suc*:
assumes $X \subseteq Y$
shows $X \subseteq \text{suc } Y$
 $\langle \text{proof} \rangle$

We build a set \mathcal{C} that is closed under applications of *suc* and contains the union of all its subsets.

inductive-set *suc-Union-closed* ($\langle \mathcal{C} \rangle$)
where
 $\text{suc}: X \in \mathcal{C} \implies \text{suc } X \in \mathcal{C}$
 $| \text{Union } [\text{unfolded Pow-iff}]: X \in \text{Pow } \mathcal{C} \implies \bigcup X \in \mathcal{C}$

Since the empty set as well as the set itself is a subset of every set, \mathcal{C} contains at least $\{\} \in \mathcal{C}$ and $\bigcup \mathcal{C} \in \mathcal{C}$.

lemma *suc-Union-closed-empty*: $\{\} \in \mathcal{C}$
and *suc-Union-closed-Union*: $\bigcup \mathcal{C} \in \mathcal{C}$
 $\langle \text{proof} \rangle$

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 $X \in \mathcal{C}$
and $\bigwedge X. X \in \mathcal{C} \implies Q X \implies Q (\text{suc } X)$
and $\bigwedge X. X \subseteq \mathcal{C} \implies \forall x \in X. Q x \implies Q (\bigcup X)$
shows $Q X$
 $\langle \text{proof} \rangle$

lemma *suc-Union-closed-cases* [*consumes 1, case-names suc Union, cases pred:*
suc-Union-closed]:

assumes $X \in \mathcal{C}$
and $\bigwedge Y. X = \text{suc } Y \implies Y \in \mathcal{C} \implies Q$
and $\bigwedge Y. X = \bigcup Y \implies Y \subseteq \mathcal{C} \implies Q$
shows Q
 $\langle \text{proof} \rangle$

On chains, *suc* yields a chain.

lemma *chain-suc*:

assumes *chain* X
shows *chain* (*suc* X)
 $\langle \text{proof} \rangle$

lemma *chain-sucD*:

assumes *chain* X
shows $\text{suc } X \subseteq A \wedge \text{chain } (\text{suc } X)$
 $\langle \text{proof} \rangle$

lemma *suc-Union-closed-total'*:

assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$
and *: $\bigwedge Z. Z \in \mathcal{C} \implies Z \subseteq Y \implies Z = Y \vee \text{suc } Z \subseteq Y$
shows $X \subseteq Y \vee \text{suc } Y \subseteq X$
 $\langle \text{proof} \rangle$

lemma *suc-Union-closed-subsetD*:

assumes $Y \subseteq X$ **and** $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$
shows $X = Y \vee \text{suc } Y \subseteq X$
 $\langle \text{proof} \rangle$

The elements of \mathcal{C} are totally ordered by the subset relation.

lemma *suc-Union-closed-total*:

assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$
shows $X \subseteq Y \vee Y \subseteq X$
 $\langle \text{proof} \rangle$

Once we hit a fixed point w.r.t. *suc*, all other elements of \mathcal{C} are subsets of this fixed point.

lemma *suc-Union-closed-suc*:

assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$ **and** $\text{suc } Y = Y$
shows $X \subseteq Y$
 $\langle \text{proof} \rangle$

lemma *eq-suc-Union*:

assumes $X \in \mathcal{C}$
shows $\text{suc } X = X \longleftrightarrow X = \bigcup \mathcal{C}$
 (is ?lhs \longleftrightarrow ?rhs)
 $\langle \text{proof} \rangle$

lemma *suc-in-carrier*:

assumes $X \subseteq A$
 shows $\text{suc } X \subseteq A$
 $\langle \text{proof} \rangle$

lemma *suc-Union-closed-in-carrier*:

assumes $X \in \mathcal{C}$
 shows $X \subseteq A$
 $\langle \text{proof} \rangle$

All elements of \mathcal{C} are chains.

lemma *suc-Union-closed-chain*:

assumes $X \in \mathcal{C}$
 shows $\text{chain } X$
 $\langle \text{proof} \rangle$

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.)

theorem *Hausdorff*: $\exists C. \text{maxchain } C$

$\langle \text{proof} \rangle$

Make notation \mathcal{C} available again.

no-notation *suc-Union-closed* ($\langle \mathcal{C} \rangle$)

lemma *chain-extend*: $\text{chain } C \implies z \in A \implies \forall x \in C. x \sqsubseteq z \implies \text{chain } (\{z\} \cup C)$

$\langle \text{proof} \rangle$

lemma *maxchain-imp-chain*: $\text{maxchain } C \implies \text{chain } C$

$\langle \text{proof} \rangle$

end

Hide constant *pred-on.suc-Union-closed*, which was just needed for the proof of Hausdorff’s maximum principle.

hide-const *pred-on.suc-Union-closed*

lemma *chain-mono*:

assumes $\bigwedge x y. x \in A \implies y \in A \implies P x y \implies Q x y$
 and $\text{pred-on.chain } A P C$
 shows $\text{pred-on.chain } A Q C$
 $\langle \text{proof} \rangle$

26.1.3 Results for the proper subset relation

interpretation *subset*: $\text{pred-on } A (\subset)$ for A $\langle \text{proof} \rangle$

lemma *subset-maxchain-max*:
assumes *subset.maxchain* $A\ C$
and $X \in A$
and $\bigcup C \subseteq X$
shows $\bigcup C = X$
 $\langle \text{proof} \rangle$

lemma *subset-chain-def*: $\bigwedge A. \text{subset.chain } A\ C = (C \subseteq A \wedge (\forall X \in C. \forall Y \in C. X \subseteq Y \vee Y \subseteq X))$
 $\langle \text{proof} \rangle$

lemma *subset-chain-insert*:
 $\text{subset.chain } A\ (\text{insert } B\ \mathcal{B}) \longleftrightarrow B \in A \wedge (\forall X \in \mathcal{B}. X \subseteq B \vee B \subseteq X) \wedge \text{subset.chain } A\ \mathcal{B}$
 $\langle \text{proof} \rangle$

26.1.4 Zorn’s lemma

If every chain has an upper bound, then there is a maximal set.

theorem *subset-Zorn*:
assumes $\bigwedge C. \text{subset.chain } A\ C \implies \exists U \in A. \forall X \in C. X \subseteq U$
shows $\exists M \in A. \forall X \in A. M \subseteq X \longrightarrow X = M$
 $\langle \text{proof} \rangle$

Alternative version of Zorn’s lemma for the subset relation.

lemma *subset-Zorn'*:
assumes $\bigwedge C. \text{subset.chain } A\ C \implies \bigcup C \in A$
shows $\exists M \in A. \forall X \in A. M \subseteq X \longrightarrow X = M$
 $\langle \text{proof} \rangle$

26.2 Zorn’s Lemma for Partial Orders

Relate old to new definitions.

definition *chain-subset* :: $'a\ \text{set}\ \text{set} \Rightarrow \text{bool}$ ($\langle \text{chain}_{\subseteq} \rangle$)
where $\text{chain}_{\subseteq}\ C \longleftrightarrow (\forall A \in C. \forall B \in C. A \subseteq B \vee B \subseteq A)$

definition *chains* :: $'a\ \text{set}\ \text{set} \Rightarrow 'a\ \text{set}\ \text{set}\ \text{set}$
where $\text{chains } A = \{C. C \subseteq A \wedge \text{chain}_{\subseteq}\ C\}$

definition *Chains* :: $('a \times 'a)\ \text{set} \Rightarrow 'a\ \text{set}\ \text{set}$
where $\text{Chains } r = \{C. \forall a \in C. \forall b \in C. (a, b) \in r \vee (b, a) \in r\}$

lemma *chains-extend*: $c \in \text{chains } S \implies z \in S \implies \forall x \in c. x \subseteq z \implies \{z\} \cup c \in \text{chains } S$
for $z :: 'a\ \text{set}$
 $\langle \text{proof} \rangle$

lemma *mono-Chains*: $r \subseteq s \implies \text{Chains } r \subseteq \text{Chains } s$
 $\langle \text{proof} \rangle$

lemma *chain-subset-alt-def*: $\text{chain}_{\subseteq} C = \text{subset.chain UNIV } C$
 $\langle \text{proof} \rangle$

lemma *chains-alt-def*: $\text{chains } A = \{C. \text{subset.chain } A \ C\}$
 $\langle \text{proof} \rangle$

lemma *Chains-subset*: $\text{Chains } r \subseteq \{C. \text{pred-on.chain UNIV } (\lambda x y. (x, y) \in r) \ C\}$
 $\langle \text{proof} \rangle$

lemma *Chains-subset'*:
assumes *refl* r
shows $\{C. \text{pred-on.chain UNIV } (\lambda x y. (x, y) \in r) \ C\} \subseteq \text{Chains } r$
 $\langle \text{proof} \rangle$

lemma *Chains-alt-def*:
assumes *refl* r
shows $\text{Chains } r = \{C. \text{pred-on.chain UNIV } (\lambda x y. (x, y) \in r) \ C\}$
 $\langle \text{proof} \rangle$

lemma *Chains-relation-of*:
assumes $C \in \text{Chains } (\text{relation-of } P \ A)$ **shows** $C \subseteq A$
 $\langle \text{proof} \rangle$

lemma *pairwise-chain-Union*:
assumes $P: \bigwedge S. S \in \mathcal{C} \implies \text{pairwise } R \ S$ **and** $\text{chain}_{\subseteq} \mathcal{C}$
shows $\text{pairwise } R \ (\bigcup \mathcal{C})$
 $\langle \text{proof} \rangle$

lemma *Zorn-Lemma*: $\forall C \in \text{chains } A. \bigcup C \in A \implies \exists M \in A. \forall X \in A. M \subseteq X \longrightarrow X = M$
 $\langle \text{proof} \rangle$

lemma *Zorn-Lemma2*: $\forall C \in \text{chains } A. \exists U \in A. \forall X \in C. X \subseteq U \implies \exists M \in A. \forall X \in A. M \subseteq X \longrightarrow X = M$
 $\langle \text{proof} \rangle$

26.3 Other variants of Zorn’s Lemma

lemma *chainsD*: $c \in \text{chains } S \implies x \in c \implies y \in c \implies x \subseteq y \vee y \subseteq x$
 $\langle \text{proof} \rangle$

lemma *chainsD2*: $c \in \text{chains } S \implies c \subseteq S$
 $\langle \text{proof} \rangle$

lemma *Zorns-po-lemma*:
assumes *po*: *Partial-order* r

and u : $\bigwedge C. C \in \text{Chains } r \implies \exists u \in \text{Field } r. \forall a \in C. (a, u) \in r$
shows $\exists m \in \text{Field } r. \forall a \in \text{Field } r. (m, a) \in r \longrightarrow a = m$
 $\langle \text{proof} \rangle$

lemma *predicate-Zorn*:

assumes po : *partial-order-on* A (*relation-of* P A)
and ch : $\bigwedge C. C \in \text{Chains } (\text{relation-of } P \ A) \implies \exists u \in A. \forall a \in C. P \ a \ u$
shows $\exists m \in A. \forall a \in A. P \ m \ a \longrightarrow a = m$
 $\langle \text{proof} \rangle$

lemma *Union-in-chain*: $\llbracket \text{finite } \mathcal{B}; \mathcal{B} \neq \{\}; \text{subset.chain } \mathcal{A} \ \mathcal{B} \rrbracket \implies \bigcup \mathcal{B} \in \mathcal{B}$
 $\langle \text{proof} \rangle$

lemma *Inter-in-chain*: $\llbracket \text{finite } \mathcal{B}; \mathcal{B} \neq \{\}; \text{subset.chain } \mathcal{A} \ \mathcal{B} \rrbracket \implies \bigcap \mathcal{B} \in \mathcal{B}$
 $\langle \text{proof} \rangle$

lemma *finite-subset-Union-chain*:

assumes $\text{finite } A \ A \subseteq \bigcup \mathcal{B} \ \mathcal{B} \neq \{\}$ **and** sub : $\text{subset.chain } \mathcal{A} \ \mathcal{B}$
obtains B **where** $B \in \mathcal{B} \ A \subseteq B$
 $\langle \text{proof} \rangle$

lemma *subset-Zorn-nonempty*:

assumes $\mathcal{A} \neq \{\}$ **and** ch : $\bigwedge C. \llbracket C \neq \{\}; \text{subset.chain } \mathcal{A} \ C \rrbracket \implies \bigcup C \in \mathcal{A}$
shows $\exists M \in \mathcal{A}. \forall X \in \mathcal{A}. M \subseteq X \longrightarrow X = M$
 $\langle \text{proof} \rangle$

26.4 The Well Ordering Theorem

definition *init-seg-of* :: $((\text{'a} \times \text{'a}) \text{ set} \times (\text{'a} \times \text{'a}) \text{ set}) \text{ set}$

where $\text{init-seg-of} = \{(r, s). r \subseteq s \wedge (\forall a \ b \ c. (a, b) \in s \wedge (b, c) \in r \longrightarrow (a, b) \in r)\}$

abbreviation *initial-segment-of-syntax* :: $(\text{'a} \times \text{'a}) \text{ set} \Rightarrow (\text{'a} \times \text{'a}) \text{ set} \Rightarrow \text{bool}$
 $(\text{infix } \langle \text{initial}'\text{-segment}'\text{-of} \rangle \ 55)$

where $r \text{ initial-segment-of } s \equiv (r, s) \in \text{init-seg-of}$

lemma *refl-on-init-seg-of* [*simp*]: $r \text{ initial-segment-of } r$
 $\langle \text{proof} \rangle$

lemma *trans-init-seg-of*:

$r \text{ initial-segment-of } s \implies s \text{ initial-segment-of } t \implies r \text{ initial-segment-of } t$
 $\langle \text{proof} \rangle$

lemma *antisym-init-seg-of*: $r \text{ initial-segment-of } s \implies s \text{ initial-segment-of } r \implies r = s$
 $\langle \text{proof} \rangle$

lemma *Chains-init-seg-of-Union*: $R \in \text{Chains init-seg-of} \implies r \in R \implies r \text{ initial-segment-of } \bigcup R$

$\langle \text{proof} \rangle$

lemma *chain-subset-trans-Union*:

assumes $\text{chain}_{\subseteq} R \ \forall r \in R. \text{trans } r$

shows $\text{trans } (\bigcup R)$

$\langle \text{proof} \rangle$

lemma *chain-subset-antisym-Union*:

assumes $\text{chain}_{\subseteq} R \ \forall r \in R. \text{antisym } r$

shows $\text{antisym } (\bigcup R)$

$\langle \text{proof} \rangle$

lemma *chain-subset-Total-Union*:

assumes $\text{chain}_{\subseteq} R$ **and** $\forall r \in R. \text{Total } r$

shows $\text{Total } (\bigcup R)$

$\langle \text{proof} \rangle$

lemma *wf-Union-wf-init-segs*:

assumes $R \in \text{Chains init-seg-of}$

and $\forall r \in R. \text{wf } r$

shows $\text{wf } (\bigcup R)$

$\langle \text{proof} \rangle$

lemma *initial-segment-of-Diff*: $p \text{ initial-segment-of } q \implies p - s \text{ initial-segment-of } q - s$

$\langle \text{proof} \rangle$

lemma *Chains-inits-DiffI*: $R \in \text{Chains init-seg-of} \implies \{r - s \mid r. r \in R\} \in \text{Chains init-seg-of}$

$\langle \text{proof} \rangle$

theorem *well-ordering*: $\exists r::'a \text{ rel. Well-order } r \wedge \text{Field } r = \text{UNIV}$

$\langle \text{proof} \rangle$

corollary *well-order-on*: $\exists r::'a \text{ rel. well-order-on } A \ r$

$\langle \text{proof} \rangle$

lemma *dependent-wf-choice*:

fixes $P :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool}$

assumes $\text{wf } R$

and $\text{adm}: \bigwedge g \ x \ r. (\bigwedge z. (z, x) \in R \implies f \ z = g \ z) \implies P \ f \ x \ r = P \ g \ x \ r$

and $P: \bigwedge x \ f. (\bigwedge y. (y, x) \in R \implies P \ f \ y \ (f \ y)) \implies \exists r. P \ f \ x \ r$

shows $\exists f. \forall x. P \ f \ x \ (f \ x)$

$\langle \text{proof} \rangle$

lemma (**in** *wellorder*) *dependent-wellorder-choice*:

assumes $\bigwedge r \ f \ g \ x. (\bigwedge y. y < x \implies f \ y = g \ y) \implies P \ f \ x \ r = P \ g \ x \ r$

and $P: \bigwedge x \ f. (\bigwedge y. y < x \implies P \ f \ y \ (f \ y)) \implies \exists r. P \ f \ x \ r$

shows $\exists f. \forall x. P \ f \ x \ (f \ x)$

$\langle proof \rangle$

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 UnderS where UnderS  $\equiv$  Order-Relation.UnderS 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 AboveS where AboveS  $\equiv$  Order-Relation.AboveS r
abbreviation ofilter where ofilter  $\equiv$  Order-Relation.ofilter r
lemmas ofilter-def = Order-Relation.ofilter-def[of r]
```

27.1 Auxiliaries

```
lemma REFL: Refl r
   $\langle proof \rangle$ 
```

```
lemma TRANS: trans r
   $\langle proof \rangle$ 
```

```
lemma ANTISYM: antisym r
   $\langle proof \rangle$ 
```

```
lemma TOTAL: Total r
   $\langle proof \rangle$ 
```

```
lemma TOTALS:  $\forall a \in \text{Field } r. \forall b \in \text{Field } r. (a,b) \in r \vee (b,a) \in r$ 
   $\langle proof \rangle$ 
```

lemma *LIN*: *Linear-order* r

$\langle \text{proof} \rangle$

lemma *WF*: *wf* $(r - Id)$

$\langle \text{proof} \rangle$

lemma *cases-Total*:

$\bigwedge \text{phi } a \text{ b. } \llbracket \{a, b\} \leq \text{Field } r; ((a, b) \in r \implies \text{phi } a \text{ b}); ((b, a) \in r \implies \text{phi } a \text{ b}) \rrbracket$
 $\implies \text{phi } a \text{ b}$

$\langle \text{proof} \rangle$

lemma *cases-Total3*:

$\bigwedge \text{phi } a \text{ b. } \llbracket \{a, b\} \leq \text{Field } r; ((a, b) \in r - Id \vee (b, a) \in r - Id \implies \text{phi } a \text{ b});$
 $(a = b \implies \text{phi } a \text{ b}) \rrbracket \implies \text{phi } a \text{ b}$

$\langle \text{proof} \rangle$

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 *IND*: $\bigwedge x. \forall y. y \neq x \wedge (y, x) \in r \longrightarrow P \text{ y} \implies P \text{ x}$

shows $P \text{ a}$

$\langle \text{proof} \rangle$

definition

$\text{worec} :: ((\text{'a} \Rightarrow \text{'b}) \Rightarrow \text{'a} \Rightarrow \text{'b}) \Rightarrow \text{'a} \Rightarrow \text{'b}$

where

$\text{worec } F \equiv \text{wfrec } (r - Id) F$

definition

$\text{adm-wo} :: ((\text{'a} \Rightarrow \text{'b}) \Rightarrow \text{'a} \Rightarrow \text{'b}) \Rightarrow \text{bool}$

where

$\text{adm-wo } H \equiv \forall f \text{ g } x. (\forall y \in \text{underS } x. f \text{ y} = g \text{ y}) \longrightarrow H \text{ f } x = H \text{ g } x$

lemma *worec-fixpoint*:

assumes *ADM*: $\text{adm-wo } H$

shows $\text{worec } H = H (\text{worec } H)$

$\langle \text{proof} \rangle$

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*, *max2*, and the minimum *of a set*, *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 *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 *max2* :: 'a \Rightarrow 'a \Rightarrow 'a
where *max2* a b \equiv if (a,b) \in r then b else a

definition *isMinim* :: 'a set \Rightarrow 'a \Rightarrow bool
where *isMinim* A b \equiv b \in A \wedge (\forall a \in A. (b,a) \in r)

definition *minim* :: 'a set \Rightarrow 'a
where *minim* A \equiv THE b. *isMinim* A b

definition *supr* :: 'a set \Rightarrow 'a
where *supr* A \equiv *minim* (Above A)

definition *suc* :: 'a set \Rightarrow 'a
where *suc* A \equiv *minim* (AboveS A)

27.3.1 Properties of max2

lemma *max2-greater-among*:
assumes a \in Field r **and** b \in Field r
shows (a, *max2* a b) \in r \wedge (b, *max2* a b) \in r \wedge *max2* a b \in {a,b}
 <proof>

lemma *max2-greater*:
assumes a \in Field r **and** b \in Field r
shows (a, *max2* a b) \in r \wedge (b, *max2* a b) \in r
 <proof>

lemma *max2-among*:
assumes a \in Field r **and** b \in Field r
shows *max2* a b \in {a, b}
 <proof>

lemma *max2-equals1*:
assumes a \in Field r **and** b \in Field r
shows (*max2* a b = a) = ((b,a) \in r)
 <proof>

lemma *max2-equals2*:

assumes $a \in \text{Field } r$ **and** $b \in \text{Field } r$
shows $(\text{max2 } a \ b = b) = ((a, b) \in r)$
 $\langle \text{proof} \rangle$

lemma *in-notinI*:

assumes $(j, i) \notin r \vee j = i$ **and** $i \in \text{Field } r$ **and** $j \in \text{Field } r$
shows $(i, j) \in r$ $\langle \text{proof} \rangle$

27.3.2 Existence and uniqueness for isMinim and well-definedness of minim

lemma *isMinim-unique*:

assumes $\text{isMinim } B \ a \ \text{isMinim } B \ a'$
shows $a = a'$
 $\langle \text{proof} \rangle$

lemma *Well-order-isMinim-exists*:

assumes $\text{SUB: } B \leq \text{Field } r$ **and** $\text{NE: } B \neq \{\}$
shows $\exists b. \text{isMinim } B \ b$
 $\langle \text{proof} \rangle$

lemma *minim-isMinim*:

assumes $\text{SUB: } B \leq \text{Field } r$ **and** $\text{NE: } B \neq \{\}$
shows $\text{isMinim } B \ (\text{minim } B)$
 $\langle \text{proof} \rangle$

27.3.3 Properties of minim

lemma *minim-in*:

assumes $B \leq \text{Field } r$ **and** $B \neq \{\}$
shows $\text{minim } B \in B$
 $\langle \text{proof} \rangle$

lemma *minim-inField*:

assumes $B \leq \text{Field } r$ **and** $B \neq \{\}$
shows $\text{minim } B \in \text{Field } r$
 $\langle \text{proof} \rangle$

lemma *minim-least*:

assumes $\text{SUB: } B \leq \text{Field } r$ **and** $\text{IN: } b \in B$
shows $(\text{minim } B, b) \in r$
 $\langle \text{proof} \rangle$

lemma *equals-minim*:

assumes $\text{SUB: } B \leq \text{Field } r$ **and** $\text{IN: } a \in B$ **and**
 $\text{LEAST: } \bigwedge b. b \in B \implies (a, b) \in r$
shows $a = \text{minim } B$
 $\langle \text{proof} \rangle$

27.3.4 Properties of successor**lemma** *suc-AboveS*:assumes $SUB: B \leq Field\ r$ and $ABOVES: AboveS\ B \neq \{\}$ shows $suc\ B \in AboveS\ B$ $\langle proof \rangle$ **lemma** *suc-greater*:assumes $SUB: B \leq Field\ r$ and $ABOVES: AboveS\ B \neq \{\}$ and $IN: b \in B$ shows $suc\ B \neq b \wedge (b, suc\ B) \in r$ $\langle proof \rangle$ **lemma** *suc-least-AboveS*:assumes $ABOVES: a \in AboveS\ B$ shows $(suc\ B, a) \in r$ $\langle proof \rangle$ **lemma** *suc-inField*:assumes $B \leq Field\ r$ and $AboveS\ B \neq \{\}$ shows $suc\ B \in Field\ r$ $\langle proof \rangle$ **lemma** *equals-suc-AboveS*:assumes $B \leq Field\ r$ and $a \in AboveS\ B$ and $\bigwedge a'. a' \in AboveS\ B \implies (a, a') \in r$ shows $a = suc\ B$ $\langle proof \rangle$ **lemma** *suc-underS*:assumes $IN: a \in Field\ r$ shows $a = suc\ (underS\ a)$ $\langle proof \rangle$ **27.3.5 Properties of order filters****lemma** *under-ofilter*: *ofilter* (*under* *a*) $\langle proof \rangle$ **lemma** *underS-ofilter*: *ofilter* (*underS* *a*) $\langle proof \rangle$ **lemma** *Field-ofilter*:*ofilter* (*Field* *r*) $\langle proof \rangle$ **lemma** *ofilter-underS-Field*:*ofilter* $A = ((\exists a \in Field\ r. A = underS\ a) \vee (A = Field\ r))$ $\langle proof \rangle$ **lemma** *ofilter-UNION*:

$$(\bigwedge i. i \in I \implies \text{ofilter}(A\ i)) \implies \text{ofilter}(\bigcup i \in I. A\ i)$$

<proof>

lemma *ofilter-under-UNION*:
assumes *ofilter A*
shows $A = (\bigcup a \in A. \text{under } a)$
<proof>

27.3.6 Other properties

lemma *ofilter-linord*:
assumes *OF1: ofilter A and OF2: ofilter B*
shows $A \leq B \vee B \leq A$
<proof>

lemma *ofilter-AboveS-Field*:
assumes *ofilter A*
shows $A \cup (\text{AboveS } A) = \text{Field } r$
<proof>

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 $b \in A$
<proof>

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: $\bigwedge i. i \in I \implies \text{wo-rel.ofilter } r \ (A \ i)$ **and**

INJ: $\bigwedge i. i \in I \implies \text{inj-on } f \ (A \ i)$

shows $\text{inj-on } f \ (\bigcup i \in I. A \ i)$

<proof>

lemma *under-underS-bij-betw*:

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: $\text{bij-betw } f \ (\text{underS } r \ a) \ (\text{underS } r' \ (f \ a))$

shows $\text{bij-betw } f \ (\text{under } r \ a) \ (\text{under } r' \ (f \ a))$

<proof>

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-on-ofilter*.) A *strict embedding* (operator *embedS*) is a non-bijective embedding and an isomorphism (operator *iso*) is a bijective embedding.

definition *embed* :: $'a \ \text{rel} \Rightarrow 'a' \ \text{rel} \Rightarrow ('a \Rightarrow 'a') \Rightarrow \text{bool}$

where

$\text{embed } r \ r' \ f \equiv \forall a \in \text{Field } r. \text{bij-betw } f \ (\text{under } r \ a) \ (\text{under } r' \ (f \ a))$

lemmas *embed-defs* = *embed-def embed-def[abs-def]*

Strict embeddings:

definition *embedS* :: $'a \ \text{rel} \Rightarrow 'a' \ \text{rel} \Rightarrow ('a \Rightarrow 'a') \Rightarrow \text{bool}$

where

$\text{embedS } r \ r' \ f \equiv \text{embed } r \ r' \ f \wedge \neg \text{bij-betw } f \ (\text{Field } r) \ (\text{Field } r')$

lemmas *embedS-defs* = *embedS-def embedS-def[abs-def]*

definition *iso* :: $'a \ \text{rel} \Rightarrow 'a' \ \text{rel} \Rightarrow ('a \Rightarrow 'a') \Rightarrow \text{bool}$

where

$\text{iso } r \ r' \ f \equiv \text{embed } r \ r' \ f \wedge \text{bij-betw } f \ (\text{Field } r) \ (\text{Field } r')$

lemmas *iso-defs* = *iso-def iso-def[abs-def]*

definition *compat* :: $'a \ \text{rel} \Rightarrow 'a' \ \text{rel} \Rightarrow ('a \Rightarrow 'a') \Rightarrow \text{bool}$

where

$\text{compat } r \ r' \ f \equiv \forall a \ b. (a, b) \in r \longrightarrow (f \ a, f \ b) \in r'$

lemma *compat-wf*:

assumes *CMP*: $\text{compat } r \ r' \ f$ **and** *WF*: $\text{wf } r'$

shows $\text{wf } r$

<proof>

lemma *id-embed*: $\text{embed } r \ r \ \text{id}$

<proof>

lemma *id-iso*: $\text{iso } r \ r \ \text{id}$

<proof>

lemma *embed-compat*:

assumes *EMB*: $\text{embed } r \ r' \ f$

shows $\text{compat } r \ r' \ f$

<proof>

lemma *embed-in-Field*:

assumes *EMB*: $\text{embed } r \ r' \ f$ **and** *IN*: $a \in \text{Field } r$

shows $f \ a \in \text{Field } r'$

<proof>

lemma *comp-embed*:

assumes *EMB*: $\text{embed } r \ r' \ f$ **and** *EMB'*: $\text{embed } r' \ r'' \ f'$

shows $\text{embed } r \ r'' \ (f' \circ f)$

<proof>

lemma *comp-iso*:

assumes *EMB*: $\text{iso } r \ r' \ f$ **and** *EMB'*: $\text{iso } r' \ r'' \ f'$

shows $\text{iso } r \ r'' \ (f' \circ f)$

<proof>

That *embedS* is also preserved by function composition shall be proved only later.

lemma *embed-Field*: $\text{embed } r \ r' \ f \implies f'(\text{Field } r) \leq \text{Field } r'$

<proof>

lemma *embed-preserves-ofilter*:

assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r' **and**

EMB: $\text{embed } r \ r' \ f$ **and** *OF*: *wo-rel.ofilter* $r \ A$

shows *wo-rel.ofilter* $r' \ (f'A)$

<proof>

lemma *embed-Field-ofilter*:

assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r' **and**

EMB: $\text{embed } r \ r' \ f$

shows *wo-rel.ofilter* $r' \ (f'(\text{Field } r))$

$\langle \text{proof} \rangle$

lemma *embed-inj-on*:

assumes *WELL*: Well-order r **and** *EMB*: $\text{embed } r \ r' \ f$
shows $\text{inj-on } f \ (\text{Field } r)$

$\langle \text{proof} \rangle$

lemma *embed-underS*:

assumes *WELL*: Well-order r **and**
EMB: $\text{embed } r \ r' \ f$ **and** *IN*: $a \in \text{Field } r$
shows $\text{bij-betw } f \ (\text{underS } r \ a) \ (\text{underS } r' \ (f \ a))$

$\langle \text{proof} \rangle$

lemma *embed-iff-compat-inj-on-ofilter*:

assumes *WELL*: Well-order r **and** *WELL'*: Well-order r'
shows $\text{embed } r \ r' \ f = (\text{compat } r \ r' \ f \wedge \text{inj-on } f \ (\text{Field } r) \wedge \text{wo-rel.ofilter } r' \ (f'(\text{Field } r)))$

$\langle \text{proof} \rangle$

lemma *inv-into-ofilter-embed*:

assumes *WELL*: Well-order r **and** *OF*: $\text{wo-rel.ofilter } r \ A$ **and**
BIJ: $\forall b \in A. \text{bij-betw } f \ (\text{under } r \ b) \ (\text{under } r' \ (f \ b))$ **and**
IMAGE: $f' \ A = \text{Field } r'$
shows $\text{embed } r' \ r \ (\text{inv-into } A \ f)$

$\langle \text{proof} \rangle$

lemma *inv-into-underS-embed*:

assumes *WELL*: Well-order r **and**
BIJ: $\forall b \in \text{underS } r \ a. \text{bij-betw } f \ (\text{under } r \ b) \ (\text{under } r' \ (f \ b))$ **and**
IN: $a \in \text{Field } r$ **and**
IMAGE: $f' \ (\text{underS } r \ a) = \text{Field } r'$
shows $\text{embed } r' \ r \ (\text{inv-into } (\text{underS } r \ a) \ f)$

$\langle \text{proof} \rangle$

lemma *inv-into-Field-embed*:

assumes *WELL*: Well-order r **and** *EMB*: $\text{embed } r \ r' \ f$ **and**
IMAGE: $\text{Field } r' \leq f' \ (\text{Field } r)$
shows $\text{embed } r' \ r \ (\text{inv-into } (\text{Field } r) \ f)$

$\langle \text{proof} \rangle$

lemma *inv-into-Field-embed-bij-betw*:

assumes *EMB*: $\text{embed } r \ r' \ f$ **and** *BIJ*: $\text{bij-betw } f \ (\text{Field } r) \ (\text{Field } r')$
shows $\text{embed } r' \ r \ (\text{inv-into } (\text{Field } r) \ f)$

$\langle \text{proof} \rangle$

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 $r :: 'a \text{ rel}$ and $r' :: 'a' \text{ rel}$. 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 $r :: 'a \text{ rel}$ **and** $r' :: 'a' \text{ rel}$ **and**
 $f :: 'a \Rightarrow 'a'$ **and** $a :: 'a$
assumes *WELL*: Well-order r **and** *WELL'*: Well-order r' **and** *IN*: $a \in \text{Field } r$
and
IH: $\forall b \in \text{underS } r \ a. \text{bij-betw } f \ (\text{under } r \ b) \ (\text{under } r' \ (f \ b))$ **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 $\text{bij-betw } f \ (\text{under } r \ a) \ (\text{under } r' \ (f \ a))$
 $\langle \text{proof} \rangle$

lemma *wellorders-totally-ordered-aux2*:

fixes $r :: 'a \text{ rel}$ **and** $r' :: 'a' \text{ rel}$ **and**
 $f :: 'a \Rightarrow 'a'$ **and** $g :: 'a \Rightarrow \text{bool}$ **and** $a :: 'a$
assumes *WELL*: Well-order r **and** *WELL'*: Well-order r' **and**
MAIN1:
 $\bigwedge a. (\text{False} \notin g'(\text{underS } r \ a) \wedge f'(\text{underS } r \ a) \neq \text{Field } r' \longrightarrow f \ a = \text{wo-rel.suc } r' \ (f'(\text{underS } r \ a)) \wedge g \ a = \text{True})$
 \wedge
 $(\neg(\text{False} \notin (g'(\text{underS } r \ a)) \wedge f'(\text{underS } r \ a) \neq \text{Field } r') \longrightarrow g \ a = \text{False})$ **and**
MAIN2: $\bigwedge a. a \in \text{Field } r \wedge \text{False} \notin g'(\text{under } r \ a) \longrightarrow \text{bij-betw } f \ (\text{under } r \ a) \ (\text{under } r' \ (f \ a))$ **and**
Case: $a \in \text{Field } r \wedge \text{False} \in g'(\text{under } r \ a)$
shows $\exists f'. \text{embed } r' \ r \ f'$
 $\langle \text{proof} \rangle$

theorem *wellorders-totally-ordered*:

fixes $r :: 'a \text{ rel}$ **and** $r' :: 'a' \text{ rel}$
assumes *WELL*: Well-order r **and** *WELL'*: Well-order r'
shows $(\exists f'. \text{embed } r \ r' \ f) \vee (\exists f'. \text{embed } r' \ r \ f')$
 $\langle \text{proof} \rangle$

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 embed-

ding, 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>

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 $a \in \text{Field } r \longrightarrow f a = g a$

<proof>

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 $(\forall a \in \text{Field } r. f'(f a) = a) \wedge (\forall a' \in \text{Field } r'. f(f' a') = a')$

<proof>

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 $(\text{Field } r)$ $(\text{Field } r')$

<proof>

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

<proof>

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 $(\text{Field } r)$ $(\text{Field } r')$

<proof>

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 embedS r r'' $(f' \circ f)$

<proof>

lemma *embed-comp-embedS*:

assumes *WELL*: Well-order r **and** *WELL'*: Well-order r'
and *EMB*: embed $r\ r' f$ **and** *EMB'*: embedS $r'\ r'' f'$
shows embedS $r\ r'' (f' \circ f)$
 $\langle \text{proof} \rangle$

lemma *embed-comp-iso*:
assumes *EMB*: embed $r\ r' f$ **and** *EMB'*: iso $r'\ r'' f'$
shows embed $r\ r'' (f' \circ f)$ $\langle \text{proof} \rangle$

lemma *iso-comp-embed*:
assumes *EMB*: iso $r\ r' f$ **and** *EMB'*: embed $r'\ r'' f'$
shows embed $r\ r'' (f' \circ f)$
 $\langle \text{proof} \rangle$

lemma *embedS-comp-iso*:
assumes *EMB*: embedS $r\ r' f$ **and** *EMB'*: iso $r'\ r'' f'$
shows embedS $r\ r'' (f' \circ f)$
 $\langle \text{proof} \rangle$

lemma *iso-comp-embedS*:
assumes *WELL*: Well-order r **and** *WELL'*: Well-order r'
and *EMB*: iso $r\ r' f$ **and** *EMB'*: embedS $r'\ r'' f'$
shows embedS $r\ r'' (f' \circ f)$
 $\langle \text{proof} \rangle$

lemma *embedS-Field*:
assumes *WELL*: Well-order r **and** *EMB*: embedS $r\ r' f$
shows $f' (Field\ r) < Field\ r'$
 $\langle \text{proof} \rangle$

lemma *embedS-iff*:
assumes *WELL*: Well-order r **and** *ISO*: embed $r\ r' f$
shows embedS $r\ r' f = (f' (Field\ r) < Field\ r')$
 $\langle \text{proof} \rangle$

lemma *iso-Field*: iso $r\ r' f \implies f' (Field\ r) = Field\ r'$
 $\langle \text{proof} \rangle$

lemma *iso-iff*:
assumes Well-order r
shows iso $r\ r' f = (embed\ r\ r' f \wedge f' (Field\ r) = Field\ r')$
 $\langle \text{proof} \rangle$

lemma *iso-iff2*: iso $r\ r' f \longleftrightarrow$
 $bij_betw\ f\ (Field\ r)\ (Field\ r') \wedge$
 $(\forall a \in Field\ r. \forall b \in Field\ r'. (a, b) \in r \longleftrightarrow (f\ a, f\ b) \in r')$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

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'*) \wedge *compat* *r* *r'* *f*)

<proof>

lemma *iso-imp-inj-on*:

assumes *iso* *r* *r'* *f* **shows** *inj-on* *f* (*Field* *r*)

<proof>

lemma *iso-backward-Field*:

assumes *x* \in *Field* *r'* *iso* *r* *r'* *f*

shows *inv-into* (*Field* *r*) *f* *x* \in *Field* *r*

<proof>

lemma *iso-backward*:

assumes (*x,y*) \in *r'* **and** *iso*: *iso* *r* *r'* *f*

shows (*inv-into* (*Field* *r*) *f* *x*, *inv-into* (*Field* *r*) *f* *y*) \in *r*

<proof>

lemma *iso-forward*:

assumes (*x,y*) \in *r* *iso* *r* *r'* *f* **shows** (*f* *x*, *f* *y*) \in *r'*

<proof>

lemma *iso-trans*:

assumes *trans* *r* **and** *iso*: *iso* *r* *r'* *f* **shows** *trans* *r'*

<proof>

lemma *iso-Total*:

assumes *Total* *r* **and** *iso*: *iso* *r* *r'* *f* **shows** *Total* *r'*

<proof>

lemma *iso-wf*:

assumes *wf* *r* **and** *iso*: *iso* *r* *r'* *f* **shows** *wf* *r'*

<proof>

end

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 *ordLeq*, of being embedded (abbreviated $\leq o$), *ordLess*, of being

strictly embedded (abbreviated $<o$), and $ordIso$, 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 $Restr\ r\ A \equiv r\ Int\ (A \times A)$

lemma *Restr-subset*:

$A \leq B \Longrightarrow Restr\ (Restr\ r\ B)\ A = Restr\ r\ A$
 $\langle proof \rangle$

lemma *Restr-Field*: $Restr\ r\ (Field\ r) = r$
 $\langle proof \rangle$

lemma *Refl-Restr*: $Refl\ r \Longrightarrow Refl\ (Restr\ r\ A)$
 $\langle proof \rangle$

lemma *linear-order-on-Restr*:

$linear-order-on\ A\ r \Longrightarrow linear-order-on\ (A \cap above\ r\ x)\ (Restr\ r\ (above\ r\ x))$
 $\langle proof \rangle$

lemma *antisym-Restr*:

$antisym\ r \Longrightarrow antisym\ (Restr\ r\ A)$
 $\langle proof \rangle$

lemma *Total-Restr*:

$Total\ r \Longrightarrow Total\ (Restr\ r\ A)$
 $\langle proof \rangle$

lemma *total-on-imp-Total-Restr*: $total-on\ A\ r \Longrightarrow Total\ (Restr\ r\ A)$
 $\langle proof \rangle$

lemma *trans-Restr*:

$trans\ r \Longrightarrow trans\ (Restr\ r\ A)$
 $\langle proof \rangle$

lemma *Preorder-Restr*:

assumes $Preorder\ r$
shows $Preorder\ (Restr\ r\ A)$
 $\langle proof \rangle$

lemma *Partial-order-Restr*:

$Partial-order\ r \Longrightarrow Partial-order\ (Restr\ r\ A)$
 $\langle proof \rangle$

lemma *Linear-order-Restr*:

Linear-order $r \implies \text{Linear-order}(\text{Restr } r \ A)$
 ⟨proof⟩

lemma *Well-order-Restr*:
 assumes *Well-order* r
 shows *Well-order*(*Restr* $r \ A$)
 ⟨proof⟩

lemma *Field-Restr-subset*: *Field*(*Restr* $r \ A$) $\leq A$
 ⟨proof⟩

lemma *Refl-Field-Restr*:
Refl $r \implies \text{Field}(\text{Restr } r \ A) = (\text{Field } r) \ \text{Int } A$
 ⟨proof⟩

lemma *Refl-Field-Restr2*:
 $\llbracket \text{Refl } r; A \leq \text{Field } r \rrbracket \implies \text{Field}(\text{Restr } r \ A) = A$
 ⟨proof⟩

lemma *well-order-on-Restr*:
 assumes *WELL*: *Well-order* r and *SUB*: $A \leq \text{Field } r$
 shows *well-order-on* $A \ (\text{Restr } r \ A)$
 ⟨proof⟩

29.2 Order filters versus restrictions and embeddings

lemma *Field-Restr-ofilter*:
 $\llbracket \text{Well-order } r; \text{wo-rel.ofilter } r \ A \rrbracket \implies \text{Field}(\text{Restr } r \ A) = A$
 ⟨proof⟩

lemma *ofilter-Restr-under*:
 assumes *WELL*: *Well-order* r and *OF*: *wo-rel.ofilter* $r \ A$ and *IN*: $a \in A$
 shows *under* (*Restr* $r \ A$) $a = \text{under } r \ a$
 ⟨proof⟩

lemma *ofilter-embed*:
 assumes *Well-order* r
 shows *wo-rel.ofilter* $r \ A = (A \leq \text{Field } r \wedge \text{embed } (\text{Restr } r \ A) \ r \ \text{id})$
 ⟨proof⟩

lemma *ofilter-Restr-Int*:
 assumes *WELL*: *Well-order* r and *OFA*: *wo-rel.ofilter* $r \ A$
 shows *wo-rel.ofilter* (*Restr* $r \ B$) ($A \ \text{Int } B$)
 ⟨proof⟩

lemma *ofilter-Restr-subset*:
 assumes *WELL*: *Well-order* r and *OFA*: *wo-rel.ofilter* $r \ A$ and *SUB*: $A \leq B$
 shows *wo-rel.ofilter* (*Restr* $r \ B$) A
 ⟨proof⟩

lemma *ofilter-subset-embed*:

assumes *WELL*: Well-order r **and**

OFA: *wo-rel.ofilter* r A **and** *OFB*: *wo-rel.ofilter* r B

shows $(A \leq B) = (\text{embed } (\text{Restr } r A) (\text{Restr } r B) \text{ id})$

<proof>

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 $((A < B) = (\text{embedS } (\text{Restr } r A) (\text{Restr } r B) \text{ id})) \wedge$

$((A = B) = (\text{iso } (\text{Restr } r A) (\text{Restr } r B) \text{ id}))$

<proof>

lemma *ofilter-subset-embedS*:

assumes *WELL*: Well-order r **and**

OFA: *wo-rel.ofilter* r A **and** *OFB*: *wo-rel.ofilter* r B

shows $(A < B) = \text{embedS } (\text{Restr } r A) (\text{Restr } r B) \text{ id}$

<proof>

lemma *embed-implys-iso-Restr*:

assumes *WELL*: Well-order r **and** *WELL'*: Well-order r' **and**

EMB: *embed* $r' r f$

shows *iso* $r' (\text{Restr } r (f ' (\text{Field } r')))$ f

<proof>

29.3 The strict inclusion on proper ofilters is well-founded

definition *ofilterIncl* :: ' a rel \Rightarrow ' a set rel

where

$\text{ofilterIncl } r \equiv \{(A, B). \text{wo-rel.ofilter } r A \wedge A \neq \text{Field } r \wedge$
 $\text{wo-rel.ofilter } r B \wedge B \neq \text{Field } r \wedge A < B\}$

lemma *wf-ofilterIncl*:

assumes *WELL*: Well-order r

shows *wf*(*ofilterIncl* r)

<proof>

29.4 Ordering the well-orders by existence of embeddings

We define three relations between well-orders:

- *ordLeq*, of being embedded (abbreviated $\leq o$);
- *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 $ordLeq :: ('a\ rel * 'a' rel)\ set$

where

$ordLeq = \{(r, r').\ Well\text{-}order\ r \wedge Well\text{-}order\ r' \wedge (\exists f.\ embed\ r\ r'\ f)\}$

abbreviation $ordLeq2 :: 'a\ rel \Rightarrow 'a' rel \Rightarrow bool$ (**infix** $\langle \leq_o \rangle$ 50)

where $r \leq_o r' \equiv (r, r') \in ordLeq$

abbreviation $ordLeq3 :: 'a\ rel \Rightarrow 'a' rel \Rightarrow bool$ (**infix** $\langle \leq_o \rangle$ 50)

where $r \leq_o r' \equiv r \leq_o r'$

definition $ordLess :: ('a\ rel * 'a' rel)\ set$

where

$ordLess = \{(r, r').\ Well\text{-}order\ r \wedge Well\text{-}order\ r' \wedge (\exists f.\ embedS\ r\ r'\ f)\}$

abbreviation $ordLess2 :: 'a\ rel \Rightarrow 'a' rel \Rightarrow bool$ (**infix** $\langle <_o \rangle$ 50)

where $r <_o r' \equiv (r, r') \in ordLess$

definition $ordIso :: ('a\ rel * 'a' rel)\ set$

where

$ordIso = \{(r, r').\ Well\text{-}order\ r \wedge Well\text{-}order\ r' \wedge (\exists f.\ iso\ r\ r'\ f)\}$

abbreviation $ordIso2 :: 'a\ rel \Rightarrow 'a' rel \Rightarrow bool$ (**infix** $\langle =_o \rangle$ 50)

where $r =_o r' \equiv (r, r') \in ordIso$

lemmas $ordRels\text{-}def = ordLeq\text{-}def\ ordLess\text{-}def\ ordIso\text{-}def$

lemma $ordLeq\text{-}Well\text{-}order\text{-}simp$:

assumes $r \leq_o r'$

shows $Well\text{-}order\ r \wedge Well\text{-}order\ r'$

$\langle proof \rangle$

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 $(('a\ rel) * ('a' rel))\ set$, i.e., to $'a\ rel\ rel$.

lemma $ordLeq\text{-}reflexive$:

$Well\text{-}order\ r \Longrightarrow r \leq_o r$

$\langle proof \rangle$

lemma $ordLeq\text{-}transitive[trans]$:

assumes $r \leq_o r'$ **and** $r' \leq_o r''$

shows $r \leq_o r''$

$\langle proof \rangle$

lemma $ordLeq\text{-}total$:

$\llbracket Well\text{-}order\ r;\ Well\text{-}order\ r' \rrbracket \Longrightarrow r \leq_o r' \vee r' \leq_o r$

$\langle proof \rangle$

lemma *ordIso-reflexive*:

Well-order $r \implies r =_o r$

$\langle proof \rangle$

lemma *ordIso-transitive*[*trans*]:

assumes *: $r =_o r'$ **and** **: $r' =_o r''$

shows $r =_o r''$

$\langle proof \rangle$

lemma *ordIso-symmetric*:

assumes *: $r =_o r'$

shows $r' =_o r$

$\langle proof \rangle$

lemma *ordLeq-ordLess-trans*[*trans*]:

assumes $r \leq_o r'$ **and** $r' <_o r''$

shows $r <_o r''$

$\langle proof \rangle$

lemma *ordLess-ordLeq-trans*[*trans*]:

assumes $r <_o r'$ **and** $r' \leq_o r''$

shows $r <_o r''$

$\langle proof \rangle$

lemma *ordLeq-ordIso-trans*[*trans*]:

assumes $r \leq_o r'$ **and** $r' =_o r''$

shows $r \leq_o r''$

$\langle proof \rangle$

lemma *ordIso-ordLeq-trans*[*trans*]:

assumes $r =_o r'$ **and** $r' \leq_o r''$

shows $r \leq_o r''$

$\langle proof \rangle$

lemma *ordLess-ordIso-trans*[*trans*]:

assumes $r <_o r'$ **and** $r' =_o r''$

shows $r <_o r''$

$\langle proof \rangle$

lemma *ordIso-ordLess-trans*[*trans*]:

assumes $r =_o r'$ **and** $r' <_o r''$

shows $r <_o r''$

$\langle proof \rangle$

lemma *ordLess-not-embed*:

assumes $r <_o r'$

shows $\neg(\exists f'. \text{embed } r' \ r \ f')$

$\langle proof \rangle$

lemma *ordLess-Field*:

assumes *OL*: $r1 <_o r2$ **and** *EMB*: $\text{embed } r1 \ r2 \ f$

shows $\neg (f'(\text{Field } r1) = \text{Field } r2)$

<proof>

lemma *ordLess-iff*:

$r <_o r' = (\text{Well-order } r \wedge \text{Well-order } r' \wedge \neg(\exists f'. \text{embed } r' \ r \ f'))$

<proof>

lemma *ordLess-irreflexive*: $\neg r <_o r$

<proof>

lemma *ordLeq-iff-ordLess-or-ordIso*:

$r \leq_o r' = (r <_o r' \vee r =_o r')$

<proof>

lemma *ordIso-iff-ordLeq*:

$(r =_o r') = (r \leq_o r' \wedge r' \leq_o r)$

<proof>

lemma *not-ordLess-ordLeq*:

$r <_o r' \implies \neg r' \leq_o r$

<proof>

lemma *not-ordLeq-ordLess*:

$r \leq_o r' \implies \neg r' <_o r$

<proof>

lemma *ordLess-or-ordLeq*:

assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'

shows $r <_o r' \vee r' \leq_o r$

<proof>

lemma *not-ordLess-ordIso*:

$r <_o r' \implies \neg r =_o r'$

<proof>

lemma *not-ordLeq-iff-ordLess*:

assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'

shows $(\neg r' \leq_o r) = (r <_o r')$

<proof>

lemma *not-ordLess-iff-ordLeq*:

assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'

shows $(\neg r' <_o r) = (r \leq_o r')$

<proof>

lemma *ordLess-transitive[trans]*:

$\llbracket r <_o r'; r' <_o r'' \rrbracket \implies r <_o r''$

$\langle \text{proof} \rangle$

corollary *ordLess-trans*: *trans ordLess*

$\langle \text{proof} \rangle$

lemmas *ordIso-equivalence* = *ordIso-transitive ordIso-reflexive ordIso-symmetric*

lemma *ordIso-imp-ordLeq*:

$r =_o r' \implies r \leq_o r'$

$\langle \text{proof} \rangle$

lemma *ordLess-imp-ordLeq*:

$r <_o r' \implies r \leq_o r'$

$\langle \text{proof} \rangle$

lemma *ofilter-subset-ordLeq*:

assumes *WELL*: *Well-order r* **and**

OFA: *wo-rel.ofilter r A* **and** *OFB*: *wo-rel.ofilter r B*

shows $(A \leq B) = (\text{Restr } r \ A \leq_o \text{Restr } r \ B)$

$\langle \text{proof} \rangle$

lemma *ofilter-subset-ordLess*:

assumes *WELL*: *Well-order r* **and**

OFA: *wo-rel.ofilter r A* **and** *OFB*: *wo-rel.ofilter r B*

shows $(A < B) = (\text{Restr } r \ A <_o \text{Restr } r \ B)$

$\langle \text{proof} \rangle$

lemma *ofilter-ordLess*:

$\llbracket \text{Well-order } r; \text{wo-rel.ofilter } r \ A \rrbracket \implies (A < \text{Field } r) = (\text{Restr } r \ A <_o r)$

$\langle \text{proof} \rangle$

corollary *underS-Restr-ordLess*:

assumes *Well-order r* **and** *Field r* $\neq \{\}$

shows $\text{Restr } r \ (\text{underS } r \ a) <_o r$

$\langle \text{proof} \rangle$

lemma *embed-ordLess-ofilterIncl*:

assumes

OL12: $r1 <_o r2$ **and** *OL23*: $r2 <_o r3$ **and**

EMB13: $\text{embed } r1 \ r3 \ f13$ **and** *EMB23*: $\text{embed } r2 \ r3 \ f23$

shows $(f13'(Field \ r1), f23'(Field \ r2)) \in (\text{ofilterIncl } r3)$

$\langle \text{proof} \rangle$

lemma *ordLess-iff-ordIso-Restr*:

assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'*

shows $(r' <_o r) = (\exists a \in Field \ r. r' =_o \text{Restr } r \ (\text{underS } r \ a))$

$\langle \text{proof} \rangle$

lemma *internalize-ordLess*:

$(r' <_o r) = (\exists p. \text{Field } p < \text{Field } r \wedge r' =_o p \wedge p <_o r)$
 $\langle \text{proof} \rangle$

lemma *internalize-ordLeq*:

$(r' \leq_o r) = (\exists p. \text{Field } p \leq \text{Field } r \wedge r' =_o p \wedge p \leq_o r)$
 $\langle \text{proof} \rangle$

lemma *ordLeq-iff-ordLess-Restr*:

assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'
shows $(r \leq_o r') = (\forall a \in \text{Field } r. \text{Restr } r (\text{underS } r a) <_o r')$
 $\langle \text{proof} \rangle$

lemma *finite-ordLess-infinite*:

assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r' **and**
FIN: *finite*(*Field* r) **and** *INF*: $\neg \text{finite}(\text{Field } r')$
shows $r <_o r'$
 $\langle \text{proof} \rangle$

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 $r =_o r'$
 $\langle \text{proof} \rangle$

29.5 $<_o$ is well-founded

Of course, it only makes sense to state that the $<_o$ is well-founded on the restricted type $'a \text{ rel } \text{rel}$. 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* $:: 'a \text{ rel} \Rightarrow 'a \text{ rel} \Rightarrow 'a \text{ set}$

where *ord-to-filter* $r0 \ r \equiv (\text{SOME } f. \text{embed } r \ r0 \ f) \text{ ' (Field } r)$

lemma *ord-to-filter-compat*:

compat (*ordLess* *Int* (*ordLess*⁻¹ “ $\{r0\} \times \text{ordLess}^{-1}$ “ $\{r0\}$ ”))
(ofilterIncl $r0$)
(ord-to-filter $r0$)
 $\langle \text{proof} \rangle$

theorem *wf-ordLess*: *wf* *ordLess*

$\langle \text{proof} \rangle$

corollary *exists-minim-Well-order*:

assumes *NE*: $R \neq \{\}$ **and** *WELL*: $\forall r \in R. \text{Well-order } r$
shows $\exists r \in R. \forall r' \in R. r \leq_o r'$
 $\langle \text{proof} \rangle$

29.6 Copy via direct images

The direct image operator is the dual of the inverse image operator *inv-image* from *Relation.thy*. It is useful for transporting a well-order between different types.

definition *dir-image* :: $'a \text{ rel} \Rightarrow ('a \Rightarrow 'a') \Rightarrow 'a' \text{ rel}$

where

$$\text{dir-image } r \ f = \{(f \ a, f \ b) \mid a \ b. (a, b) \in r\}$$

lemma *dir-image-Field*:

$$\text{Field}(\text{dir-image } r \ f) = f \cdot (\text{Field } r)$$

<proof>

lemma *dir-image-minus-Id*:

$$\text{inj-on } f \ (\text{Field } r) \implies (\text{dir-image } r \ f) - \text{Id} = \text{dir-image } (r - \text{Id}) \ f$$

<proof>

lemma *dir-image-subset*:

assumes $r \subseteq A \times B$

shows $\text{dir-image } r \ f \subseteq f \cdot A \times f \cdot B$

<proof>

lemma *Refl-dir-image*:

assumes *Refl* r

shows *Refl* $(\text{dir-image } r \ f)$

<proof>

lemma *trans-dir-image*:

assumes *TRANS*: *trans* r **and** *INJ*: *inj-on* $f \ (\text{Field } r)$

shows *trans* $(\text{dir-image } r \ f)$

<proof>

lemma *Preorder-dir-image*:

assumes *Preorder* r **and** *inj*: *inj-on* $f \ (\text{Field } r)$

shows *Preorder* $(\text{dir-image } r \ f)$

<proof>

lemma *antisym-dir-image*:

assumes *AN*: *antisym* r **and** *INJ*: *inj-on* $f \ (\text{Field } r)$

shows *antisym* $(\text{dir-image } r \ f)$

<proof>

lemma *Partial-order-dir-image*:

$$\llbracket \text{Partial-order } r; \text{inj-on } f \ (\text{Field } r) \rrbracket \implies \text{Partial-order } (\text{dir-image } r \ f)$$

<proof>

lemma *Total-dir-image*:

assumes *TOT*: *Total* r **and** *INJ*: *inj-on* $f \ (\text{Field } r)$

shows *Total* $(\text{dir-image } r \ f)$

$\langle proof \rangle$

lemma *Linear-order-dir-image:*

$\llbracket \text{Linear-order } r; \text{inj-on } f \text{ (Field } r) \rrbracket \implies \text{Linear-order (dir-image } r \text{ } f)$

$\langle proof \rangle$

lemma *wf-dir-image:*

assumes *WF: wf r and INJ: inj-on f (Field r)*

shows *wf(dir-image r f)*

$\langle proof \rangle$

lemma *Well-order-dir-image:*

$\llbracket \text{Well-order } r; \text{inj-on } f \text{ (Field } r) \rrbracket \implies \text{Well-order (dir-image } r \text{ } f)$

$\langle proof \rangle$

lemma *dir-image-bij-betw:*

$\llbracket \text{inj-on } f \text{ (Field } r) \rrbracket \implies \text{bij-betw } f \text{ (Field } r) \text{ (Field (dir-image } r \text{ } f))$

$\langle proof \rangle$

lemma *dir-image-compat:*

compat r (dir-image r f) f

$\langle proof \rangle$

lemma *dir-image-iso:*

$\llbracket \text{Well-order } r; \text{inj-on } f \text{ (Field } r) \rrbracket \implies \text{iso } r \text{ (dir-image } r \text{ } f) \text{ } f$

$\langle proof \rangle$

lemma *dir-image-ordIso:*

$\llbracket \text{Well-order } r; \text{inj-on } f \text{ (Field } r) \rrbracket \implies r =_o \text{dir-image } r \text{ } f$

$\langle proof \rangle$

lemma *Well-order-iso-copy:*

assumes *WELL: well-order-on A r and BIJ: bij-betw f A A'*

shows $\exists r'. \text{well-order-on } A' \text{ } r' \wedge r =_o r'$

$\langle proof \rangle$

29.7 Bounded square

This construction essentially defines, for an order relation r , a lexicographic order $bsqr \ r$ on $(Field \ r) \times (Field \ r)$, 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 $bsqr :: 'a \text{ rel} \Rightarrow ('a * 'a) \text{ rel}$

where

$$bsqr \ r = \{((a1, a2), (b1, b2)). \\ \{a1, a2, b1, b2\} \leq Field \ r \wedge \\ (a1 = b1 \wedge a2 = b2 \vee \\ (wo\text{-}rel.\text{max2} \ r \ a1 \ a2, wo\text{-}rel.\text{max2} \ r \ b1 \ b2) \in r - Id \vee \\ wo\text{-}rel.\text{max2} \ r \ a1 \ a2 = wo\text{-}rel.\text{max2} \ r \ b1 \ b2 \wedge (a1, b1) \in r - Id \vee \\ wo\text{-}rel.\text{max2} \ r \ a1 \ a2 = wo\text{-}rel.\text{max2} \ r \ b1 \ b2 \wedge a1 = b1 \wedge (a2, b2) \in r - Id \\)\}$$

lemma *Field-bsqr*:

$Field \ (bsqr \ r) = Field \ r \times Field \ r$

<proof>

lemma *bsqr-subset*:

assumes $r \subseteq Field \ r \times Field \ r$

shows $bsqr \ r \subseteq Field \ (bsqr \ r) \times Field \ (bsqr \ r)$

<proof>

lemma *bsqr-Refl*: $Refl \ (bsqr \ r)$

<proof>

lemma *bsqr-Trans*:

assumes *Well-order* r

shows *trans* $(bsqr \ r)$

<proof>

lemma *bsqr-antisym*:

assumes *Well-order* r

shows *antisym* $(bsqr \ r)$

<proof>

lemma *bsqr-Total*:

assumes *Well-order* r

shows *Total* $(bsqr \ r)$

<proof>

lemma *bsqr-Linear-order*:

assumes *Well-order* r

shows *Linear-order* $(bsqr \ r)$

<proof>

lemma *bsqr-Well-order*:

assumes *Well-order* r

shows *Well-order*(*bsqr r*)
 ⟨*proof*⟩

lemma *bsqr-max2*:

assumes *WELL*: *Well-order r* **and** *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*⟩

lemma *bsqr-ofilter*:

assumes *WELL*: *Well-order r* **and**

OF: *wo-rel.ofilter* (*bsqr r*) *D* **and** *SUB*: $D < \text{Field } r \times \text{Field } r$ **and**

NE: $\neg (\exists a. \text{Field } r = \text{under } r \ a)$

shows $\exists A. \text{wo-rel.ofilter } r \ A \wedge A < \text{Field } r \wedge D \leq A \times A$

⟨*proof*⟩

definition *Func* **where**

Func $A \ B = \{f . (\forall a \in A. f \ a \in B) \wedge (\forall a. a \notin A \longrightarrow f \ a = \text{undefined})\}$

lemma *Func-empty*:

Func $\{\}$ $B = \{\lambda x. \text{undefined}\}$

⟨*proof*⟩

lemma *Func-elim*:

assumes $g \in \text{Func } A \ B$ **and** $a \in A$

shows $\exists b. b \in B \wedge g \ a = b$

⟨*proof*⟩

definition *curr* **where**

curr $A \ f \equiv \lambda a. \text{if } a \in A \text{ then } \lambda b. f \ (a, b) \text{ else undefined}$

lemma *curr-in*:

assumes $f: f \in \text{Func } (A \times B) \ C$

shows $\text{curr } A \ f \in \text{Func } A \ (\text{Func } B \ C)$

⟨*proof*⟩

lemma *curr-inj*:

assumes $f1 \in \text{Func } (A \times B) \ C$ **and** $f2 \in \text{Func } (A \times B) \ C$

shows $\text{curr } A \ f1 = \text{curr } A \ f2 \longleftrightarrow f1 = f2$

⟨*proof*⟩

lemma *curr-surj*:

assumes $g \in \text{Func } A \ (\text{Func } B \ C)$

shows $\exists f \in \text{Func } (A \times B) \ C. \text{curr } A \ f = g$

⟨*proof*⟩

lemma *bij-betw-curr*:

bij-betw (*curr* A) (*Func* $(A \times B) \ C$) (*Func* $A \ (\text{Func } B \ C)$)

⟨*proof*⟩

definition *Func-map* **where**

Func-map $B2\ f1\ f2\ g\ b2 \equiv \text{if } b2 \in B2 \text{ then } f1\ (g\ (f2\ b2)) \text{ else undefined}$

lemma *Func-map*:

assumes $g: g \in \text{Func } A2\ A1$ **and** $f1: f1 \text{ ‘ } A1 \subseteq B1$ **and** $f2: f2 \text{ ‘ } B2 \subseteq A2$

shows $\text{Func-map } B2\ f1\ f2\ g \in \text{Func } B2\ B1$

$\langle \text{proof} \rangle$

lemma *Func-non-emp*:

assumes $B \neq \{\}$

shows $\text{Func } A\ B \neq \{\}$

$\langle \text{proof} \rangle$

lemma *Func-is-emp*:

$\text{Func } A\ B = \{\} \longleftrightarrow A \neq \{\} \wedge B = \{\}$ (**is** $?L \longleftrightarrow ?R$)

$\langle \text{proof} \rangle$

lemma *Func-map-surj*:

assumes $B1: f1 \text{ ‘ } A1 = B1$ **and** $A2: \text{inj-on } f2\ B2\ f2 \text{ ‘ } B2 \subseteq A2$

and $B2A2: B2 = \{\} \implies A2 = \{\}$

shows $\text{Func } B2\ B1 = \text{Func-map } B2\ f1\ f2 \text{ ‘ } \text{Func } A2\ A1$

$\langle \text{proof} \rangle$

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, powersets, 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 *minim* w.r.t. the order-embedding relation, \leq_o (which is the same as being *minimal* w.r.t. the strict order-embedding relation, $<_o$), among all the well-orders on its field.

definition *card-order-on* :: 'a set \Rightarrow 'a rel \Rightarrow bool

where

card-order-on A r \equiv *well-order-on* A r \wedge ($\forall r'. \text{well-order-on } A \ r' \longrightarrow r \leq_o r'$)

abbreviation *Card-order* r \equiv *card-order-on* (Field r) r

abbreviation *card-order* r \equiv *card-order-on* UNIV r

lemma *card-order-on-well-order-on*:

assumes *card-order-on* A r

shows *well-order-on* A r

<proof>

lemma *card-order-on-Card-order*:

card-order-on A r \implies A = Field r \wedge *Card-order* r

<proof>

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 *card-order-on*: $\exists r. \text{card-order-on } A \ r$

<proof>

lemma *card-order-on-ordIso*:

assumes CO: *card-order-on* A r **and** CO': *card-order-on* A r'

shows $r =_o r'$

<proof>

lemma *Card-order-ordIso*:

assumes CO: *Card-order* r **and** ISO: $r' =_o r$

shows *Card-order* r'

<proof>

lemma *Card-order-ordIso2*:

assumes *CO*: Card-order r **and** *ISO*: $r =_o r'$
shows Card-order r'
 $\langle proof \rangle$

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 \Rightarrow 'a rel ($\langle \langle \text{open-block notation} = \langle \text{mixfix card-of} \rangle \rangle | \cdot | \rangle$)
where *card-of* $A = (\text{SOME } r. \text{ card-order-on } A \ r)$

lemma *card-of-card-order-on*: card-order-on $A \ |A|$
 $\langle proof \rangle$

lemma *card-of-well-order-on*: well-order-on $A \ |A|$
 $\langle proof \rangle$

lemma *Field-card-of*: Field $|A| = A$
 $\langle proof \rangle$

lemma *card-of-Card-order*: Card-order $|A|$
 $\langle proof \rangle$

corollary *ordIso-card-of-imp-Card-order*:
 $r =_o |A| \implies \text{Card-order } r$
 $\langle proof \rangle$

lemma *card-of-Well-order*: Well-order $|A|$
 $\langle proof \rangle$

lemma *card-of-refl*: $|A| =_o |A|$
 $\langle proof \rangle$

lemma *card-of-least*: well-order-on $A \ r \implies |A| \leq_o r$
 $\langle proof \rangle$

lemma *card-of-ordIso*:
 $(\exists f. \text{bij-betw } f \ A \ B) = (|A| =_o |B|)$
 $\langle proof \rangle$

lemma *card-of-ordLeq*:
 $(\exists f. \text{inj-on } f \ A \wedge f' A \leq B) = (|A| \leq_o |B|)$
 $\langle proof \rangle$

lemma *card-of-ordLeq2*:
 $A \neq \{\}$ $\implies (\exists g. g' B = A) = (|A| \leq_o |B|)$
 $\langle proof \rangle$

lemma *card-of-ordLess*:

$(\neg(\exists f. \text{inj-on } f \ A \wedge f \text{ ` } A \leq B)) = (|B| <_o |A|)$
 $\langle \text{proof} \rangle$

lemma *card-of-ordLess2*:

$B \neq \{\}$ $\implies (\neg(\exists f. f \text{ ` } A = B)) = (|A| <_o |B|)$
 $\langle \text{proof} \rangle$

lemma *card-of-ordIsoI*:

assumes *bij-betw* $f \ A \ B$
shows $|A| =_o |B|$
 $\langle \text{proof} \rangle$

lemma *card-of-ordLeqI*:

assumes *inj-on* $f \ A$ **and** $\bigwedge a. a \in A \implies f \ a \in B$
shows $|A| \leq_o |B|$
 $\langle \text{proof} \rangle$

lemma *card-of-unique*:

card-order-on $A \ r \implies r =_o |A|$
 $\langle \text{proof} \rangle$

lemma *card-of-mono1*:

$A \leq B \implies |A| \leq_o |B|$
 $\langle \text{proof} \rangle$

lemma *card-of-mono2*:

assumes $r \leq_o r'$
shows $|\text{Field } r| \leq_o |\text{Field } r'|$
 $\langle \text{proof} \rangle$

lemma *card-of-cong*: $r =_o r' \implies |\text{Field } r| =_o |\text{Field } r'|$

$\langle \text{proof} \rangle$

lemma *card-of-Field-ordIso*:

assumes *Card-order* r
shows $|\text{Field } r| =_o r$
 $\langle \text{proof} \rangle$

lemma *Card-order-iff-ordIso-card-of*:

Card-order $r = (r =_o |\text{Field } r|)$
 $\langle \text{proof} \rangle$

lemma *Card-order-iff-ordLeq-card-of*:

Card-order $r = (r \leq_o |\text{Field } r|)$
 $\langle \text{proof} \rangle$

lemma *Card-order-iff-Restr-underS*:

assumes *Well-order* r

shows $\text{Card-order } r = (\forall a \in \text{Field } r. \text{Restr } r (\text{underS } r \ a) <_o |\text{Field } r|)$
 $\langle \text{proof} \rangle$

lemma *card-of-underS*:
assumes r : *Card-order* r **and** a : $a \in \text{Field } r$
shows $|\text{underS } r \ a| <_o r$
 $\langle \text{proof} \rangle$

lemma *ordLess-Field*:
assumes $r <_o r'$
shows $|\text{Field } r| <_o r'$
 $\langle \text{proof} \rangle$

lemma *internalize-card-of-ordLeq*:
 $(|A| \leq_o r) = (\exists B \leq \text{Field } r. |A| =_o |B| \wedge |B| \leq_o r)$
 $\langle \text{proof} \rangle$

lemma *internalize-card-of-ordLeq2*:
 $(|A| \leq_o |C|) = (\exists B \leq C. |A| =_o |B| \wedge |B| \leq_o |C|)$
 $\langle \text{proof} \rangle$

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*: $|\{\}| \leq_o |A|$
 $\langle \text{proof} \rangle$

lemma *card-of-empty1*:
assumes *Well-order* $r \vee$ *Card-order* r
shows $|\{\}| \leq_o r$
 $\langle \text{proof} \rangle$

corollary *Card-order-empty*:
 $\text{Card-order } r \implies |\{\}| \leq_o r$ $\langle \text{proof} \rangle$

lemma *card-of-empty2*:
assumes $|A| =_o |\{\}|$
shows $A = \{\}$
 $\langle \text{proof} \rangle$

lemma *card-of-empty3*:
assumes $|A| \leq_o |\{\}|$
shows $A = \{\}$
 $\langle \text{proof} \rangle$

lemma *card-of-empty-ordIso*:

$$|\{\}::'a \text{ set}| =_o |\{\}::'b \text{ set}|$$

$\langle \text{proof} \rangle$

lemma *card-of-image*:

$$|f \cdot A| \leq_o |A|$$

$\langle \text{proof} \rangle$

lemma *surj-imp-ordLeq*:

assumes $B \subseteq f \cdot A$

shows $|B| \leq_o |A|$

$\langle \text{proof} \rangle$

lemma *card-of-singl-ordLeq*:

assumes $A \neq \{\}$

shows $|\{b\}| \leq_o |A|$

$\langle \text{proof} \rangle$

corollary *Card-order-singl-ordLeq*:

$$\llbracket \text{Card-order } r; \text{Field } r \neq \{\} \rrbracket \implies |\{b\}| \leq_o r$$

$\langle \text{proof} \rangle$

lemma *card-of-Pow*: $|A| <_o |\text{Pow } A|$

$\langle \text{proof} \rangle$

corollary *Card-order-Pow*:

$$\text{Card-order } r \implies r <_o |\text{Pow}(\text{Field } r)|$$

$\langle \text{proof} \rangle$

lemma *card-of-Plus1*: $|A| \leq_o |A <+> B|$ **and** *card-of-Plus2*: $|B| \leq_o |A <+> B|$

$\langle \text{proof} \rangle$

corollary *Card-order-Plus1*:

$$\text{Card-order } r \implies r \leq_o |(Field\ r) <+> B|$$

$\langle \text{proof} \rangle$

corollary *Card-order-Plus2*:

$$\text{Card-order } r \implies r \leq_o |A <+> (Field\ r)|$$

$\langle \text{proof} \rangle$

lemma *card-of-Plus-empty1*: $|A| =_o |A <+> \{\}|$

$\langle \text{proof} \rangle$

lemma *card-of-Plus-empty2*: $|A| =_o |\{\} <+> A|$

$\langle \text{proof} \rangle$

lemma *card-of-Plus-commute*: $|A <+> B| =_o |B <+> A|$

$\langle \text{proof} \rangle$

lemma *card-of-Plus-assoc*:

fixes $A :: 'a \text{ set}$ **and** $B :: 'b \text{ set}$ **and** $C :: 'c \text{ set}$

shows $|(A <+> B) <+> C| =_o |A <+> B <+> C|$

<proof>

lemma *card-of-Plus-mono1*:

assumes $|A| \leq_o |B|$

shows $|A <+> C| \leq_o |B <+> C|$

<proof>

corollary *ordLeq-Plus-mono1*:

assumes $r \leq_o r'$

shows $|(Field\ r) <+> C| \leq_o |(Field\ r') <+> C|$

<proof>

lemma *card-of-Plus-mono2*:

assumes $|A| \leq_o |B|$

shows $|C <+> A| \leq_o |C <+> B|$

<proof>

corollary *ordLeq-Plus-mono2*:

assumes $r \leq_o r'$

shows $|A <+> (Field\ r)| \leq_o |A <+> (Field\ r')|$

<proof>

lemma *card-of-Plus-mono*:

assumes $|A| \leq_o |B|$ **and** $|C| \leq_o |D|$

shows $|A <+> C| \leq_o |B <+> D|$

<proof>

corollary *ordLeq-Plus-mono*:

assumes $r \leq_o r'$ **and** $p \leq_o p'$

shows $|(Field\ r) <+> (Field\ p)| \leq_o |(Field\ r') <+> (Field\ p')|$

<proof>

lemma *card-of-Plus-cong1*:

assumes $|A| =_o |B|$

shows $|A <+> C| =_o |B <+> C|$

<proof>

corollary *ordIso-Plus-cong1*:

assumes $r =_o r'$

shows $|(Field\ r) <+> C| =_o |(Field\ r') <+> C|$

<proof>

lemma *card-of-Plus-cong2*:

assumes $|A| =_o |B|$

shows $|C <+> A| =_o |C <+> B|$

<proof>

corollary *ordIso-Plus-cong2*:

assumes $r =_o r'$

shows $|A <+> (Field\ r)| =_o |A <+> (Field\ r')|$

<proof>

lemma *card-of-Plus-cong*:

assumes $|A| =_o |B|$ **and** $|C| =_o |D|$

shows $|A <+> C| =_o |B <+> D|$

<proof>

corollary *ordIso-Plus-cong*:

assumes $r =_o r'$ **and** $p =_o p'$

shows $|(Field\ r) <+> (Field\ p)| =_o |(Field\ r') <+> (Field\ p')|$

<proof>

lemma *card-of-Un-Plus-ordLeq*:

$|A \cup B| \leq_o |A <+> B|$

<proof>

lemma *card-of-Times1*:

assumes $A \neq \{\}$

shows $|B| \leq_o |B \times A|$

<proof>

lemma *card-of-Times-commute*: $|A \times B| =_o |B \times A|$

<proof>

lemma *card-of-Times2*:

assumes $A \neq \{\}$ **shows** $|B| \leq_o |A \times B|$

<proof>

corollary *Card-order-Times1*:

$\llbracket Card\text{-}order\ r; B \neq \{\} \rrbracket \implies r \leq_o |(Field\ r) \times B|$

<proof>

corollary *Card-order-Times2*:

$\llbracket Card\text{-}order\ r; A \neq \{\} \rrbracket \implies r \leq_o |A \times (Field\ r)|$

<proof>

lemma *card-of-Times3*: $|A| \leq_o |A \times A|$

<proof>

lemma *card-of-Plus-Times-bool*: $|A <+> A| =_o |A \times (UNIV::bool\ set)|$

<proof>

lemma *card-of-Times-mono1*:

assumes $|A| \leq_o |B|$

shows $|A \times C| \leq_o |B \times C|$

$\langle proof \rangle$

corollary *ordLeq-Times-mono1*:

assumes $r \leq_o r'$

shows $|(Field\ r) \times C| \leq_o |(Field\ r') \times C|$

$\langle proof \rangle$

lemma *card-of-Times-mono2*:

assumes $|A| \leq_o |B|$

shows $|C \times A| \leq_o |C \times B|$

$\langle proof \rangle$

corollary *ordLeq-Times-mono2*:

assumes $r \leq_o r'$

shows $|A \times (Field\ r)| \leq_o |A \times (Field\ r')|$

$\langle proof \rangle$

lemma *card-of-Sigma-mono1*:

assumes $\forall i \in I. |A\ i| \leq_o |B\ i|$

shows $|SIGMA\ i : I. A\ i| \leq_o |SIGMA\ i : I. B\ i|$

$\langle proof \rangle$

lemma *card-of-UNION-Sigma*:

$|\bigcup i \in I. A\ i| \leq_o |SIGMA\ i : I. A\ i|$

$\langle proof \rangle$

lemma *card-of-bool*:

assumes $a1 \neq a2$

shows $|UNIV::bool\ set| =_o |\{a1, a2\}|$

$\langle proof \rangle$

lemma *card-of-Plus-Times-aux*:

assumes $A2: a1 \neq a2 \wedge \{a1, a2\} \leq A$ **and**

$LEQ: |A| \leq_o |B|$

shows $|A <+> B| \leq_o |A \times B|$

$\langle proof \rangle$

lemma *card-of-Plus-Times*:

assumes $A2: a1 \neq a2 \wedge \{a1, a2\} \leq A$ **and** $B2: b1 \neq b2 \wedge \{b1, b2\} \leq B$

shows $|A <+> B| \leq_o |A \times B|$

$\langle proof \rangle$

lemma *card-of-Times-Plus-distrib*:

$|A \times (B <+> C)| =_o |A \times B <+> A \times C|$ (**is** $|?RHS| =_o |?LHS|$)

$\langle proof \rangle$

lemma *card-of-ordLeq-finite*:

assumes $|A| \leq_o |B|$ **and** *finite B*

shows *finite A*

$\langle proof \rangle$

lemma *card-of-ordLeq-infinite*:

assumes $|A| \leq_o |B|$ **and** $\neg finite\ A$

shows $\neg finite\ B$

$\langle proof \rangle$

lemma *card-of-ordIso-finite*:

assumes $|A| =_o |B|$

shows $finite\ A = finite\ B$

$\langle proof \rangle$

lemma *card-of-ordIso-finite-Field*:

assumes *Card-order* r **and** $r =_o |A|$

shows $finite(Field\ r) = finite\ A$

$\langle proof \rangle$

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*:

$\neg finite\ A \longleftrightarrow (|UNIV::nat\ set| \leq_o |A|)$

$\langle proof \rangle$

The next two results correspond to the ZF fact that all infinite cardinals are limit ordinals:

lemma *Card-order-infinite-not-under*:

assumes *CARD*: *Card-order* r **and** *INF*: $\neg finite\ (Field\ r)$

shows $\neg (\exists a. Field\ r = under\ r\ a)$

$\langle proof \rangle$

lemma *infinite-Card-order-limit*:

assumes r : *Card-order* r **and** $\neg finite\ (Field\ r)$

and a : $a \in Field\ r$

shows $\exists b \in Field\ r. a \neq b \wedge (a, b) \in r$

$\langle proof \rangle$

theorem *Card-order-Times-same-infinite*:

assumes *CO*: *Card-order* r **and** *INF*: $\neg finite(Field\ r)$

shows $|Field\ r \times Field\ r| \leq_o r$

$\langle proof \rangle$

corollary *card-of-Times-same-infinite*:

assumes $\neg \text{finite } A$
shows $|A \times A| =_o |A|$
 $\langle \text{proof} \rangle$

lemma *card-of-Times-infinite*:

assumes *INF*: $\neg \text{finite } A$ **and** *NE*: $B \neq \{\}$ **and** *LEQ*: $|B| \leq_o |A|$
shows $|A \times B| =_o |A| \wedge |B \times A| =_o |A|$
 $\langle \text{proof} \rangle$

corollary *Card-order-Times-infinite*:

assumes *INF*: $\neg \text{finite}(\text{Field } r)$ **and** *CARD*: *Card-order* r **and**
NE: $\text{Field } p \neq \{\}$ **and** *LEQ*: $p \leq_o r$
shows $|(\text{Field } r) \times (\text{Field } p)| =_o r \wedge |(\text{Field } p) \times (\text{Field } r)| =_o r$
 $\langle \text{proof} \rangle$

lemma *card-of-Sigma-ordLeq-infinite*:

assumes *INF*: $\neg \text{finite } B$ **and**
LEQ-I: $|I| \leq_o |B|$ **and** *LEQ*: $\forall i \in I. |A \ i| \leq_o |B|$
shows $|\text{SIGMA } i : I. A \ i| \leq_o |B|$
 $\langle \text{proof} \rangle$

lemma *card-of-Sigma-ordLeq-infinite-Field*:

assumes *INF*: $\neg \text{finite}(\text{Field } r)$ **and** r : *Card-order* r **and**
LEQ-I: $|I| \leq_o r$ **and** *LEQ*: $\forall i \in I. |A \ i| \leq_o r$
shows $|\text{SIGMA } i : I. A \ i| \leq_o r$
 $\langle \text{proof} \rangle$

lemma *card-of-Times-ordLeq-infinite-Field*:

$\llbracket \neg \text{finite}(\text{Field } r); |A| \leq_o r; |B| \leq_o r; \text{Card-order } r \rrbracket \implies |A \times B| \leq_o r$
 $\langle \text{proof} \rangle$

lemma *card-of-Times-infinite-simps*:

$\llbracket \neg \text{finite } A; B \neq \{\}; |B| \leq_o |A| \rrbracket \implies |A \times B| =_o |A|$
 $\llbracket \neg \text{finite } A; B \neq \{\}; |B| \leq_o |A| \rrbracket \implies |A| =_o |A \times B|$
 $\llbracket \neg \text{finite } A; B \neq \{\}; |B| \leq_o |A| \rrbracket \implies |B \times A| =_o |A|$
 $\llbracket \neg \text{finite } A; B \neq \{\}; |B| \leq_o |A| \rrbracket \implies |A| =_o |B \times A|$
 $\langle \text{proof} \rangle$

lemma *card-of-UNION-ordLeq-infinite*:

assumes *INF*: $\neg \text{finite } B$ **and** *LEQ-I*: $|I| \leq_o |B|$ **and** *LEQ*: $\forall i \in I. |A \ i| \leq_o |B|$
shows $|\bigcup i \in I. A \ i| \leq_o |B|$
 $\langle \text{proof} \rangle$

corollary *card-of-UNION-ordLeq-infinite-Field*:

assumes *INF*: $\neg \text{finite}(\text{Field } r)$ **and** r : *Card-order* r **and**
LEQ-I: $|I| \leq_o r$ **and** *LEQ*: $\forall i \in I. |A \ i| \leq_o r$
shows $|\bigcup i \in I. A \ i| \leq_o r$
 $\langle \text{proof} \rangle$

lemma *card-of-Plus-infinite1*:
 assumes *INF*: $\neg \text{finite } A$ and *LEQ*: $|B| \leq_o |A|$
 shows $|A <+> B| =_o |A|$
 $\langle \text{proof} \rangle$

lemma *card-of-Plus-infinite2*:
 assumes *INF*: $\neg \text{finite } A$ and *LEQ*: $|B| \leq_o |A|$
 shows $|B <+> A| =_o |A|$
 $\langle \text{proof} \rangle$

lemma *card-of-Plus-infinite*:
 assumes *INF*: $\neg \text{finite } A$ and *LEQ*: $|B| \leq_o |A|$
 shows $|A <+> B| =_o |A| \wedge |B <+> A| =_o |A|$
 $\langle \text{proof} \rangle$

corollary *Card-order-Plus-infinite*:
 assumes *INF*: $\neg \text{finite}(\text{Field } r)$ and *CARD*: *Card-order* r and
LEQ: $p \leq_o r$
 shows $|(\text{Field } r) <+> (\text{Field } p)| =_o r \wedge |(\text{Field } p) <+> (\text{Field } r)| =_o r$
 $\langle \text{proof} \rangle$

30.5 The cardinal ω and the finite cardinals

The cardinal ω , of natural numbers, shall be the standard non-strict order relation on *nat*, that we abbreviate by *natLeq*. The finite cardinals shall be the restrictions of these relations to the numbers smaller than fixed numbers n , that we abbreviate by *natLeq-on* n .

definition $(\text{natLeq}::(\text{nat} * \text{nat}) \text{ set}) \equiv \{(x,y). x \leq y\}$

definition $(\text{natLess}::(\text{nat} * \text{nat}) \text{ set}) \equiv \{(x,y). x < y\}$

abbreviation *natLeq-on* $:: \text{nat} \Rightarrow (\text{nat} * \text{nat}) \text{ set}$
 where *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 $\neg \text{finite } (A \times B)$
 $\langle \text{proof} \rangle$

30.5.1 First as well-orders

lemma *Field-natLeq*: *Field* *natLeq* = (*UNIV*::*nat set*)
 $\langle \text{proof} \rangle$

lemma *natLeq-Refl*: *Refl* *natLeq*
 $\langle \text{proof} \rangle$

lemma *natLeq-trans*: *trans* *natLeq*
 $\langle \text{proof} \rangle$

lemma *natLeq-Preorder: Preorder natLeq*
 $\langle \text{proof} \rangle$

lemma *natLeq-antisym: antisym natLeq*
 $\langle \text{proof} \rangle$

lemma *natLeq-Partial-order: Partial-order natLeq*
 $\langle \text{proof} \rangle$

lemma *natLeq-Total: Total natLeq*
 $\langle \text{proof} \rangle$

lemma *natLeq-Linear-order: Linear-order natLeq*
 $\langle \text{proof} \rangle$

lemma *natLeq-natLess-Id: natLess = natLeq - Id*
 $\langle \text{proof} \rangle$

lemma *natLeq-Well-order: Well-order natLeq*
 $\langle \text{proof} \rangle$

lemma *Field-natLeq-on: Field (natLeq-on n) = {x. x < n}*
 $\langle \text{proof} \rangle$

lemma *natLeq-underS-less: underS natLeq n = {x. x < n}*
 $\langle \text{proof} \rangle$

lemma *Restr-natLeq: Restr natLeq {x. x < n} = natLeq-on n*
 $\langle \text{proof} \rangle$

lemma *Restr-natLeq2:*
Restr natLeq (underS natLeq n) = natLeq-on n
 $\langle \text{proof} \rangle$

lemma *natLeq-on-Well-order: Well-order(natLeq-on n)*
 $\langle \text{proof} \rangle$

corollary *natLeq-on-well-order-on: well-order-on {x. x < n} (natLeq-on n)*
 $\langle \text{proof} \rangle$

lemma *natLeq-on-wo-rel: wo-rel(natLeq-on n)*
 $\langle \text{proof} \rangle$

30.5.2 Then as cardinals

lemma *natLeq-Card-order: Card-order natLeq*
 $\langle \text{proof} \rangle$

corollary *card-of-Field-natLeq:*

$|Field\ natLeq| =_o\ natLeq$
 $\langle proof \rangle$

corollary *card-of-nat*:

$|UNIV::nat\ set| =_o\ natLeq$
 $\langle proof \rangle$

corollary *infinite-iff-natLeq-ordLeq*:

$\neg finite\ A = (\ natLeq \leq_o\ |A|)$
 $\langle proof \rangle$

corollary *finite-iff-ordLess-natLeq*:

$finite\ A = (|A| <_o\ natLeq)$
 $\langle proof \rangle$

30.6 The successor of a cardinal

First we define *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* $:: 'a\ rel \Rightarrow 'a\ set\ rel \Rightarrow bool$

where

$isCardSuc\ r\ r' \equiv$
 $Card\text{-}order\ r' \wedge r <_o\ r' \wedge$
 $(\forall (r'': 'a\ set\ rel). Card\text{-}order\ r'' \wedge r <_o\ r'' \longrightarrow r' \leq_o\ r'')$

Now we introduce the cardinal-successor operator *cardSuc*, by picking *some* cardinal-order relation fulfilling *isCardSuc*. Again, the picked item shall be proved unique up to order-isomorphism.

definition *cardSuc* $:: 'a\ rel \Rightarrow 'a\ set\ rel$

where $cardSuc\ r \equiv SOME\ r'.\ isCardSuc\ r\ r'$

lemma *exists-minim-Card-order*:

$\llbracket R \neq \{\}; \forall r \in R. Card\text{-}order\ r \rrbracket \Longrightarrow \exists r \in R. \forall r' \in R. r \leq_o\ r'$
 $\langle proof \rangle$

lemma *exists-isCardSuc*:

assumes *Card-order* r

shows $\exists r'. isCardSuc\ r\ r'$

$\langle proof \rangle$

lemma *cardSuc-isCardSuc*:

assumes *Card-order* r

shows $isCardSuc\ r\ (cardSuc\ r)$

$\langle proof \rangle$

lemma *cardSuc-Card-order*:

$Card\text{-}order\ r \Longrightarrow Card\text{-}order(cardSuc\ r)$

$\langle \text{proof} \rangle$

lemma *cardSuc-greater:*

Card-order $r \implies r <_o \text{cardSuc } r$

$\langle \text{proof} \rangle$

lemma *cardSuc-ordLeq:*

Card-order $r \implies r \leq_o \text{cardSuc } r$

$\langle \text{proof} \rangle$

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:*

$\llbracket \text{Card-order } (r::'a \text{ rel}); \text{Card-order } (r'::'a \text{ set rel}); r <_o r' \rrbracket \implies \text{cardSuc } r \leq_o r'$

$\langle \text{proof} \rangle$

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 $\text{cardSuc } r \leq_o r'$

$\langle \text{proof} \rangle$

lemma *cardSuc-ordLess-ordLeq:*

assumes *CARD*: *Card-order* r **and** *CARD'*: *Card-order* r'

shows $(r <_o r') = (\text{cardSuc } r \leq_o r')$

$\langle \text{proof} \rangle$

lemma *cardSuc-ordLeq-ordLess:*

assumes *CARD*: *Card-order* r **and** *CARD'*: *Card-order* r'

shows $(r' <_o \text{cardSuc } r) = (r' \leq_o r)$

$\langle \text{proof} \rangle$

lemma *cardSuc-mono-ordLeq:*

assumes *CARD*: *Card-order* r **and** *CARD'*: *Card-order* r'

shows $(\text{cardSuc } r \leq_o \text{cardSuc } r') = (r \leq_o r')$

$\langle \text{proof} \rangle$

lemma *cardSuc-invar-ordIso:*

assumes *CARD*: *Card-order* r **and** *CARD'*: *Card-order* r'

shows $(\text{cardSuc } r =_o \text{cardSuc } r') = (r =_o r')$

$\langle \text{proof} \rangle$

lemma *card-of-cardSuc-finite:*

$\text{finite}(\text{Field}(\text{cardSuc } |A|)) = \text{finite } A$

$\langle \text{proof} \rangle$

lemma *cardSuc-finite:*

assumes *Card-order* r

shows $\text{finite}(\text{Field}(\text{cardSuc } r)) = \text{finite}(\text{Field } r)$

$\langle proof \rangle$

lemma *Field-cardSuc-not-empty:*

assumes *Card-order* r

shows *Field* ($cardSuc\ r$) $\neq \{\}$

$\langle proof \rangle$

typedef $'a\ suc = Field\ (cardSuc\ |\ UNIV :: 'a\ set|)$

$\langle proof \rangle$

definition $card-suc :: 'a\ rel \Rightarrow 'a\ suc\ rel$ **where**

$card-suc \equiv \lambda-. map-prod\ Abs-suc\ Abs-suc\ ' cardSuc\ |\ UNIV :: 'a\ set|$

lemma *Field-card-suc:* $Field\ (card-suc\ r) = UNIV$

$\langle proof \rangle$

lemma *card-suc-alt:* $card-suc\ r = dir-image\ (cardSuc\ |\ UNIV :: 'a\ set|)\ Abs-suc$

$\langle proof \rangle$

lemma *cardSuc-Well-order:* $Card-order\ r \Longrightarrow Well-order(cardSuc\ r)$

$\langle proof \rangle$

lemma *cardSuc-ordIso-card-suc:*

assumes *card-order* r

shows $cardSuc\ r =_o card-suc\ (r :: 'a\ rel)$

$\langle proof \rangle$

lemma *Card-order-card-suc:* $card-order\ r \Longrightarrow Card-order\ (card-suc\ r)$

$\langle proof \rangle$

lemma *card-order-card-suc:* $card-order\ r \Longrightarrow card-order\ (card-suc\ r)$

$\langle proof \rangle$

lemma *card-suc-greater:* $card-order\ r \Longrightarrow r <_o card-suc\ r$

$\langle proof \rangle$

lemma *card-of-Plus-ordLess-infinite:*

assumes *INF:* $\neg finite\ C$ **and** *LESS1:* $|A| <_o |C|$ **and** *LESS2:* $|B| <_o |C|$

shows $|A| <_+ B| <_o |C|$

$\langle proof \rangle$

lemma *card-of-Plus-ordLess-infinite-Field:*

assumes *INF:* $\neg finite\ (Field\ r)$ **and** $r: Card-order\ r$ **and**

LESS1: $|A| <_o r$ **and** *LESS2:* $|B| <_o r$

shows $|A| <_+ B| <_o r$

$\langle proof \rangle$

lemma *card-of-Plus-ordLeq-infinite-Field:*

assumes $r: \neg finite\ (Field\ r)$ **and** $A: |A| \leq_o r$ **and** $B: |B| \leq_o r$

and c : *Card-order* r
shows $|A <+> B| \leq_o r$
 $\langle proof \rangle$

lemma *card-of-Un-ordLeq-infinite-Field*:
assumes C : $\neg finite (Field\ r)$ **and** A : $|A| \leq_o r$ **and** B : $|B| \leq_o r$
and *Card-order* r
shows $|A\ Un\ B| \leq_o r$
 $\langle proof \rangle$

lemma *card-of-Un-ordLess-infinite*:
assumes INF : $\neg finite\ C$ **and**
 $LESS1$: $|A| <_o |C|$ **and** $LESS2$: $|B| <_o |C|$
shows $|A \cup B| <_o |C|$
 $\langle proof \rangle$

lemma *card-of-Un-ordLess-infinite-Field*:
assumes INF : $\neg finite (Field\ r)$ **and** r : *Card-order* r **and**
 $LESS1$: $|A| <_o r$ **and** $LESS2$: $|B| <_o r$
shows $|A\ Un\ B| <_o r$
 $\langle proof \rangle$

30.7 Regular cardinals

definition *cofinal* **where**
 $cofinal\ A\ r \equiv \forall a \in Field\ r. \exists b \in A. a \neq b \wedge (a, b) \in r$

definition *regularCard* **where**
 $regularCard\ r \equiv \forall K. K \leq Field\ r \wedge cofinal\ K\ r \longrightarrow |K| =_o r$

definition *relChain* **where**
 $relChain\ r\ As \equiv \forall i\ j. (i, j) \in r \longrightarrow As\ i \leq As\ j$

lemma *regularCard-UNION*:
assumes r : *Card-order* r $regularCard\ r$
and As : $relChain\ r\ As$
and $Bsub$: $B \leq (\bigcup i \in Field\ r. As\ i)$
and $cardB$: $|B| <_o r$
shows $\exists i \in Field\ r. B \leq As\ i$
 $\langle proof \rangle$

lemma *infinite-cardSuc-regularCard*:
assumes $r-inf$: $\neg finite (Field\ r)$ **and** $r-card$: *Card-order* r
shows $regularCard\ (cardSuc\ r)$
 $\langle proof \rangle$

lemma *cardSuc-UNION*:
assumes r : *Card-order* r **and** $\neg finite (Field\ r)$
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 $\exists i \in Field (cardSuc\ r). B \leq As\ i$
 $\langle proof \rangle$

30.8 Others

lemma *card-of-Func-Times*:
 $|Func\ (A \times B)\ C| =_o |Func\ A\ (Func\ B\ C)|$
 $\langle proof \rangle$

lemma *card-of-Pow-Func*:
 $|Pow\ A| =_o |Func\ A\ (UNIV::bool\ set)|$
 $\langle proof \rangle$

lemma *card-of-Func-UNIV*:
 $|Func\ (UNIV::'a\ set)\ (B::'b\ set)| =_o |\{f::'a \Rightarrow 'b. range\ f \subseteq B\}|$
 $\langle proof \rangle$

lemma *Func-Times-Range*:
 $|Func\ A\ (B \times C)| =_o |Func\ A\ B \times Func\ A\ C|$ (**is** $|?LHS| =_o |?RHS|$)
 $\langle proof \rangle$

30.9 Regular vs. stable cardinals

definition *stable* :: $'a\ rel \Rightarrow bool$
where
 $stable\ r \equiv \forall (A::'a\ set)\ (F::'a \Rightarrow 'a\ set).$
 $|A| <_o r \wedge (\forall a \in A. |F\ a| <_o r)$
 $\longrightarrow |SIGMA\ a : A. F\ a| <_o r$

lemma *regularCard-stable*:
assumes cr : *Card-order* r **and** ir : $\neg finite\ (Field\ r)$ **and** reg : *regularCard* r
shows *stable* r
 $\langle proof \rangle$

lemma *stable-regularCard*:
assumes cr : *Card-order* r **and** ir : $\neg finite\ (Field\ r)$ **and** st : *stable* r
shows *regularCard* r
 $\langle proof \rangle$

lemma *internalize-card-of-ordLess*:
 $(|A| <_o r) = (\exists B < Field\ r. |A| =_o |B| \wedge |B| <_o r)$
 $\langle proof \rangle$

lemma *card-of-Sigma-cong1*:
assumes $\forall i \in I. |A\ i| =_o |B\ i|$
shows $|SIGMA\ i : I. A\ i| =_o |SIGMA\ i : I. B\ i|$
 $\langle proof \rangle$

lemma *card-of-Sigma-cong2*:

assumes *bij-betw* f ($I::'i$ set) ($J::'j$ set)

shows $|\text{SIGMA } i : I. (A::'j \Rightarrow 'a \text{ set}) (f i)| =_o |\text{SIGMA } j : J. A j|$
 $\langle \text{proof} \rangle$

lemma *card-of-Sigma-cong*:

assumes *BIJ*: *bij-betw* f I J **and** *ISO*: $\forall j \in J. |A j| =_o |B j|$

shows $|\text{SIGMA } i : I. A (f i)| =_o |\text{SIGMA } j : J. B j|$
 $\langle \text{proof} \rangle$

lemma *stable-elim*:

assumes *ST*: *stable* r **and** *A-LESS*: $|A| <_o r$ **and**

F-LESS: $\bigwedge a. a \in A \Rightarrow |F a| <_o r$

shows $|\text{SIGMA } a : A. F a| <_o r$
 $\langle \text{proof} \rangle$

lemma *stable-natLeq*: *stable* *natLeq*

$\langle \text{proof} \rangle$

corollary *regularCard-natLeq*: *regularCard* *natLeq*

$\langle \text{proof} \rangle$

lemma *stable-ordIso1*:

assumes *ST*: *stable* r **and** *ISO*: $r' =_o r$

shows *stable* r'
 $\langle \text{proof} \rangle$

lemma *stable-UNION*:

assumes *stable* r **and** $|A| <_o r$ **and** $\bigwedge a. a \in A \Rightarrow |F a| <_o r$

shows $|\bigcup a \in A. F a| <_o r$

$\langle \text{proof} \rangle$

corollary *card-of-UNION-ordLess-infinite*:

assumes *stable* $|B|$ **and** $|I| <_o |B|$ **and** $\forall i \in I. |A i| <_o |B|$

shows $|\bigcup i \in I. A i| <_o |B|$

$\langle \text{proof} \rangle$

corollary *card-of-UNION-ordLess-infinite-Field*:

assumes *ST*: *stable* r **and** r : *Card-order* r **and**

LEQ-I: $|I| <_o r$ **and** *LEQ*: $\forall i \in I. |A i| <_o r$

shows $|\bigcup i \in I. A i| <_o r$

$\langle \text{proof} \rangle$

end

31 Cardinal Arithmetic as Needed by Bounded Natural Functors

theory *BNF-Cardinal-Arithmetic*
imports *BNF-Cardinal-Order-Relation*
begin

lemma *dir-image*: $\llbracket \bigwedge x y. (f x = f y) = (x = y); \text{Card-order } r \rrbracket \implies r =_o \text{dir-image } r f$
 $\langle \text{proof} \rangle$

lemma *card-order-dir-image*:
assumes *bij*: *bij f* **and** *co*: *card-order r*
shows *card-order (dir-image r f)*
 $\langle \text{proof} \rangle$

lemma *ordIso-refl*: *Card-order r* $\implies r =_o r$
 $\langle \text{proof} \rangle$

lemma *ordLeq-refl*: *Card-order r* $\implies r \leq_o r$
 $\langle \text{proof} \rangle$

lemma *card-of-ordIso-subst*: $A = B \implies |A| =_o |B|$
 $\langle \text{proof} \rangle$

lemma *Field-card-order*: *card-order r* $\implies \text{Field } r = \text{UNIV}$
 $\langle \text{proof} \rangle$

31.1 Zero

definition *czero* **where**
czero = *card-of* $\{\}$

lemma *czero-ordIso*: *czero* =_o *czero*
 $\langle \text{proof} \rangle$

lemma *card-of-ordIso-czero-iff-empty*:
 $|A| =_o (\text{czero} :: 'b \text{ rel}) \longleftrightarrow A = (\{\} :: 'a \text{ set})$
 $\langle \text{proof} \rangle$

abbreviation *Cnotzero* **where**
Cnotzero ($r :: 'a \text{ rel}$) $\equiv \neg(r =_o (\text{czero} :: 'a \text{ rel})) \wedge \text{Card-order } r$

lemma *Cnotzero-imp-not-empty*: *Cnotzero r* $\implies \text{Field } r \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *czeroI*:

$\llbracket \text{Card-order } r; \text{Field } r = \{\} \rrbracket \implies r =_o \text{czero}$
 $\langle \text{proof} \rangle$

lemma *czeroE*:

$r =_o \text{czero} \implies \text{Field } r = \{\}$
 $\langle \text{proof} \rangle$

lemma *Cnotzero-mono*:

$\llbracket \text{Cnotzero } r; \text{Card-order } q; r \leq_o q \rrbracket \implies \text{Cnotzero } q$
 $\langle \text{proof} \rangle$

31.2 (In)finite cardinals

definition *cinfinite* where

$\text{cinfinite } r \equiv (\neg \text{finite } (\text{Field } r))$

abbreviation *Cinfinite* where

$\text{Cinfinite } r \equiv \text{cinfinite } r \wedge \text{Card-order } r$

definition *cfinite* where

$\text{cfinite } r = \text{finite } (\text{Field } r)$

abbreviation *Cfinite* where

$\text{Cfinite } r \equiv \text{cfinite } r \wedge \text{Card-order } r$

lemma *Cfinite-ordLess-Cinfinite*: $\llbracket \text{Cfinite } r; \text{Cinfinite } s \rrbracket \implies r <_o s$
 $\langle \text{proof} \rangle$

lemmas *natLeq-card-order* = *natLeq-Card-order*[*unfolded Field-natLeq*]

lemma *natLeq-cinfinite*: $\text{cinfinite } \text{natLeq}$
 $\langle \text{proof} \rangle$

lemma *natLeq-Cinfinite*: $\text{Cinfinite } \text{natLeq}$
 $\langle \text{proof} \rangle$

lemma *natLeq-ordLeq-cinfinite*:

assumes *inf*: $\text{Cinfinite } r$

shows $\text{natLeq } \leq_o r$

$\langle \text{proof} \rangle$

lemma *cinfinite-not-czero*: $\text{cinfinite } r \implies \neg (r =_o (\text{czero} :: 'a \text{ rel}))$
 $\langle \text{proof} \rangle$

lemma *Cinfinite-Cnotzero*: $\text{Cinfinite } r \implies \text{Cnotzero } r$
 $\langle \text{proof} \rangle$

lemma *Cinfinite-cong*: $\llbracket r1 =_o r2; \text{Cinfinite } r1 \rrbracket \implies \text{Cinfinite } r2$
 $\langle \text{proof} \rangle$

lemma *cinfinite-mono*: $\llbracket r1 \leq_o r2; \text{cinfinite } r1 \rrbracket \implies \text{cinfinite } r2$
 ⟨proof⟩

lemma *regularCard-ordIso*:
 assumes $k =_o k'$ and *Cinfinite* k and *regularCard* k
 shows *regularCard* k'
 ⟨proof⟩

corollary *card-of-UNION-ordLess-infinite-Field-regularCard*:
 assumes *regularCard* r and *Cinfinite* r and $|I| <_o r$ and $\forall i \in I. |A \ i| <_o r$
 shows $|\bigcup i \in I. A \ i| <_o r$
 ⟨proof⟩

31.3 Binary sum

definition *csum* (**infixr** $\langle +_c \rangle$ 65)
 where $r1 +_c r2 \equiv |\text{Field } r1 \langle + \rangle \text{Field } r2|$

lemma *Field-csum*: $\text{Field } (r +_c s) = \text{Inl} \text{ ` } \text{Field } r \cup \text{Inr} \text{ ` } \text{Field } s$
 ⟨proof⟩

lemma *Card-order-csum*: *Card-order* $(r1 +_c r2)$
 ⟨proof⟩

lemma *csum-Cnotzero1*: *Cnotzero* $r1 \implies \text{Cnotzero } (r1 +_c r2)$
 ⟨proof⟩

lemma *card-order-csum*:
 assumes *card-order* $r1$ *card-order* $r2$
 shows *card-order* $(r1 +_c r2)$
 ⟨proof⟩

lemma *cinfinite-csum*:
 $\text{cinfinite } r1 \vee \text{cinfinite } r2 \implies \text{cinfinite } (r1 +_c r2)$
 ⟨proof⟩

lemma *Cinfinite-csum*:
 $\text{Cinfinite } r1 \vee \text{Cinfinite } r2 \implies \text{Cinfinite } (r1 +_c r2)$
 ⟨proof⟩

lemma *Cinfinite-csum1*: *Cinfinite* $r1 \implies \text{Cinfinite } (r1 +_c r2)$
 ⟨proof⟩

lemma *Cinfinite-csum-weak*:
 $\llbracket \text{Cinfinite } r1; \text{Cinfinite } r2 \rrbracket \implies \text{Cinfinite } (r1 +_c r2)$
 ⟨proof⟩

lemma *csum-cong*: $\llbracket p1 =_o r1; p2 =_o r2 \rrbracket \implies p1 +_c p2 =_o r1 +_c r2$

$\langle proof \rangle$

lemma *csum-cong1*: $p1 =_o r1 \implies p1 +_c q =_o r1 +_c q$
 $\langle proof \rangle$

lemma *csum-cong2*: $p2 =_o r2 \implies q +_c p2 =_o q +_c r2$
 $\langle proof \rangle$

lemma *csum-mono*: $\llbracket p1 \leq_o r1; p2 \leq_o r2 \rrbracket \implies p1 +_c p2 \leq_o r1 +_c r2$
 $\langle proof \rangle$

lemma *csum-mono1*: $p1 \leq_o r1 \implies p1 +_c q \leq_o r1 +_c q$
 $\langle proof \rangle$

lemma *csum-mono2*: $p2 \leq_o r2 \implies q +_c p2 \leq_o q +_c r2$
 $\langle proof \rangle$

lemma *ordLeq-csum1*: *Card-order* $p1 \implies p1 \leq_o p1 +_c p2$
 $\langle proof \rangle$

lemma *ordLeq-csum2*: *Card-order* $p2 \implies p2 \leq_o p1 +_c p2$
 $\langle proof \rangle$

lemma *csum-com*: $p1 +_c p2 =_o p2 +_c p1$
 $\langle proof \rangle$

lemma *csum-assoc*: $(p1 +_c p2) +_c p3 =_o p1 +_c p2 +_c p3$
 $\langle proof \rangle$

lemma *Cfinite-csum*: $\llbracket Cfinite\ r; Cfinite\ s \rrbracket \implies Cfinite\ (r +_c s)$
 $\langle proof \rangle$

lemma *csum-csum*: $(r1 +_c r2) +_c (r3 +_c r4) =_o (r1 +_c r3) +_c (r2 +_c r4)$
 $\langle proof \rangle$

lemma *Plus-csum*: $|A <+> B| =_o |A| +_c |B|$
 $\langle proof \rangle$

lemma *Un-csum*: $|A \cup B| \leq_o |A| +_c |B|$
 $\langle proof \rangle$

31.4 One

definition *cone* **where**
cone = *card-of* $\{()\}$

lemma *Card-order-cone*: *Card-order* *cone*
 $\langle proof \rangle$

lemma *Cfinite-cone*: *Cfinite cone*
 ⟨*proof*⟩

lemma *cone-not-czero*: $\neg (cone =_o czero)$
 ⟨*proof*⟩

lemma *cone-ordLeq-Cnotzero*: $Cnotzero\ r \implies cone \leq_o r$
 ⟨*proof*⟩

31.5 Two

definition *ctwo* **where**
 $ctwo = |UNIV :: bool\ set|$

lemma *Card-order-ctwo*: *Card-order ctwo*
 ⟨*proof*⟩

lemma *ctwo-not-czero*: $\neg (ctwo =_o czero)$
 ⟨*proof*⟩

lemma *ctwo-Cnotzero*: *Cnotzero ctwo*
 ⟨*proof*⟩

31.6 Family sum

definition *Csum* **where**
 $Csum\ r\ rs \equiv |SIGMA\ i : Field\ r.\ Field\ (rs\ i)|$

syntax *-Csum* ::
 $pttrn \Rightarrow ('a * 'a)\ set \Rightarrow 'b * 'b\ set \Rightarrow (('a * 'b) * ('a * 'b))\ set$
 (⟨⟨*indent*=3 *notation*=⟨*binder CSUM*⟩⟩*CSUM* *:-.* *-*)⟩ [0, 51, 10] 10)

syntax-consts
 $-Csum == Csum$

translations
 $CSUM\ i:r.\ rs == CONST\ Csum\ r\ (\%i.\ rs)$

lemma *SIGMA-CSUM*: $|SIGMA\ i : I.\ As\ i| = (CSUM\ i : |I|. |As\ i|)$
 ⟨*proof*⟩

31.7 Product

definition *cprod* (**infixr** $\langle *c \rangle\ 80$) **where**
 $r1 *c r2 = |Field\ r1 \times Field\ r2|$

lemma *card-order-cprod*:
assumes *card-order r1 card-order r2*
shows *card-order (r1 *c r2)*

$\langle \text{proof} \rangle$

lemma *Card-order-cprod*: *Card-order* ($r1 *c r2$)
 $\langle \text{proof} \rangle$

lemma *cprod-mono1*: $p1 \leq_o r1 \implies p1 *c q \leq_o r1 *c q$
 $\langle \text{proof} \rangle$

lemma *cprod-mono2*: $p2 \leq_o r2 \implies q *c p2 \leq_o q *c r2$
 $\langle \text{proof} \rangle$

lemma *cprod-mono*: $\llbracket p1 \leq_o r1; p2 \leq_o r2 \rrbracket \implies p1 *c p2 \leq_o r1 *c r2$
 $\langle \text{proof} \rangle$

lemma *ordLeq-cprod2*: $\llbracket \text{Cnotzero } p1; \text{Card-order } p2 \rrbracket \implies p2 \leq_o p1 *c p2$
 $\langle \text{proof} \rangle$

lemma *cinfinite-cprod*: $\llbracket \text{cinfinite } r1; \text{cinfinite } r2 \rrbracket \implies \text{cinfinite } (r1 *c r2)$
 $\langle \text{proof} \rangle$

lemma *cinfinite-cprod2*: $\llbracket \text{Cnotzero } r1; \text{Cinfinite } r2 \rrbracket \implies \text{cinfinite } (r1 *c r2)$
 $\langle \text{proof} \rangle$

lemma *Cinfinite-cprod2*: $\llbracket \text{Cnotzero } r1; \text{Cinfinite } r2 \rrbracket \implies \text{Cinfinite } (r1 *c r2)$
 $\langle \text{proof} \rangle$

lemma *cprod-cong*: $\llbracket p1 =_o r1; p2 =_o r2 \rrbracket \implies p1 *c p2 =_o r1 *c r2$
 $\langle \text{proof} \rangle$

lemma *cprod-cong1*: $\llbracket p1 =_o r1 \rrbracket \implies p1 *c p2 =_o r1 *c p2$
 $\langle \text{proof} \rangle$

lemma *cprod-cong2*: $p2 =_o r2 \implies q *c p2 =_o q *c r2$
 $\langle \text{proof} \rangle$

lemma *cprod-com*: $p1 *c p2 =_o p2 *c p1$
 $\langle \text{proof} \rangle$

lemma *card-of-Csum-Times*:
 $\forall i \in I. |A \ i| \leq_o |B| \implies (\text{CSUM } i : |I|. |A \ i|) \leq_o |I| *c |B|$
 $\langle \text{proof} \rangle$

lemma *card-of-Csum-Times'*:
assumes *Card-order* $r \ \forall i \in I. |A \ i| \leq_o r$
shows $(\text{CSUM } i : |I|. |A \ i|) \leq_o |I| *c r$
 $\langle \text{proof} \rangle$

lemma *cprod-csum-distrib1*: $r1 *c r2 +_c r1 *c r3 =_o r1 *c (r2 +_c r3)$
 $\langle \text{proof} \rangle$

lemma *csum-absorb2'*: $\llbracket \text{Card-order } r2; r1 \leq_o r2; \text{cinfinitive } r1 \vee \text{cinfinitive } r2 \rrbracket \implies$
 $r1 +_c r2 =_o r2$
 $\langle \text{proof} \rangle$

lemma *csum-absorb1'*:
assumes *card*: *Card-order* *r2*
and *r12*: $r1 \leq_o r2$ **and** *cr12*: $\text{cinfinitive } r1 \vee \text{cinfinitive } r2$
shows $r2 +_c r1 =_o r2$
 $\langle \text{proof} \rangle$

lemma *csum-absorb1*: $\llbracket \text{Cinfinitive } r2; r1 \leq_o r2 \rrbracket \implies r2 +_c r1 =_o r2$
 $\langle \text{proof} \rangle$

lemma *csum-absorb2*: $\llbracket \text{Cinfinitive } r2 ; r1 \leq_o r2 \rrbracket \implies r1 +_c r2 =_o r2$
 $\langle \text{proof} \rangle$

lemma *regularCard-csum*:
assumes *Cinfinitive* *r* *Cinfinitive* *s* *regularCard* *r* *regularCard* *s*
shows *regularCard* $(r +_c s)$
 $\langle \text{proof} \rangle$

lemma *csum-mono-strict*:
assumes *Card-order*: *Card-order* *r* *Card-order* *q*
and *Cinfinitive*: *Cinfinitive* *r'* *Cinfinitive* *q'*
and *less*: $r <_o r' \wedge q <_o q'$
shows $r +_c q <_o r' +_c q'$
 $\langle \text{proof} \rangle$

31.8 Exponentiation

definition *cexp* (**infixr** $\hat{^}_c$ 90) **where**
 $r1 \hat{^}_c r2 \equiv |\text{Func } (\text{Field } r2) (\text{Field } r1)|$

lemma *Card-order-cexp*: *Card-order* $(r1 \hat{^}_c r2)$
 $\langle \text{proof} \rangle$

lemma *cexp-mono'*:
assumes 1: $p1 \leq_o r1$ **and** 2: $p2 \leq_o r2$
and *n*: $\text{Field } p2 = \{\} \implies \text{Field } r2 = \{\}$
shows $p1 \hat{^}_c p2 \leq_o r1 \hat{^}_c r2$
 $\langle \text{proof} \rangle$

lemma *cexp-mono*:
assumes 1: $p1 \leq_o r1$ **and** 2: $p2 \leq_o r2$
and *n*: $p2 =_o \text{czero} \implies r2 =_o \text{czero}$ **and** *card*: *Card-order* *p2*
shows $p1 \hat{^}_c p2 \leq_o r1 \hat{^}_c r2$
 $\langle \text{proof} \rangle$

lemma *cexp-mono1*:

assumes 1: $p1 \leq_o r1$ **and** q : *Card-order* q

shows $p1 \hat{^c} q \leq_o r1 \hat{^c} q$

<proof>

lemma *cexp-mono2'*:

assumes 2: $p2 \leq_o r2$ **and** q : *Card-order* q

and n : *Field* $p2 = \{\}$ \implies *Field* $r2 = \{\}$

shows $q \hat{^c} p2 \leq_o q \hat{^c} r2$

<proof>

lemma *cexp-mono2*:

assumes 2: $p2 \leq_o r2$ **and** q : *Card-order* q

and n : $p2 =_o \text{czero} \implies r2 =_o \text{czero}$ **and** card : *Card-order* $p2$

shows $q \hat{^c} p2 \leq_o q \hat{^c} r2$

<proof>

lemma *cexp-mono2-Cnotzero*:

assumes $p2 \leq_o r2$ *Card-order* q *Cnotzero* $p2$

shows $q \hat{^c} p2 \leq_o q \hat{^c} r2$

<proof>

lemma *cexp-cong*:

assumes 1: $p1 =_o r1$ **and** 2: $p2 =_o r2$

and Cr : *Card-order* $r2$

and Cp : *Card-order* $p2$

shows $p1 \hat{^c} p2 =_o r1 \hat{^c} r2$

<proof>

lemma *cexp-cong1*:

assumes 1: $p1 =_o r1$ **and** q : *Card-order* q

shows $p1 \hat{^c} q =_o r1 \hat{^c} q$

<proof>

lemma *cexp-cong2*:

assumes 2: $p2 =_o r2$ **and** q : *Card-order* q **and** p : *Card-order* $p2$

shows $q \hat{^c} p2 =_o q \hat{^c} r2$

<proof>

lemma *cexp-cone*:

assumes *Card-order* r

shows $r \hat{^c} \text{cone} =_o r$

<proof>

lemma *cexp-cprod*:

assumes $r1$: *Card-order* $r1$

shows $(r1 \hat{^c} r2) \hat{^c} r3 =_o r1 \hat{^c} (r2 *_c r3)$ (**is** ? $L =_o$? R)

<proof>

lemma *cprod-infinite1*': $\llbracket \text{Cinfinite } r; \text{Cnotzero } p; p \leq_o r \rrbracket \implies r *c p =_o r$
 $\langle \text{proof} \rangle$

lemma *cprod-infinite*: $\text{Cinfinite } r \implies r *c r =_o r$
 $\langle \text{proof} \rangle$

lemma *cexp-cprod-ordLeq*:
assumes $r1$: *Card-order* $r1$ **and** $r2$: *Cinfinite* $r2$
and $r3$: *Cnotzero* $r3$ $r3 \leq_o r2$
shows $(r1 \hat{c} r2) \hat{c} r3 =_o r1 \hat{c} r2$ (**is** ? $L =_o$? R)
 $\langle \text{proof} \rangle$

lemma *Cnotzero-UNIV*: *Cnotzero* $|UNIV|$
 $\langle \text{proof} \rangle$

lemma *ordLess-ctwo-cexp*:
assumes *Card-order* r
shows $r <_o \text{ctwo} \hat{c} r$
 $\langle \text{proof} \rangle$

lemma *ordLeq-cexp1*:
assumes *Cnotzero* r *Card-order* q
shows $q \leq_o q \hat{c} r$
 $\langle \text{proof} \rangle$

lemma *ordLeq-cexp2*:
assumes $\text{ctwo} \leq_o q$ *Card-order* r
shows $r \leq_o q \hat{c} r$
 $\langle \text{proof} \rangle$

lemma *cinfinite-cexp*: $\llbracket \text{ctwo} \leq_o q; \text{Cinfinite } r \rrbracket \implies \text{cinfinite } (q \hat{c} r)$
 $\langle \text{proof} \rangle$

lemma *Cinfinite-cexp*:
 $\llbracket \text{ctwo} \leq_o q; \text{Cinfinite } r \rrbracket \implies \text{Cinfinite } (q \hat{c} r)$
 $\langle \text{proof} \rangle$

lemma *card-order-cexp*:
assumes *card-order* $r1$ *card-order* $r2$
shows *card-order* $(r1 \hat{c} r2)$
 $\langle \text{proof} \rangle$

lemma *ctwo-ordLess-natLeq*: $\text{ctwo} <_o \text{natLeq}$
 $\langle \text{proof} \rangle$

lemma *ctwo-ordLess-Cinfinite*: $\text{Cinfinite } r \implies \text{ctwo} <_o r$
 $\langle \text{proof} \rangle$

lemma *ctwo-ordLeq-Cinfinite*:

assumes $Cinfinite\ r$
shows $ctwo \leq_o r$
 $\langle proof \rangle$

lemma *Un-Cinfinite-bound*: $\llbracket |A| \leq_o r; |B| \leq_o r; Cinfinite\ r \rrbracket \implies |A \cup B| \leq_o r$
 $\langle proof \rangle$

lemma *Un-Cinfinite-bound-strict*: $\llbracket |A| <_o r; |B| <_o r; Cinfinite\ r \rrbracket \implies |A \cup B| <_o r$
 $\langle proof \rangle$

lemma *UNION-Cinfinite-bound*: $\llbracket |I| \leq_o r; \forall i \in I. |A\ i| \leq_o r; Cinfinite\ r \rrbracket \implies |\bigcup i \in I. A\ i| \leq_o r$
 $\langle proof \rangle$

lemma *csum-cinfinite-bound*:
assumes $p \leq_o r\ q \leq_o r\ Card\text{-}order\ p\ Card\text{-}order\ q\ Cinfinite\ r$
shows $p +_c q \leq_o r$
 $\langle proof \rangle$

lemma *cprod-cinfinite-bound*:
assumes $p \leq_o r\ q \leq_o r\ Card\text{-}order\ p\ Card\text{-}order\ q\ Cinfinite\ r$
shows $p *_c q \leq_o r$
 $\langle proof \rangle$

lemma *cprod-infinite2'*: $\llbracket Cnotzero\ r1; Cinfinite\ r2; r1 \leq_o r2 \rrbracket \implies r1 *_c r2 =_o r2$
 $\langle proof \rangle$

lemma *regularCard-cprod*:
assumes $Cinfinite\ r\ Cinfinite\ s\ regularCard\ r\ regularCard\ s$
shows $regularCard\ (r *_c s)$
 $\langle proof \rangle$

lemma *cprod-csum-cexp*:
 $r1 *_c r2 \leq_o (r1 +_c r2) \wedge_c ctwo$
 $\langle proof \rangle$

lemma *Cfinite-cprod-Cinfinite*: $\llbracket Cfinite\ r; Cinfinite\ s \rrbracket \implies r *_c s \leq_o s$
 $\langle proof \rangle$

lemma *cprod-cexp*: $(r *_c s) \wedge_c t =_o r \wedge_c t *_c s \wedge_c t$
 $\langle proof \rangle$

lemma *cprod-cexp-csum-cexp-Cinfinite*:
assumes $t: Cinfinite\ t$
shows $(r *_c s) \wedge_c t \leq_o (r +_c s) \wedge_c t$
 $\langle proof \rangle$

lemma *Cfinite-cexp-Cinfinite:*

assumes *s: Cfinite s and t: Cinfinite t*

shows $s \hat{c} t \leq_o \text{ctwo} \hat{c} t$

<proof>

lemma *csum-Cfinite-cexp-Cinfinite:*

assumes *r: Card-order r and s: Cfinite s and t: Cinfinite t*

shows $(r +_c s) \hat{c} t \leq_o (r +_c \text{ctwo}) \hat{c} t$

<proof>

lemma *Cinfinite-cardSuc: Cinfinite r \implies Cinfinite (cardSuc r)*

<proof>

lemma *cardSuc-UNION-Cinfinite:*

assumes *Cinfinite r relChain (cardSuc r) As B $\leq (\bigcup i \in \text{Field (cardSuc r)}. As$*

i) |B| \leq_o r

shows $\exists i \in \text{Field (cardSuc r)}. B \leq As i$

<proof>

lemma *Cinfinite-card-suc: $\llbracket \text{Cinfinite } r ; \text{card-order } r \rrbracket \implies \text{Cinfinite (card-suc } r)$*

<proof>

lemma *card-suc-least: $\llbracket \text{card-order } r ; \text{Card-order } s ; r <_o s \rrbracket \implies \text{card-suc } r \leq_o s$*

<proof>

lemma *regularCard-cardSuc: Cinfinite k \implies regularCard (cardSuc k)*

<proof>

lemma *regularCard-card-suc: card-order r \implies Cinfinite r \implies regularCard (card-suc r)*

<proof>

end

32 Function Definition Base

theory *Fun-Def-Base*

imports *Ctr-Sugar Set Wellfounded*

begin

<ML>

named-theorems *termination-simp simplification rules for termination proofs*

<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*: $x \in \text{Collect } (\text{case-prod } A) \implies A \text{ (fst } x) \text{ (snd } x)$
 $\langle \text{proof} \rangle$

inductive

$\text{rel-sum} :: ('a \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'd \Rightarrow \text{bool}) \Rightarrow 'a + 'b \Rightarrow 'c + 'd \Rightarrow \text{bool}$

for *R1 R2*

where

$R1 \ a \ c \implies \text{rel-sum } R1 \ R2 \ (\text{Inl } a) \ (\text{Inl } c)$

$| \ R2 \ b \ d \implies \text{rel-sum } R1 \ R2 \ (\text{Inr } b) \ (\text{Inr } d)$

definition

$\text{rel-fun} :: ('a \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'd \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'd) \Rightarrow \text{bool}$

where

$\text{rel-fun } A \ B = (\lambda f \ g. \forall x \ y. A \ x \ y \longrightarrow B \ (f \ x) \ (g \ y))$

lemma *rel-funI [intro]*:

assumes $\bigwedge x \ y. A \ x \ y \implies B \ (f \ x) \ (g \ y)$

shows $\text{rel-fun } A \ B \ f \ g$

$\langle \text{proof} \rangle$

lemma *rel-funD*:

assumes $\text{rel-fun } A \ B \ f \ g$ **and** $A \ x \ y$

shows $B \ (f \ x) \ (g \ y)$

$\langle \text{proof} \rangle$

lemma *rel-fun-mono*:

$\llbracket \text{rel-fun } X \ A \ f \ g; \bigwedge x \ y. Y \ x \ y \longrightarrow X \ x \ y; \bigwedge x \ y. A \ x \ y \implies B \ x \ y \rrbracket \implies \text{rel-fun } Y \ B \ f \ g$

$\langle \text{proof} \rangle$

lemma *rel-fun-mono' [mono]*:

$\llbracket \bigwedge x \ y. Y \ x \ y \longrightarrow X \ x \ y; \bigwedge x \ y. A \ x \ y \longrightarrow B \ x \ y \rrbracket \implies \text{rel-fun } X \ A \ f \ g \longrightarrow \text{rel-fun } Y \ B \ f \ g$

$\langle \text{proof} \rangle$

definition *rel-set* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow \text{bool}$

where $\text{rel-set } R = (\lambda A \ B. (\forall x \in A. \exists y \in B. R \ x \ y) \wedge (\forall y \in B. \exists x \in A. R \ x \ y))$

lemma *rel-setI*:

assumes $\bigwedge x. x \in A \implies \exists y \in B. R \ x \ y$

assumes $\bigwedge y. y \in B \implies \exists x \in A. R\ x\ y$
shows *rel-set* $R\ A\ B$
 $\langle proof \rangle$

lemma *predicate2-transferD*:
 $\llbracket rel\text{-}fun\ R1\ (rel\text{-}fun\ R2\ (=))\ P\ Q; a \in A; b \in B; A \subseteq \{(x, y). R1\ x\ y\}; B \subseteq \{(x, y). R2\ x\ y\} \rrbracket \implies$
 $P\ (fst\ a)\ (fst\ b) \longleftrightarrow Q\ (snd\ a)\ (snd\ b)$
 $\langle proof \rangle$

definition *collect where*
 $collect\ F\ x = (\bigcup f \in F. f\ x)$

lemma *fstI*: $x = (y, z) \implies fst\ x = y$
 $\langle proof \rangle$

lemma *sndI*: $x = (y, z) \implies snd\ x = z$
 $\langle proof \rangle$

lemma *bijI*: $\llbracket \bigwedge x\ y. (f\ x = f\ y) = (x = y); \bigwedge y. \exists x. y = f\ x \rrbracket \implies bij\ f$
 $\langle proof \rangle$

definition $Gr\ A\ f = \{(a, f\ a) \mid a. a \in A\}$

definition $Grp\ A\ f = (\lambda a\ b. b = f\ a \wedge a \in A)$

definition *vimage2p where*
 $vimage2p\ f\ g\ R = (\lambda x\ y. R\ (f\ x)\ (g\ y))$

lemma *collect-comp*: $collect\ F \circ g = collect\ ((\lambda f. f \circ g)\ 'F)$
 $\langle proof \rangle$

definition *convol* ($\langle (\langle indent=1\ notation=\langle mixfix\ convol \rangle \langle -, / - \rangle) \rangle$) **where**
 $\langle f, g \rangle \equiv \lambda a. (f\ a, g\ a)$

lemma *fst-convol*: $fst \circ \langle f, g \rangle = f$
 $\langle proof \rangle$

lemma *snd-convol*: $snd \circ \langle f, g \rangle = g$
 $\langle proof \rangle$

lemma *convol-mem-GrpI*:
 $x \in A \implies \langle id, g \rangle x \in (Collect\ (case\text{-}prod\ (Grp\ A\ g)))$
 $\langle proof \rangle$

definition *csquare where*
 $csquare\ A\ f1\ f2\ p1\ p2 \longleftrightarrow (\forall\ a \in A. f1\ (p1\ a) = f2\ (p2\ a))$

lemma *eq-alt*: $(=) = \text{Grp UNIV id}$
 $\langle \text{proof} \rangle$

lemma *leq-conversepI*: $R = (=) \implies R \leq R^{-1-1}$
 $\langle \text{proof} \rangle$

lemma *leq-OOI*: $R = (=) \implies R \leq R \text{ OO } R$
 $\langle \text{proof} \rangle$

lemma *OO-Grp-alt*: $(\text{Grp } A \text{ } f)^{-1-1} \text{ OO } \text{Grp } A \text{ } g = (\lambda x \text{ } y. \exists z. z \in A \wedge f \text{ } z = x \wedge g \text{ } z = y)$
 $\langle \text{proof} \rangle$

lemma *Grp-UNIV-id*: $f = \text{id} \implies (\text{Grp UNIV } f)^{-1-1} \text{ OO } \text{Grp UNIV } f = \text{Grp UNIV } f$
 $\langle \text{proof} \rangle$

lemma *Grp-UNIV-idI*: $x = y \implies \text{Grp UNIV id } x \text{ } y$
 $\langle \text{proof} \rangle$

lemma *Grp-mono*: $A \leq B \implies \text{Grp } A \text{ } f \leq \text{Grp } B \text{ } f$
 $\langle \text{proof} \rangle$

lemma *GrpI*: $\llbracket f \text{ } x = y; x \in A \rrbracket \implies \text{Grp } A \text{ } f \text{ } x \text{ } y$
 $\langle \text{proof} \rangle$

lemma *GrpE*: $\text{Grp } A \text{ } f \text{ } x \text{ } y \implies (\llbracket f \text{ } x = y; x \in A \rrbracket \implies R) \implies R$
 $\langle \text{proof} \rangle$

lemma *Collect-case-prod-Grp-eqD*: $z \in \text{Collect } (\text{case-prod } (\text{Grp } A \text{ } f)) \implies (f \circ \text{fst}) z = \text{snd } z$
 $\langle \text{proof} \rangle$

lemma *Collect-case-prod-Grp-in*: $z \in \text{Collect } (\text{case-prod } (\text{Grp } A \text{ } f)) \implies \text{fst } z \in A$
 $\langle \text{proof} \rangle$

definition *pick-middlep* $P \text{ } Q \text{ } a \text{ } c = (\text{SOME } b. P \text{ } a \text{ } b \wedge Q \text{ } b \text{ } c)$

lemma *pick-middlep*:
 $(P \text{ OO } Q) \text{ } a \text{ } c \implies P \text{ } a \text{ } (\text{pick-middlep } P \text{ } Q \text{ } a \text{ } c) \wedge Q \text{ } (\text{pick-middlep } P \text{ } Q \text{ } a \text{ } c) \text{ } c$
 $\langle \text{proof} \rangle$

definition *fstOp* **where**
 $\text{fstOp } P \text{ } Q \text{ } ac = (\text{fst } ac, \text{pick-middlep } P \text{ } Q \text{ } (\text{fst } ac) \text{ } (\text{snd } ac))$

definition *sndOp* **where**
 $\text{sndOp } P \text{ } Q \text{ } ac = (\text{pick-middlep } P \text{ } Q \text{ } (\text{fst } ac) \text{ } (\text{snd } ac), (\text{snd } ac))$

lemma *fstOp-in*: $ac \in \text{Collect } (\text{case-prod } (P \text{ OO } Q)) \implies \text{fstOp } P \text{ } Q \text{ } ac \in \text{Collect}$

(*case-prod P*)
 ⟨*proof*⟩

lemma *fst-fstOp*: $\text{fst } bc = (\text{fst} \circ \text{fstOp } P \ Q) \ bc$
 ⟨*proof*⟩

lemma *snd-sndOp*: $\text{snd } bc = (\text{snd} \circ \text{sndOp } P \ Q) \ bc$
 ⟨*proof*⟩

lemma *sndOp-in*: $ac \in \text{Collect } (\text{case-prod } (P \ OO \ Q)) \implies \text{sndOp } P \ Q \ ac \in \text{Collect } (\text{case-prod } Q)$
 ⟨*proof*⟩

lemma *csquare-fstOp-sndOp*:
 $\text{csquare } (\text{Collect } (f \ (P \ OO \ Q))) \ \text{snd } \text{fst } (\text{fstOp } P \ Q) \ (\text{sndOp } P \ Q)$
 ⟨*proof*⟩

lemma *snd-fst-flip*: $\text{snd } xy = (\text{fst} \circ (\% (x, y). (y, x))) \ xy$
 ⟨*proof*⟩

lemma *fst-snd-flip*: $\text{fst } xy = (\text{snd} \circ (\% (x, y). (y, x))) \ xy$
 ⟨*proof*⟩

lemma *flip-pred*: $A \subseteq \text{Collect } (\text{case-prod } (R^{-1-1})) \implies (\% (x, y). (y, x)) \ ' \ A \subseteq \text{Collect } (\text{case-prod } R)$
 ⟨*proof*⟩

lemma *predicate2-eqD*: $A = B \implies A \ a \ b \longleftrightarrow B \ a \ b$
 ⟨*proof*⟩

lemma *case-sum-o-inj*: $\text{case-sum } f \ g \circ \text{Inl} = f \ \text{case-sum } f \ g \circ \text{Inr} = g$
 ⟨*proof*⟩

lemma *map-sum-o-inj*: $\text{map-sum } f \ g \circ \text{Inl} = \text{Inl} \circ f \ \text{map-sum } f \ g \circ \text{Inr} = \text{Inr} \circ g$
 ⟨*proof*⟩

lemma *card-order-csum-cone-cexp-def*:
 $\text{card-order } r \implies (|A1| + c \ \text{cone}) \ \hat{\ }_c \ r = |\text{Func } \text{UNIV } (\text{Inl} \ ' \ A1 \cup \{\text{Inr } ()\})|$
 ⟨*proof*⟩

lemma *If-the-inv-into-in-Func*:
 $\llbracket \text{inj-on } g \ C; \ C \subseteq B \cup \{x\} \rrbracket \implies$
 $(\lambda i. \text{if } i \in g \ ' \ C \text{ then the-inv-into } C \ g \ i \text{ else } x) \in \text{Func } \text{UNIV } (B \cup \{x\})$
 ⟨*proof*⟩

lemma *If-the-inv-into-f-f*:
 $\llbracket i \in C; \ \text{inj-on } g \ C \rrbracket \implies ((\lambda i. \text{if } i \in g \ ' \ C \text{ then the-inv-into } C \ g \ i \text{ else } x) \circ g) \ i = \text{id } i$
 ⟨*proof*⟩

lemma *the-inv-f-o-f-id*: $\text{inj } f \implies (\text{the-inv } f \circ f) \, z = \text{id } z$
 $\langle \text{proof} \rangle$

lemma *vimage2pI*: $R \, (f \, x) \, (g \, y) \implies \text{vimage2p } f \, g \, R \, x \, y$
 $\langle \text{proof} \rangle$

lemma *rel-fun-iff-leq-vimage2p*: $(\text{rel-fun } R \, S) \, f \, g = (R \leq \text{vimage2p } f \, g \, S)$
 $\langle \text{proof} \rangle$

lemma *convol-image-vimage2p*: $\langle f \circ \text{fst}, g \circ \text{snd} \rangle \, \text{Collect } (\text{case-prod } (\text{vimage2p } f \, g \, R)) \subseteq \text{Collect } (\text{case-prod } R)$
 $\langle \text{proof} \rangle$

lemma *vimage2p-Grp*: $\text{vimage2p } f \, g \, P = \text{Grp } \text{UNIV } f \, \text{OO } P \, \text{OO } (\text{Grp } \text{UNIV } g)^{-1-1}$
 $\langle \text{proof} \rangle$

lemma *subst-Pair*: $P \, x \, y \implies a = (x, y) \implies P \, (\text{fst } a) \, (\text{snd } a)$
 $\langle \text{proof} \rangle$

lemma *comp-apply-eq*: $f \, (g \, x) = h \, (k \, x) \implies (f \circ g) \, x = (h \circ k) \, x$
 $\langle \text{proof} \rangle$

lemma *refl-ge-eq*: $(\bigwedge x. R \, x \, x) \implies (=) \leq R$
 $\langle \text{proof} \rangle$

lemma *ge-eq-refl*: $(=) \leq R \implies R \, x \, x$
 $\langle \text{proof} \rangle$

lemma *reflp-eq*: $\text{reflp } R = ((=) \leq R)$
 $\langle \text{proof} \rangle$

lemma *transp-relcompp*: $\text{transp } r \longleftrightarrow r \, \text{OO } r \leq r$
 $\langle \text{proof} \rangle$

lemma *symp-conversep*: $\text{symp } R = (R^{-1-1} \leq R)$
 $\langle \text{proof} \rangle$

lemma *diag-imp-eq-le*: $(\bigwedge x. x \in A \implies R \, x \, x) \implies \forall x \, y. x \in A \longrightarrow y \in A \longrightarrow x = y \longrightarrow R \, x \, y$
 $\langle \text{proof} \rangle$

definition *eq-onp* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$
where $\text{eq-onp } R = (\lambda x \, y. R \, x \wedge x = y)$

lemma *eq-onp-Grp*: $\text{eq-onp } P = \text{BNF-Def.Grp } (\text{Collect } P) \, \text{id}$
 $\langle \text{proof} \rangle$

lemma *eq-onp-to-eq*: $eq\text{-onp } P \ x \ y \implies x = y$
 $\langle proof \rangle$

lemma *eq-onp-top-eq-eq*: $eq\text{-onp } top = (=)$
 $\langle proof \rangle$

lemma *eq-onp-same-args*: $eq\text{-onp } P \ x \ x = P \ x$
 $\langle proof \rangle$

lemma *eq-onp-eqD*: $eq\text{-onp } P = Q \implies P \ x = Q \ x \ x$
 $\langle proof \rangle$

lemma *Ball-Collect*: $Ball \ A \ P = (A \subseteq (Collect \ P))$
 $\langle proof \rangle$

lemma *eq-onp-mono0*: $\forall x \in A. P \ x \longrightarrow Q \ x \implies \forall x \in A. \forall y \in A. eq\text{-onp } P \ x \ y \longrightarrow eq\text{-onp } Q \ x \ y$
 $\langle proof \rangle$

lemma *eq-onp-True*: $eq\text{-onp } (\lambda\cdot. True) = (=)$
 $\langle proof \rangle$

lemma *Ball-image-comp*: $Ball \ (f \text{ ‘ } A) \ g = Ball \ A \ (g \circ f)$
 $\langle proof \rangle$

lemma *rel-fun-Collect-case-prodD*:
 $rel\text{-fun } A \ B \ f \ g \implies X \subseteq Collect \ (case\text{-prod } A) \implies x \in X \implies B \ ((f \circ fst) \ x) ((g \circ snd) \ x)$
 $\langle proof \rangle$

lemma *eq-onp-mono-iff*: $eq\text{-onp } P \leq eq\text{-onp } Q \longleftrightarrow P \leq Q$
 $\langle proof \rangle$

$\langle ML \rangle$

end

34 Composition of Bounded Natural Functors

theory *BNF-Composition*

imports *BNF-Def*

begin

lemma *ssubst-mem*: $\llbracket t = s; s \in X \rrbracket \implies t \in X$
 $\langle proof \rangle$

lemma *empty-natural*: $(\lambda\cdot. \{\}) \circ f = image \ g \circ (\lambda\cdot. \{\})$
 $\langle proof \rangle$

lemma *Cinfinite-gt-empty*: $\text{Cinfinite } r \implies |\{\}| <_o r$
 ⟨proof⟩

lemma *Union-natural*: $\text{Union} \circ \text{image } f = \text{image } f \circ \text{Union}$
 ⟨proof⟩

lemma *in-Union-o-assoc*: $x \in (\text{Union} \circ \text{gset} \circ \text{gmap}) A \implies x \in (\text{Union} \circ (\text{gset} \circ \text{gmap})) A$
 ⟨proof⟩

lemma *regularCard-UNION-bound*:
 assumes *Cinfinite* r *regularCard* r and $|I| <_o r \wedge i. i \in I \implies |A \ i| <_o r$
 shows $|\bigcup_{i \in I}. A \ i| <_o r$
 ⟨proof⟩

lemma *comp-single-set-bd-strict*:
 assumes *fbd*: *Cinfinite* *fbd* *regularCard* *fbd* and
 gbd: *Cinfinite* *gbd* *regularCard* *gbd* and
 fset-bd: $\bigwedge x. |\text{fset } x| <_o \text{fbd}$ and
 gset-bd: $\bigwedge x. |\text{gset } x| <_o \text{gbd}$
 shows $|\bigcup (\text{fset} \text{ ‘ } \text{gset } x)| <_o \text{gbd} * c \text{ fbd}$
 ⟨proof⟩

lemma *comp-single-set-bd*:
 assumes *fbd-Card-order*: *Card-order* *fbd* and
 fset-bd: $\bigwedge x. |\text{fset } x| \leq_o \text{fbd}$ and
 gset-bd: $\bigwedge x. |\text{gset } x| \leq_o \text{gbd}$
 shows $|\bigcup (\text{fset} \text{ ‘ } \text{gset } x)| \leq_o \text{gbd} * c \text{ fbd}$
 ⟨proof⟩

lemma *csum-dup*: $\text{cinfinite } r \implies \text{Card-order } r \implies p +_c p' =_o r +_c r \implies p +_c p' =_o r$
 ⟨proof⟩

lemma *cprod-dup*: $\text{cinfinite } r \implies \text{Card-order } r \implies p *_c p' =_o r *_c r \implies p *_c p' =_o r$
 ⟨proof⟩

lemma *Union-image-insert*: $\bigcup (f \text{ ‘ } \text{insert } a \ B) = f \ a \cup \bigcup (f \text{ ‘ } B)$
 ⟨proof⟩

lemma *Union-image-empty*: $A \cup \bigcup (f \text{ ‘ } \{\}) = A$
 ⟨proof⟩

lemma *image-o-collect*: $\text{collect } ((\lambda f. \text{image } g \circ f) \text{ ‘ } F) = \text{image } g \circ \text{collect } F$
 ⟨proof⟩

lemma *conj-subset-def*: $A \subseteq \{x. P \ x \wedge Q \ x\} = (A \subseteq \{x. P \ x\} \wedge A \subseteq \{x. Q \ x\})$
 ⟨proof⟩

lemma *UN-image-subset*: $\bigcup (f \text{ ‘ } g \ x) \subseteq X = (g \ x \subseteq \{x. f \ x \subseteq X\})$
 ⟨proof⟩

lemma *comp-set-bd-Union-o-collect*: $|\bigcup (\bigcup ((\lambda f. f \ x) \text{ ‘ } X))| \leq_o \text{ hbd} \implies |(Union \circ collect \ X) \ x| \leq_o \text{ hbd}$
 ⟨proof⟩

lemma *comp-set-bd-Union-o-collect-strict*: $|\bigcup (\bigcup ((\lambda f. f \ x) \text{ ‘ } X))| <_o \text{ hbd} \implies |(Union \circ collect \ X) \ x| <_o \text{ hbd}$
 ⟨proof⟩

lemma *Collect-inj*: $Collect \ P = Collect \ Q \implies P = Q$
 ⟨proof⟩

lemma *Grp-fst-snd*: $(Grp \ (Collect \ (case\text{-}prod \ R)) \ fst)^{-1-1} \ OO \ Grp \ (Collect \ (case\text{-}prod \ R)) \ snd = R$
 ⟨proof⟩

lemma *OO-Grp-cong*: $A = B \implies (Grp \ A \ f)^{-1-1} \ OO \ Grp \ A \ g = (Grp \ B \ f)^{-1-1} \ OO \ Grp \ B \ g$
 ⟨proof⟩

lemma *vimage2p-relcompp-mono*: $R \ OO \ S \leq T \implies vimage2p \ f \ g \ R \ OO \ vimage2p \ g \ h \ S \leq vimage2p \ f \ h \ T$
 ⟨proof⟩

lemma *type-copy-map-cong0*: $M \ (g \ x) = N \ (h \ x) \implies (f \circ M \circ g) \ x = (f \circ N \circ h) \ x$
 ⟨proof⟩

lemma *type-copy-set-bd*: $(\bigwedge y. |S \ y| <_o \text{ bd}) \implies |(S \circ Rep) \ x| <_o \text{ bd}$
 ⟨proof⟩

lemma *vimage2p-cong*: $R = S \implies vimage2p \ f \ g \ R = vimage2p \ f \ g \ S$
 ⟨proof⟩

lemma *Ball-comp-iff*: $(\lambda x. Ball \ (A \ x) \ f) \circ g = (\lambda x. Ball \ ((A \circ g) \ x) \ f)$
 ⟨proof⟩

lemma *conj-comp-iff*: $(\lambda x. P \ x \wedge Q \ x) \circ g = (\lambda x. (P \circ g) \ x \wedge (Q \circ g) \ x)$
 ⟨proof⟩

context

fixes *Rep Abs*

assumes *type-copy*: *type-definition Rep Abs UNIV*

begin

lemma *type-copy-map-id0*: $M = id \implies Abs \circ M \circ Rep = id$

$\langle \text{proof} \rangle$

lemma *type-copy-map-comp0*: $M = M1 \circ M2 \implies f \circ M \circ g = (f \circ M1 \circ Rep) \circ (Abs \circ M2 \circ g)$
 $\langle \text{proof} \rangle$

lemma *type-copy-set-map0*: $S \circ M = \text{image } f \circ S' \implies (S \circ Rep) \circ (Abs \circ M \circ g) = \text{image } f \circ (S' \circ g)$
 $\langle \text{proof} \rangle$

lemma *type-copy-wit*: $x \in (S \circ Rep) (Abs \ y) \implies x \in S \ y$
 $\langle \text{proof} \rangle$

lemma *type-copy-vimage2p-Grp-Rep*: $\text{vimage2p } f \text{ Rep } (Grp \ (Collect \ P) \ h) = Grp \ (Collect \ (\lambda x. \ P \ (f \ x))) \ (Abs \circ h \circ f)$
 $\langle \text{proof} \rangle$

lemma *type-copy-vimage2p-Grp-Abs*:
 $\bigwedge h. \text{vimage2p } g \text{ Abs } (Grp \ (Collect \ P) \ h) = Grp \ (Collect \ (\lambda x. \ P \ (g \ x))) \ (Rep \circ h \circ g)$
 $\langle \text{proof} \rangle$

lemma *type-copy-ex-RepI*: $(\exists b. \ F \ b) = (\exists b. \ F \ (Rep \ b))$
 $\langle \text{proof} \rangle$

lemma *vimage2p-relcompp-converse*:
 $\text{vimage2p } f \ g \ (R^{-1-1} \circ S) = (\text{vimage2p } Rep \ f \ R)^{-1-1} \circ \text{vimage2p } Rep \ g \ S$
 $\langle \text{proof} \rangle$

end

bnf *DEADID*: $'a$
 $\text{map}: id :: 'a \Rightarrow 'a$
 $bd: natLeq$
 $rel: (=) :: 'a \Rightarrow 'a \Rightarrow bool$
 $\langle \text{proof} \rangle$

definition *id-bnf* :: $'a \Rightarrow 'a$ **where**
 $id-bnf \equiv (\lambda x. \ x)$

lemma *id-bnf-apply*: $id-bnf \ x = x$
 $\langle \text{proof} \rangle$

bnf *ID*: $'a$
 $\text{map}: id-bnf :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
 $sets: \lambda x. \ \{x\}$
 $bd: natLeq$
 $rel: id-bnf :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
 $pred: id-bnf :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$

<proof>

lemma *type-definition-id-bnf-UNIV: type-definition id-bnf id-bnf UNIV*
<proof>

<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 \Rightarrow 'a set for s :: 'a + 'b where*

s = Inl x \Rightarrow x \in setl s

inductive-set *setr :: 'a + 'b \Rightarrow 'b set for s :: 'a + 'b where*

s = Inr x \Rightarrow x \in setr s

lemma *sum-set-defs [code]:*

setl = (λx . case x of Inl z \Rightarrow {z} | - \Rightarrow {})

setr = (λx . case x of Inr z \Rightarrow {z} | - \Rightarrow {})

<proof>

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

<proof>

inductive

$pred\text{-}sum :: ('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a + 'b \Rightarrow bool$ **for** $P1\ P2$
where
 $P1\ a \Longrightarrow pred\text{-}sum\ P1\ P2\ (Inl\ a)$
 $| P2\ b \Longrightarrow pred\text{-}sum\ P1\ P2\ (Inr\ b)$

lemma $pred\text{-}sum\text{-}inject$ [*code*, *simp*]:
 $pred\text{-}sum\ P1\ P2\ (Inl\ a) \longleftrightarrow P1\ a$
 $pred\text{-}sum\ P1\ P2\ (Inr\ b) \longleftrightarrow P2\ b$
 $\langle proof \rangle$

bnf $'a + 'b$
 map : $map\text{-}sum$
 $sets$: $setl\ setr$
 bd : $natLeq$
 $wits$: $Inl\ Inr$
 rel : $rel\text{-}sum$
 $pred$: $pred\text{-}sum$
 $\langle proof \rangle$

inductive-set $fsts :: 'a \times 'b \Rightarrow 'a\ set$ **for** $p :: 'a \times 'b$ **where**
 $fst\ p \in fsts\ p$
inductive-set $snds :: 'a \times 'b \Rightarrow 'b\ set$ **for** $p :: 'a \times 'b$ **where**
 $snd\ p \in snds\ p$

lemma $prod\text{-}set\text{-}defs[code]$: $fsts = (\lambda p. \{fst\ p\})\ snds = (\lambda p. \{snd\ p\})$
 $\langle proof \rangle$

inductive
 $rel\text{-}prod :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('c \Rightarrow 'd \Rightarrow bool) \Rightarrow 'a \times 'c \Rightarrow 'b \times 'd \Rightarrow bool$
for $R1\ R2$
where
 $\llbracket R1\ a\ b; R2\ c\ d \rrbracket \Longrightarrow rel\text{-}prod\ R1\ R2\ (a, c)\ (b, d)$

inductive
 $pred\text{-}prod :: ('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a \times 'b \Rightarrow bool$ **for** $P1\ P2$
where
 $\llbracket P1\ a; P2\ b \rrbracket \Longrightarrow pred\text{-}prod\ P1\ P2\ (a, b)$

lemma $rel\text{-}prod\text{-}inject$ [*code*, *simp*]:
 $rel\text{-}prod\ R1\ R2\ (a, b)\ (c, d) \longleftrightarrow R1\ a\ c \wedge R2\ b\ d$
 $\langle proof \rangle$

lemma $pred\text{-}prod\text{-}inject$ [*code*, *simp*]:
 $pred\text{-}prod\ P1\ P2\ (a, b) \longleftrightarrow P1\ a \wedge P2\ b$
 $\langle proof \rangle$

lemma $rel\text{-}prod\text{-}conv$:
 $rel\text{-}prod\ R1\ R2 = (\lambda(a, b)\ (c, d). R1\ a\ c \wedge R2\ b\ d)$
 $\langle proof \rangle$

definition

$$\text{pred-fun} :: ('a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$$
where

$$\text{pred-fun } A \ B = (\lambda f. \forall x. A \ x \longrightarrow B \ (f \ x))$$

lemma *pred-funI*: $(\bigwedge x. A \ x \Longrightarrow B \ (f \ x)) \Longrightarrow \text{pred-fun } A \ B \ f$

<proof>

bnf $'a \times 'b$

map: *map-prod*

sets: *fsts snds*

bd: *natLeq*

rel: *rel-prod*

pred: *pred-prod*

<proof>

lemma *card-order-bd-fun*: *card-order* (*natLeq* + *c card-suc* ($|UNIV|$))

<proof>

lemma *Cinfinite-bd-fun*: *Cinfinite* (*natLeq* + *c card-suc* ($|UNIV|$))

<proof>

lemma *regularCard-bd-fun*: *regularCard* (*natLeq* + *c card-suc* ($|UNIV|$))

(**is** *regularCard* ($- + c \text{ card-suc } ?U$))

<proof>

lemma *ordLess-bd-fun*: $|UNIV::'a \text{ set}| <_o \text{ natLeq} + c \text{ card-suc } (|UNIV::'a \text{ set}|)$

(**is** $- <_o (- + c \text{ card-suc } (?U :: 'a \text{ rel}))$)

<proof>

bnf $'a \Rightarrow 'b$

map: (\circ)

sets: *range*

bd: *natLeq* + *c card-suc* ($|UNIV::'a \text{ set}|$)

rel: *rel-fun* ($=$)

pred: *pred-fun* ($\lambda-. \text{ True}$)

<proof>

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*: $(P \wedge Q \implies PROP R) \equiv (P \implies Q \implies PROP R)$
 $\langle proof \rangle$

lemma *predicate2D-conj*: $P \leq Q \wedge R \implies R \wedge (P x y \longrightarrow Q x y)$
 $\langle proof \rangle$

lemma *eq-sym-Unity-conv*: $(x = (() = ())) = x$
 $\langle proof \rangle$

lemma *case-unit-Unity*: $(case\ u\ of\ () \Rightarrow f) = f$
 $\langle proof \rangle$

lemma *case-prod-Pair-iden*: $(case\ p\ of\ (x, y) \Rightarrow (x, y)) = p$
 $\langle proof \rangle$

lemma *unit-all-impI*: $(P () \implies Q ()) \implies \forall x. P x \longrightarrow Q x$
 $\langle proof \rangle$

lemma *pointfree-idE*: $f \circ g = id \implies f (g x) = x$
 $\langle proof \rangle$

lemma *o-bij*:
 assumes *gf*: $g \circ f = id$ and *fg*: $f \circ g = id$
 shows *bij f*
 $\langle proof \rangle$

lemma *case-sum-step*:
 $case-sum\ (case-sum\ f'\ g')\ g\ (Inl\ p) = case-sum\ f'\ g'\ p$
 $case-sum\ f\ (case-sum\ f'\ g')\ (Inr\ p) = case-sum\ f'\ g'\ p$
 $\langle proof \rangle$

lemma *obj-one-pointE*: $\forall x. s = x \longrightarrow P \implies P$
 $\langle proof \rangle$

lemma *type-copy-obj-one-point-absE*:
 assumes *type-definition Rep Abs UNIV* $\forall x. s = Abs\ x \longrightarrow P$ shows *P*
 $\langle proof \rangle$

lemma *obj-sumE-f*:
 assumes $\forall x. s = f\ (Inl\ x) \longrightarrow P \ \forall x. s = f\ (Inr\ x) \longrightarrow P$
 shows $\forall x. s = f\ x \longrightarrow P$
 $\langle proof \rangle$

lemma *case-sum-if*:
 $case-sum\ f\ g\ (if\ p\ then\ Inl\ x\ else\ Inr\ y) = (if\ p\ then\ f\ x\ else\ g\ y)$
 $\langle proof \rangle$

lemma *prod-set-simps[simp]*:
 $fsts\ (x, y) = \{x\}$

$snds\ (x, y) = \{y\}$
 $\langle proof \rangle$

lemma *sum-set-simps*[simp]:

$setl\ (Inl\ x) = \{x\}$
 $setl\ (Inr\ x) = \{\}$
 $setr\ (Inl\ x) = \{\}$
 $setr\ (Inr\ x) = \{x\}$
 $\langle proof \rangle$

lemma *Inl-Inr-False*: $(Inl\ x = Inr\ y) = False$
 $\langle proof \rangle$

lemma *Inr-Inl-False*: $(Inr\ x = Inl\ y) = False$
 $\langle proof \rangle$

lemma *spec2*: $\forall x\ y. P\ x\ y \implies P\ x\ y$
 $\langle proof \rangle$

lemma *rewriteR-comp-comp*: $\llbracket g \circ h = r \rrbracket \implies f \circ g \circ h = f \circ r$
 $\langle proof \rangle$

lemma *rewriteR-comp-comp2*: $\llbracket g \circ h = r1 \circ r2; f \circ r1 = l \rrbracket \implies f \circ g \circ h = l \circ r2$
 $\langle proof \rangle$

lemma *rewriteL-comp-comp*: $\llbracket f \circ g = l \rrbracket \implies f \circ (g \circ h) = l \circ h$
 $\langle proof \rangle$

lemma *rewriteL-comp-comp2*: $\llbracket f \circ g = l1 \circ l2; l2 \circ h = r \rrbracket \implies f \circ (g \circ h) = l1 \circ r$
 $\langle proof \rangle$

lemma *convol-o*: $\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle$
 $\langle proof \rangle$

lemma *map-prod-o-convol*: $map\text{-}prod\ h1\ h2 \circ \langle f, g \rangle = \langle h1 \circ f, h2 \circ g \rangle$
 $\langle proof \rangle$

lemma *map-prod-o-convol-id*: $(map\text{-}prod\ f\ id \circ \langle id, g \rangle)\ x = \langle id \circ f, g \rangle\ x$
 $\langle proof \rangle$

lemma *o-case-sum*: $h \circ case\text{-}sum\ f\ g = case\text{-}sum\ (h \circ f)\ (h \circ g)$
 $\langle proof \rangle$

lemma *case-sum-o-map-sum*: $case\text{-}sum\ f\ g \circ map\text{-}sum\ h1\ h2 = case\text{-}sum\ (f \circ h1)\ (g \circ h2)$
 $\langle proof \rangle$

lemma *case-sum-o-map-sum-id*: $(\text{case-sum } id \ g \circ \text{map-sum } f \ id) \ x = \text{case-sum } (f \circ id) \ g \ x$
 $\langle \text{proof} \rangle$

lemma *rel-fun-def-butlast*:
 $\text{rel-fun } R \ (\text{rel-fun } S \ T) \ f \ g = (\forall x \ y. \ R \ x \ y \longrightarrow (\text{rel-fun } S \ T) \ (f \ x) \ (g \ y))$
 $\langle \text{proof} \rangle$

lemma *subst-eq-imp*: $(\forall a \ b. \ a = b \longrightarrow P \ a \ b) \equiv (\forall a. \ P \ a \ a)$
 $\langle \text{proof} \rangle$

lemma *eq-subset*: $(=) \leq (\lambda a \ b. \ P \ a \ b \vee a = b)$
 $\langle \text{proof} \rangle$

lemma *eq-le-Grp-id-iff*: $((=) \leq \text{Grp } (\text{Collect } R) \ id) = (\text{All } R)$
 $\langle \text{proof} \rangle$

lemma *Grp-id-mono-subst*: $(\bigwedge x \ y. \ \text{Grp } P \ id \ x \ y \Longrightarrow \text{Grp } Q \ id \ (f \ x) \ (f \ y)) \equiv$
 $(\bigwedge x. \ x \in P \Longrightarrow f \ x \in Q)$
 $\langle \text{proof} \rangle$

lemma *vimage2p-mono*: $\text{vimage2p } f \ g \ R \ x \ y \Longrightarrow R \leq S \Longrightarrow \text{vimage2p } f \ g \ S \ x \ y$
 $\langle \text{proof} \rangle$

lemma *vimage2p-refl*: $(\bigwedge x. \ R \ x \ x) \Longrightarrow \text{vimage2p } f \ f \ R \ x \ x$
 $\langle \text{proof} \rangle$

lemma
assumes *type-definition Rep Abs UNIV*
shows *type-copy-Rep-o-Abs*: $\text{Rep} \circ \text{Abs} = id$ **and** *type-copy-Abs-o-Rep*: $\text{Abs} \circ \text{Rep} = id$
 $\langle \text{proof} \rangle$

lemma *type-copy-map-comp0-undo*:
assumes *type-definition Rep Abs UNIV*
type-definition Rep' Abs' UNIV
type-definition Rep'' Abs'' UNIV
shows $\text{Abs}' \circ M \circ \text{Rep}'' = (\text{Abs}' \circ M1 \circ \text{Rep}) \circ (\text{Abs} \circ M2 \circ \text{Rep}'') \Longrightarrow M1 \circ M2 = M$
 $\langle \text{proof} \rangle$

lemma *vimage2p-id*: $\text{vimage2p } id \ id \ R = R$
 $\langle \text{proof} \rangle$

lemma *vimage2p-comp*: $\text{vimage2p } (f1 \circ f2) \ (g1 \circ g2) = \text{vimage2p } f2 \ g2 \circ \text{vimage2p } f1 \ g1$
 $\langle \text{proof} \rangle$

lemma *vimage2p-rel-fun*: $\text{rel-fun } (\text{vimage2p } f \ g \ R) \ R \ f \ g$

$\langle \text{proof} \rangle$

lemma *fun-cong-unused-0*: $f = (\lambda x. g) \implies f (\lambda x. 0) = g$
 $\langle \text{proof} \rangle$

lemma *inj-on-convol-ident*: $\text{inj-on } (\lambda x. (x, f x)) \ X$
 $\langle \text{proof} \rangle$

lemma *map-sum-if-distrib-then*:

$\bigwedge f g e x y. \text{map-sum } f g \ (\text{if } e \text{ then } \text{Inl } x \text{ else } y) = (\text{if } e \text{ then } \text{Inl } (f x) \text{ else } \text{map-sum } f g y)$
 $\bigwedge f g e x y. \text{map-sum } f g \ (\text{if } e \text{ then } \text{Inr } x \text{ else } y) = (\text{if } e \text{ then } \text{Inr } (g x) \text{ else } \text{map-sum } f g y)$
 $\langle \text{proof} \rangle$

lemma *map-sum-if-distrib-else*:

$\bigwedge f g e x y. \text{map-sum } f g \ (\text{if } e \text{ then } x \text{ else } \text{Inl } y) = (\text{if } e \text{ then } \text{map-sum } f g x \text{ else } \text{Inl } (f y))$
 $\bigwedge f g e x y. \text{map-sum } f g \ (\text{if } e \text{ then } x \text{ else } \text{Inr } y) = (\text{if } e \text{ then } \text{map-sum } f g x \text{ else } \text{Inr } (g y))$
 $\langle \text{proof} \rangle$

lemma *case-prod-app*: $\text{case-prod } f x y = \text{case-prod } (\lambda l r. f l r y) x$
 $\langle \text{proof} \rangle$

lemma *case-sum-map-sum*: $\text{case-sum } l r \ (\text{map-sum } f g x) = \text{case-sum } (l \circ f) \ (r \circ g) x$
 $\langle \text{proof} \rangle$

lemma *case-sum-transfer*:

$\text{rel-fun } (\text{rel-fun } R \ T) \ (\text{rel-fun } (\text{rel-fun } S \ T) \ (\text{rel-fun } (\text{rel-sum } R \ S) \ T)) \ \text{case-sum}$
 case-sum
 $\langle \text{proof} \rangle$

lemma *case-prod-map-prod*: $\text{case-prod } h \ (\text{map-prod } f g x) = \text{case-prod } (\lambda l r. h (f l) (g r)) x$
 $\langle \text{proof} \rangle$

lemma *case-prod-o-map-prod*: $\text{case-prod } f \circ \text{map-prod } g1 g2 = \text{case-prod } (\lambda l r. f (g1 l) (g2 r))$
 $\langle \text{proof} \rangle$

lemma *case-prod-transfer*:

$(\text{rel-fun } (\text{rel-fun } A \ (\text{rel-fun } B \ C)) \ (\text{rel-fun } (\text{rel-prod } A \ B) \ C)) \ \text{case-prod case-prod}$
 $\langle \text{proof} \rangle$

lemma *eq-ifI*: $(P \longrightarrow t = u1) \implies (\neg P \longrightarrow t = u2) \implies t = (\text{if } P \text{ then } u1 \text{ else } u2)$
 $\langle \text{proof} \rangle$

lemma *comp-transfer*:

rel-fun (*rel-fun* *B C*) (*rel-fun* (*rel-fun* *A B*) (*rel-fun* *A C*)) (*o*) (*o*)
 ⟨*proof*⟩

lemma *If-transfer*: *rel-fun* (=) (*rel-fun* *A* (*rel-fun* *A A*)) *If If*

⟨*proof*⟩

lemma *Abs-transfer*:

assumes *type-copy1*: *type-definition* *Rep1 Abs1 UNIV*
assumes *type-copy2*: *type-definition* *Rep2 Abs2 UNIV*
shows *rel-fun* *R* (*vimage2p* *Rep1 Rep2 R*) *Abs1 Abs2*
 ⟨*proof*⟩

lemma *Inl-transfer*:

rel-fun *S* (*rel-sum* *S T*) *Inl Inl*
 ⟨*proof*⟩

lemma *Inr-transfer*:

rel-fun *T* (*rel-sum* *S T*) *Inr Inr*
 ⟨*proof*⟩

lemma *Pair-transfer*: *rel-fun* *A* (*rel-fun* *B* (*rel-prod* *A B*)) *Pair Pair*

⟨*proof*⟩

lemma *eq-onp-live-step*: $x = y \implies \text{eq-onp } P \ a \ a \wedge x \longleftrightarrow P \ a \wedge y$

⟨*proof*⟩

lemma *top-conj*: $\text{top } x \wedge P \longleftrightarrow P \ P \wedge \text{top } x \longleftrightarrow P$

⟨*proof*⟩

lemma *fst-convol'*: $\text{fst } (\langle f, g \rangle \ x) = f \ x$

⟨*proof*⟩

lemma *snd-convol'*: $\text{snd } (\langle f, g \rangle \ x) = g \ x$

⟨*proof*⟩

lemma *convol-expand-snd*: $\text{fst} \circ f = g \implies \langle g, \text{snd} \circ f \rangle = f$

⟨*proof*⟩

lemma *convol-expand-snd'*:

assumes ($\text{fst} \circ f = g$)
shows $h = \text{snd} \circ f \longleftrightarrow \langle g, h \rangle = f$

⟨*proof*⟩

lemma *case-sum-expand-Inr-pointfree*: $f \circ \text{Inl} = g \implies \text{case-sum } g \ (f \circ \text{Inr}) = f$

⟨*proof*⟩

lemma *case-sum-expand-Inr'*: $f \circ \text{Inl} = g \implies h = f \circ \text{Inr} \longleftrightarrow \text{case-sum } g \ h = f$

<proof>

lemma *case-sum-expand-Inr*: $f \circ \text{Inl} = g \implies f\ x = \text{case-sum}\ g\ (f \circ \text{Inr})\ x$
<proof>

lemma *id-transfer*: $\text{rel-fun}\ A\ A\ \text{id}\ \text{id}$
<proof>

lemma *fst-transfer*: $\text{rel-fun}\ (\text{rel-prod}\ A\ B)\ A\ \text{fst}\ \text{fst}$
<proof>

lemma *snd-transfer*: $\text{rel-fun}\ (\text{rel-prod}\ A\ B)\ B\ \text{snd}\ \text{snd}$
<proof>

lemma *convol-transfer*:
 $\text{rel-fun}\ (\text{rel-fun}\ R\ S)\ (\text{rel-fun}\ (\text{rel-fun}\ R\ T)\ (\text{rel-fun}\ R\ (\text{rel-prod}\ S\ T)))\ \text{BNF-Def.convolve}$
<proof>

lemma *Let-const*: $\text{Let}\ x\ (\lambda\cdot. c) = c$
<proof>

<ML>

end

37 Least Fixpoint (Datatype) Operation on Bounded Natural Functors

theory *BNF-Least-Fixpoint*
imports *BNF-Fixpoint-Base*
keywords
 $\text{datatype} :: \text{thy-defn}$ **and**
 $\text{datatype-compat} :: \text{thy-defn}$
begin

lemma *subset-emptyI*: $(\bigwedge x. x \in A \implies \text{False}) \implies A \subseteq \{\}$
<proof>

lemma *image-Collect-subsetI*: $(\bigwedge x. P\ x \implies f\ x \in B) \implies f\ ` \{x. P\ x\} \subseteq B$
<proof>

lemma *Collect-restrict*: $\{x. x \in X \wedge P\ x\} \subseteq X$
<proof>

lemma *prop-restrict*: $\llbracket x \in Z; Z \subseteq \{x. x \in X \wedge P\ x\} \rrbracket \implies P\ x$
<proof>

lemma *underS-I*: $\llbracket i \neq j; (i, j) \in R \rrbracket \implies i \in \text{underS } R \ j$
 $\langle \text{proof} \rangle$

lemma *underS-E*: $i \in \text{underS } R \ j \implies i \neq j \wedge (i, j) \in R$
 $\langle \text{proof} \rangle$

lemma *underS-Field*: $i \in \text{underS } R \ j \implies i \in \text{Field } R$
 $\langle \text{proof} \rangle$

lemma *ex-bij-betw*: $|A| \leq_o (r :: 'b \text{ rel}) \implies \exists f B :: 'b \text{ set. } \text{bij-betw } f \ B \ A$
 $\langle \text{proof} \rangle$

lemma *bij-betwI'*:
 $\llbracket \bigwedge x y. \llbracket x \in X; y \in X \rrbracket \implies (f \ x = f \ y) = (x = y);$
 $\bigwedge x. x \in X \implies f \ x \in Y;$
 $\bigwedge y. y \in Y \implies \exists x \in X. y = f \ x \rrbracket \implies \text{bij-betw } f \ X \ Y$
 $\langle \text{proof} \rangle$

lemma *surj-fun-eq*:
assumes *surj-on*: $f \text{ ' } X = \text{UNIV}$ **and** *eq-on*: $\forall x \in X. (g1 \circ f) \ x = (g2 \circ f) \ x$
shows $g1 = g2$
 $\langle \text{proof} \rangle$

lemma *Card-order-wo-rel*: $\text{Card-order } r \implies \text{wo-rel } r$
 $\langle \text{proof} \rangle$

lemma *Cinfinite-limit*: $\llbracket x \in \text{Field } r; \text{Cinfinite } r \rrbracket \implies \exists y \in \text{Field } r. x \neq y \wedge (x, y) \in r$
 $\langle \text{proof} \rangle$

lemma *Card-order-trans*:
 $\llbracket \text{Card-order } r; x \neq y; (x, y) \in r; y \neq z; (y, z) \in r \rrbracket \implies x \neq z \wedge (x, z) \in r$
 $\langle \text{proof} \rangle$

lemma *Cinfinite-limit2*:
assumes *x1*: $x1 \in \text{Field } r$ **and** *x2*: $x2 \in \text{Field } r$ **and** *r*: $\text{Cinfinite } r$
shows $\exists y \in \text{Field } r. (x1 \neq y \wedge (x1, y) \in r) \wedge (x2 \neq y \wedge (x2, y) \in r)$
 $\langle \text{proof} \rangle$

lemma *Cinfinite-limit-finite*:
 $\llbracket \text{finite } X; X \subseteq \text{Field } r; \text{Cinfinite } r \rrbracket \implies \exists y \in \text{Field } r. \forall x \in X. (x \neq y \wedge (x, y) \in r)$
 $\langle \text{proof} \rangle$

lemma *insert-subsetI*: $\llbracket x \in A; X \subseteq A \rrbracket \implies \text{insert } x \ X \subseteq A$
 $\langle \text{proof} \rangle$

lemmas *well-order-induct-imp* = $\text{wo-rel.well-order-induct}[of \ r \ \lambda x. x \in \text{Field } r \longrightarrow P \ x \text{ for } r \ P]$

lemma *meta-spec2*:

assumes $(\bigwedge x y. PROP P x y)$

shows $PROP P x y$

<proof>

lemma *nchotomy-relcomppE*:

assumes $\bigwedge y. \exists x. y = f x (r OO s) a c \bigwedge b. r a (f b) \implies s (f b) c \implies P$

shows P

<proof>

lemma *predicate2D-vimage2p*: $\llbracket R \leq vimage2p f g S; R x y \rrbracket \implies S (f x) (g y)$

<proof>

lemma *ssubst-Pair-rhs*: $\llbracket (r, s) \in R; s' = s \rrbracket \implies (r, s') \in R$

<proof>

lemma *all-mem-range1*:

$(\bigwedge y. y \in range f \implies P y) \equiv (\bigwedge x. P (f x))$

<proof>

lemma *all-mem-range2*:

$(\bigwedge fa y. fa \in range f \implies y \in range fa \implies P y) \equiv (\bigwedge x xa. P (f x xa))$

<proof>

lemma *all-mem-range3*:

$(\bigwedge fa fb y. fa \in range f \implies fb \in range fa \implies y \in range fb \implies P y) \equiv (\bigwedge x xa xb. P (f x xa xb))$

<proof>

lemma *all-mem-range4*:

$(\bigwedge fa fb fc y. fa \in range f \implies fb \in range fa \implies fc \in range fb \implies y \in range fc \implies P y) \equiv$

$(\bigwedge x xa xb xc. P (f x xa xb xc))$

<proof>

lemma *all-mem-range5*:

$(\bigwedge fa fb fc fd y. fa \in range f \implies fb \in range fa \implies fc \in range fb \implies fd \in range fc \implies$

$y \in range fd \implies P y) \equiv$

$(\bigwedge x xa xb xc xd. P (f x xa xb xc xd))$

<proof>

lemma *all-mem-range6*:

$(\bigwedge fa fb fc fd fe ff y. fa \in range f \implies fb \in range fa \implies fc \in range fb \implies fd \in range fc \implies$

$fe \in range fd \implies ff \in range fe \implies y \in range ff \implies P y) \equiv$

$(\bigwedge x xa xb xc xd xe xf. P (f x xa xb xc xd xe xf))$

<proof>

lemma *all-mem-range7*:

$(\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ fg\ y. fa \in \text{range } f \implies fb \in \text{range } fa \implies fc \in \text{range } fb \implies fd \in \text{range } fc \implies$
 $fe \in \text{range } fd \implies ff \in \text{range } fe \implies fg \in \text{range } ff \implies y \in \text{range } fg \implies P\ y) \equiv$
 $(\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf\ xg. P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf\ xg))$
 $\langle \text{proof} \rangle$

lemma *all-mem-range8*:

$(\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ fg\ fh\ y. fa \in \text{range } f \implies fb \in \text{range } fa \implies fc \in \text{range } fb \implies$
 $fd \in \text{range } fc \implies$
 $fe \in \text{range } fd \implies ff \in \text{range } fe \implies fg \in \text{range } ff \implies fh \in \text{range } fg \implies y \in$
 $\text{range } fh \implies P\ y) \equiv$
 $(\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf\ xg\ xh. P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf\ xg\ xh))$
 $\langle \text{proof} \rangle$

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* $id\ p \implies \text{pred-fun } (\lambda x. \text{True})\ p\ f = \text{pred-fun}$
 $(\lambda x. \text{True})\ id\ (p \circ f)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

end

38 Equivalence Relations in Higher-Order Set Theory

theory *Equiv-Relations*
imports *BNF-Least-Fixpoint*
begin

38.1 Equivalence relations – set version

definition *equiv* :: $'a\ \text{set} \Rightarrow ('a \times 'a)\ \text{set} \Rightarrow \text{bool}$
where $\text{equiv } A\ r \iff r \subseteq A \times A \wedge \text{refl-on } A\ r \wedge \text{sym } r \wedge \text{trans } r$

lemma *equivI*: $r \subseteq A \times A \implies \text{refl-on } A\ r \implies \text{sym } r \implies \text{trans } r \implies \text{equiv } A\ r$
 $\langle \text{proof} \rangle$

lemma *equivE*:

assumes $\text{equiv } A\ r$
obtains $r \subseteq A \times A$ **and** $\text{refl-on } A\ r$ **and** $\text{sym } r$ **and** $\text{trans } r$
 $\langle \text{proof} \rangle$

Suppes, Theorem 70: r is an equiv relation iff $r^{-1} \circ r = r$.

First half: $\text{equiv } A \ r \implies r^{-1} \circ r = r$.

lemma *sym-trans-comp-subset*:

assumes $r \subseteq A \times A$ **and** *sym-on* $A \ r$ **and** *trans-on* $A \ r$

shows $r^{-1} \circ r \subseteq r$

<proof>

lemma *refl-on-comp-subset*: $r \subseteq A \times A \implies \text{refl-on } A \ r \implies r \subseteq r^{-1} \circ r$

<proof>

lemma *equiv-comp-eq*: $\text{equiv } A \ r \implies r^{-1} \circ r = r$

<proof>

Second half.

lemma *comp-equivI*:

assumes $r^{-1} \circ r = r$ *Domain* $r = A$

shows *equiv* $A \ r$

<proof>

38.2 Equivalence classes

lemma *equiv-class-subset*: $\text{equiv } A \ r \implies (a, b) \in r \implies r^{\cdot\cdot}\{a\} \subseteq r^{\cdot\cdot}\{b\}$

— lemma for the next result

<proof>

theorem *equiv-class-eq*:

assumes *equiv* $A \ r$ **and** $(a, b) \in r$

shows $r^{\cdot\cdot}\{a\} = r^{\cdot\cdot}\{b\}$

<proof>

lemma *equiv-class-self*: $\text{equiv } A \ r \implies a \in A \implies a \in r^{\cdot\cdot}\{a\}$

<proof>

lemma *subset-equiv-class*: $\text{equiv } A \ r \implies r^{\cdot\cdot}\{b\} \subseteq r^{\cdot\cdot}\{a\} \implies b \in A \implies (a, b) \in r$

— lemma for the next result

<proof>

lemma *eq-equiv-class*: $r^{\cdot\cdot}\{a\} = r^{\cdot\cdot}\{b\} \implies \text{equiv } A \ r \implies b \in A \implies (a, b) \in r$

<proof>

lemma *equiv-class-nondisjoint*: $\text{equiv } A \ r \implies x \in (r^{\cdot\cdot}\{a\} \cap r^{\cdot\cdot}\{b\}) \implies (a, b) \in r$

<proof>

lemma *equiv-type*: $\text{equiv } A \ r \implies r \subseteq A \times A$

<proof>

lemma *equiv-class-eq-iff*: $\text{equiv } A \ r \implies (x, y) \in r \iff r^{\cdot\cdot}\{x\} = r^{\cdot\cdot}\{y\} \wedge x \in A$

$\wedge y \in A$

<proof>

lemma *eq-equiv-class-iff*: $\text{equiv } A \ r \implies x \in A \implies y \in A \implies r^{\{\{x\}\}} = r^{\{\{y\}\}} \longleftrightarrow (x, y) \in r$
 ⟨proof⟩

lemma *disjnt-equiv-class*: $\text{equiv } A \ r \implies \text{disjnt } (r^{\{\{a\}\}}) (r^{\{\{b\}\}}) \longleftrightarrow (a, b) \notin r$
 ⟨proof⟩

38.3 Quotients

definition *quotient* :: $'a \text{ set} \Rightarrow ('a \times 'a) \text{ set} \Rightarrow 'a \text{ set set}$ (**infixl** $\langle '/' \rangle$ 90)
 where $A//r = (\bigcup x \in A. \{r^{\{\{x\}\}}\})$ — set of equiv classes

lemma *quotientI*: $x \in A \implies r^{\{\{x\}\}} \in A//r$
 ⟨proof⟩

lemma *quotientE*: $X \in A//r \implies (\bigwedge x. X = r^{\{\{x\}\}} \implies x \in A \implies P) \implies P$
 ⟨proof⟩

lemma *Union-quotient*: $\text{equiv } A \ r \implies \bigcup (A//r) = A$
 ⟨proof⟩

lemma *quotient-disj-strong*:
 assumes $r \subseteq A \times A$ and *sym-on* $A \ r$ and *trans-on* $A \ r$ and $X \in A//r$ and $Y \in A//r$
 shows $X = Y \vee X \cap Y = \{\}$
 ⟨proof⟩

lemma *quotient-disj*: $\text{equiv } A \ r \implies X \in A//r \implies Y \in A//r \implies X = Y \vee X \cap Y = \{\}$
 ⟨proof⟩

lemma *quotient-eqI*:
 assumes $\text{equiv } A \ r$ $X \in A//r$ $Y \in A//r$ and *xy*: $x \in X \ y \in Y \ (x, y) \in r$
 shows $X = Y$
 ⟨proof⟩

lemma *quotient-eq-iff*:
 assumes $\text{equiv } A \ r$ $X \in A//r$ $Y \in A//r$ and *xy*: $x \in X \ y \in Y$
 shows $X = Y \longleftrightarrow (x, y) \in r$
 ⟨proof⟩

lemma *eq-equiv-class-iff2*: $\text{equiv } A \ r \implies x \in A \implies y \in A \implies \{x\}//r = \{y\}//r \longleftrightarrow (x, y) \in r$
 ⟨proof⟩

lemma *quotient-empty* [*simp*]: $\{\}//r = \{\}$
 ⟨proof⟩

lemma *quotient-is-empty* [*iff*]: $A//r = \{\} \longleftrightarrow A = \{\}$

$\langle \text{proof} \rangle$

lemma *quotient-is-empty2* [iff]: $\{\} = A//r \longleftrightarrow A = \{\}$
 $\langle \text{proof} \rangle$

lemma *singleton-quotient*: $\{x\} // r = \{r \text{ “ } \{x\}\}$
 $\langle \text{proof} \rangle$

lemma *quotient-diff1*: *inj-on* $(\lambda a. \{a\} // r) A \implies a \in A \implies (A - \{a\}) // r = A // r - \{a\} // r$
 $\langle \text{proof} \rangle$

38.4 Refinement of one equivalence relation WRT another

lemma *refines-equiv-class-eq*: $R \subseteq S \implies \text{equiv } A \ R \implies \text{equiv } A \ S \implies R \text{ “ } (S \text{ “ } \{a\}) = S \text{ “ } \{a\}$
 $\langle \text{proof} \rangle$

lemma *refines-equiv-class-eq2*: $R \subseteq S \implies \text{equiv } A \ R \implies \text{equiv } A \ S \implies S \text{ “ } (R \text{ “ } \{a\}) = S \text{ “ } \{a\}$
 $\langle \text{proof} \rangle$

lemma *refines-equiv-image-eq*: $R \subseteq S \implies \text{equiv } A \ R \implies \text{equiv } A \ S \implies (\lambda X. S \text{ “ } X) \text{ “ } (A // R) = A // S$
 $\langle \text{proof} \rangle$

lemma *finite-refines-finite*:
 $\text{finite } (A // R) \implies R \subseteq S \implies \text{equiv } A \ R \implies \text{equiv } A \ S \implies \text{finite } (A // S)$
 $\langle \text{proof} \rangle$

lemma *finite-refines-card-le*:
 $\text{finite } (A // R) \implies R \subseteq S \implies \text{equiv } A \ R \implies \text{equiv } A \ S \implies \text{card } (A // S) \leq \text{card } (A // R)$
 $\langle \text{proof} \rangle$

38.5 Defining unary operations upon equivalence classes

A congruence-preserving function.

definition *congruent* :: $('a \times 'a) \text{ set} \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$
where $\text{congruent } r \ f \longleftrightarrow (\forall (y, z) \in r. f \ y = f \ z)$

lemma *congruentI*: $(\bigwedge y \ z. (y, z) \in r \implies f \ y = f \ z) \implies \text{congruent } r \ f$
 $\langle \text{proof} \rangle$

lemma *congruentD*: $\text{congruent } r \ f \implies (y, z) \in r \implies f \ y = f \ z$
 $\langle \text{proof} \rangle$

abbreviation *RESPECTS* :: $('a \Rightarrow 'b) \Rightarrow ('a \times 'a) \text{ set} \Rightarrow \text{bool}$ (**infixr** $\langle \text{respects} \rangle$ 80)

where f respects $r \equiv$ congruent r f

lemma *UN-constant-eq*: $a \in A \implies \forall y \in A. f\ y = c \implies (\bigcup y \in A. f\ y) = c$
 — lemma required to prove *UN-equiv-class*
 $\langle proof \rangle$

lemma *UN-equiv-class*:
assumes *equiv* A r f respects r $a \in A$
shows $(\bigcup x \in r^{-1}\{a\}. f\ x) = f\ a$
 — Conversion rule
 $\langle proof \rangle$

lemma *UN-equiv-class-type*:
assumes r : *equiv* A r f respects r **and** X : $X \in A//r$ **and** AB : $\bigwedge x. x \in A \implies f\ x \in B$
shows $(\bigcup x \in X. f\ x) \in B$
 $\langle proof \rangle$

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; *bcong* could be $\bigwedge y. y \in A \implies f\ y \in B$.

lemma *UN-equiv-class-inject*:
assumes *equiv* A r f respects r
and *eq*: $(\bigcup x \in X. f\ x) = (\bigcup y \in Y. f\ y)$
and X : $X \in A//r$ **and** Y : $Y \in A//r$
and *fr*: $\bigwedge x\ y. x \in A \implies y \in A \implies f\ x = f\ y \implies (x, y) \in r$
shows $X = Y$
 $\langle proof \rangle$

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 $\Rightarrow ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow bool$
where *congruent2* $r1$ $r2$ $f \longleftrightarrow (\forall (y1, z1) \in r1. \forall (y2, z2) \in r2. f\ y1\ y2 = f\ z1\ z2)$

lemma *congruent2I'*:
assumes $\bigwedge y1\ z1\ y2\ z2. (y1, z1) \in r1 \implies (y2, z2) \in r2 \implies f\ y1\ y2 = f\ z1\ z2$
shows *congruent2* $r1$ $r2$ f
 $\langle proof \rangle$

lemma *congruent2D*: *congruent2* $r1$ $r2$ $f \implies (y1, z1) \in r1 \implies (y2, z2) \in r2 \implies f\ y1\ y2 = f\ z1\ z2$
 $\langle proof \rangle$

Abbreviation for the common case where the relations are identical.

abbreviation *RESPECTS2*:: $('a \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('a \times 'a)$ set $\Rightarrow bool$ (**infixr** $\langle respects2 \rangle$ 80)

where $f \text{ respects2 } r \equiv \text{congruent2 } r \ r \ f$

lemma *congruent2-implies-congruent*:

$\text{equiv } A \ r1 \implies \text{congruent2 } r1 \ r2 \ f \implies a \in A \implies \text{congruent } r2 \ (f \ a)$
 $\langle \text{proof} \rangle$

lemma *congruent2-implies-congruent-UN*:

assumes $\text{equiv } A1 \ r1 \ \text{equiv } A2 \ r2 \ \text{congruent2 } r1 \ r2 \ f \ a \in A2$
shows $\text{congruent } r1 \ (\lambda x1. \bigcup x2 \in r2^{\{a\}}. f \ x1 \ x2)$
 $\langle \text{proof} \rangle$

lemma *UN-equiv-class2*:

$\text{equiv } A1 \ r1 \implies \text{equiv } A2 \ r2 \implies \text{congruent2 } r1 \ r2 \ f \implies a1 \in A1 \implies a2 \in A2$
 \implies
 $(\bigcup x1 \in r1^{\{a1\}}. \bigcup x2 \in r2^{\{a2\}}. f \ x1 \ x2) = f \ a1 \ a2$
 $\langle \text{proof} \rangle$

lemma *UN-equiv-class-type2*:

$\text{equiv } A1 \ r1 \implies \text{equiv } A2 \ r2 \implies \text{congruent2 } r1 \ r2 \ f$
 $\implies X1 \in A1 // r1 \implies X2 \in A2 // r2$
 $\implies (\bigwedge x1 \ x2. x1 \in A1 \implies x2 \in A2 \implies f \ x1 \ x2 \in B)$
 $\implies (\bigcup x1 \in X1. \bigcup x2 \in X2. f \ x1 \ x2) \in B$
 $\langle \text{proof} \rangle$

lemma *UN-UN-split-split-eq*:

$(\bigcup (x1, x2) \in X. \bigcup (y1, y2) \in Y. A \ x1 \ x2 \ y1 \ y2) =$
 $(\bigcup x \in X. \bigcup y \in Y. (\lambda (x1, x2). (\lambda (y1, y2). A \ x1 \ x2 \ y1 \ y2) \ y) \ x)$
— Allows a natural expression of binary operators,
— without explicit calls to *split*
 $\langle \text{proof} \rangle$

lemma *congruent2I*:

$\text{equiv } A1 \ r1 \implies \text{equiv } A2 \ r2$
 $\implies (\bigwedge y \ z \ w. w \in A2 \implies (y, z) \in r1 \implies f \ y \ w = f \ z \ w)$
 $\implies (\bigwedge y \ z \ w. w \in A1 \implies (y, z) \in r2 \implies f \ w \ y = f \ w \ z)$
 $\implies \text{congruent2 } r1 \ r2 \ f$
— Suggested by John Harrison – the two subproofs may be
— *much* simpler than the direct proof.
 $\langle \text{proof} \rangle$

lemma *congruent2-commuteI*:

assumes $\text{equivA: equiv } A \ r$
and *commute*: $\bigwedge y \ z. y \in A \implies z \in A \implies f \ y \ z = f \ z \ y$
and *cong*: $\bigwedge y \ z \ w. w \in A \implies (y, z) \in r \implies f \ w \ y = f \ w \ z$
shows $f \text{ respects2 } r$
 $\langle \text{proof} \rangle$

38.7 Quotients and finiteness

Suggested by Florian Kammüller

lemma *finite-quotient*:

assumes *finite* A $r \subseteq A \times A$

shows *finite* $(A//r)$

— recall *equiv* $?A$ $?r \implies ?r \subseteq ?A \times ?A$

<proof>

lemma *finite-equiv-class*: *finite* $A \implies r \subseteq A \times A \implies X \in A//r \implies \text{finite } X$

<proof>

lemma *equiv-imp-dvd-card*:

assumes *finite* A *equiv* A $r \wedge X. X \in A//r \implies k \text{ dvd card } X$

shows $k \text{ dvd card } A$

<proof>

38.8 Kernel of a Function

definition *kernel* :: $('a \Rightarrow 'b) \Rightarrow ('a * 'a) \text{ set}$ **where**

kernel $f = \{(x,y). f\ x = f\ y\}$

lemma *equiv-kernel*: *equiv* *UNIV* (*kernel* f)

<proof>

lemma *respects-kernel*: f *respects* (*kernel* f)

<proof>

lemma *inj-on-vimage-image*: *inj-on* $(\lambda b. f - \{b\})$ $(f - A)$

<proof>

lemma *kernel-Image*: *kernel* $f - A = f - (f - A)$

<proof>

lemma *quotient-kernel-eq-image*: $A // \text{kernel } f = (\lambda b. f - \{b\}) - f - A$

<proof>

lemma *bij-betw-image-quotient-kernel*: *bij-betw* $(\lambda b. f - \{b\})$ $(f - A)$ $(A // \text{kernel } f)$

<proof>

38.9 Projection

definition *proj* :: $('b \times 'a) \text{ set} \Rightarrow 'b \Rightarrow 'a \text{ set}$

where *proj* $r\ x = r - \{x\}$

lemma *proj-preserves*: $x \in A \implies \text{proj } r\ x \in A//r$

<proof>

lemma *proj-in-iff*:
assumes *equiv A r*
shows $\text{proj } r \ x \in A // r \longleftrightarrow x \in A$
 (**is** $?lhs \longleftrightarrow ?rhs$)
 $\langle \text{proof} \rangle$

lemma *proj-iff*: $\text{equiv } A \ r \implies \{x, y\} \subseteq A \implies \text{proj } r \ x = \text{proj } r \ y \longleftrightarrow (x, y) \in r$
 $\langle \text{proof} \rangle$

lemma *proj-image*: $\text{proj } r \ ` \ A = A // r$
 $\langle \text{proof} \rangle$

lemma *in-quotient-imp-non-empty*: $\text{equiv } A \ r \implies X \in A // r \implies X \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *in-quotient-imp-in-rel*: $\text{equiv } A \ r \implies X \in A // r \implies \{x, y\} \subseteq X \implies (x, y) \in r$
 $\langle \text{proof} \rangle$

lemma *in-quotient-imp-closed*: $\text{equiv } A \ r \implies X \in A // r \implies x \in X \implies (x, y) \in r \implies y \in X$
 $\langle \text{proof} \rangle$

lemma *in-quotient-imp-subset*: $\text{equiv } A \ r \implies X \in A // r \implies X \subseteq A$
 $\langle \text{proof} \rangle$

38.10 Equivalence relations – predicate version

Partial equivalences.

definition *part-equivp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where $\text{part-equivp } R \longleftrightarrow (\exists x. R \ x \ x) \wedge (\forall x \ y. R \ x \ y \longleftrightarrow R \ x \ x \wedge R \ y \ y \wedge R \ x \ y = R \ y)$
 — John-Harrison-style characterization

lemma *part-equivpI*: $\exists x. R \ x \ x \implies \text{symp } R \implies \text{transp } R \implies \text{part-equivp } R$
 $\langle \text{proof} \rangle$

lemma *part-equivpE*:
assumes *part-equivp R*
obtains *x* **where** $R \ x \ x$ **and** *symp R* **and** *transp R*
 $\langle \text{proof} \rangle$

lemma *part-equivp-refl-symp-transp*: $\text{part-equivp } R \longleftrightarrow (\exists x. R \ x \ x) \wedge \text{symp } R \wedge \text{transp } R$
 $\langle \text{proof} \rangle$

lemma *part-equivp-symp*: $\text{part-equivp } R \implies R \ x \ y \implies R \ y \ x$

$\langle \text{proof} \rangle$

lemma *part-equivp-transp*: $\text{part-equivp } R \implies R \ x \ y \implies R \ y \ z \implies R \ x \ z$
 $\langle \text{proof} \rangle$

lemma *part-equivp-typedef*: $\text{part-equivp } R \implies \exists d. d \in \{c. \exists x. R \ x \ x \wedge c = \text{Collect } (R \ x)\}$
 $\langle \text{proof} \rangle$

Total equivalences.

definition *equivp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *equivp* $R \longleftrightarrow (\forall x \ y. R \ x \ y = (R \ x = R \ y))$ — John-Harrison-style
 characterization

lemma *equivpI*: $\text{reflp } R \implies \text{symp } R \implies \text{transp } R \implies \text{equivp } R$
 $\langle \text{proof} \rangle$

lemma *equivpE*:
assumes *equivp* R
obtains *reflp* R **and** *symp* R **and** *transp* R
 $\langle \text{proof} \rangle$

lemma *equivp-implies-part-equivp*: $\text{equivp } R \implies \text{part-equivp } R$
 $\langle \text{proof} \rangle$

lemma *equivp-equiv*: $\text{equiv } \text{UNIV } A \longleftrightarrow \text{equivp } (\lambda x \ y. (x, y) \in A)$
 $\langle \text{proof} \rangle$

lemma *equivp-reflp-symp-transp*: $\text{equivp } R \longleftrightarrow \text{reflp } R \wedge \text{symp } R \wedge \text{transp } R$
 $\langle \text{proof} \rangle$

lemma *identity-equivp*: $\text{equivp } (=)$
 $\langle \text{proof} \rangle$

lemma *equivp-reflp*: $\text{equivp } R \implies R \ x \ x$
 $\langle \text{proof} \rangle$

lemma *equivp-symp*: $\text{equivp } R \implies R \ x \ y \implies R \ y \ x$
 $\langle \text{proof} \rangle$

lemma *equivp-transp*: $\text{equivp } R \implies R \ x \ y \implies R \ y \ z \implies R \ x \ z$
 $\langle \text{proof} \rangle$

lemma *equivp-rtranclp*: $\text{symp } r \implies \text{equivp } r^{**}$
 $\langle \text{proof} \rangle$

lemmas *equivp-rtranclp-symclp* [*simp*] = *equivp-rtranclp*[*OF symp-on-symclp*]

lemma *equivp-vimage2p*: $\text{equivp } R \implies \text{equivp } (\text{vimage2p } f \ f \ R)$

$\langle proof \rangle$

lemma *equivp-imp-transp*: $equivp\ R \implies transp\ R$
 $\langle proof \rangle$

38.11 Equivalence closure

definition *equivclp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
equivclp $r = (symclp\ r)^{**}$

lemma *transp-equivclp* [*simp*]: $transp\ (equivclp\ r)$
 $\langle proof \rangle$

lemma *reflp-equivclp* [*simp*]: $reflp\ (equivclp\ r)$
 $\langle proof \rangle$

lemma *symp-equivclp* [*simp*]: $symp\ (equivclp\ r)$
 $\langle proof \rangle$

lemma *equivp-evquivclp* [*simp*]: $equivp\ (equivclp\ r)$
 $\langle proof \rangle$

lemma *tranclp-equivclp* [*simp*]: $(equivclp\ r)^{++} = equivclp\ r$
 $\langle proof \rangle$

lemma *rtranclp-equivclp* [*simp*]: $(equivclp\ r)^{**} = equivclp\ r$
 $\langle proof \rangle$

lemma *symclp-equivclp* [*simp*]: $symclp\ (equivclp\ r) = equivclp\ r$
 $\langle proof \rangle$

lemma *equivclp-symclp* [*simp*]: $equivclp\ (symclp\ r) = equivclp\ r$
 $\langle proof \rangle$

lemma *equivclp-conversep* [*simp*]: $equivclp\ (conversep\ r) = equivclp\ r$
 $\langle proof \rangle$

lemma *equivclp-sym* [*sym*]: $equivclp\ r\ x\ y \implies equivclp\ r\ y\ x$
 $\langle proof \rangle$

lemma *equivclp-OO-equivclp-le-equivclp*: $equivclp\ r\ OO\ equivclp\ r \leq equivclp\ r$
 $\langle proof \rangle$

lemma *rtranclp-le-equivclp*: $r^{**} \leq equivclp\ r$
 $\langle proof \rangle$

lemma *rtranclp-conversep-le-equivclp*: $r^{-1-1**} \leq equivclp\ r$
 $\langle proof \rangle$

lemma *symclp-rtrancpl-le-equivclp*: $\text{symclp } r^{**} \leq \text{equivclp } r$
 ⟨proof⟩

lemma *r-OO-conversep-into-equivclp*:
 $r^{**} \text{ OO } r^{-1-1**} \leq \text{equivclp } r$
 ⟨proof⟩

lemma *equivclp-induct* [consumes 1, case-names base step, induct pred: equivclp]:
 assumes a : $\text{equivclp } r \ a \ b$
 and cases: $P \ a \ \bigwedge y \ z. \text{equivclp } r \ a \ y \implies r \ y \ z \vee r \ z \ y \implies P \ y \implies P \ z$
 shows $P \ b$
 ⟨proof⟩

lemma *converse-equivclp-induct* [consumes 1, case-names base step]:
 assumes major: $\text{equivclp } r \ a \ b$
 and cases: $P \ b \ \bigwedge y \ z. r \ y \ z \vee r \ z \ y \implies \text{equivclp } r \ z \ b \implies P \ z \implies P \ y$
 shows $P \ a$
 ⟨proof⟩

lemma *equivclp-refl* [simp]: $\text{equivclp } r \ x \ x$
 ⟨proof⟩

lemma *r-into-equivclp* [intro]: $r \ x \ y \implies \text{equivclp } r \ x \ y$
 ⟨proof⟩

lemma *converse-r-into-equivclp* [intro]: $r \ y \ x \implies \text{equivclp } r \ x \ y$
 ⟨proof⟩

lemma *rtrancpl-into-equivclp*: $r^{**} \ x \ y \implies \text{equivclp } r \ x \ y$
 ⟨proof⟩

lemma *converse-rtrancpl-into-equivclp*: $r^{**} \ y \ x \implies \text{equivclp } r \ x \ y$
 ⟨proof⟩

lemma *equivclp-into-equivclp*: $\llbracket \text{equivclp } r \ a \ b; r \ b \ c \vee r \ c \ b \rrbracket \implies \text{equivclp } r \ a \ c$
 ⟨proof⟩

lemma *equivclp-trans* [trans]: $\llbracket \text{equivclp } r \ a \ b; \text{equivclp } r \ b \ c \rrbracket \implies \text{equivclp } r \ a \ c$
 ⟨proof⟩

hide-const (open) *proj*

end

theory *Basic-BNF-LFPs*
imports *BNF-Least-Fixpoint*
begin

definition $xtor :: 'a \Rightarrow 'a$ **where**
 $xtor\ x = x$

lemma $xtor\text{-}map$: $f\ (xtor\ x) = xtor\ (f\ x)$
 $\langle proof \rangle$

lemma $xtor\text{-}map\text{-}unique$: $u \circ xtor = xtor \circ f \implies u = f$
 $\langle proof \rangle$

lemma $xtor\text{-}set$: $f\ (xtor\ x) = f\ x$
 $\langle proof \rangle$

lemma $xtor\text{-}rel$: $R\ (xtor\ x)\ (xtor\ y) = R\ x\ y$
 $\langle proof \rangle$

lemma $xtor\text{-}induct$: $(\bigwedge x. P\ (xtor\ x)) \implies P\ z$
 $\langle proof \rangle$

lemma $xtor\text{-}xtor$: $xtor\ (xtor\ x) = x$
 $\langle proof \rangle$

lemmas $xtor\text{-}inject = xtor\text{-}rel[of\ (=)]$

lemma $xtor\text{-}rel\text{-}induct$: $(\bigwedge x\ y. vimage2p\ id\text{-}bnf\ id\text{-}bnf\ R\ x\ y \implies IR\ (xtor\ x)\ (xtor\ y)) \implies R \leq IR$
 $\langle proof \rangle$

lemma $xtor\text{-}iff\text{-}xtor$: $u = xtor\ w \longleftrightarrow xtor\ u = w$
 $\langle proof \rangle$

lemma $Inl\text{-}def\text{-}alt$: $Inl \equiv (\lambda a. xtor\ (id\text{-}bnf\ (Inl\ a)))$
 $\langle proof \rangle$

lemma $Inr\text{-}def\text{-}alt$: $Inr \equiv (\lambda a. xtor\ (id\text{-}bnf\ (Inr\ a)))$
 $\langle proof \rangle$

lemma $Pair\text{-}def\text{-}alt$: $Pair \equiv (\lambda a\ b. xtor\ (id\text{-}bnf\ (a, b)))$
 $\langle proof \rangle$

definition $ctor\text{-}rec :: 'a \Rightarrow 'a$ **where**
 $ctor\text{-}rec\ x = x$

lemma $ctor\text{-}rec$: $g = id \implies ctor\text{-}rec\ f\ (xtor\ x) = f\ ((id\text{-}bnf \circ g \circ id\text{-}bnf)\ x)$
 $\langle proof \rangle$

lemma $ctor\text{-}rec\text{-}unique$: $g = id \implies f \circ xtor = s \circ (id\text{-}bnf \circ g \circ id\text{-}bnf) \implies f = ctor\text{-}rec\ s$
 $\langle proof \rangle$

lemma *ctor-rec-def-alt*: $f = \text{ctor-rec } (f \circ \text{id-bnf})$
 $\langle \text{proof} \rangle$

lemma *ctor-rec-o-map*: $\text{ctor-rec } f \circ g = \text{ctor-rec } (f \circ (\text{id-bnf} \circ g \circ \text{id-bnf}))$
 $\langle \text{proof} \rangle$

lemma *ctor-rec-transfer*: $\text{rel-fun } (\text{rel-fun } (\text{vimage2p id-bnf id-bnf } R) S) (\text{rel-fun } R S) \text{ ctor-rec ctor-rec}$
 $\langle \text{proof} \rangle$

lemma *eq-fst-iff*: $a = \text{fst } p \longleftrightarrow (\exists b. p = (a, b))$
 $\langle \text{proof} \rangle$

lemma *eq-snd-iff*: $b = \text{snd } p \longleftrightarrow (\exists a. p = (a, b))$
 $\langle \text{proof} \rangle$

lemma *ex-neg-all-pos*: $((\exists x. P x) \implies Q) \equiv (\bigwedge x. P x \implies Q)$
 $\langle \text{proof} \rangle$

lemma *hypsubst-in-prems*: $(\bigwedge x. y = x \implies z = f x \implies P) \equiv (z = f y \implies P)$
 $\langle \text{proof} \rangle$

lemma *isl-map-sum*:
 $\text{isl } (\text{map-sum } f g s) = \text{isl } s$
 $\langle \text{proof} \rangle$

lemma *map-sum-sel*:
 $\text{isl } s \implies \text{projl } (\text{map-sum } f g s) = f (\text{projl } s)$
 $\neg \text{isl } s \implies \text{projr } (\text{map-sum } f g s) = g (\text{projr } s)$
 $\langle \text{proof} \rangle$

lemma *set-sum-sel*:
 $\text{isl } s \implies \text{projl } s \in \text{setl } s$
 $\neg \text{isl } s \implies \text{projr } s \in \text{setr } s$
 $\langle \text{proof} \rangle$

lemma *rel-sum-sel*: $\text{rel-sum } R1 R2 a b = (\text{isl } a = \text{isl } b \wedge$
 $(\text{isl } a \longrightarrow \text{isl } b \longrightarrow R1 (\text{projl } a) (\text{projl } b)) \wedge$
 $(\neg \text{isl } a \longrightarrow \neg \text{isl } b \longrightarrow R2 (\text{projr } a) (\text{projr } b)))$
 $\langle \text{proof} \rangle$

lemma *isl-transfer*: $\text{rel-fun } (\text{rel-sum } A B) (=) \text{isl } \text{isl}$
 $\langle \text{proof} \rangle$

lemma *rel-prod-sel*: $\text{rel-prod } R1 R2 p q = (R1 (\text{fst } p) (\text{fst } q) \wedge R2 (\text{snd } p) (\text{snd } q))$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

```

declare prod.size [no-atp]

hide-const (open) xlor ctor-rec

hide-fact (open)
  xlor-def xlor-map xlor-set xlor-rel xlor-induct xlor-xlor xlor-inject ctor-rec-def
ctor-rec
  ctor-rec-def-alt ctor-rec-o-map xlor-rel-induct Inl-def-alt Inr-def-alt Pair-def-alt

end

```

39 MESON Proof Method

```

theory Meson
imports Nat
begin

```

39.1 Negation Normal Form

de Morgan laws

```

lemma not-conjD:  $\neg(P \wedge Q) \implies \neg P \vee \neg Q$ 
and not-disjD:  $\neg(P \vee Q) \implies \neg P \wedge \neg Q$ 
and not-notD:  $\neg\neg P \implies P$ 
and not-allD:  $\bigwedge P. \neg(\forall x. P(x)) \implies \exists x. \neg P(x)$ 
and not-exD:  $\bigwedge P. \neg(\exists x. P(x)) \implies \forall x. \neg P(x)$ 
 $\langle proof \rangle$ 

```

Removal of \longrightarrow and \longleftrightarrow (positive and negative occurrences)

```

lemma imp-to-disjD:  $P \longrightarrow Q \implies \neg P \vee Q$ 
and not-impD:  $\neg(P \longrightarrow Q) \implies P \wedge \neg Q$ 
and iff-to-disjD:  $P = Q \implies (\neg P \vee Q) \wedge (\neg Q \vee P)$ 
and not-iffD:  $\neg(P = Q) \implies (P \vee Q) \wedge (\neg P \vee \neg Q)$ 
  — Much more efficient than  $P \wedge \neg Q \vee Q \wedge \neg P$  for computing CNF
and not-refl-disjD:  $x \neq x \vee P \implies P$ 
 $\langle proof \rangle$ 

```

39.2 Pulling out the existential quantifiers

Conjunction

```

lemma conj-exD1:  $\bigwedge P Q. (\exists x. P(x)) \wedge Q \implies \exists x. P(x) \wedge Q$ 
and conj-exD2:  $\bigwedge P Q. P \wedge (\exists x. Q(x)) \implies \exists x. P \wedge Q(x)$ 
 $\langle proof \rangle$ 

```

Disjunction

```

lemma disj-exD:  $\bigwedge P Q. (\exists x. P(x)) \vee (\exists x. Q(x)) \implies \exists x. P(x) \vee Q(x)$ 
  — DO NOT USE with forall-Skolemization: makes fewer schematic variables!!

```

— With ex-Skolemization, makes fewer Skolem constants
and *disj-exD1*: $\bigwedge P \ Q. (\exists x. P(x)) \vee Q \implies \exists x. P(x) \vee Q$
and *disj-exD2*: $\bigwedge P \ Q. P \vee (\exists x. Q(x)) \implies \exists x. P \vee Q(x)$
 $\langle proof \rangle$

lemma *disj-assoc*: $(P \vee Q) \vee R \implies P \vee (Q \vee R)$
and *disj-comm*: $P \vee Q \implies Q \vee P$
and *disj-FalseD1*: $False \vee P \implies P$
and *disj-FalseD2*: $P \vee False \implies P$
 $\langle proof \rangle$

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*: $\neg P \vee Q \implies ((\neg P \implies P) \implies Q)$
 $\langle proof \rangle$

Version for Plaisted’s “Positive refinement” of the Meson procedure

lemma *make-refined-neg-rule*: $\neg P \vee Q \implies (P \implies Q)$
 $\langle proof \rangle$

P should be a literal

lemma *make-pos-rule*: $P \vee Q \implies ((P \implies \neg P) \implies Q)$
 $\langle proof \rangle$

Versions of *make-neg-rule* and *make-pos-rule* that don’t insert new assumptions, for ordinary resolution.

lemmas *make-neg-rule'* = *make-refined-neg-rule*

lemma *make-pos-rule'*: $\llbracket P \vee Q; \neg P \rrbracket \implies Q$
 $\langle proof \rangle$

Generation of a goal clause – put away the final literal

lemma *make-neg-goal*: $\neg P \implies ((\neg P \implies P) \implies False)$
 $\langle proof \rangle$

lemma *make-pos-goal*: $P \implies ((P \implies \neg P) \implies False)$
 $\langle proof \rangle$

39.3 Lemmas for Forward Proof

There is a similarity to congruence rules. They are also useful in ordinary proofs.

lemma *conj-forward*: $\llbracket P' \wedge Q'; P' \implies P; Q' \implies Q \rrbracket \implies P \wedge Q$
 $\langle proof \rangle$

lemma *disj-forward*: $\llbracket P \vee Q'; P' \Longrightarrow P; Q' \Longrightarrow Q \rrbracket \Longrightarrow P \vee Q$
 $\langle proof \rangle$

lemma *imp-forward*: $\llbracket P' \longrightarrow Q'; P \Longrightarrow P'; Q' \Longrightarrow Q \rrbracket \Longrightarrow P \longrightarrow Q$
 $\langle proof \rangle$

lemma *imp-forward2*: $\llbracket P' \longrightarrow Q'; P \Longrightarrow P'; P' \Longrightarrow Q' \Longrightarrow Q \rrbracket \Longrightarrow P \longrightarrow Q$
 $\langle proof \rangle$

lemma *disj-forward2*: $\llbracket P \vee Q'; P' \Longrightarrow P; \llbracket Q'; P \Longrightarrow False \rrbracket \Longrightarrow Q \rrbracket \Longrightarrow P \vee Q$
 $\langle proof \rangle$

lemma *all-forward*: $\llbracket \forall x. P'(x); !!x. P'(x) \Longrightarrow P(x) \rrbracket \Longrightarrow \forall x. P(x)$
 $\langle proof \rangle$

lemma *ex-forward*: $\llbracket \exists x. P'(x); !!x. P'(x) \Longrightarrow P(x) \rrbracket \Longrightarrow \exists x. P(x)$
 $\langle proof \rangle$

39.4 Clausification helper

lemma *TruepropI*: $P \equiv Q \Longrightarrow \text{Trueprop } P \equiv \text{Trueprop } Q$
 $\langle proof \rangle$

lemma *ext-cong-neq*: $F g \neq F h \Longrightarrow F g \neq F h \wedge (\exists x. g x \neq h x)$
 $\langle proof \rangle$

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* :: $('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ **where**
 $\text{COMBB } P \ Q \ R = P \ (Q \ R)$

definition *COMBC* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$ **where**
 $\text{COMBC } P \ Q \ R = P \ R \ Q$

definition *COMBS* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ **where**
 $\text{COMBS } P \ Q \ R = P \ R \ (Q \ R)$

lemma *abs-S*: $\lambda x. (f x) (g x) \equiv \text{COMBS } f \ g$
 $\langle proof \rangle$

lemma *abs-I*: $\lambda x. x \equiv \text{COMBI}$
 $\langle proof \rangle$

lemma *abs-K*: $\lambda x. y \equiv \text{COMBK } y$
 $\langle \text{proof} \rangle$

lemma *abs-B*: $\lambda x. a (g \ x) \equiv \text{COMBB } a \ g$
 $\langle \text{proof} \rangle$

lemma *abs-C*: $\lambda x. (f \ x) \ b \equiv \text{COMBC } f \ b$
 $\langle \text{proof} \rangle$

39.5 Skolemization helpers

definition *skolem* :: $'a \Rightarrow 'a$ **where**
skolem = $(\lambda x. x)$

lemma *skolem-COMBK-iff*: $P \longleftrightarrow \text{skolem } (\text{COMBK } P \ (i::\text{nat}))$
 $\langle \text{proof} \rangle$

lemmas *skolem-COMBK-I* = *iffD1* [*OF skolem-COMBK-iff*]

39.6 Meson package

$\langle \text{ML} \rangle$

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

$\langle \text{ML} \rangle$

40.2 Higher-order reasoning helpers

definition *fFalse* :: *bool* **where**
fFalse $\longleftrightarrow \text{False}$

definition *fTrue* :: *bool* **where**

$fTrue \longleftrightarrow True$

definition $fNot :: bool \Rightarrow bool$ **where**
 $fNot P \longleftrightarrow \neg P$

definition $fComp :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ **where**
 $fComp P = (\lambda x. \neg P x)$

definition $fconj :: bool \Rightarrow bool \Rightarrow bool$ **where**
 $fconj P Q \longleftrightarrow P \wedge Q$

definition $fdisj :: bool \Rightarrow bool \Rightarrow bool$ **where**
 $fdisj P Q \longleftrightarrow P \vee Q$

definition $fimplies :: bool \Rightarrow bool \Rightarrow bool$ **where**
 $fimplies P Q \longleftrightarrow (P \longrightarrow Q)$

definition $fAll :: ('a \Rightarrow bool) \Rightarrow bool$ **where**
 $fAll P \longleftrightarrow All P$

definition $fEx :: ('a \Rightarrow bool) \Rightarrow bool$ **where**
 $fEx P \longleftrightarrow Ex P$

definition $fequal :: 'a \Rightarrow 'a \Rightarrow bool$ **where**
 $fequal x y \longleftrightarrow (x = y)$

definition $fChoice :: ('a \Rightarrow bool) \Rightarrow 'a$ **where**
 $fChoice \equiv Hilbert-Choice.Eps$

lemma $fTrue-ne-fFalse$: $fFalse \neq fTrue$
 $\langle proof \rangle$

lemma $fNot-table$:
 $fNot fFalse = fTrue$
 $fNot fTrue = fFalse$
 $\langle proof \rangle$

lemma $fconj-table$:
 $fconj fFalse P = fFalse$
 $fconj P fFalse = fFalse$
 $fconj fTrue fTrue = fTrue$
 $\langle proof \rangle$

lemma $fdisj-table$:
 $fdisj fTrue P = fTrue$
 $fdisj P fTrue = fTrue$
 $fdisj fFalse fFalse = fFalse$
 $\langle proof \rangle$

lemma *fimplies-table:*

fimplies P *fTrue* = *fTrue*

fimplies *fFalse* P = *fTrue*

fimplies *fTrue* *fFalse* = *fFalse*

$\langle \text{proof} \rangle$

lemma *fAll-table:*

Ex (*fComp* P) \vee *fAll* P = *fTrue*

All $P \vee$ *fAll* P = *fFalse*

$\langle \text{proof} \rangle$

lemma *fEx-table:*

All (*fComp* P) \vee *fEx* P = *fTrue*

Ex $P \vee$ *fEx* P = *fFalse*

$\langle \text{proof} \rangle$

lemma *fequal-table:*

fequal x x = *fTrue*

$x = y \vee$ *fequal* x y = *fFalse*

$\langle \text{proof} \rangle$

lemma *fNot-law:*

fNot $P \neq P$

$\langle \text{proof} \rangle$

lemma *fComp-law:*

fComp P $x \longleftrightarrow \neg P$ x

$\langle \text{proof} \rangle$

lemma *fconj-laws:*

fconj P $P \longleftrightarrow P$

fconj P $Q \longleftrightarrow$ *fconj* Q P

fNot (*fconj* P Q) \longleftrightarrow *fdisj* (*fNot* P) (*fNot* Q)

$\langle \text{proof} \rangle$

lemma *fdisj-laws:*

fdisj P $P \longleftrightarrow P$

fdisj P $Q \longleftrightarrow$ *fdisj* Q P

fNot (*fdisj* P Q) \longleftrightarrow *fconj* (*fNot* P) (*fNot* Q)

$\langle \text{proof} \rangle$

lemma *fimplies-laws:*

fimplies P $Q \longleftrightarrow$ *fdisj* ($\neg P$) Q

fNot (*fimplies* P Q) \longleftrightarrow *fconj* P (*fNot* Q)

$\langle \text{proof} \rangle$

lemma *fAll-law:*

fNot (*fAll* R) \longleftrightarrow *fEx* (*fComp* R)

$\langle \text{proof} \rangle$

lemma *fEx-law*:
 $fNot (fEx R) \longleftrightarrow fAll (fComp R)$
 $\langle proof \rangle$

lemma *fequal-laws*:
 $fequal\ x\ y = fequal\ y\ x$
 $fequal\ x\ y = fFalse \vee fequal\ y\ z = fFalse \vee fequal\ x\ z = fTrue$
 $fequal\ x\ y = fFalse \vee fequal\ (f\ x)\ (f\ y) = fTrue$
 $\langle proof \rangle$

lemma *fChoice-iff-Ex*: $P (fChoice\ P) \longleftrightarrow HOL.Ex\ P$
 $\langle proof \rangle$

We use the *Ex* constant on the right-hand side of *fChoice-iff-Ex* because we want to use the TPTP-native version if *fChoice* is introduced in a logic that supports FOOL. In logics that don’t support it, it gets replaced by *fEx* during processing. Notice that we cannot use $\exists x. P\ x$, as existentials are not skolemized by the metis proof method but *Ex* *P* is eta-expanded if FOOL is supported.

40.3 Basic connection between ATPs and HOL

$\langle ML \rangle$

end

41 Metis Proof Method

theory *Metis*
imports *ATP*
begin

context notes $[[ML-catch-all]]$
begin
 $\langle ML \rangle$
end

41.1 Literal selection and lambda-lifting helpers

definition *select* :: $'a \Rightarrow 'a$ **where**
 $select = (\lambda x. x)$

lemma *not-atomize*: $(\neg A \Longrightarrow False) \equiv Trueprop\ A$
 $\langle proof \rangle$

lemma *atomize-not-select*: $(A \Longrightarrow select\ False) \equiv Trueprop\ (\neg A)$
 $\langle proof \rangle$

lemma *not-atomize-select*: $(\neg A \implies \text{select False}) \equiv \text{Trueprop } A$
 $\langle \text{proof} \rangle$

lemma *select-FalseI*: $\text{False} \implies \text{select False}$
 $\langle \text{proof} \rangle$

definition *lambda* :: $'a \Rightarrow 'a$ **where**
lambda = $(\lambda x. x)$

lemma *eq-lambdaI*: $x \equiv y \implies x \equiv \text{lambda } y$
 $\langle \text{proof} \rangle$

41.2 Metis package

$\langle \text{ML} \rangle$

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 se-*
lect-FalseI

fFalse-def fTrue-def fNot-def fconj-def fdisj-def fimplies-def fAll-def fEx-def fe-
qual-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-laws fdisj-laws fim-
plies-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** $\langle === \rangle$ 55)

notation *map-fun* (**infixr** $\langle --- \rangle$ 55)

end

context **includes** *lifting-syntax*

begin

lemma *rel-funD2*:

assumes *rel-fun* $A B f g$ **and** $A x x$

shows $B (f x) (g x)$
 $\langle proof \rangle$

lemma *rel-funE*:
assumes *rel-fun* $A B f g$ **and** $A x y$
obtains $B (f x) (g y)$
 $\langle proof \rangle$

lemmas *rel-fun-eq* = *fun.rel-eq*

lemma *rel-fun-eq-rel*:
shows *rel-fun* $(=) R = (\lambda f g. \forall x. R (f x) (g x))$
 $\langle proof \rangle$

42.2 Transfer method

Explicit tag for relation membership allows for backward proof methods.

definition *Rel* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
where *Rel* $r \equiv r$

Handling of equality relations

definition *is-equality* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where *is-equality* $R \longleftrightarrow R = (=)$

lemma *is-equality-eq*: *is-equality* $(=)$
 $\langle proof \rangle$

Reverse implication for monotonicity rules

definition *rev-implies* **where**
rev-implies $x y \longleftrightarrow (y \longrightarrow x)$

Handling of meta-logic connectives

definition *transfer-forall* **where**
transfer-forall $\equiv All$

definition *transfer-implies* **where**
transfer-implies $\equiv (\longrightarrow)$

definition *transfer-bforall* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where *transfer-bforall* $\equiv (\lambda P Q. \forall x. P x \longrightarrow Q x)$

lemma *transfer-forall-eq*: $(\bigwedge x. P x) \equiv Trueprop (transfer-forall (\lambda x. P x))$
 $\langle proof \rangle$

lemma *transfer-implies-eq*: $(A \Longrightarrow B) \equiv Trueprop (transfer-implies A B)$
 $\langle proof \rangle$

lemma *transfer-bforall-unfold*:

Trueprop (*transfer-bforall* $P (\lambda x. Q x)$) $\equiv (\bigwedge x. P x \implies Q x)$
 $\langle \text{proof} \rangle$

lemma *transfer-start*: $\llbracket P; \text{Rel } (=) P Q \rrbracket \implies Q$
 $\langle \text{proof} \rangle$

lemma *transfer-start'*: $\llbracket P; \text{Rel } (\longrightarrow) P Q \rrbracket \implies Q$
 $\langle \text{proof} \rangle$

lemma *transfer-prover-start*: $\llbracket x = x'; \text{Rel } R x' y \rrbracket \implies \text{Rel } R x y$
 $\langle \text{proof} \rangle$

lemma *untransfer-start*: $\llbracket Q; \text{Rel } (=) P Q \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *Rel-eq-reft*: $\text{Rel } (=) x x$
 $\langle \text{proof} \rangle$

lemma *Rel-app*:
 assumes $\text{Rel } (A ==> B) f g$ and $\text{Rel } A x y$
 shows $\text{Rel } B (f x) (g y)$
 $\langle \text{proof} \rangle$

lemma *Rel-abs*:
 assumes $\bigwedge x y. \text{Rel } A x y \implies \text{Rel } B (f x) (g y)$
 shows $\text{Rel } (A ==> B) (\lambda x. f x) (\lambda y. g y)$
 $\langle \text{proof} \rangle$

42.3 Predicates on relations, i.e. “class constraints”

See also *left-unique* and *right-unique*.

definition *left-total* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 where $\text{left-total } R \longleftrightarrow (\forall x. \exists y. R x y)$

definition *right-total* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 where $\text{right-total } R \longleftrightarrow (\forall y. \exists x. R x y)$

definition *bi-total* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 where $\text{bi-total } R \longleftrightarrow (\forall x. \exists y. R x y) \wedge (\forall y. \exists x. R x y)$

definition *bi-unique* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 where $\text{bi-unique } R \longleftrightarrow$
 $(\forall x y z. R x y \longrightarrow R x z \longrightarrow y = z) \wedge$
 $(\forall x y z. R x z \longrightarrow R y z \longrightarrow x = y)$

lemma *left-totalI*:
 $(\bigwedge x. \exists y. R x y) \implies \text{left-total } R$
 $\langle \text{proof} \rangle$

lemma *left-totalE*:
assumes *left-total R*
obtains $(\bigwedge x. \exists y. R\ x\ y)$
 $\langle proof \rangle$

lemma *bi-uniqueDr*: $\llbracket bi\text{-}unique\ A; A\ x\ y; A\ x\ z \rrbracket \implies y = z$
 $\langle proof \rangle$

lemma *bi-uniqueDl*: $\llbracket bi\text{-}unique\ A; A\ x\ y; A\ z\ y \rrbracket \implies x = z$
 $\langle proof \rangle$

lemma *bi-unique-iff*: $bi\text{-}unique\ R \longleftrightarrow (\forall z. \exists_{\leq 1} x. R\ x\ z) \wedge (\forall z. \exists_{\leq 1} x. R\ z\ x)$
 $\langle proof \rangle$

lemma *right-totalI*: $(\bigwedge y. \exists x. A\ x\ y) \implies right\text{-}total\ A$
 $\langle proof \rangle$

lemma *right-totalE*:
assumes *right-total A*
obtains x **where** $A\ x\ y$
 $\langle proof \rangle$

lemma *right-total-alt-def2*:
 $right\text{-}total\ R \longleftrightarrow ((R\ == => (\longrightarrow)) == => (\longrightarrow))\ All\ All\ (\mathbf{is}\ ?lhs = ?rhs)$
 $\langle proof \rangle$

lemma *right-unique-alt-def2*:
 $right\text{-}unique\ R \longleftrightarrow (R\ == => R\ == => (\longrightarrow))\ (=)\ (=)$
 $\langle proof \rangle$

lemma *bi-total-alt-def2*:
 $bi\text{-}total\ R \longleftrightarrow ((R\ == => (=)) == => (=))\ All\ All\ (\mathbf{is}\ ?lhs = ?rhs)$
 $\langle proof \rangle$

lemma *bi-unique-alt-def2*:
 $bi\text{-}unique\ R \longleftrightarrow (R\ == => R\ == => (=))\ (=)\ (=)$
 $\langle proof \rangle$

lemma [*simp*]:
shows *left-total-conversep*: $left\text{-}total\ A^{-1-1} \longleftrightarrow right\text{-}total\ A$
and *right-total-conversep*: $right\text{-}total\ A^{-1-1} \longleftrightarrow left\text{-}total\ A$
 $\langle proof \rangle$

lemma *bi-unique-conversep* [*simp*]: $bi\text{-}unique\ R^{-1-1} = bi\text{-}unique\ R$
 $\langle proof \rangle$

lemma *bi-total-conversep* [*simp*]: $bi\text{-}total\ R^{-1-1} = bi\text{-}total\ R$
 $\langle proof \rangle$

lemma *right-unique-alt-def*: $\text{right-unique } R = (\text{conversep } R \text{ OO } R \leq (=)) \langle \text{proof} \rangle$

lemma *left-unique-alt-def*: $\text{left-unique } R = (R \text{ OO } (\text{conversep } R) \leq (=)) \langle \text{proof} \rangle$

lemma *right-total-alt-def*: $\text{right-total } R = (\text{conversep } R \text{ OO } R \geq (=)) \langle \text{proof} \rangle$

lemma *left-total-alt-def*: $\text{left-total } R = (R \text{ OO } \text{conversep } R \geq (=)) \langle \text{proof} \rangle$

lemma *bi-total-alt-def*: $\text{bi-total } A = (\text{left-total } A \wedge \text{right-total } A) \langle \text{proof} \rangle$

lemma *bi-unique-alt-def*: $\text{bi-unique } A = (\text{left-unique } A \wedge \text{right-unique } A) \langle \text{proof} \rangle$

lemma *bi-totalI*: $\text{left-total } R \implies \text{right-total } R \implies \text{bi-total } R \langle \text{proof} \rangle$

lemma *bi-uniqueI*: $\text{left-unique } R \implies \text{right-unique } R \implies \text{bi-unique } R \langle \text{proof} \rangle$

end

lemma *is-equality-lemma*: $(\bigwedge R. \text{is-equality } R \implies \text{PROP } (P \ R)) \equiv \text{PROP } (P \ (=)) \langle \text{proof} \rangle$

lemma *Domainp-lemma*: $(\bigwedge R. \text{Domainp } T = R \implies \text{PROP } (P \ R)) \equiv \text{PROP } (P \ (\text{Domainp } T)) \langle \text{proof} \rangle$

$\langle ML \rangle$

declare *refl* [*transfer-rule*]

hide-const (**open**) *Rel*

context includes *lifting-syntax*

begin

Handling of domains

lemma *Domainp-iff*: $\text{Domainp } T \ x \longleftrightarrow (\exists y. T \ x \ y) \langle \text{proof} \rangle$

lemma *Domainp-refl*[*transfer-domain-rule*]:
 $\text{Domainp } T = \text{Domainp } T \langle \text{proof} \rangle$

lemma *Domain-eq-top*[*transfer-domain-rule*]: $\text{Domainp } (=) = \text{top} \langle \text{proof} \rangle$

lemma *Domainp-pred-fun-eq*[*relator-domain*]:

assumes *left-unique* *T*

shows $\text{Domainp } (T \implies S) = \text{pred-fun } (\text{Domainp } T) (\text{Domainp } S) \quad (\text{is } ?lhs = ?rhs)$

$\langle proof \rangle$

Properties are preserved by relation composition.

lemma *OO-def*: $R \text{ OO } S = (\lambda x z. \exists y. R \ x \ y \wedge S \ y \ z)$
 $\langle proof \rangle$

lemma *bi-total-OO*: $\llbracket bi\text{-total } A; bi\text{-total } B \rrbracket \implies bi\text{-total } (A \text{ OO } B)$
 $\langle proof \rangle$

lemma *bi-unique-OO*: $\llbracket bi\text{-unique } A; bi\text{-unique } B \rrbracket \implies bi\text{-unique } (A \text{ OO } B)$
 $\langle proof \rangle$

lemma *right-total-OO*:
 $\llbracket right\text{-total } A; right\text{-total } B \rrbracket \implies right\text{-total } (A \text{ OO } B)$
 $\langle proof \rangle$

lemma *right-unique-OO*:
 $\llbracket right\text{-unique } A; right\text{-unique } B \rrbracket \implies right\text{-unique } (A \text{ OO } B)$
 $\langle proof \rangle$

lemma *left-total-OO*: $left\text{-total } R \implies left\text{-total } S \implies left\text{-total } (R \text{ OO } S)$
 $\langle proof \rangle$

lemma *left-unique-OO*: $left\text{-unique } R \implies left\text{-unique } S \implies left\text{-unique } (R \text{ OO } S)$
 $\langle proof \rangle$

42.4 Properties of relators

lemma *left-total-eq[transfer-rule]*: $left\text{-total } (=)$
 $\langle proof \rangle$

lemma *left-unique-eq[transfer-rule]*: $left\text{-unique } (=)$
 $\langle proof \rangle$

lemma *right-total-eq [transfer-rule]*: $right\text{-total } (=)$
 $\langle proof \rangle$

lemma *right-unique-eq [transfer-rule]*: $right\text{-unique } (=)$
 $\langle proof \rangle$

lemma *bi-total-eq[transfer-rule]*: $bi\text{-total } (=)$
 $\langle proof \rangle$

lemma *bi-unique-eq[transfer-rule]*: $bi\text{-unique } (=)$
 $\langle proof \rangle$

lemma *left-total-fun[transfer-rule]*:
 assumes $left\text{-unique } A \text{ left-total } B$
 shows $left\text{-total } (A \implies B)$

$\langle \text{proof} \rangle$

lemma *left-unique-fun* [*transfer-rule*]:
 $\llbracket \text{left-total } A; \text{left-unique } B \rrbracket \implies \text{left-unique } (A \implies B)$
 $\langle \text{proof} \rangle$

lemma *right-total-fun* [*transfer-rule*]:
assumes *right-unique A right-total B*
shows *right-total (A \implies B)*
 $\langle \text{proof} \rangle$

lemma *right-unique-fun* [*transfer-rule*]:
 $\llbracket \text{right-total } A; \text{right-unique } B \rrbracket \implies \text{right-unique } (A \implies B)$
 $\langle \text{proof} \rangle$

lemma *bi-total-fun* [*transfer-rule*]:
 $\llbracket \text{bi-unique } A; \text{bi-total } B \rrbracket \implies \text{bi-total } (A \implies B)$
 $\langle \text{proof} \rangle$

lemma *bi-unique-fun* [*transfer-rule*]:
 $\llbracket \text{bi-total } A; \text{bi-unique } B \rrbracket \implies \text{bi-unique } (A \implies B)$
 $\langle \text{proof} \rangle$

end

lemma *if-conn*:
 $(\text{if } P \wedge Q \text{ then } t \text{ else } e) = (\text{if } P \text{ then if } Q \text{ then } t \text{ else } e \text{ else } e)$
 $(\text{if } P \vee Q \text{ then } t \text{ else } e) = (\text{if } P \text{ then } t \text{ else if } Q \text{ then } t \text{ else } e)$
 $(\text{if } P \longrightarrow Q \text{ then } t \text{ else } e) = (\text{if } P \text{ then if } Q \text{ then } t \text{ else } e \text{ else } t)$
 $(\text{if } \neg P \text{ then } t \text{ else } e) = (\text{if } P \text{ then } e \text{ else } t)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

declare *pred-fun-def* [*simp*]
declare *rel-fun-eq* [*relator-eq*]

declare *fun.Domainp-rel* [*relator-domain del*]

42.5 Transfer rules

context **includes** *lifting-syntax*
begin

lemma *Domainp-forall-transfer* [*transfer-rule*]:
assumes *right-total A*
shows $((A \implies (=)) \implies (=))$
 $(\text{transfer-bforall } (\text{Domainp } A)) \text{ transfer-forall}$

<proof>

Transfer rules using implication instead of equality on booleans.

lemma *transfer-forall-transfer* [transfer-rule]:

bi-total $A \implies ((A \implies (=)) \implies (=))$ *transfer-forall transfer-forall*
right-total $A \implies ((A \implies (=)) \implies \text{implies})$ *transfer-forall transfer-forall*
right-total $A \implies ((A \implies \text{implies}) \implies \text{implies})$ *transfer-forall transfer-forall*
bi-total $A \implies ((A \implies (=)) \implies \text{rev-implies})$ *transfer-forall transfer-forall*
bi-total $A \implies ((A \implies \text{rev-implies}) \implies \text{rev-implies})$ *transfer-forall transfer-forall*
<proof>

lemma *transfer-implies-transfer* [transfer-rule]:

$((=) \implies (=) \implies (=))$ *transfer-implies transfer-implies*
 $(\text{rev-implies} \implies \text{implies} \implies \text{implies})$ *transfer-implies transfer-implies*
 $(\text{rev-implies} \implies (=) \implies \text{implies})$ *transfer-implies transfer-implies*
 $((=) \implies \text{implies} \implies \text{implies})$ *transfer-implies transfer-implies*
 $((=) \implies (=) \implies \text{implies})$ *transfer-implies transfer-implies*
 $(\text{implies} \implies \text{rev-implies} \implies \text{rev-implies})$ *transfer-implies transfer-implies*
 $(\text{implies} \implies (=) \implies \text{rev-implies})$ *transfer-implies transfer-implies*
 $((=) \implies \text{rev-implies} \implies \text{rev-implies})$ *transfer-implies transfer-implies*
 $((=) \implies (=) \implies \text{rev-implies})$ *transfer-implies transfer-implies*
<proof>

lemma *eq-imp-transfer* [transfer-rule]:

right-unique $A \implies (A \implies A \implies (\longrightarrow)) (=) (=)$
<proof>

Transfer rules using equality.

lemma *left-unique-transfer* [transfer-rule]:

assumes *right-total* A
assumes *right-total* B
assumes *bi-unique* A
shows $((A \implies B \implies (=)) \implies \text{implies})$ *left-unique left-unique*
<proof>

lemma *eq-transfer* [transfer-rule]:

assumes *bi-unique* A
shows $(A \implies A \implies (=)) (=) (=)$
<proof>

lemma *right-total-Ex-transfer* [transfer-rule]:

assumes *right-total* A
shows $((A \implies (=)) \implies (=)) (B \text{ex } (\text{Collect } (\text{Domainp } A))) \text{Ex}$
<proof>

lemma *right-total-All-transfer* [transfer-rule]:

assumes *right-total* A

```

shows (( $A \implies (=)$ )  $\implies (=)$ ) ( $Ball\ (Collect\ (Domainp\ A))$ )  $All$ 
 $\langle proof \rangle$ 

context
  includes lifting-syntax
begin

lemma right-total-fun-eq-transfer:
  assumes [transfer-rule]: right-total A bi-unique B
  shows (( $A \implies B \implies (A \implies B) \implies (=)$ ) ( $\lambda f g. \forall x \in Collect(Domainp\ A). f\ x = g\ x$ )  $(=)$ )
   $\langle proof \rangle$ 

end

lemma All-transfer [transfer-rule]:
  assumes bi-total A
  shows (( $A \implies (=)$ )  $\implies (=)$ )  $All\ All$ 
   $\langle proof \rangle$ 

lemma Ex-transfer [transfer-rule]:
  assumes bi-total A
  shows (( $A \implies (=)$ )  $\implies (=)$ )  $Ex\ Ex$ 
   $\langle proof \rangle$ 

lemma Ex1-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-unique A bi-total A
  shows (( $A \implies (=)$ )  $\implies (=)$ )  $Ex1\ Ex1$ 
   $\langle proof \rangle$ 

declare If-transfer [transfer-rule]

lemma Let-transfer [transfer-rule]: ( $A \implies (A \implies B) \implies B$ )  $Let\ Let$ 
   $\langle proof \rangle$ 

declare id-transfer [transfer-rule]

declare comp-transfer [transfer-rule]

lemma curry-transfer [transfer-rule]:
  (( $rel\text{-}prod\ A\ B \implies C \implies A \implies B \implies C$ )  $curry\ curry$ )
   $\langle proof \rangle$ 

lemma fun-upd-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (( $A \implies B \implies A \implies B \implies A \implies B$ )  $fun\text{-}upd\ fun\text{-}upd$ )
   $\langle proof \rangle$ 

lemma case-nat-transfer [transfer-rule]:

```

$(A \text{ ===> } ((=) \text{ ===> } A) \text{ ===> } (=) \text{ ===> } A) \text{ case-nat case-nat}$
 $\langle \text{proof} \rangle$

lemma *rec-nat-transfer* [transfer-rule]:
 $(A \text{ ===> } ((=) \text{ ===> } A \text{ ===> } A) \text{ ===> } (=) \text{ ===> } A) \text{ rec-nat rec-nat}$
 $\langle \text{proof} \rangle$

lemma *funpow-transfer* [transfer-rule]:
 $((=) \text{ ===> } (A \text{ ===> } A) \text{ ===> } (A \text{ ===> } A)) \text{ compow compow}$
 $\langle \text{proof} \rangle$

lemma *mono-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-total* A
assumes [transfer-rule]: $(A \text{ ===> } A \text{ ===> } (=)) (\leq) (\leq)$
assumes [transfer-rule]: $(B \text{ ===> } B \text{ ===> } (=)) (\leq) (\leq)$
shows $((A \text{ ===> } B) \text{ ===> } (=)) \text{ mono mono}$
 $\langle \text{proof} \rangle$

lemma *right-total-relcompp-transfer* [transfer-rule]:
assumes [transfer-rule]: *right-total* B
shows $((A \text{ ===> } B \text{ ===> } (=)) \text{ ===> } (B \text{ ===> } C \text{ ===> } (=)) \text{ ===> } A \text{ ===> } C \text{ ===> } (=))$
 $(\lambda R \ S \ x \ z. \exists y \in \text{Collect } (\text{Domainp } B). R \ x \ y \wedge S \ y \ z) (OO)$
 $\langle \text{proof} \rangle$

lemma *relcompp-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-total* B
shows $((A \text{ ===> } B \text{ ===> } (=)) \text{ ===> } (B \text{ ===> } C \text{ ===> } (=)) \text{ ===> } A \text{ ===> } C \text{ ===> } (=)) (OO) (OO)$
 $\langle \text{proof} \rangle$

lemma *right-total-Domainp-transfer* [transfer-rule]:
assumes [transfer-rule]: *right-total* B
shows $((A \text{ ===> } B \text{ ===> } (=)) \text{ ===> } A \text{ ===> } (=)) (\lambda T \ x. \exists y \in \text{Collect } (\text{Domainp } B). T \ x \ y) \text{ Domainp}$
 $\langle \text{proof} \rangle$

lemma *Domainp-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-total* B
shows $((A \text{ ===> } B \text{ ===> } (=)) \text{ ===> } A \text{ ===> } (=)) \text{ Domainp Domainp}$
 $\langle \text{proof} \rangle$

lemma *reflp-transfer* [transfer-rule]:
bi-total $A \implies ((A \text{ ===> } A \text{ ===> } (=)) \text{ ===> } (=)) \text{ reflp reflp}$
right-total $A \implies ((A \text{ ===> } A \text{ ===> } \text{implies}) \text{ ===> } \text{implies}) \text{ reflp reflp}$
right-total $A \implies ((A \text{ ===> } A \text{ ===> } (=)) \text{ ===> } \text{implies}) \text{ reflp reflp}$
bi-total $A \implies ((A \text{ ===> } A \text{ ===> } \text{rev-implies}) \text{ ===> } \text{rev-implies}) \text{ reflp reflp}$
bi-total $A \implies ((A \text{ ===> } A \text{ ===> } (=)) \text{ ===> } \text{rev-implies}) \text{ reflp reflp}$

$\langle \text{proof} \rangle$

lemma *right-unique-transfer* [transfer-rule]:

[[*right-total* A ; *right-total* B ; *bi-unique* B]]

$\implies ((A \implies B \implies (=)) \implies \text{implies})$ *right-unique right-unique*

$\langle \text{proof} \rangle$

lemma *left-total-parametric* [transfer-rule]:

assumes [transfer-rule]: *bi-total* A *bi-total* B

shows $((A \implies B \implies (=)) \implies (=))$ *left-total left-total*

$\langle \text{proof} \rangle$

lemma *right-total-parametric* [transfer-rule]:

assumes [transfer-rule]: *bi-total* A *bi-total* B

shows $((A \implies B \implies (=)) \implies (=))$ *right-total right-total*

$\langle \text{proof} \rangle$

lemma *left-unique-parametric* [transfer-rule]:

assumes [transfer-rule]: *bi-unique* A *bi-total* A *bi-total* B

shows $((A \implies B \implies (=)) \implies (=))$ *left-unique left-unique*

$\langle \text{proof} \rangle$

lemma *prod-pred-parametric* [transfer-rule]:

$((A \implies (=)) \implies (B \implies (=)) \implies \text{rel-prod } A \ B \implies (=))$

pred-prod pred-prod

$\langle \text{proof} \rangle$

lemma *apfst-parametric* [transfer-rule]:

$((A \implies B) \implies \text{rel-prod } A \ C \implies \text{rel-prod } B \ C)$ *apfst apfst*

$\langle \text{proof} \rangle$

lemma *rel-fun-eq-onp*: $((=) \implies \text{eq-onp } P) = \text{eq-onp } (\lambda f. \forall x. P(f\ x))$

$\langle \text{proof} \rangle$

lemma *rel-fun-eq-onp-rel*:

shows $((\text{eq-onp } R) \implies S) = (\lambda f\ g. \forall x. R\ x \longrightarrow S\ (f\ x)\ (g\ x))$

$\langle \text{proof} \rangle$

lemma *eq-onp-transfer* [transfer-rule]:

assumes [transfer-rule]: *bi-unique* A

shows $((A \implies (=)) \implies A \implies A \implies (=))$ *eq-onp eq-onp*

$\langle \text{proof} \rangle$

lemma *rtrancp-parametric* [transfer-rule]:

assumes *bi-unique* A *bi-total* A

shows $((A \implies A \implies (=)) \implies A \implies A \implies (=))$ *rtrancp*

rtrancp

$\langle \text{proof} \rangle$

```

lemma right-unique-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-total A bi-unique B bi-total B
  shows  $((A \text{ ===> } B \text{ ===> } (=)) \text{ ===> } (=))$  right-unique right-unique
   $\langle \text{proof} \rangle$ 

```

```

lemma map-fun-parametric [transfer-rule]:
   $((A \text{ ===> } B) \text{ ===> } (C \text{ ===> } D) \text{ ===> } (B \text{ ===> } C) \text{ ===> } A \text{ ===> } D)$  map-fun map-fun
   $\langle \text{proof} \rangle$ 

```

end

42.6 *of-bool* and *of-nat*

```

context
  includes lifting-syntax
begin

```

```

lemma transfer-rule-of-bool:
   $\langle ((\longleftrightarrow) \text{ ===> } (\cong)) \text{ of-bool of-bool} \rangle$ 
  if [transfer-rule]:  $\langle 0 \cong 0 \rangle \langle 1 \cong 1 \rangle$ 
  for  $R :: \langle 'a :: \text{zero-neq-one} \Rightarrow 'b :: \text{zero-neq-one} \Rightarrow \text{bool} \rangle$  (infix  $\langle \cong \rangle$  50)
   $\langle \text{proof} \rangle$ 

```

```

lemma transfer-rule-of-nat:
   $((=) \text{ ===> } (\cong)) \text{ of-nat of-nat}$ 
  if [transfer-rule]:  $\langle 0 \cong 0 \rangle \langle 1 \cong 1 \rangle$ 
   $\langle ((\cong) \text{ ===> } (\cong) \text{ ===> } (\cong)) (+) (+) \rangle$ 
  for  $R :: \langle 'a :: \text{semiring-1} \Rightarrow 'b :: \text{semiring-1} \Rightarrow \text{bool} \rangle$  (infix  $\langle \cong \rangle$  50)
   $\langle \text{proof} \rangle$ 

```

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*:
 $(id \dashrightarrow id) = id$
 $\langle proof \rangle$

43.2 Quotient Predicate

definition *Quotient* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$

where

$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
 $(\forall a. Abs\ (Rep\ a) = a) \wedge$
 $(\forall a. R\ (Rep\ a)\ (Rep\ a)) \wedge$
 $(\forall r\ s. R\ r\ s \longleftrightarrow R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s) \wedge$
 $T = (\lambda x\ y. R\ x\ x \wedge Abs\ x = y)$

lemma *QuotientI*:

assumes $\bigwedge a. Abs\ (Rep\ a) = a$
and $\bigwedge a. R\ (Rep\ a)\ (Rep\ a)$
and $\bigwedge r\ s. R\ r\ s \longleftrightarrow R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s$
and $T = (\lambda x\ y. R\ x\ x \wedge Abs\ x = y)$
shows $Quotient\ R\ Abs\ Rep\ T$
 $\langle proof \rangle$

context

fixes $R\ Abs\ Rep\ T$
assumes $a: Quotient\ R\ Abs\ Rep\ T$
begin

lemma *Quotient-abs-rep*: $Abs\ (Rep\ a) = a$
 $\langle proof \rangle$

lemma *Quotient-rep-reflp*: $R\ (Rep\ a)\ (Rep\ a)$
 $\langle proof \rangle$

lemma *Quotient-rel*:

$R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s \longleftrightarrow R\ r\ s$ — orientation does not loop on rewriting
 $\langle proof \rangle$

lemma *Quotient-cr-rel*: $T = (\lambda x\ y. R\ x\ x \wedge Abs\ x = y)$
 $\langle proof \rangle$

lemma *Quotient-refl1*: $R\ r\ s \Longrightarrow R\ r\ r$
 $\langle proof \rangle$

lemma *Quotient-refl2*: $R\ r\ s \Longrightarrow R\ s\ s$
 $\langle proof \rangle$

lemma *Quotient-rel-rep*: $R \text{ (Rep } a) \text{ (Rep } b) \longleftrightarrow a = b$
 $\langle \text{proof} \rangle$

lemma *Quotient-rep-abs*: $R \text{ } r \text{ } r \implies R \text{ (Rep (Abs } r)) \text{ } r$
 $\langle \text{proof} \rangle$

lemma *Quotient-rep-abs-eq*: $R \text{ } t \text{ } t \implies R \leq (=) \implies \text{Rep (Abs } t) = t$
 $\langle \text{proof} \rangle$

lemma *Quotient-rep-abs-fold-unmap*:
 assumes $x' \equiv \text{Abs } x$ and $R \text{ } x \text{ } x$ and $\text{Rep } x' \equiv \text{Rep}' x'$
 shows $R \text{ (Rep}' x') \text{ } x$
 $\langle \text{proof} \rangle$

lemma *Quotient-Rep-eq*:
 assumes $x' \equiv \text{Abs } x$
 shows $\text{Rep } x' \equiv \text{Rep } x'$
 $\langle \text{proof} \rangle$

lemma *Quotient-rel-abs*: $R \text{ } r \text{ } s \implies \text{Abs } r = \text{Abs } s$
 $\langle \text{proof} \rangle$

lemma *Quotient-rel-abs2*:
 assumes $R \text{ (Rep } x) \text{ } y$
 shows $x = \text{Abs } y$
 $\langle \text{proof} \rangle$

lemma *Quotient-symp*: $\text{symp } R$
 $\langle \text{proof} \rangle$

lemma *Quotient-transp*: $\text{transp } R$
 $\langle \text{proof} \rangle$

lemma *Quotient-part-equivp*: $\text{part-equivp } R$
 $\langle \text{proof} \rangle$

end

lemma *identity-quotient*: $\text{Quotient } (=) \text{ id id } (=)$
 $\langle \text{proof} \rangle$

TODO: Use one of these alternatives as the real definition.

lemma *Quotient-alt-def*:
 $\text{Quotient } R \text{ Abs Rep } T \longleftrightarrow$
 $(\forall a \text{ } b. T \text{ } a \text{ } b \longrightarrow \text{Abs } a = b) \wedge$
 $(\forall b. T \text{ (Rep } b) \text{ } b) \wedge$
 $(\forall x \text{ } y. R \text{ } x \text{ } y \longleftrightarrow T \text{ } x \text{ (Abs } x) \wedge T \text{ } y \text{ (Abs } y) \wedge \text{Abs } x = \text{Abs } y)$
 $\langle \text{proof} \rangle$

lemma *Quotient-alt-def2:*

Quotient R Abs Rep T \longleftrightarrow
 $(\forall a\ b. T\ a\ b \longrightarrow Abs\ a = b) \wedge$
 $(\forall b. T\ (Rep\ b)\ b) \wedge$
 $(\forall x\ y. R\ x\ y \longleftrightarrow T\ x\ (Abs\ y) \wedge T\ y\ (Abs\ x))$
 $\langle proof \rangle$

lemma *Quotient-alt-def3:*

Quotient R Abs Rep T \longleftrightarrow
 $(\forall a\ b. T\ a\ b \longrightarrow Abs\ a = b) \wedge (\forall b. T\ (Rep\ b)\ b) \wedge$
 $(\forall x\ y. R\ x\ y \longleftrightarrow (\exists z. T\ x\ z \wedge T\ y\ z))$
 $\langle proof \rangle$

lemma *Quotient-alt-def4:*

Quotient R Abs Rep T \longleftrightarrow
 $(\forall a\ b. T\ a\ b \longrightarrow Abs\ a = b) \wedge (\forall b. T\ (Rep\ b)\ b) \wedge R = T\ OO\ conversep\ T$
 $\langle proof \rangle$

lemma *Quotient-alt-def5:*

Quotient R Abs Rep T \longleftrightarrow
 $T \leq BNF-Def.Grp\ UNIV\ Abs \wedge BNF-Def.Grp\ UNIV\ Rep \leq T^{-1-1} \wedge R = T$
 $OO\ T^{-1-1}$
 $\langle proof \rangle$

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)*
 $\langle proof \rangle$

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 $R2\ (f\ x)\ (g\ y)$
 $\langle proof \rangle$

lemma *apply-rsp':*

assumes $a: (R1 ==> R2)\ f\ g\ R1\ x\ y$
shows $R2\ (f\ x)\ (g\ y)$
 $\langle proof \rangle$

lemma *apply-rsp'':*

assumes *Quotient R Abs Rep T*
and $(R ==> S)\ f\ f$
shows $S\ (f\ (Rep\ x))\ (f\ (Rep\ x))$
 $\langle proof \rangle$

43.3 Quotient composition

lemma *Quotient-compose:*

assumes 1: *Quotient* $R1\ Abs1\ Rep1\ T1$

assumes 2: *Quotient* $R2\ Abs2\ Rep2\ T2$

shows *Quotient* $(T1\ OO\ R2\ OO\ conversep\ T1)\ (Abs2\ \circ\ Abs1)\ (Rep1\ \circ\ Rep2)$
 $(T1\ OO\ T2)$

<proof>

lemma *equivp-reflp2:*

equivp $R \implies reflp\ R$

<proof>

43.4 Respects predicate

definition *Respects* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ set$

where *Respects* $R = \{x.\ R\ x\ x\}$

lemma *in-respects:* $x \in Respects\ R \longleftrightarrow R\ x\ x$

<proof>

lemma *UNIV-typedef-to-Quotient:*

assumes *type-definition* $Rep\ Abs\ UNIV$

and *T-def:* $T \equiv (\lambda x\ y.\ x = Rep\ y)$

shows *Quotient* $(=)\ Abs\ Rep\ T$

<proof>

lemma *UNIV-typedef-to-equivp:*

fixes $Abs :: 'a \Rightarrow 'b$

and $Rep :: 'b \Rightarrow 'a$

assumes *type-definition* $Rep\ Abs\ (UNIV :: 'a\ set)$

shows *equivp* $((=) :: 'a \Rightarrow 'a \Rightarrow bool)$

<proof>

lemma *typedef-to-Quotient:*

assumes *type-definition* $Rep\ Abs\ S$

and *T-def:* $T \equiv (\lambda x\ y.\ x = Rep\ y)$

shows *Quotient* $(eq-onp\ (\lambda x.\ x \in S))\ Abs\ Rep\ T$

<proof>

lemma *typedef-to-part-equivp:*

assumes *type-definition* $Rep\ Abs\ S$

shows *part-equivp* $(eq-onp\ (\lambda x.\ x \in S))$

<proof>

lemma *open-typedef-to-Quotient:*

assumes *type-definition* $Rep\ Abs\ \{x.\ P\ x\}$

and *T-def:* $T \equiv (\lambda x\ y.\ x = Rep\ y)$

shows *Quotient* $(eq-onp\ P)\ Abs\ Rep\ T$

<proof>

lemma *open-typedef-to-part-equivp*:
assumes *type-definition Rep Abs* $\{x. P\ x\}$
shows *part-equivp* (*eq-onp P*)
 $\langle proof \rangle$

lemma *type-definition-Quotient-not-empty*: *Quotient* (*eq-onp P*) *Abs Rep T* \implies
 $\exists x. P\ x$
 $\langle proof \rangle$

lemma *type-definition-Quotient-not-empty-witness*: *Quotient* (*eq-onp P*) *Abs Rep*
 $T \implies P\ (Rep\ undefined)$
 $\langle proof \rangle$

Generating transfer rules for quotients.

context
fixes *R Abs Rep T*
assumes *1: Quotient R Abs Rep T*
begin

lemma *Quotient-right-unique: right-unique T*
 $\langle proof \rangle$

lemma *Quotient-right-total: right-total T*
 $\langle proof \rangle$

lemma *Quotient-rel-eq-transfer*: $(T \implies T \implies (=))\ R\ (=)$
 $\langle proof \rangle$

lemma *Quotient-abs-induct*:
assumes $\bigwedge y. R\ y\ y \implies P\ (Abs\ y)$ **shows** $P\ x$
 $\langle proof \rangle$

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: left-total T*
 $\langle proof \rangle$

lemma *Quotient-bi-total: bi-total T*
 $\langle proof \rangle$

lemma *Quotient-id-abs-transfer*: $((=) \implies T)\ (\lambda x. x)\ Abs$
 $\langle proof \rangle$

lemma *Quotient-total-abs-induct*: $(\bigwedge y. P (Abs\ y)) \implies P\ x$
 $\langle proof \rangle$

lemma *Quotient-total-abs-eq-iff*: $Abs\ x = Abs\ y \longleftrightarrow R\ x\ y$
 $\langle proof \rangle$

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: left-unique T*
 $\langle proof \rangle$

lemma *typedef-bi-unique: bi-unique T*
 $\langle proof \rangle$

lemma *typedef-right-unique: right-unique T*
 $\langle proof \rangle$

lemma *typedef-right-total: right-total T*
 $\langle proof \rangle$

lemma *typedef-rep-transfer: $(T \implies\implies\implies (=)) (\lambda x. x)\ Rep$*
 $\langle proof \rangle$

end

Generating the correspondence rule for a constant defined with *lift-definition*.

lemma *Quotient-to-transfer*:
 assumes *Quotient R Abs Rep T and $R\ c\ c$ and $c' \equiv Abs\ c$*
 shows *$T\ c\ c'$*
 $\langle proof \rangle$

Proving reflexivity

lemma *Quotient-to-left-total*:
 assumes *q: Quotient R Abs Rep T*
 and *r-R: reflp R*
 shows *left-total T*
 $\langle proof \rangle$

lemma *Quotient-composition-ge-eq*:

assumes *left-total* T
assumes $R \geq (=)$
shows $(T \circ\circ R \circ\circ T^{-1-1}) \geq (=)$
 $\langle proof \rangle$

lemma *Quotient-composition-le-eq*:
assumes *left-unique* T
assumes $R \leq (=)$
shows $(T \circ\circ R \circ\circ T^{-1-1}) \leq (=)$
 $\langle proof \rangle$

lemma *eq-onp-le-eq*:
 $eq\text{-}onp\ P \leq (=) \langle proof \rangle$

lemma *reflp-ge-eq*:
 $reflp\ R \implies R \geq (=) \langle proof \rangle$

Proving a parametrized correspondence relation

definition $POS :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
 $POS\ A\ B \equiv A \leq B$

definition $NEG :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
 $NEG\ A\ B \equiv B \leq A$

lemma *pos-OO-eq*:
shows $POS\ (A \circ\circ (=))\ A$
 $\langle proof \rangle$

lemma *pos-eq-OO*:
shows $POS\ ((=) \circ\circ A)\ A$
 $\langle proof \rangle$

lemma *neg-OO-eq*:
shows $NEG\ (A \circ\circ (=))\ A$
 $\langle proof \rangle$

lemma *neg-eq-OO*:
shows $NEG\ ((=) \circ\circ A)\ A$
 $\langle proof \rangle$

lemma *POS-trans*:
assumes $POS\ A\ B$
assumes $POS\ B\ C$
shows $POS\ A\ C$
 $\langle proof \rangle$

lemma *NEG-trans*:
assumes $NEG\ A\ B$
assumes $NEG\ B\ C$

shows $NEG\ A\ C$
 $\langle proof \rangle$

lemma *POS-NEG*:
 $POS\ A\ B \equiv NEG\ B\ A$
 $\langle proof \rangle$

lemma *NEG-POS*:
 $NEG\ A\ B \equiv POS\ B\ A$
 $\langle proof \rangle$

lemma *POS-pcr-rule*:
assumes $POS\ (A\ OO\ B)\ C$
shows $POS\ (A\ OO\ B\ OO\ X)\ (C\ OO\ X)$
 $\langle proof \rangle$

lemma *NEG-pcr-rule*:
assumes $NEG\ (A\ OO\ B)\ C$
shows $NEG\ (A\ OO\ B\ OO\ X)\ (C\ OO\ X)$
 $\langle proof \rangle$

lemma *POS-apply*:
assumes $POS\ R\ R'$
assumes $R\ f\ g$
shows $R'\ f\ g$
 $\langle proof \rangle$

Proving a parametrized correspondence relation

lemma *fun-mono*:
assumes $A \geq C$
assumes $B \leq D$
shows $(A \implies B) \leq (C \implies D)$
 $\langle proof \rangle$

lemma *pos-fun-distr*: $((R \implies S)\ OO\ (R' \implies S')) \leq ((R\ OO\ R') \implies (S\ OO\ S'))$
 $\langle proof \rangle$

lemma *functional-relation*: $right\text{-}unique\ R \implies left\text{-}total\ R \implies \forall x. \exists! y. R\ x\ y$
 $\langle proof \rangle$

lemma *functional-converse-relation*: $left\text{-}unique\ R \implies right\text{-}total\ R \implies \forall y. \exists! x. R\ x\ y$
 $\langle proof \rangle$

lemma *neg-fun-distr1*:
assumes 1: $left\text{-}unique\ R\ right\text{-}total\ R$
assumes 2: $right\text{-}unique\ R'\ left\text{-}total\ R'$
shows $(R\ OO\ R' \implies S\ OO\ S') \leq ((R \implies S)\ OO\ (R' \implies S'))$

$\langle \text{proof} \rangle$

lemma *neg-fun-distr2*:

assumes *1*: *right-unique* R' *left-total* R'

assumes *2*: *left-unique* S' *right-total* S'

shows $(R \text{ OO } R' \implies S \text{ OO } S') \leq ((R \implies S) \text{ OO } (R' \implies S'))$

$\langle \text{proof} \rangle$

43.5 Domains

lemma *composed-equiv-rel-eq-onp*:

assumes *left-unique* R

assumes $(R \implies (=)) \ P \ P'$

assumes *Domainp* $R = P''$

shows $(R \text{ OO } \text{eq-onp } P' \text{ OO } R^{-1-1}) = \text{eq-onp } (\inf P'' \ P)$

$\langle \text{proof} \rangle$

lemma *composed-equiv-rel-eq-eq-onp*:

assumes *left-unique* R

assumes *Domainp* $R = P$

shows $(R \text{ OO } (=) \text{ OO } R^{-1-1}) = \text{eq-onp } P$

$\langle \text{proof} \rangle$

lemma *pcr-Domainp-par-left-total*:

assumes *Domainp* $B = P$

assumes *left-total* A

assumes $(A \implies (=)) \ P' \ P$

shows *Domainp* $(A \text{ OO } B) = P'$

$\langle \text{proof} \rangle$

lemma *pcr-Domainp-par*:

assumes *Domainp* $B = P2$

assumes *Domainp* $A = P1$

assumes $(A \implies (=)) \ P2' \ P2$

shows *Domainp* $(A \text{ OO } B) = (\inf P1 \ P2')$

$\langle \text{proof} \rangle$

definition *rel-pred-comp* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow \text{bool}$

where *rel-pred-comp* $R \ P \equiv \lambda x. \exists y. R \ x \ y \wedge P \ y$

lemma *pcr-Domainp*:

assumes *Domainp* $B = P$

shows *Domainp* $(A \text{ OO } B) = (\lambda x. \exists y. A \ x \ y \wedge P \ y)$

$\langle \text{proof} \rangle$

lemma *pcr-Domainp-total*:

assumes *left-total* B

assumes *Domainp* $A = P$

shows *Domainp* $(A \text{ OO } B) = P$

<proof>

lemma *Quotient-to-Domainp*:

assumes *Quotient R Abs Rep T*

shows *Domainp T = ($\lambda x. R x x$)*

<proof>

lemma *eq-onp-to-Domainp*:

assumes *Quotient (eq-onp P) Abs Rep T*

shows *Domainp T = P*

<proof>

end

lemma *right-total-UNIV-transfer*:

assumes *right-total A*

shows *(rel-set A) (Collect (Domainp A)) UNIV*

<proof>

43.6 ML setup

<ML>

named-theorems *relator-eq-onp*

theorems that a relator of an eq-onp is an eq-onp of the corresponding predicate

<ML>

declare *fun-quotient[quot-map]*

declare *fun-mono[relator-mono]*

lemmas *[relator-distr] = pos-fun-distr neg-fun-distr1 neg-fun-distr2*

<ML>

lemma *pred-prod-beta*: *pred-prod P Q xy \longleftrightarrow P (fst xy) \wedge Q (snd xy)*

<proof>

lemma *pred-prod-split*: *P (pred-prod Q R xy) \longleftrightarrow ($\forall x y. xy = (x, y) \longrightarrow P (Q x \wedge R y)$)*

<proof>

hide-const (**open**) *POS NEG*

end

44 Definition of Quotient Types

theory *Quotient*

imports *Lifting*

keywords

print-quotmapsQ3 print-quotientsQ3 print-quotconsts :: diag and
quotient-type :: thy-goal-defn and / and
quotient-definition :: thy-goal-defn and
copy-bnf :: thy-defn and
lift-bnf :: thy-goal-defn

begin

Basic definition for equivalence relations that are represented by predicates.

Composition of Relations

abbreviation

rel-conj :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool (infixr 'OOO'
75)

where

r1 OOO r2 \equiv r1 OO r2 OO r1

lemma *eq-comp-r*:

shows $((=) \text{ OOO } R) = R$
<proof>

context *includes lifting-syntax*

begin

44.1 Quotient Predicate

definition

Quotient3 R Abs Rep \longleftrightarrow
 $(\forall a. \text{Abs } (\text{Rep } a) = a) \wedge (\forall a. R (\text{Rep } a) (\text{Rep } a)) \wedge$
 $(\forall r s. R r s \longleftrightarrow R r r \wedge R s s \wedge \text{Abs } r = \text{Abs } s)$

lemma *Quotient3I*:

assumes $\bigwedge a. \text{Abs } (\text{Rep } a) = a$
and $\bigwedge a. R (\text{Rep } a) (\text{Rep } a)$
and $\bigwedge r s. R r s \longleftrightarrow R r r \wedge R s s \wedge \text{Abs } r = \text{Abs } s$
shows *Quotient3 R Abs Rep*
<proof>

context

fixes *R Abs Rep*

assumes *a: Quotient3 R Abs Rep*

begin

lemma *Quotient3-abs-rep*:

Abs (Rep a) = a
<proof>

lemma *Quotient3-rep-refl*:

R (Rep a) (Rep a)

$\langle \text{proof} \rangle$

lemma *Quotient3-rel*:

$R\ r\ r \wedge R\ s\ s \wedge \text{Abs}\ r = \text{Abs}\ s \longleftrightarrow R\ r\ s$ — orientation does not loop on rewriting

$\langle \text{proof} \rangle$

lemma *Quotient3-refl1*:

$R\ r\ s \Longrightarrow R\ r\ r$

$\langle \text{proof} \rangle$

lemma *Quotient3-refl2*:

$R\ r\ s \Longrightarrow R\ s\ s$

$\langle \text{proof} \rangle$

lemma *Quotient3-rel-rep*:

$R\ (\text{Rep}\ a)\ (\text{Rep}\ b) \longleftrightarrow a = b$

$\langle \text{proof} \rangle$

lemma *Quotient3-rep-abs*:

$R\ r\ r \Longrightarrow R\ (\text{Rep}\ (\text{Abs}\ r))\ r$

$\langle \text{proof} \rangle$

lemma *Quotient3-rel-abs*:

$R\ r\ s \Longrightarrow \text{Abs}\ r = \text{Abs}\ s$

$\langle \text{proof} \rangle$

lemma *Quotient3-symp*:

$\text{symp}\ R$

$\langle \text{proof} \rangle$

lemma *Quotient3-transp*:

$\text{transp}\ R$

$\langle \text{proof} \rangle$

lemma *Quotient3-part-equivp*:

$\text{part-equivp}\ R$

$\langle \text{proof} \rangle$

lemma *abs-o-rep*:

$\text{Abs} \circ \text{Rep} = \text{id}$

$\langle \text{proof} \rangle$

lemma *equals-rsp*:

assumes $b: R\ x\ a\ x\ b\ R\ y\ a\ y\ b$

shows $R\ x\ a\ y\ a = R\ x\ b\ y\ b$

$\langle \text{proof} \rangle$

lemma *rep-abs-rsp*:

assumes $b: R\ x\ 1\ x\ 2$

```

shows  $R\ x1\ (Rep\ (Abs\ x2))$ 
 $\langle proof \rangle$ 

lemma rep-abs-rsp-left:
  assumes  $b: R\ x1\ x2$ 
  shows  $R\ (Rep\ (Abs\ x1))\ x2$ 
   $\langle proof \rangle$ 

end

lemma identity-quotient3:
   $Quotient3\ (=)\ id\ id$ 
   $\langle proof \rangle$ 

lemma fun-quotient3:
  assumes  $q1: Quotient3\ R1\ abs1\ rep1$ 
  and  $q2: Quotient3\ R2\ abs2\ rep2$ 
  shows  $Quotient3\ (R1\ ==\>\ R2)\ (rep1\ ---->\ abs2)\ (abs1\ ---->\ rep2)$ 
   $\langle proof \rangle$ 

lemma lambda-prs:
  assumes  $q1: Quotient3\ R1\ Abs1\ Rep1$ 
  and  $q2: Quotient3\ R2\ Abs2\ Rep2$ 
  shows  $(Rep1\ ---->\ Abs2)\ (\lambda x. Rep2\ (f\ (Abs1\ x))) = (\lambda x. f\ x)$ 
   $\langle proof \rangle$ 

lemma lambda-prs1:
  assumes  $q1: Quotient3\ R1\ Abs1\ Rep1$ 
  and  $q2: Quotient3\ R2\ Abs2\ Rep2$ 
  shows  $(Rep1\ ---->\ Abs2)\ (\lambda x. (Abs1\ ---->\ Rep2)\ f\ x) = (\lambda x. f\ x)$ 
   $\langle proof \rangle$ 

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  $R2\ (f\ x)\ (g\ y)$ 
   $\langle proof \rangle$ 

lemma apply-rspQ3'':
  assumes  $Quotient3\ R\ Abs\ Rep$ 
  and  $(R\ ==\>\ S)\ f\ f$ 
  shows  $S\ (f\ (Rep\ x))\ (f\ (Rep\ x))$ 
   $\langle proof \rangle$ 

```

44.2 lemmas for regularisation of ball and bex

lemma *ball-reg-equiv*:

fixes $P :: 'a \Rightarrow \text{bool}$

assumes a : *equivp* R

shows $\text{Ball } (\text{Respects } R) P = (\text{All } P)$

$\langle \text{proof} \rangle$

lemma *bex-reg-equiv*:

fixes $P :: 'a \Rightarrow \text{bool}$

assumes a : *equivp* R

shows $\text{Bex } (\text{Respects } R) P = (\text{Ex } P)$

$\langle \text{proof} \rangle$

lemma *ball-reg-right*:

assumes a : $\bigwedge x. x \in R \Longrightarrow P x \longrightarrow Q x$

shows $\text{All } P \longrightarrow \text{Ball } R Q$

$\langle \text{proof} \rangle$

lemma *bex-reg-left*:

assumes a : $\bigwedge x. x \in R \Longrightarrow Q x \longrightarrow P x$

shows $\text{Bex } R Q \longrightarrow \text{Ex } P$

$\langle \text{proof} \rangle$

lemma *ball-reg-left*:

assumes a : *equivp* R

shows $(\bigwedge x. (Q x \longrightarrow P x)) \Longrightarrow \text{Ball } (\text{Respects } R) Q \longrightarrow \text{All } P$

$\langle \text{proof} \rangle$

lemma *bex-reg-right*:

assumes a : *equivp* R

shows $(\bigwedge x. (Q x \longrightarrow P x)) \Longrightarrow \text{Ex } Q \longrightarrow \text{Bex } (\text{Respects } R) P$

$\langle \text{proof} \rangle$

lemma *ball-reg-equiv-range*:

fixes $P :: 'a \Rightarrow \text{bool}$

and $x :: 'a$

assumes a : *equivp* $R2$

shows $(\text{Ball } (\text{Respects } (R1 ==> R2)) (\lambda f. P (f x))) = \text{All } (\lambda f. P (f x))$

$\langle \text{proof} \rangle$

lemma *bex-reg-equiv-range*:

assumes a : *equivp* $R2$

shows $(\text{Bex } (\text{Respects } (R1 ==> R2)) (\lambda f. P (f x))) = \text{Ex } (\lambda f. P (f x))$

$\langle \text{proof} \rangle$

lemma *all-reg*:

assumes a : $\forall x :: 'a. (P x \longrightarrow Q x)$

and b : $\text{All } P$

shows $All\ Q$
 $\langle proof \rangle$

lemma *ex-reg*:
assumes $a: \forall x :: 'a. (P\ x \longrightarrow Q\ x)$
and $b: Ex\ P$
shows $Ex\ Q$
 $\langle proof \rangle$

lemma *ball-reg*:
assumes $a: \forall x :: 'a. (x \in R \longrightarrow P\ x \longrightarrow Q\ x)$
and $b: Ball\ R\ P$
shows $Ball\ R\ Q$
 $\langle proof \rangle$

lemma *bex-reg*:
assumes $a: \forall x :: 'a. (x \in R \longrightarrow P\ x \longrightarrow Q\ x)$
and $b: Bex\ R\ P$
shows $Bex\ R\ Q$
 $\langle proof \rangle$

lemma *ball-all-comm*:
assumes $\bigwedge y. (\forall x \in P. A\ x\ y) \longrightarrow (\forall x. B\ x\ y)$
shows $(\forall x \in P. \forall y. A\ x\ y) \longrightarrow (\forall x. \forall y. B\ x\ y)$
 $\langle proof \rangle$

lemma *bex-ex-comm*:
assumes $(\exists y. \exists x. A\ x\ y) \longrightarrow (\exists y. \exists x \in P. B\ x\ y)$
shows $(\exists x. \exists y. A\ x\ y) \longrightarrow (\exists x \in P. \exists y. B\ x\ y)$
 $\langle proof \rangle$

44.3 Bounded abstraction

definition

$Babs :: 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$

where

$x \in p \Longrightarrow Babs\ p\ m\ x = m\ x$

lemma

babs-rsp:

assumes $q: Quotient3\ R1\ Abs1\ Rep1$
and $a: (R1 \Longrightarrow R2)\ f\ g$
shows $(R1 \Longrightarrow R2)\ (Babs\ (Respects\ R1)\ f)\ (Babs\ (Respects\ R1)\ g)$
 $\langle proof \rangle$

lemma

babs-prs:

assumes $q1: Quotient3\ R1\ Abs1\ Rep1$
and $q2: Quotient3\ R2\ Abs2\ Rep2$
shows $((Rep1 \dashrightarrow Abs2)\ (Babs\ (Respects\ R1)\ ((Abs1 \dashrightarrow Rep2)\ f))) = f$

$\langle proof \rangle$

lemma *babs-simp*:

assumes q : *Quotient3* $R1$ Abs Rep
shows $((R1 ==> R2) (Babs (Respects R1) f) (Babs (Respects R1) g)) = ((R1 ==> R2) f g)$
 (is ?lhs = ?rhs)
 $\langle proof \rangle$

If a user proves that a particular functional relation is an equivalence, this may be useful in regularising

lemma *babs-reg-epv*:

shows *equivp* $R \implies Babs (Respects R) P = P$
 $\langle proof \rangle$

lemma *ball-rsp*:

assumes a : $(R ==> (=)) f g$
shows $Ball (Respects R) f = Ball (Respects R) g$
 $\langle proof \rangle$

lemma *bex-rsp*:

assumes a : $(R ==> (=)) f g$
shows $(Bex (Respects R) f = Bex (Respects R) g)$
 $\langle proof \rangle$

lemma *bex1-rsp*:

assumes a : $(R ==> (=)) f g$
shows $Ex1 (\lambda x. x \in Respects R \wedge f x) = Ex1 (\lambda x. x \in Respects R \wedge g x)$
 $\langle proof \rangle$

Two lemmas needed for cleaning of quantifiers

lemma *all-prs*:

assumes a : *Quotient3* R *absf* *repf*
shows $Ball (Respects R) ((absf ---> id) f) = All f$
 $\langle proof \rangle$

lemma *ex-prs*:

assumes a : *Quotient3* R *absf* *repf*
shows $Bex (Respects R) ((absf ---> id) f) = Ex f$
 $\langle proof \rangle$

44.4 *Bex1-rel* quantifier

definition

$Bex1-rel :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
 where

$Bex1\text{-rel } R \ P \longleftrightarrow (\exists x \in \text{Respects } R. P \ x) \wedge (\forall x \in \text{Respects } R. \forall y \in \text{Respects } R. ((P \ x \wedge P \ y) \longrightarrow (R \ x \ y)))$

lemma *bex1-rel-aux*:

$\llbracket \forall xa \ ya. R \ xa \ ya \longrightarrow x \ xa = y \ ya; Bex1\text{-rel } R \ x \rrbracket \Longrightarrow Bex1\text{-rel } R \ y$
 $\langle \text{proof} \rangle$

lemma *bex1-rel-aux2*:

$\llbracket \forall xa \ ya. R \ xa \ ya \longrightarrow x \ xa = y \ ya; Bex1\text{-rel } R \ y \rrbracket \Longrightarrow Bex1\text{-rel } R \ x$
 $\langle \text{proof} \rangle$

lemma *bex1-rel-rsp*:

assumes *a*: *Quotient3* *R* *absf* *repf*
shows $((R \Longrightarrow (=)) \Longrightarrow (=)) \ (Bex1\text{-rel } R) \ (Bex1\text{-rel } R)$
 $\langle \text{proof} \rangle$

lemma *ex1-prs*:

assumes *Quotient3* *R* *absf* *repf*
shows $((\text{absf} \dashrightarrow id) \dashrightarrow id) \ (Bex1\text{-rel } R) \ f = Ex1 \ f$
 (is *?lhs* = *?rhs*)
 $\langle \text{proof} \rangle$

lemma *bex1-bexeq-reg*:

shows $(\exists !x \in \text{Respects } R. P \ x) \longrightarrow (Bex1\text{-rel } R \ (\lambda x. P \ x))$
 $\langle \text{proof} \rangle$

lemma *bex1-bexeq-reg-equiv*:

assumes *a*: *equivp* *R*
shows $(\exists !x. P \ x) \longrightarrow Bex1\text{-rel } R \ P$
 $\langle \text{proof} \rangle$

44.5 Various respects and preserve lemmas

lemma *quot-rel-rsp*:

assumes *a*: *Quotient3* *R* *Abs* *Rep*
shows $(R \Longrightarrow (=) \ R \Longrightarrow (=)) \ R \ R$
 $\langle \text{proof} \rangle$

lemma *o-prs*:

assumes *q1*: *Quotient3* *R1* *Abs1* *Rep1*
and *q2*: *Quotient3* *R2* *Abs2* *Rep2*
and *q3*: *Quotient3* *R3* *Abs3* *Rep3*
shows $((\text{Abs2} \dashrightarrow \text{Rep3}) \dashrightarrow (\text{Abs1} \dashrightarrow \text{Rep2}) \dashrightarrow (\text{Rep1} \dashrightarrow \text{Abs3})) \ (\circ) = (\circ)$
and $(id \dashrightarrow (\text{Abs1} \dashrightarrow id) \dashrightarrow \text{Rep1} \dashrightarrow id) \ (\circ) = (\circ)$
 $\langle \text{proof} \rangle$

lemma *o-rsp*:

$((R2 \Longrightarrow R3) \Longrightarrow (R1 \Longrightarrow R2) \Longrightarrow (R1 \Longrightarrow R3)) \ (\circ) \ (\circ)$

$((=) == => (R1 == => (=)) == => R1 == => (=)) (\circ) (\circ)$
 $\langle proof \rangle$

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)*
 $\langle proof \rangle$

lemma *if-prs*:

assumes *q*: *Quotient3 R Abs Rep*
shows $(id \dashrightarrow Rep \dashrightarrow Rep \dashrightarrow Abs) If = If$
 $\langle proof \rangle$

lemma *if-rsp*:

assumes *q*: *Quotient3 R Abs Rep*
shows $((=) == => R == => R == => R) If If$
 $\langle proof \rangle$

lemma *let-prs*:

assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows $(Rep2 \dashrightarrow (Abs2 \dashrightarrow Rep1) \dashrightarrow Abs1) Let = Let$
 $\langle proof \rangle$

lemma *let-rsp*:

shows $(R1 == => (R1 == => R2) == => R2) Let Let$
 $\langle proof \rangle$

lemma *id-rsp*:

shows $(R == => R) id id$
 $\langle proof \rangle$

lemma *id-prs*:

assumes *a*: *Quotient3 R Abs Rep*
shows $(Rep \dashrightarrow Abs) id = id$
 $\langle proof \rangle$

end

locale *quot-type* =

fixes *R* :: *'a* \Rightarrow *'a* \Rightarrow *bool*
and *Abs* :: *'a set* \Rightarrow *'b*
and *Rep* :: *'b* \Rightarrow *'a set*
assumes *equivp*: *part-equivp R*
and *rep-prop*: $\bigwedge y. \exists x. R\ x\ x \wedge Rep\ y = Collect\ (R\ x)$
and *rep-inverse*: $\bigwedge x. Abs\ (Rep\ x) = x$
and *abs-inverse*: $\bigwedge c. (\exists x. ((R\ x\ x) \wedge (c = Collect\ (R\ x)))) \implies (Rep\ (Abs\ c)) = c$
and *rep-inject*: $\bigwedge x\ y. (Rep\ x = Rep\ y) = (x = y)$

begin

definition

$abs :: 'a \Rightarrow 'b$

where

$abs\ x = Abs\ (Collect\ (R\ x))$

definition

$rep :: 'b \Rightarrow 'a$

where

$rep\ a = (SOME\ x.\ x \in Rep\ a)$

lemma *some-collect*:

assumes $R\ r\ r$

shows $R\ (SOME\ x.\ x \in Collect\ (R\ r)) = R\ r$

<proof>

lemma *Quotient*: *Quotient3* $R\ abs\ rep$

<proof>

end

44.6 Quotient composition

lemma *OOO-quotient3*:

fixes $R1 :: 'a \Rightarrow 'a \Rightarrow bool$

fixes $Abs1 :: 'a \Rightarrow 'b$ **and** $Rep1 :: 'b \Rightarrow 'a$

fixes $Abs2 :: 'b \Rightarrow 'c$ **and** $Rep2 :: 'c \Rightarrow 'b$

fixes $R2' :: 'a \Rightarrow 'a \Rightarrow bool$

fixes $R2 :: 'b \Rightarrow 'b \Rightarrow bool$

assumes $R1: Quotient3\ R1\ Abs1\ Rep1$

assumes $R2: Quotient3\ R2\ Abs2\ Rep2$

assumes $Abs1: \bigwedge x\ y.\ R2'\ x\ y \Longrightarrow R1\ x\ x \Longrightarrow R1\ y\ y \Longrightarrow R2\ (Abs1\ x)\ (Abs1\ y)$

assumes $Rep1: \bigwedge x\ y.\ R2\ x\ y \Longrightarrow R2'\ (Rep1\ x)\ (Rep1\ y)$

shows $Quotient3\ (R1\ OO\ R2'\ OO\ R1)\ (Abs2 \circ Abs1)\ (Rep1 \circ Rep2)$

<proof>

lemma *OOO-eq-quotient3*:

fixes $R1 :: 'a \Rightarrow 'a \Rightarrow bool$

fixes $Abs1 :: 'a \Rightarrow 'b$ **and** $Rep1 :: 'b \Rightarrow 'a$

fixes $Abs2 :: 'b \Rightarrow 'c$ **and** $Rep2 :: 'c \Rightarrow 'b$

assumes $R1: Quotient3\ R1\ Abs1\ Rep1$

assumes $R2: Quotient3\ (=)\ Abs2\ Rep2$

shows $Quotient3\ (R1\ OO\ (=))\ (Abs2 \circ Abs1)\ (Rep1 \circ Rep2)$

<proof>

44.7 Quotient3 to Quotient

lemma *Quotient3-to-Quotient*:

assumes $Quotient3\ R\ Abs\ Rep$

and $T \equiv \lambda x y. R\ x\ x \wedge Abs\ x = y$
shows *Quotient R Abs Rep T*
 $\langle proof \rangle$

lemma *Quotient3-to-Quotient-equivp:*
assumes $q: Quotient3\ R\ Abs\ Rep$
and $T\text{-def}: T \equiv \lambda x y. Abs\ x = y$
and $eR: equivp\ R$
shows *Quotient R Abs Rep T*
 $\langle proof \rangle$

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*
 $\langle ML \rangle$

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.

$\langle ML \rangle$

Definitions of the quotient types.

$\langle ML \rangle$

Definitions for quotient constants.

$\langle ML \rangle$

An auxiliary constant for recording some information about the lifted theorem in a tactic.

definition

$$\text{Quot-True} :: 'a \Rightarrow \text{bool}$$
where

$$\text{Quot-True } x \longleftrightarrow \text{True}$$
lemma

$$\text{shows } QT\text{-all}: \text{Quot-True } (\text{All } P) \Longrightarrow \text{Quot-True } P$$

$$\text{and } QT\text{-ex}: \text{Quot-True } (\text{Ex } P) \Longrightarrow \text{Quot-True } P$$

$$\text{and } QT\text{-ex1}: \text{Quot-True } (\text{Ex1 } P) \Longrightarrow \text{Quot-True } P$$

$$\text{and } QT\text{-lam}: \text{Quot-True } (\lambda x. P x) \Longrightarrow (\bigwedge x. \text{Quot-True } (P x))$$

$$\text{and } QT\text{-ext}: (\bigwedge x. \text{Quot-True } (a x) \Longrightarrow f x = g x) \Longrightarrow (\text{Quot-True } a \Longrightarrow f = g)$$

<proof>

lemma $QT\text{-imp}: \text{Quot-True } a \equiv \text{Quot-True } b$

<proof>

context includes *lifting-syntax*

begin

Tactics for proving the lifted theorems

<ML>

end

44.9 Methods / Interface

<ML>

no-notation *rel-conj* (**infixr** *<OOO>* 75)

45 Lifting of BNFs

lemma *sum-insert-Inl-unit*: $x \in A \Longrightarrow (\bigwedge y. x = \text{Inr } y \Longrightarrow \text{Inr } y \in B) \Longrightarrow x \in \text{insert } (\text{Inl } ()) B$

<proof>

lemma *lift-sum-unit-vimage-commute*:

$$\text{insert } (\text{Inl } ()) (\text{Inr } ' f -' A) = \text{map-sum id } f -' \text{insert } (\text{Inl } ()) (\text{Inr } ' A)$$

<proof>

lemma *insert-Inl-int-map-sum-unit*: $\text{insert } (\text{Inl } ()) A \cap \text{range } (\text{map-sum id } f) \neq \{\}$

<proof>

lemma *image-map-sum-unit-subset*:

$$A \subseteq \text{insert } (\text{Inl } ()) (\text{Inr } ' B) \Longrightarrow \text{map-sum id } f -' A \subseteq \text{insert } (\text{Inl } ()) (\text{Inr } ' f -' B)$$

<proof>

lemma *subset-lift-sum-unitD*: $A \subseteq \text{insert } (\text{Inl } ()) (\text{Inr } \cdot B) \implies \text{Inr } x \in A \implies x \in B$
 ⟨proof⟩

lemma *UNIV-sum-unit-conv*: $\text{insert } (\text{Inl } ()) (\text{range } \text{Inr}) = \text{UNIV}$
 ⟨proof⟩

lemma *subset-vimage-image-subset*: $A \subseteq f^{-1} B \implies f^{-1} A \subseteq B$
 ⟨proof⟩

lemma *relcompp-mem-Grp-neq-bot*:
 $A \cap \text{range } f \neq \{\} \implies (\lambda x y. x \in A \wedge y \in A) \text{ OO } (\text{Grp } \text{UNIV } f)^{-1-1} \neq \text{bot}$
 ⟨proof⟩

lemma *comp-projr-Inr*: $\text{projr} \circ \text{Inr} = \text{id}$
 ⟨proof⟩

lemma *in-rel-sum-in-image-projr*:
 $B \subseteq \{(x,y). \text{rel-sum } ((=) :: \text{unit} \Rightarrow \text{unit} \Rightarrow \text{bool}) A x y\} \implies$
 $\text{Inr } \cdot C = \text{fst } \cdot B \implies \text{snd } \cdot B = \text{Inr } \cdot D \implies \text{map-prod } \text{projr } \text{projr } \cdot B \subseteq \{(x,y). A x y\}$
 ⟨proof⟩

lemma *subset-rel-sumI*: $B \subseteq \{(x,y). A x y\} \implies \text{rel-sum } ((=) :: \text{unit} \Rightarrow \text{unit} \Rightarrow \text{bool}) A$
 $(\text{if } x \in B \text{ then } \text{Inr } (\text{fst } x) \text{ else } \text{Inl } ())$
 $(\text{if } x \in B \text{ then } \text{Inr } (\text{snd } x) \text{ else } \text{Inl } ())$
 ⟨proof⟩

lemma *relcompp-eq-Grp-neq-bot*: $(=) \text{ OO } (\text{Grp } \text{UNIV } f)^{-1-1} \neq \text{bot}$
 ⟨proof⟩

lemma *rel-fun-rel-OO1*: $(\text{rel-fun } Q (\text{rel-fun } R (=))) A B \implies \text{conversep } Q \text{ OO } A \text{ OO } R \leq B$
 ⟨proof⟩

lemma *rel-fun-rel-OO2*: $(\text{rel-fun } Q (\text{rel-fun } R (=))) A B \implies Q \text{ OO } B \text{ OO } \text{conversep } R \leq A$
 ⟨proof⟩

lemma *rel-sum-eq2-nonempty*: $\text{rel-sum } (=) A \text{ OO } \text{rel-sum } (=) B \neq \text{bot}$
 ⟨proof⟩

lemma *rel-sum-eq3-nonempty*: $\text{rel-sum } (=) A \text{ OO } (\text{rel-sum } (=) B \text{ OO } \text{rel-sum } (=) C) \neq \text{bot}$
 ⟨proof⟩

lemma *hypsubst*: $A = B \implies x \in B \implies (x \in A \implies P) \implies P$ ⟨proof⟩

lemma *Quotient-crel-quotient*: $\text{Quotient } R \text{ Abs Rep } T \implies \text{equivp } R \implies T \equiv (\lambda x y. \text{Abs } x = y)$
 ⟨proof⟩

lemma *Quotient-crel-typedef*: $\text{Quotient } (\text{eq-onp } P) \text{ Abs Rep } T \implies T \equiv (\lambda x y. x = \text{Rep } y)$
 ⟨proof⟩

lemma *Quotient-crel-typecopy*: $\text{Quotient } (=) \text{ Abs Rep } T \implies T \equiv (\lambda x y. x = \text{Rep } y)$
 ⟨proof⟩

lemma *equivp-add-relconj*:
 assumes *equiv*: $\text{equivp } R \text{ equivp } R'$ and *le*: $S \text{ OO } T \text{ OO } U \leq R \text{ OO } STU \text{ OO } R'$
 shows $R \text{ OO } S \text{ OO } T \text{ OO } U \text{ OO } R' \leq R \text{ OO } STU \text{ OO } R'$
 ⟨proof⟩

lemma *Grp-conversep-eq-onp*: $((\text{BNF-Def.Grp UNIV } f)^{-1-1} \text{ OO } \text{BNF-Def.Grp UNIV } f) = \text{eq-onp } (\lambda x. x \in \text{range } f)$
 ⟨proof⟩

lemma *Grp-conversep-nonempty*: $(\text{BNF-Def.Grp UNIV } f)^{-1-1} \text{ OO } \text{BNF-Def.Grp UNIV } f \neq \text{bot}$
 ⟨proof⟩

lemma *relcomppI2*: $r \ a \ b \implies s \ b \ c \implies t \ c \ d \implies (r \text{ OO } s \text{ OO } t) \ a \ d$
 ⟨proof⟩

lemma *rel-conj-eq-onp*: $\text{equivp } R \implies \text{rel-conj } R (\text{eq-onp } P) \leq R$
 ⟨proof⟩

lemma *Quotient-Quotient3*: $\text{Quotient } R \text{ Abs Rep } T \implies \text{Quotient3 } R \text{ Abs Rep}$
 ⟨proof⟩

lemma *Quotient-reflp-imp-equivp*: $\text{Quotient } R \text{ Abs Rep } T \implies \text{reflp } R \implies \text{equivp } R$
 ⟨proof⟩

lemma *Quotient-eq-onp-typedef*:
 $\text{Quotient } (\text{eq-onp } P) \text{ Abs Rep } cr \implies \text{type-definition } \text{Rep Abs } \{x. P \ x\}$
 ⟨proof⟩

lemma *Quotient-eq-onp-type-copy*:
 $\text{Quotient } (=) \text{ Abs Rep } cr \implies \text{type-definition } \text{Rep Abs UNIV}$
 ⟨proof⟩

⟨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-
    comppI2 rel-conj-eq-onp
    Quotient-reftp-imp-equivp Quotient-Quotient3
end

```

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 ⇒ 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 ⇒ nat⟩
where
  ⟨nat-of-num One = Suc 0⟩
| ⟨nat-of-num (Bit0 x) = nat-of-num x + nat-of-num x⟩
| ⟨nat-of-num (Bit1 x) = Suc (nat-of-num x + nat-of-num x)⟩

```

```

primrec num-of-nat :: ⟨nat ⇒ num⟩
where
  ⟨num-of-nat 0 = One⟩
| ⟨num-of-nat (Suc n) = (if 0 < n then inc (num-of-nat n) else One)⟩

```

```

lemma nat-of-num-pos: ⟨0 < nat-of-num x⟩
  ⟨proof⟩

```

```

lemma nat-of-num-neq-0: ⟨ nat-of-num x ≠ 0 ⟩
  ⟨proof⟩

```

```

lemma nat-of-num-inc: ⟨nat-of-num (inc x) = Suc (nat-of-num x)⟩
  ⟨proof⟩

```

lemma *num-of-nat-double*: $\langle 0 < n \implies \text{num-of-nat } (n + n) = \text{Bit0 } (\text{num-of-nat } n) \rangle$
 $\langle \text{proof} \rangle$

Type *num* is isomorphic to the strictly positive natural numbers.

lemma *nat-of-num-inverse*: $\langle \text{num-of-nat } (\text{nat-of-num } x) = x \rangle$
 $\langle \text{proof} \rangle$

lemma *num-of-nat-inverse*: $\langle 0 < n \implies \text{nat-of-num } (\text{num-of-nat } n) = n \rangle$
 $\langle \text{proof} \rangle$

lemma *num-eq-iff*: $\langle x = y \iff \text{nat-of-num } x = \text{nat-of-num } y \rangle$
 $\langle \text{proof} \rangle$

lemma *num-induct* [*case-names One inc*]:
fixes *P* :: $\langle \text{num} \Rightarrow \text{bool} \rangle$
assumes *One*: $\langle P \text{ One} \rangle$
and *inc*: $\langle \bigwedge x. P x \implies P (\text{inc } x) \rangle$
shows $\langle P x \rangle$
 $\langle \text{proof} \rangle$

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* :: $\langle \{ \text{plus}, \text{times}, \text{linorder} \} \rangle$
begin

definition [*code del*]: $\langle m + n = \text{num-of-nat } (\text{nat-of-num } m + \text{nat-of-num } n) \rangle$

definition [*code del*]: $\langle m * n = \text{num-of-nat } (\text{nat-of-num } m * \text{nat-of-num } n) \rangle$

definition [*code del*]: $\langle m \leq n \iff \text{nat-of-num } m \leq \text{nat-of-num } n \rangle$

definition [*code del*]: $\langle m < n \iff \text{nat-of-num } m < \text{nat-of-num } n \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma *nat-of-num-add*: $\langle \text{nat-of-num } (x + y) = \text{nat-of-num } x + \text{nat-of-num } y \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-of-num-mult*: $\langle \text{nat-of-num } (x * y) = \text{nat-of-num } x * \text{nat-of-num } y \rangle$
 $\langle \text{proof} \rangle$

lemma *add-num-simps* [*simp*, *code*]:

$\langle One + One = Bit0\ One \rangle$
 $\langle One + Bit0\ n = Bit1\ n \rangle$
 $\langle One + Bit1\ n = Bit0\ (n + One) \rangle$
 $\langle Bit0\ m + One = Bit1\ m \rangle$
 $\langle Bit0\ m + Bit0\ n = Bit0\ (m + n) \rangle$
 $\langle Bit0\ m + Bit1\ n = Bit1\ (m + n) \rangle$
 $\langle Bit1\ m + One = Bit0\ (m + One) \rangle$
 $\langle Bit1\ m + Bit0\ n = Bit1\ (m + n) \rangle$
 $\langle Bit1\ m + Bit1\ n = Bit0\ (m + n + One) \rangle$
 $\langle proof \rangle$

lemma *mult-num-simps* [*simp*, *code*]:

$\langle m * One = m \rangle$
 $\langle One * n = n \rangle$
 $\langle Bit0\ m * Bit0\ n = Bit0\ (Bit0\ (m * n)) \rangle$
 $\langle Bit0\ m * Bit1\ n = Bit0\ (m * Bit1\ n) \rangle$
 $\langle Bit1\ m * Bit0\ n = Bit0\ (Bit1\ m * n) \rangle$
 $\langle Bit1\ m * Bit1\ n = Bit1\ (m + n + Bit0\ (m * n)) \rangle$
 $\langle proof \rangle$

lemma *eq-num-simps*:

$\langle One = One \longleftrightarrow True \rangle$
 $\langle One = Bit0\ n \longleftrightarrow False \rangle$
 $\langle One = Bit1\ n \longleftrightarrow False \rangle$
 $\langle Bit0\ m = One \longleftrightarrow False \rangle$
 $\langle Bit1\ m = One \longleftrightarrow False \rangle$
 $\langle Bit0\ m = Bit0\ n \longleftrightarrow m = n \rangle$
 $\langle Bit0\ m = Bit1\ n \longleftrightarrow False \rangle$
 $\langle Bit1\ m = Bit0\ n \longleftrightarrow False \rangle$
 $\langle Bit1\ m = Bit1\ n \longleftrightarrow m = n \rangle$
 $\langle proof \rangle$

lemma *le-num-simps* [*simp*, *code*]:

$\langle One \leq n \longleftrightarrow True \rangle$
 $\langle Bit0\ m \leq One \longleftrightarrow False \rangle$
 $\langle Bit1\ m \leq One \longleftrightarrow False \rangle$
 $\langle Bit0\ m \leq Bit0\ n \longleftrightarrow m \leq n \rangle$
 $\langle Bit0\ m \leq Bit1\ n \longleftrightarrow m \leq n \rangle$
 $\langle Bit1\ m \leq Bit1\ n \longleftrightarrow m \leq n \rangle$
 $\langle Bit1\ m \leq Bit0\ n \longleftrightarrow m < n \rangle$
 $\langle proof \rangle$

lemma *less-num-simps* [*simp*, *code*]:

$\langle m < One \longleftrightarrow False \rangle$
 $\langle One < Bit0\ n \longleftrightarrow True \rangle$
 $\langle One < Bit1\ n \longleftrightarrow True \rangle$
 $\langle Bit0\ m < Bit0\ n \longleftrightarrow m < n \rangle$
 $\langle Bit0\ m < Bit1\ n \longleftrightarrow m \leq n \rangle$

$\langle \text{Bit1 } m < \text{Bit1 } n \longleftrightarrow m < n \rangle$
 $\langle \text{Bit1 } m < \text{Bit0 } n \longleftrightarrow m < n \rangle$
 $\langle \text{proof} \rangle$

lemma *le-num-One-iff*: $\langle x \leq \text{One} \longleftrightarrow x = \text{One} \rangle$
 $\langle \text{proof} \rangle$

Rules using *One* and *inc* as constructors.

lemma *add-One*: $\langle x + \text{One} = \text{inc } x \rangle$
 $\langle \text{proof} \rangle$

lemma *add-One-commute*: $\langle \text{One} + n = n + \text{One} \rangle$
 $\langle \text{proof} \rangle$

lemma *add-inc*: $\langle x + \text{inc } y = \text{inc } (x + y) \rangle$
 $\langle \text{proof} \rangle$

lemma *mult-inc*: $\langle x * \text{inc } y = x * y + x \rangle$
 $\langle \text{proof} \rangle$

The *num-of-nat* conversion.

lemma *num-of-nat-One*: $\langle n \leq 1 \implies \text{num-of-nat } n = \text{One} \rangle$
 $\langle \text{proof} \rangle$

lemma *num-of-nat-plus-distrib*:
 $\langle 0 < m \implies 0 < n \implies \text{num-of-nat } (m + n) = \text{num-of-nat } m + \text{num-of-nat } n \rangle$
 $\langle \text{proof} \rangle$

A double-and-decrement function.

primrec *BitM* :: $\langle \text{num} \Rightarrow \text{num} \rangle$
where
 $\langle \text{BitM } \text{One} = \text{One} \rangle$
 $| \langle \text{BitM } (\text{Bit0 } n) = \text{Bit1 } (\text{BitM } n) \rangle$
 $| \langle \text{BitM } (\text{Bit1 } n) = \text{Bit1 } (\text{Bit0 } n) \rangle$

lemma *BitM-plus-one*: $\langle \text{BitM } n + \text{One} = \text{Bit0 } n \rangle$
 $\langle \text{proof} \rangle$

lemma *one-plus-BitM*: $\langle \text{One} + \text{BitM } n = \text{Bit0 } n \rangle$
 $\langle \text{proof} \rangle$

lemma *BitM-inc-eq*:
 $\langle \text{BitM } (\text{inc } n) = \text{Bit1 } n \rangle$
 $\langle \text{proof} \rangle$

lemma *inc-BitM-eq*:
 $\langle \text{inc } (\text{BitM } n) = \text{Bit0 } n \rangle$
 $\langle \text{proof} \rangle$

Squaring and exponentiation.

```
primrec sqr :: ⟨num ⇒ num⟩
  where
    ⟨sqr One = One⟩
  | ⟨sqr (Bit0 n) = Bit0 (Bit0 (sqr n))⟩
  | ⟨sqr (Bit1 n) = Bit1 (Bit0 (sqr n + n))⟩
```

```
primrec pow :: ⟨num ⇒ num ⇒ num⟩
  where
    ⟨pow x One = x⟩
  | ⟨pow x (Bit0 y) = sqr (pow x y)⟩
  | ⟨pow x (Bit1 y) = sqr (pow x y) * x⟩
```

```
lemma nat-of-num-sqr: ⟨nat-of-num (sqr x) = nat-of-num x * nat-of-num x⟩
  ⟨proof⟩
```

```
lemma sqr-conv-mult: ⟨sqr x = x * x⟩
  ⟨proof⟩
```

```
lemma num-double [simp]:
  ⟨Bit0 num.One * n = Bit0 n⟩
  ⟨proof⟩
```

46.3 Binary numerals

We embed binary representations into a generic algebraic structure using *numeral*.

```
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)⟩
  ⟨proof⟩
```

```
lemma one-plus-numeral-commute: ⟨1 + numeral x = numeral x + 1⟩
  ⟨proof⟩
```

```
lemma numeral-inc: ⟨numeral (inc x) = numeral x + 1⟩
  ⟨proof⟩
```

declare *numeral.simps* [*simp del*]

abbreviation $\langle \text{Numeral1} \equiv \text{numeral One} \rangle$

declare *numeral-One* [*code-post*]

end

Numeral syntax.

syntax

$\text{-Numeral} :: \langle \text{num-const} \Rightarrow 'a \rangle \ (\langle \langle \text{open-block notation} = \langle \text{literal number} \rangle \rangle \rangle)$

$\langle ML \rangle$

46.4 Class-specific numeral rules

numeral is a morphism.

46.4.1 Structures with addition: class *numeral*

context *numeral*

begin

lemma *numeral-add*: $\langle \text{numeral } (m + n) = \text{numeral } m + \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-plus-numeral*: $\langle \text{numeral } m + \text{numeral } n = \text{numeral } (m + n) \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-plus-one*: $\langle \text{numeral } n + 1 = \text{numeral } (n + \text{One}) \rangle$
 $\langle \text{proof} \rangle$

lemma *one-plus-numeral*: $\langle 1 + \text{numeral } n = \text{numeral } (\text{One} + n) \rangle$
 $\langle \text{proof} \rangle$

lemma *one-add-one*: $\langle 1 + 1 = 2 \rangle$
 $\langle \text{proof} \rangle$

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*: $\langle - \text{Numeral1} = - 1 \rangle$

$\langle \text{proof} \rangle$

Numerals form an abelian subgroup.

inductive *is-num* :: $\langle 'a \Rightarrow \text{bool} \rangle$

where

$\langle \text{is-num } 1 \rangle$

$\mid \langle \text{is-num } x \Longrightarrow \text{is-num } (-\ x) \rangle$

$\mid \langle \text{is-num } x \Longrightarrow \text{is-num } y \Longrightarrow \text{is-num } (x + y) \rangle$

lemma *is-num-numeral*: $\langle \text{is-num } (\text{numeral } k) \rangle$

$\langle \text{proof} \rangle$

lemma *is-num-add-commute*: $\langle \text{is-num } x \Longrightarrow \text{is-num } y \Longrightarrow x + y = y + x \rangle$

$\langle \text{proof} \rangle$

lemma *is-num-add-left-commute*: $\langle \text{is-num } x \Longrightarrow \text{is-num } y \Longrightarrow x + (y + z) = y + (x + z) \rangle$

$\langle \text{proof} \rangle$

lemmas *is-num-normalize* =

add.assoc is-num-add-commute is-num-add-left-commute

is-num.intros is-num-numeral

minus-add

definition *dbl* :: $\langle 'a \Rightarrow 'a \rangle$

where $\langle \text{dbl } x = x + x \rangle$

definition *dbl-inc* :: $\langle 'a \Rightarrow 'a \rangle$

where $\langle \text{dbl-inc } x = x + x + 1 \rangle$

definition *dbl-dec* :: $\langle 'a \Rightarrow 'a \rangle$

where $\langle \text{dbl-dec } x = x + x - 1 \rangle$

definition *sub* :: $\langle \text{num} \Rightarrow \text{num} \Rightarrow 'a \rangle$

where $\langle \text{sub } k\ l = \text{numeral } k - \text{numeral } l \rangle$

lemma *numeral-BitM*: $\langle \text{numeral } (\text{BitM } n) = \text{numeral } (\text{Bit0 } n) - 1 \rangle$

$\langle \text{proof} \rangle$

lemma *sub-inc-One-eq*:

$\langle \text{sub } (\text{inc } n)\ \text{num.One} = \text{numeral } n \rangle$

$\langle \text{proof} \rangle$

lemma *dbl-simps* [*simp*]:

$\langle \text{dbl } (-\ \text{numeral } k) = -\ \text{dbl } (\text{numeral } k) \rangle$

$\langle \text{dbl } 0 = 0 \rangle$

$\langle \text{dbl } 1 = 2 \rangle$

$\langle \text{dbl } (-\ 1) = -\ 2 \rangle$

$\langle \text{dbl } (\text{numeral } k) = \text{numeral } (\text{Bit0 } k) \rangle$

$\langle \text{proof} \rangle$

lemma *dbl-inc-simps* [simp]:

$\langle \text{dbl-inc } (- \text{ numeral } k) = - \text{ dbl-dec } (\text{ numeral } k) \rangle$
 $\langle \text{dbl-inc } 0 = 1 \rangle$
 $\langle \text{dbl-inc } 1 = 3 \rangle$
 $\langle \text{dbl-inc } (- 1) = - 1 \rangle$
 $\langle \text{dbl-inc } (\text{ numeral } k) = \text{ numeral } (\text{Bit1 } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *dbl-dec-simps* [simp]:

$\langle \text{dbl-dec } (- \text{ numeral } k) = - \text{ dbl-inc } (\text{ numeral } k) \rangle$
 $\langle \text{dbl-dec } 0 = - 1 \rangle$
 $\langle \text{dbl-dec } 1 = 1 \rangle$
 $\langle \text{dbl-dec } (- 1) = - 3 \rangle$
 $\langle \text{dbl-dec } (\text{ numeral } k) = \text{ numeral } (\text{BitM } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *sub-num-simps* [simp]:

$\langle \text{sub One One} = 0 \rangle$
 $\langle \text{sub One } (\text{Bit0 } l) = - \text{ numeral } (\text{BitM } l) \rangle$
 $\langle \text{sub One } (\text{Bit1 } l) = - \text{ numeral } (\text{Bit0 } l) \rangle$
 $\langle \text{sub } (\text{Bit0 } k) \text{ One} = \text{ numeral } (\text{BitM } k) \rangle$
 $\langle \text{sub } (\text{Bit1 } k) \text{ One} = \text{ numeral } (\text{Bit0 } k) \rangle$
 $\langle \text{sub } (\text{Bit0 } k) (\text{Bit0 } l) = \text{dbl } (\text{sub } k l) \rangle$
 $\langle \text{sub } (\text{Bit0 } k) (\text{Bit1 } l) = \text{dbl-dec } (\text{sub } k l) \rangle$
 $\langle \text{sub } (\text{Bit1 } k) (\text{Bit0 } l) = \text{dbl-inc } (\text{sub } k l) \rangle$
 $\langle \text{sub } (\text{Bit1 } k) (\text{Bit1 } l) = \text{dbl } (\text{sub } k l) \rangle$
 $\langle \text{proof} \rangle$

lemma *add-neg-numeral-simps*:

$\langle \text{numeral } m + - \text{ numeral } n = \text{sub } m n \rangle$
 $\langle - \text{ numeral } m + \text{ numeral } n = \text{sub } n m \rangle$
 $\langle - \text{ numeral } m + - \text{ numeral } n = - (\text{numeral } m + \text{ numeral } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *add-neg-numeral-special*:

$\langle 1 + - \text{ numeral } m = \text{sub One } m \rangle$
 $\langle - \text{ numeral } m + 1 = \text{sub One } m \rangle$
 $\langle \text{numeral } m + - 1 = \text{sub } m \text{ One} \rangle$
 $\langle - 1 + \text{ numeral } n = \text{sub } n \text{ One} \rangle$
 $\langle - 1 + - \text{ numeral } n = - \text{ numeral } (\text{inc } n) \rangle$
 $\langle - \text{ numeral } m + - 1 = - \text{ numeral } (\text{inc } m) \rangle$
 $\langle 1 + - 1 = 0 \rangle$
 $\langle - 1 + 1 = 0 \rangle$
 $\langle - 1 + - 1 = - 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *diff-numeral-simps*:

```

⟨numeral 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⟩
⟨proof⟩

```

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⟩
⟨proof⟩

```

end

46.4.3 Structures with multiplication: class *semiring-numeral*

```

class semiring-numeral = semiring + monoid-mult
begin

```

```

subclass numeral ⟨proof⟩

```

```

lemma numeral-mult: ⟨numeral (m * n) = numeral m * numeral n⟩
⟨proof⟩

```

```

lemma numeral-times-numeral: ⟨numeral m * numeral n = numeral (m * n)⟩
⟨proof⟩

```

```

lemma mult-2: ⟨2 * z = z + z⟩
⟨proof⟩

```

```

lemma mult-2-right: ⟨z * 2 = z + z⟩
⟨proof⟩

```

```

lemma left-add-twice:
⟨a + (a + b) = 2 * a + b⟩
⟨proof⟩

```

```

lemma numeral-Bit0-eq-double:
⟨numeral (Bit0 n) = 2 * numeral n⟩
⟨proof⟩

```

lemma *numeral-Bit1-eq-inc-double*:
 $\langle \text{numeral } (\text{Bit1 } n) = 2 * \text{numeral } n + 1 \rangle$
 $\langle \text{proof} \rangle$

end

46.4.4 Structures with a zero: class *semiring-1*

context *semiring-1*
begin

subclass *semiring-numeral* $\langle \text{proof} \rangle$

lemma *of-nat-numeral* [*simp*]: $\langle \text{of-nat } (\text{numeral } n) = \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

end

lemma *nat-of-num-numeral* [*code-abbrev*]: $\langle \text{nat-of-num} = \text{numeral} \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-of-num-code* [*code*]:
 $\langle \text{nat-of-num } \text{One} = 1 \rangle$
 $\langle \text{nat-of-num } (\text{Bit0 } n) = (\text{let } m = \text{nat-of-num } n \text{ in } m + m) \rangle$
 $\langle \text{nat-of-num } (\text{Bit1 } n) = (\text{let } m = \text{nat-of-num } n \text{ in } \text{Suc } (m + m)) \rangle$
 $\langle \text{proof} \rangle$

46.4.5 Equality: class *semiring-char-0*

context *semiring-char-0*
begin

lemma *numeral-eq-iff*: $\langle \text{numeral } m = \text{numeral } n \longleftrightarrow m = n \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-eq-one-iff*: $\langle \text{numeral } n = 1 \longleftrightarrow n = \text{One} \rangle$
 $\langle \text{proof} \rangle$

lemma *one-eq-numeral-iff*: $\langle 1 = \text{numeral } n \longleftrightarrow \text{One} = n \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-neq-zero*: $\langle \text{numeral } n \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *zero-neq-numeral*: $\langle 0 \neq \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

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*: $\langle \text{numeral } m \leq \text{numeral } n \longleftrightarrow m \leq n \rangle$
 $\langle \text{proof} \rangle$

lemma *one-le-numeral*: $\langle 1 \leq \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-le-one-iff*: $\langle \text{numeral } n \leq 1 \longleftrightarrow n \leq \text{One} \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-less-iff*: $\langle \text{numeral } m < \text{numeral } n \longleftrightarrow m < n \rangle$
 $\langle \text{proof} \rangle$

lemma *not-numeral-less-one*: $\langle \neg \text{numeral } n < 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *one-less-numeral-iff*: $\langle 1 < \text{numeral } n \longleftrightarrow \text{One} < n \rangle$
 $\langle \text{proof} \rangle$

lemma *zero-le-numeral*: $\langle 0 \leq \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *zero-less-numeral*: $\langle 0 < \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *not-numeral-le-zero*: $\langle \neg \text{numeral } n \leq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *not-numeral-less-zero*: $\langle \neg \text{numeral } n < 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *one-of-nat-le-iff* [simp]: $\langle 1 \leq \text{of-nat } k \longleftrightarrow 1 \leq k \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-nat-le-iff* [simp]: $\langle \text{numeral } n \leq \text{of-nat } k \longleftrightarrow \text{numeral } n \leq k \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-le-1-iff* [simp]: $\langle \text{of-nat } k \leq 1 \longleftrightarrow k \leq 1 \rangle$

⟨proof⟩

lemma *of-nat-le-numeral-iff* [simp]: $\langle \text{of-nat } k \leq \text{numeral } n \longleftrightarrow k \leq \text{numeral } n \rangle$
⟨proof⟩

lemma *one-of-nat-less-iff* [simp]: $\langle 1 < \text{of-nat } k \longleftrightarrow 1 < k \rangle$
⟨proof⟩

lemma *numeral-nat-less-iff* [simp]: $\langle \text{numeral } n < \text{of-nat } k \longleftrightarrow \text{numeral } n < k \rangle$
⟨proof⟩

lemma *of-nat-less-1-iff* [simp]: $\langle \text{of-nat } k < 1 \longleftrightarrow k < 1 \rangle$
⟨proof⟩

lemma *of-nat-less-numeral-iff* [simp]: $\langle \text{of-nat } k < \text{numeral } n \longleftrightarrow k < \text{numeral } n \rangle$
⟨proof⟩

lemma *of-nat-eq-numeral-iff* [simp]: $\langle \text{of-nat } k = \text{numeral } n \longleftrightarrow k = \text{numeral } n \rangle$
⟨proof⟩

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'* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$
defines $\langle \text{min}' \equiv \text{min} \rangle$
shows
 $\langle \text{min}' 0 1 = 0 \rangle$
 $\langle \text{min}' 1 0 = 0 \rangle$
 $\langle \text{min}' 0 (\text{numeral } x) = 0 \rangle$

```

    ⟨min' (numeral x) 0 = 0⟩
    ⟨min' 1 (numeral x) = 1⟩
    ⟨min' (numeral x) 1 = 1⟩
    ⟨proof⟩

```

```

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⟩
    ⟨proof⟩

```

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] =
  min-def [of ⟨numeral u⟩ ⟨numeral v⟩]
  min-def [of ⟨numeral u⟩ ⟨← numeral v⟩]
  min-def [of ⟨← numeral u⟩ ⟨numeral v⟩]
  min-def [of ⟨← numeral u⟩ ⟨← numeral v⟩] for u v

```

46.4.7 Multiplication and negation: class *ring-1*

```

context ring-1
begin

```

```

subclass neg-numeral ⟨proof⟩

```

```

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)⟩
  ⟨proof⟩

```

```

lemma mult-minus1 [simp]: ⟨← 1 * z = ← z⟩
  ⟨proof⟩

```

```

lemma mult-minus1-right [simp]: ⟨z * ← 1 = ← z⟩

```

⟨proof⟩

lemma *minus-sub-one-diff-one* [simp]:
 ⟨ $\text{sub } m \text{ One} - 1 = - \text{numeral } m$ ⟩
 ⟨proof⟩

end

46.4.8 Equality using *iszero* for rings with non-zero characteristic

context *ring-1*
begin

definition *iszero* :: ⟨ $a \Rightarrow \text{bool}$ ⟩
where ⟨ $\text{iszero } z \longleftrightarrow z = 0$ ⟩

lemma *iszero-0* [simp]: ⟨ $\text{iszero } 0$ ⟩
 ⟨proof⟩

lemma *not-iszero-1* [simp]: ⟨ $\neg \text{iszero } 1$ ⟩
 ⟨proof⟩

lemma *not-iszero-Numeral1*: ⟨ $\neg \text{iszero } \text{Numeral1}$ ⟩
 ⟨proof⟩

lemma *not-iszero-neg-1* [simp]: ⟨ $\neg \text{iszero } (- 1)$ ⟩
 ⟨proof⟩

lemma *not-iszero-neg-Numeral1*: ⟨ $\neg \text{iszero } (- \text{Numeral1})$ ⟩
 ⟨proof⟩

lemma *iszero-neg-numeral* [simp]: ⟨ $\text{iszero } (- \text{numeral } w) \longleftrightarrow \text{iszero } (\text{numeral } w)$ ⟩
 ⟨proof⟩

lemma *eq-iff-iszero-diff*: ⟨ $x = y \longleftrightarrow \text{iszero } (x - y)$ ⟩
 ⟨proof⟩

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*:
 ⟨ $\text{numeral } x = \text{numeral } y \longleftrightarrow \text{iszero } (\text{sub } x \ y)$ ⟩
 ⟨ $\text{numeral } x = - \text{numeral } y \longleftrightarrow \text{iszero } (\text{numeral } (x + y))$ ⟩

```

  ⟨- 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)⟩
  ⟨proof⟩

```

end

46.4.9 Equality and negation: class *ring-char-0*

context *ring-char-0*

begin

lemma *not-iszero-numeral* [simp]: ⟨¬ iszero (numeral w)⟩
 ⟨proof⟩

lemma *neg-numeral-eq-iff*: ⟨- numeral m = - numeral n ⟷ m = n⟩
 ⟨proof⟩

lemma *numeral-neq-neg-numeral*: ⟨numeral m ≠ - numeral n⟩
 ⟨proof⟩

lemma *neg-numeral-neq-numeral*: ⟨- numeral m ≠ numeral n⟩
 ⟨proof⟩

lemma *zero-neq-neg-numeral*: ⟨0 ≠ - numeral n⟩
 ⟨proof⟩

lemma *neg-numeral-neq-zero*: ⟨- numeral n ≠ 0⟩
 ⟨proof⟩

lemma *one-neq-neg-numeral*: ⟨1 ≠ - numeral n⟩
 ⟨proof⟩

lemma *neg-numeral-neq-one*: ⟨- numeral n ≠ 1⟩
 ⟨proof⟩

lemma *neg-one-neq-numeral*: ⟨- 1 ≠ numeral n⟩
 ⟨proof⟩

lemma *numeral-neq-neg-one*: ⟨numeral n ≠ - 1⟩
 ⟨proof⟩

lemma *neg-one-eq-numeral-iff*: $\langle -1 = -\text{numeral } n \longleftrightarrow n = \text{One} \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-eq-neg-one-iff*: $\langle -\text{numeral } n = -1 \longleftrightarrow n = \text{One} \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-one-neq-zero*: $\langle -1 \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *zero-neq-neg-one*: $\langle 0 \neq -1 \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-one-neq-one*: $\langle -1 \neq 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *one-neq-neg-one*: $\langle 1 \neq -1 \rangle$
 $\langle \text{proof} \rangle$

lemmas *eq-neg-numeral-simps* [*simp*] =
neg-numeral-eq-iff
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-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* $\langle \text{proof} \rangle$

lemma *neg-numeral-le-iff*: $\langle -\text{numeral } m \leq -\text{numeral } n \longleftrightarrow n \leq m \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-less-iff*: $\langle -\text{numeral } m < -\text{numeral } n \longleftrightarrow n < m \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-less-zero*: $\langle -\text{numeral } n < 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-le-zero*: $\langle -\text{numeral } n \leq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *not-zero-less-neg-numeral*: $\langle \neg 0 < - \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *not-zero-le-neg-numeral*: $\langle \neg 0 \leq - \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-less-numeral*: $\langle - \text{numeral } m < \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-le-numeral*: $\langle - \text{numeral } m \leq \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *not-numeral-less-neg-numeral*: $\langle \neg \text{numeral } m < - \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *not-numeral-le-neg-numeral*: $\langle \neg \text{numeral } m \leq - \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-less-one*: $\langle - \text{numeral } m < 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-le-one*: $\langle - \text{numeral } m \leq 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *not-one-less-neg-numeral*: $\langle \neg 1 < - \text{numeral } m \rangle$
 $\langle \text{proof} \rangle$

lemma *not-one-le-neg-numeral*: $\langle \neg 1 \leq - \text{numeral } m \rangle$
 $\langle \text{proof} \rangle$

lemma *not-numeral-less-neg-one*: $\langle \neg \text{numeral } m < - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *not-numeral-le-neg-one*: $\langle \neg \text{numeral } m \leq - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-one-less-numeral*: $\langle - 1 < \text{numeral } m \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-one-le-numeral*: $\langle - 1 \leq \text{numeral } m \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-less-neg-one-iff*: $\langle - \text{numeral } m < - 1 \longleftrightarrow m \neq \text{One} \rangle$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-le-neg-one*: $\langle - \text{numeral } m \leq - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *not-neg-one-less-neg-numeral*: $\langle \neg - 1 < - \text{numeral } m \rangle$

$\langle \text{proof} \rangle$

lemma *not-neg-one-le-neg-numeral-iff*: $\langle \neg - 1 \leq - \text{numeral } m \longleftrightarrow m \neq \text{One} \rangle$
 $\langle \text{proof} \rangle$

lemma *sub-non-negative*: $\langle \text{sub } n \ m \geq 0 \longleftrightarrow n \geq m \rangle$
 $\langle \text{proof} \rangle$

lemma *sub-positive*: $\langle \text{sub } n \ m > 0 \longleftrightarrow n > m \rangle$
 $\langle \text{proof} \rangle$

lemma *sub-non-positive*: $\langle \text{sub } n \ m \leq 0 \longleftrightarrow n \leq m \rangle$
 $\langle \text{proof} \rangle$

lemma *sub-negative*: $\langle \text{sub } n \ m < 0 \longleftrightarrow n < m \rangle$
 $\langle \text{proof} \rangle$

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]:
 $\langle - 1 \leq 0 \rangle$
 $\langle - 1 \leq 1 \rangle$
 $\langle \neg 0 \leq - 1 \rangle$
 $\langle \neg 1 \leq - 1 \rangle$
 $\langle \text{proof} \rangle$

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]:
 $\langle - 1 < 0 \rangle$
 $\langle - 1 < 1 \rangle$
 $\langle \neg 0 < - 1 \rangle$
 $\langle \neg 1 < - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *abs-numeral* [simp]: $\langle |\text{numeral } n| = \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *abs-neg-numeral* [*simp*]: $\langle |- \text{numeral } n | = \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *abs-neg-one* [*simp*]: $\langle |- 1 | = 1 \rangle$
 $\langle \text{proof} \rangle$

end

46.4.11 Natural numbers

lemma *numeral-num-of-nat*:
 $\langle \text{numeral } (\text{num-of-nat } n) = n \rangle \text{ if } \langle n > 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *Suc-1* [*simp*]: $\langle \text{Suc } 1 = 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *Suc-numeral* [*simp*]: $\langle \text{Suc } (\text{numeral } n) = \text{numeral } (n + \text{One}) \rangle$
 $\langle \text{proof} \rangle$

definition *pred-numeral* :: $\langle \text{num} \Rightarrow \text{nat} \rangle$
where $\langle \text{pred-numeral } k = \text{numeral } k - 1 \rangle$

declare [[*code drop: pred-numeral*]]

lemma *numeral-eq-Suc*: $\langle \text{numeral } k = \text{Suc } (\text{pred-numeral } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *eval-nat-numeral*:
 $\langle \text{numeral } \text{One} = \text{Suc } 0 \rangle$
 $\langle \text{numeral } (\text{Bit0 } n) = \text{Suc } (\text{numeral } (\text{BitM } n)) \rangle$
 $\langle \text{numeral } (\text{Bit1 } n) = \text{Suc } (\text{numeral } (\text{Bit0 } n)) \rangle$
 $\langle \text{proof} \rangle$

lemma *pred-numeral-simps* [*simp*]:
 $\langle \text{pred-numeral } \text{One} = 0 \rangle$
 $\langle \text{pred-numeral } (\text{Bit0 } k) = \text{numeral } (\text{BitM } k) \rangle$
 $\langle \text{pred-numeral } (\text{Bit1 } k) = \text{numeral } (\text{Bit0 } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *pred-numeral-inc* [*simp*]:
 $\langle \text{pred-numeral } (\text{inc } k) = \text{numeral } k \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-2-eq-2*: $\langle 2 = \text{Suc } (\text{Suc } 0) \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-3-eq-3*: $\langle 3 = \text{Suc } (\text{Suc } (\text{Suc } 0)) \rangle$

$\langle \text{proof} \rangle$

lemma *numeral-1-eq-Suc-0*: $\langle \text{Numeral1} = \text{Suc } 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *Suc-nat-number-of-add*: $\langle \text{Suc } (\text{numeral } v + n) = \text{numeral } (v + \text{One}) + n \rangle$
 $\langle \text{proof} \rangle$

lemma *numerals*: $\langle \text{Numeral1} = (1::\text{nat}) \rangle \langle 2 = \text{Suc } (\text{Suc } 0) \rangle$
 $\langle \text{proof} \rangle$

lemmas *numeral-nat* = *eval-nat-numeral BitM.simps One-nat-def*

Comparisons involving *Suc*.

lemma *eq-numeral-Suc* [simp]: $\langle \text{numeral } k = \text{Suc } n \longleftrightarrow \text{pred-numeral } k = n \rangle$
 $\langle \text{proof} \rangle$

lemma *Suc-eq-numeral* [simp]: $\langle \text{Suc } n = \text{numeral } k \longleftrightarrow n = \text{pred-numeral } k \rangle$
 $\langle \text{proof} \rangle$

lemma *less-numeral-Suc* [simp]: $\langle \text{numeral } k < \text{Suc } n \longleftrightarrow \text{pred-numeral } k < n \rangle$
 $\langle \text{proof} \rangle$

lemma *less-Suc-numeral* [simp]: $\langle \text{Suc } n < \text{numeral } k \longleftrightarrow n < \text{pred-numeral } k \rangle$
 $\langle \text{proof} \rangle$

lemma *le-numeral-Suc* [simp]: $\langle \text{numeral } k \leq \text{Suc } n \longleftrightarrow \text{pred-numeral } k \leq n \rangle$
 $\langle \text{proof} \rangle$

lemma *le-Suc-numeral* [simp]: $\langle \text{Suc } n \leq \text{numeral } k \longleftrightarrow n \leq \text{pred-numeral } k \rangle$
 $\langle \text{proof} \rangle$

lemma *diff-Suc-numeral* [simp]: $\langle \text{Suc } n - \text{numeral } k = n - \text{pred-numeral } k \rangle$
 $\langle \text{proof} \rangle$

lemma *diff-numeral-Suc* [simp]: $\langle \text{numeral } k - \text{Suc } n = \text{pred-numeral } k - n \rangle$
 $\langle \text{proof} \rangle$

lemma *max-Suc-numeral* [simp]: $\langle \text{max } (\text{Suc } n) (\text{numeral } k) = \text{Suc } (\text{max } n (\text{pred-numeral } k)) \rangle$
 $\langle \text{proof} \rangle$

lemma *max-numeral-Suc* [simp]: $\langle \text{max } (\text{numeral } k) (\text{Suc } n) = \text{Suc } (\text{max } (\text{pred-numeral } k) n) \rangle$
 $\langle \text{proof} \rangle$

lemma *min-Suc-numeral* [simp]: $\langle \text{min } (\text{Suc } n) (\text{numeral } k) = \text{Suc } (\text{min } n (\text{pred-numeral } k)) \rangle$
 $\langle \text{proof} \rangle$

lemma *min-numeral-Suc* [simp]: $\langle \text{min} (\text{numeral } k) (\text{Suc } n) = \text{Suc} (\text{min} (\text{pred-numeral } k) n) \rangle$
 $\langle \text{proof} \rangle$

For *case-nat* and *rec-nat*.

lemma *case-nat-numeral* [simp]: $\langle \text{case-nat } a f (\text{numeral } v) = (\text{let } pv = \text{pred-numeral } v \text{ in } f \text{ } pv) \rangle$
 $\langle \text{proof} \rangle$

lemma *case-nat-add-eq-if* [simp]:
 $\langle \text{case-nat } a f ((\text{numeral } v) + n) = (\text{let } pv = \text{pred-numeral } v \text{ in } f (pv + n)) \rangle$
 $\langle \text{proof} \rangle$

lemma *rec-nat-numeral* [simp]:
 $\langle \text{rec-nat } a f (\text{numeral } v) = (\text{let } pv = \text{pred-numeral } v \text{ in } f \text{ } pv (\text{rec-nat } a f \text{ } pv)) \rangle$
 $\langle \text{proof} \rangle$

lemma *rec-nat-add-eq-if* [simp]:
 $\langle \text{rec-nat } a f (\text{numeral } v + n) = (\text{let } pv = \text{pred-numeral } v \text{ in } f (pv + n) (\text{rec-nat } a f (pv + n))) \rangle$
 $\langle \text{proof} \rangle$

Case analysis on $n < (2::'a)$.

lemma *less-2-cases*: $\langle n < 2 \implies n = 0 \vee n = \text{Suc } 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *less-2-cases-iff*: $\langle n < 2 \longleftrightarrow n = 0 \vee n = \text{Suc } 0 \rangle$
 $\langle \text{proof} \rangle$

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]: $\langle 2 + n = \text{Suc} (\text{Suc } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *add-2-eq-Suc'* [simp]: $\langle n + 2 = \text{Suc} (\text{Suc } n) \rangle$
 $\langle \text{proof} \rangle$

Can be used to eliminate long strings of Sucs, but not by default.

lemma *Suc3-eq-add-3*: $\langle \text{Suc} (\text{Suc} (\text{Suc } n)) = 3 + n \rangle$
 $\langle \text{proof} \rangle$

lemmas *nat-1-add-1 = one-add-one* [where 'a=nat]

context *semiring-numeral*
begin

lemma *numeral-add-unfold-funpow*:

$\langle \text{numeral } k + a = ((+) \ 1 \ \sim\text{numeral } k) \ a \rangle$
 $\langle \text{proof} \rangle$

end

context *semiring-1*

begin

lemma *numeral-unfold-funpow*:

$\langle \text{numeral } k = ((+) \ 1 \ \sim\text{numeral } k) \ 0 \rangle$
 $\langle \text{proof} \rangle$

end

context

includes *lifting-syntax*

begin

lemma *transfer-rule-numeral*:

$\langle ((=) \implies R) \ \text{numeral} \ \text{numeral} \rangle$
if [*transfer-rule*]: $\langle R \ 0 \ 0 \rangle \ \langle R \ 1 \ 1 \rangle$
 $\langle (R \implies R \implies R) \ (+) \ (+) \rangle$
for $R :: \langle 'a :: \{\text{semiring-numeral}, \text{monoid-add}\} \Rightarrow 'b :: \{\text{semiring-numeral}, \text{monoid-add}\} \Rightarrow \text{bool} \rangle$
 $\langle \text{proof} \rangle$

end

46.5 Particular lemmas concerning $2::'a$

context *linordered-field*

begin

subclass *field-char-0* $\langle \text{proof} \rangle$

lemma *half-gt-zero-iff*: $\langle 0 < a / 2 \iff 0 < a \rangle$
 $\langle \text{proof} \rangle$

lemma *half-gt-zero* [*simp*]: $\langle 0 < a \implies 0 < a / 2 \rangle$
 $\langle \text{proof} \rangle$

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* $\langle \text{numeral } v \rangle$] **for** *v*

lemmas *minus-less-iff-numeral* [*no-atp*] =
minus-less-iff [*of* - $\langle \text{numeral } v \rangle$] **for** *v*

Cancellation of constant factors in comparisons ($<$ and \leq)

lemmas *mult-less-cancel-left-numeral* [*simp, no-atp*] = *mult-less-cancel-left* [*of* $\langle \text{numeral } v \rangle$] **for** *v*

lemmas *mult-less-cancel-right-numeral* [*simp, no-atp*] = *mult-less-cancel-right* [*of* - $\langle \text{numeral } v \rangle$] **for** *v*

lemmas *mult-le-cancel-left-numeral* [*simp, no-atp*] = *mult-le-cancel-left* [*of* $\langle \text{numeral } v \rangle$] **for** *v*

lemmas *mult-le-cancel-right-numeral* [*simp, no-atp*] = *mult-le-cancel-right* [*of* - $\langle \text{numeral } v \rangle$] **for** *v*

Multiplying out constant divisors in comparisons ($<$, \leq and $=$)

named-theorems *divide-const-simps* \langle *simplification rules to simplify comparisons involving constant divisors* \rangle

lemmas *le-divide-eq-numeral1* [*simp, divide-const-simps*] =
pos-le-divide-eq [*of* $\langle \text{numeral } w \rangle$, *OF zero-less-numeral*]
neg-le-divide-eq [*of* $\langle - \text{numeral } w \rangle$, *OF neg-numeral-less-zero*] **for** *w*

lemmas *divide-le-eq-numeral1* [*simp, divide-const-simps*] =
pos-divide-le-eq [*of* $\langle \text{numeral } w \rangle$, *OF zero-less-numeral*]
neg-divide-le-eq [*of* $\langle - \text{numeral } w \rangle$, *OF neg-numeral-less-zero*] **for** *w*

lemmas *less-divide-eq-numeral1* [*simp, divide-const-simps*] =
pos-less-divide-eq [*of* $\langle \text{numeral } w \rangle$, *OF zero-less-numeral*]
neg-less-divide-eq [*of* $\langle - \text{numeral } w \rangle$, *OF neg-numeral-less-zero*] **for** *w*

lemmas *divide-less-eq-numeral1* [*simp, divide-const-simps*] =
pos-divide-less-eq [*of* $\langle \text{numeral } w \rangle$, *OF zero-less-numeral*]
neg-divide-less-eq [*of* $\langle - \text{numeral } w \rangle$, *OF neg-numeral-less-zero*] **for** *w*

lemmas *eq-divide-eq-numeral1* [*simp, divide-const-simps*] =
eq-divide-eq [*of* - - $\langle \text{numeral } w \rangle$]
eq-divide-eq [*of* - - $\langle - \text{numeral } w \rangle$] **for** *w*

lemmas *divide-eq-eq-numeral1* [*simp, divide-const-simps*] =
divide-eq-eq [*of* - $\langle \text{numeral } w \rangle$]
divide-eq-eq [*of* - $\langle - \text{numeral } w \rangle$] **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 $\langle \text{numeral } w \rangle$]
le-divide-eq [of $\langle - \text{ numeral } w \rangle$] **for** w

lemmas *divide-le-eq-numeral* [*divide-const-simps*] =
divide-le-eq [of - - $\langle \text{numeral } w \rangle$]
divide-le-eq [of - - $\langle - \text{ numeral } w \rangle$] **for** w

lemmas *less-divide-eq-numeral* [*divide-const-simps*] =
less-divide-eq [of $\langle \text{numeral } w \rangle$]
less-divide-eq [of $\langle - \text{ numeral } w \rangle$] **for** w

lemmas *divide-less-eq-numeral* [*divide-const-simps*] =
divide-less-eq [of - - $\langle \text{numeral } w \rangle$]
divide-less-eq [of - - $\langle - \text{ numeral } w \rangle$] **for** w

lemmas *eq-divide-eq-numeral* [*divide-const-simps*] =
eq-divide-eq [of $\langle \text{numeral } w \rangle$]
eq-divide-eq [of $\langle - \text{ numeral } w \rangle$] **for** w

lemmas *divide-eq-eq-numeral* [*divide-const-simps*] =
divide-eq-eq [of - - $\langle \text{numeral } w \rangle$]
divide-eq-eq [of - - $\langle - \text{ numeral } w \rangle$] **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*: $\langle \text{Numeral1} * a = a \rangle$
for $a :: \langle 'a::\text{semiring-numeral} \rangle$
 $\langle \text{proof} \rangle$

lemma *mult-numeral-1-right*: $\langle a * \text{Numeral1} = a \rangle$
for $a :: \langle 'a::\text{semiring-numeral} \rangle$
 $\langle \text{proof} \rangle$

lemma *divide-numeral-1*: $\langle a / \text{Numeral1} = a \rangle$
for $a :: \langle 'a::\text{field} \rangle$
 $\langle \text{proof} \rangle$

lemma *inverse-numeral-1*: $\langle \text{inverse } \text{Numeral1} = (\text{Numeral1}::'a::\text{division-ring}) \rangle$
 $\langle \text{proof} \rangle$

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*:
 $\langle \text{Numeral1} * a = a \rangle$
 $\langle a * \text{Numeral1} = a \rangle$

for $a :: \langle 'a :: \text{semiring-numeral} \rangle$
 $\langle \text{proof} \rangle$

lemma *mult-1s-ring-1* :
 $\langle - \text{Numeral1} * b = - b \rangle$
 $\langle b * - \text{Numeral1} = - b \rangle$
for $b :: \langle 'a :: \text{ring-1} \rangle$
 $\langle \text{proof} \rangle$

lemmas *mult-1s = mult-1s-semiring-numeral mult-1s-ring-1*

$\langle \text{ML} \rangle$

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 mult-neg-numeral-simps
mult-zero-left mult-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 mult-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 =*
neg-le-iff-le
minus-zero left-minus right-minus
mult-1-left mult-1-right
mult-minus-left mult-minus-right
minus-add-distrib minus-minus mult.assoc

lemmas *of-nat-simps =*
of-nat-0 of-nat-1 of-nat-Suc of-nat-add of-nat-mult

Simplification of relational operations.

lemmas *eq-numeral-extra =*
zero-neq-one one-neq-zero

lemmas *rel-simps* =
le-num-simps less-num-simps eq-num-simps
le-numeral-simps le-neg-numeral-simps le-minus-one-simps le-numeral-extra
less-numeral-simps less-neg-numeral-simps less-minus-one-simps less-numeral-extra
eq-numeral-simps eq-neg-numeral-simps eq-numeral-extra

lemma *Let-numeral [simp]*: $\langle \text{Let } (\text{numeral } v) f = f (\text{numeral } v) \rangle$
 — Unfold all *lets* involving constants
 $\langle \text{proof} \rangle$

lemma *Let-neg-numeral [simp]*: $\langle \text{Let } (- \text{numeral } v) f = f (- \text{numeral } v) \rangle$
 — Unfold all *lets* involving constants
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

46.7.2 Simplification of arithmetic when nested to the right

lemma *add-numeral-left [simp]*: $\langle \text{numeral } v + (\text{numeral } w + z) = (\text{numeral}(v + w) + z) \rangle$
 $\langle \text{proof} \rangle$

lemma *add-neg-numeral-left [simp]*:
 $\langle \text{numeral } v + (- \text{numeral } w + y) = (\text{sub } v \ w \ + \ y) \rangle$
 $\langle - \text{numeral } v + (\text{numeral } w + y) = (\text{sub } w \ v \ + \ y) \rangle$
 $\langle - \text{numeral } v + (- \text{numeral } w + y) = (- \text{numeral}(v + w) + y) \rangle$
 $\langle \text{proof} \rangle$

lemma *mult-numeral-left-semiring-numeral*:
 $\langle \text{numeral } v * (\text{numeral } w * z) = (\text{numeral}(v * w) * z :: 'a::\text{semiring-numeral}) \rangle$
 $\langle \text{proof} \rangle$

lemma *mult-numeral-left-ring-1*:
 $\langle - \text{numeral } v * (\text{numeral } w * y) = (- \text{numeral}(v * w) * y :: 'a::\text{ring-1}) \rangle$
 $\langle \text{numeral } v * (- \text{numeral } w * y) = (- \text{numeral}(v * w) * y :: 'a::\text{ring-1}) \rangle$
 $\langle - \text{numeral } v * (- \text{numeral } w * y) = (\text{numeral}(v * w) * y :: 'a::\text{ring-1}) \rangle$
 $\langle \text{proof} \rangle$

lemmas *mult-numeral-left [simp]* =
mult-numeral-left-semiring-numeral
mult-numeral-left-ring-1

46.8 Code module namespace

code-identifier
`code-module` *Num* \rightarrow (*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 (inc n)⟩
⟨proof⟩

```

lemmas [code-post] = inc.simps

46.10 More on auxiliary conversion

context *semiring-1*

begin

lemma *num-of-nat-numeral-eq* [simp]:

```

⟨num-of-nat (numeral q) = q⟩
⟨proof⟩

```

lemma *numeral-num-of-nat-unfold*:

```

⟨numeral (num-of-nat n) = (if n = 0 then 1 else of-nat n)⟩
⟨proof⟩

```

end

hide-const (open) *One Bit0 Bit1 BitM inc pow sqr sub dbl dbl-inc dbl-dec*

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 (⟨(-)⟩ [1000] 1000)

Special syntax for squares.

abbreviation *power2* :: 'a ⇒ 'a (⟨(⟨notation=⟨postfix 2⟩⟩-2)⟩ [1000] 999)

where $x^2 \equiv x \wedge 2$

end

context

includes *lifting-syntax*

begin

lemma *power-transfer* [*transfer-rule*]:

$\langle (R \implies (=) \implies R) (\wedge) (\wedge) \rangle$

if [*transfer-rule*]: $\langle R \ 1 \ 1 \rangle$

$\langle (R \implies R \implies R) (*) (*) \rangle$

for $R :: \langle 'a::\text{power} \Rightarrow 'b::\text{power} \Rightarrow \text{bool} \rangle$

$\langle \text{proof} \rangle$

end

context *monoid-mult*

begin

subclass *power* $\langle \text{proof} \rangle$

lemma *power-one* [*simp*]: $1 \wedge n = 1$

$\langle \text{proof} \rangle$

lemma *power-one-right* [*simp*]: $a \wedge 1 = a$

$\langle \text{proof} \rangle$

lemma *power-Suc0-right* [*simp*]: $a \wedge \text{Suc } 0 = a$

$\langle \text{proof} \rangle$

lemma *power-commutes*: $a \wedge n * a = a * a \wedge n$

$\langle \text{proof} \rangle$

lemma *power-Suc2*: $a \wedge \text{Suc } n = a \wedge n * a$

$\langle \text{proof} \rangle$

lemma *power-add*: $a \wedge (m + n) = a \wedge m * a \wedge n$

$\langle \text{proof} \rangle$

lemma *power-mult*: $a \wedge (m * n) = (a \wedge m) \wedge n$

$\langle \text{proof} \rangle$

lemma *power-even-eq*: $a \wedge (2 * n) = (a \wedge n)^2$

$\langle \text{proof} \rangle$

lemma *power-odd-eq*: $a \wedge \text{Suc } (2*n) = a * (a \wedge n)^2$

$\langle \text{proof} \rangle$

lemma *power-numeral-even*: $z \wedge \text{numeral } (\text{Num.Bit0 } w) = (\text{let } w = z \wedge (\text{numeral$

w) in $w * w$)
 $\langle \text{proof} \rangle$

lemma *power-numeral-odd*: $z \wedge \text{numeral } (\text{Num.Bit1 } w) = (\text{let } w = z \wedge (\text{numeral } w) \text{ in } z * w * w)$
 $\langle \text{proof} \rangle$

lemma *power2-eq-square*: $a^2 = a * a$
 $\langle \text{proof} \rangle$

lemma *power3-eq-cube*: $a \wedge 3 = a * a * a$
 $\langle \text{proof} \rangle$

lemma *power4-eq-xxxx*: $x \wedge 4 = x * x * x * x$
 $\langle \text{proof} \rangle$

lemma *power-numeral-reduce*: $x \wedge \text{numeral } n = x * x \wedge \text{pred-numeral } n$
 $\langle \text{proof} \rangle$

lemma *funpow-times-power*: $(\text{times } x \wedge\wedge f x) = \text{times } (x \wedge f x)$
 $\langle \text{proof} \rangle$

lemma *power-commuting-commutes*:
 assumes $x * y = y * x$
 shows $x \wedge n * y = y * x \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-minus-mult*: $0 < n \implies a \wedge (n - 1) * a = a \wedge n$
 $\langle \text{proof} \rangle$

lemma *left-right-inverse-power*:
 assumes $x * y = 1$
 shows $x \wedge n * y \wedge n = 1$
 $\langle \text{proof} \rangle$

end

context *comm-monoid-mult*
begin

lemma *power-mult-distrib* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*, *divide-simps*]:
 $(a * b) \wedge n = (a \wedge n) * (b \wedge n)$
 $\langle \text{proof} \rangle$

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 $a :: 'a::\text{monoid-mult}$
 $\langle \text{proof} \rangle$

lemma *power-add-numeral2* [*simp*]: $a^{\text{numeral } m} * (a^{\text{numeral } n} * b) = a^{\text{numeral } (m + n)} * b$
for $a :: 'a::\text{monoid-mult}$
 $\langle \text{proof} \rangle$

lemma *power-mult-numeral* [*simp*]: $(a^{\text{numeral } m})^{\text{numeral } n} = a^{\text{numeral } (m * n)}$
for $a :: 'a::\text{monoid-mult}$
 $\langle \text{proof} \rangle$

context *semiring-numeral*
begin

lemma *numeral-sqr*: $\text{numeral } (\text{Num.sqr } k) = \text{numeral } k * \text{numeral } k$
 $\langle \text{proof} \rangle$

lemma *numeral-pow*: $\text{numeral } (\text{Num.pow } k \ l) = \text{numeral } k^{\text{numeral } l}$
 $\langle \text{proof} \rangle$

lemma *power-numeral* [*simp*]: $\text{numeral } k^{\text{numeral } l} = \text{numeral } (\text{Num.pow } k \ l)$
 $\langle \text{proof} \rangle$

end

context *semiring-1*
begin

lemma *of-nat-power* [*simp*]: $\text{of-nat } (m^{\text{numeral } n}) = \text{of-nat } m^{\text{numeral } n}$
 $\langle \text{proof} \rangle$

lemma *zero-power*: $0 < n \implies 0^{\text{numeral } n} = 0$
 $\langle \text{proof} \rangle$

lemma *power-zero-numeral* [*simp*]: $0^{\text{numeral } k} = 0$
 $\langle \text{proof} \rangle$

lemma *zero-power2*: $0^2 = 0$
 $\langle \text{proof} \rangle$

lemma *one-power2*: $1^2 = 1$
 $\langle \text{proof} \rangle$

lemma *power-0-Suc* [simp]: $0 \wedge \text{Suc } n = 0$
 ⟨proof⟩

It looks plausible as a simprule, but its effect can be strange.

lemma *power-0-left*: $0 \wedge n = (\text{if } n = 0 \text{ then } 1 \text{ else } 0)$
 ⟨proof⟩

end

context *semiring-char-0* **begin**

lemma *numeral-power-eq-of-nat-cancel-iff* [simp]:
 $\text{numeral } x \wedge n = \text{of-nat } y \longleftrightarrow \text{numeral } x \wedge n = y$
 ⟨proof⟩

lemma *real-of-nat-eq-numeral-power-cancel-iff* [simp]:
 $\text{of-nat } y = \text{numeral } x \wedge n \longleftrightarrow y = \text{numeral } x \wedge n$
 ⟨proof⟩

lemma *of-nat-eq-of-nat-power-cancel-iff* [simp]: $(\text{of-nat } b) \wedge w = \text{of-nat } x \longleftrightarrow b \wedge w = x$
 ⟨proof⟩

lemma *of-nat-power-eq-of-nat-cancel-iff* [simp]: $\text{of-nat } x = (\text{of-nat } b) \wedge w \longleftrightarrow x = b \wedge w$
 ⟨proof⟩

end

context *comm-semiring-1*
begin

The divides relation.

lemma *le-imp-power-dvd*:
 assumes $m \leq n$
 shows $a \wedge m \text{ dvd } a \wedge n$
 ⟨proof⟩

lemma *power-le-dvd*: $a \wedge n \text{ dvd } b \implies m \leq n \implies a \wedge m \text{ dvd } b$
 ⟨proof⟩

lemma *dvd-power-same*: $x \text{ dvd } y \implies x \wedge n \text{ dvd } y \wedge n$
 ⟨proof⟩

lemma *dvd-power-le*: $x \text{ dvd } y \implies m \geq n \implies x \wedge n \text{ dvd } y \wedge m$
 ⟨proof⟩

lemma *dvd-power* [simp]:
 fixes $n :: \text{nat}$

```

assumes  $n > 0 \vee x = 1$ 
shows  $x \text{ dvd } (x \wedge n)$ 
 $\langle \text{proof} \rangle$ 

end

context semiring-1-no-zero-divisors
begin

subclass power  $\langle \text{proof} \rangle$ 

lemma power-eq-0-iff [simp]:  $a \wedge n = 0 \longleftrightarrow a = 0 \wedge n > 0$ 
 $\langle \text{proof} \rangle$ 

lemma power-not-zero:  $a \neq 0 \implies a \wedge n \neq 0$ 
 $\langle \text{proof} \rangle$ 

lemma zero-eq-power2 [simp]:  $a^2 = 0 \longleftrightarrow a = 0$ 
 $\langle \text{proof} \rangle$ 

end

context ring-1
begin

lemma power-minus:  $(-a) \wedge n = (-1) \wedge n * a \wedge n$ 
 $\langle \text{proof} \rangle$ 

lemma power-minus': NO-MATCH  $1\ x \implies (-x) \wedge n = (-1) \wedge n * x \wedge n$ 
 $\langle \text{proof} \rangle$ 

lemma power-minus-Bit0:  $(-x) \wedge \text{numeral } (\text{Num.Bit0 } k) = x \wedge \text{numeral } (\text{Num.Bit0 } k)$ 
 $\langle \text{proof} \rangle$ 

lemma power-minus-Bit1:  $(-x) \wedge \text{numeral } (\text{Num.Bit1 } k) = - (x \wedge \text{numeral } (\text{Num.Bit1 } k))$ 
 $\langle \text{proof} \rangle$ 

lemma power2-minus [simp]:  $(-a)^2 = a^2$ 
 $\langle \text{proof} \rangle$ 

lemma power-minus1-even [simp]:  $(-1) \wedge (2*n) = 1$ 
 $\langle \text{proof} \rangle$ 

lemma power-minus1-odd:  $(-1) \wedge \text{Suc } (2*n) = -1$ 
 $\langle \text{proof} \rangle$ 

lemma power-minus-even [simp]:  $(-a) \wedge (2*n) = a \wedge (2*n)$ 

```

```

    <proof>

end

context ring-1-no-zero-divisors
begin

lemma power2-eq-1-iff:  $a^2 = 1 \longleftrightarrow a = 1 \vee a = -1$ 
  <proof>

end

context idom
begin

lemma power2-eq-iff:  $x^2 = y^2 \longleftrightarrow x = y \vee x = -y$ 
  <proof>

end

context semidom-divide
begin

lemma power-diff:
   $a \wedge (m - n) = (a \wedge m) \text{ div } (a \wedge n)$  if  $a \neq 0$  and  $n \leq m$ 
  <proof>

lemma power-diff-if:
   $a \wedge (m - n) = (a \wedge m) \text{ div } (a \wedge n)$  else 1 if  $a \neq 0$ 
  <proof>

end

context algebraic-semidom
begin

lemma div-power:  $b \text{ dvd } a \implies (a \text{ div } b) \wedge n = a \wedge n \text{ div } b \wedge n$ 
  <proof>

lemma is-unit-power-iff:  $\text{is-unit } (a \wedge n) \longleftrightarrow \text{is-unit } a \vee n = 0$ 
  <proof>

lemma dvd-power-iff:
  assumes  $x \neq 0$ 
  shows  $x \wedge m \text{ dvd } x \wedge n \longleftrightarrow \text{is-unit } x \vee m \leq n$ 
  <proof>

end

```

context *normalization-semidom-multiplicative*
begin

lemma *normalize-power*: $\text{normalize } (a \wedge n) = \text{normalize } a \wedge n$
 $\langle \text{proof} \rangle$

lemma *unit-factor-power*: $\text{unit-factor } (a \wedge n) = \text{unit-factor } a \wedge n$
 $\langle \text{proof} \rangle$

end

context *division-ring*
begin

Perhaps these should be simprules.

lemma *power-inverse* [*field-simps, field-split-simps, divide-simps*]: $\text{inverse } a \wedge n = \text{inverse } (a \wedge n)$
 $\langle \text{proof} \rangle$

lemma *power-one-over* [*field-simps, field-split-simps, divide-simps*]: $(1 / a) \wedge n = 1 / a \wedge n$
 $\langle \text{proof} \rangle$

end

context *field*
begin

lemma *power-divide* [*field-simps, field-split-simps, divide-simps*]: $(a / b) \wedge n = a \wedge n / b \wedge n$
 $\langle \text{proof} \rangle$

end

47.2 Exponentiation on ordered types

context *ordered-semiring-1*
begin

lemma *zero-le-power* [*simp*]: $0 \leq a \implies 0 \leq a \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-mono*: $a \leq b \implies 0 \leq a \implies a \wedge n \leq b \wedge n$
 $\langle \text{proof} \rangle$

lemma *one-le-power* [*simp*]: $1 \leq a \implies 1 \leq a \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-le-one*: $0 \leq a \implies a \leq 1 \implies a \wedge n \leq 1$
 ⟨proof⟩

lemma *power-gt1-lemma*:
 assumes *gt1*: $1 < a$
 shows $1 < a * a \wedge n$
 ⟨proof⟩

lemma *power-gt1*: $1 < a \implies 1 < a \wedge \text{Suc } n$
 ⟨proof⟩

lemma *one-less-power* [*simp*]: $1 < a \implies 0 < n \implies 1 < a \wedge n$
 ⟨proof⟩

Proof resembles that of *power-strict-decreasing*.

lemma *power-increasing*: $n \leq N \implies 1 \leq a \implies a \wedge n \leq a \wedge N$
 ⟨proof⟩

Proof resembles that of *power-strict-decreasing*.

lemma *power-decreasing*: $n \leq N \implies 0 \leq a \implies a \leq 1 \implies a \wedge N \leq a \wedge n$
 ⟨proof⟩

lemma *power-Suc-le-self*: $0 \leq a \implies a \leq 1 \implies a \wedge \text{Suc } n \leq a$
 ⟨proof⟩

end

context *linordered-semidom*
begin

lemma *zero-less-power* [*simp*]: $0 < a \implies 0 < a \wedge n$
 ⟨proof⟩

lemma *power-le-imp-le-exp*:
 assumes *gt1*: $1 < a$
 shows $a \wedge m \leq a \wedge n \implies m \leq n$
 ⟨proof⟩

lemma *of-nat-zero-less-power-iff* [*simp*]: $\text{of-nat } x \wedge n > 0 \longleftrightarrow x > 0 \vee n = 0$
 ⟨proof⟩

lemma *power-strict-mono*: $a < b \implies 0 \leq a \implies 0 < n \implies a \wedge n < b \wedge n$
 ⟨proof⟩

lemma *power-mono-iff* [*simp*]:
 shows $\llbracket a \geq 0; b \geq 0; n > 0 \rrbracket \implies a \wedge n \leq b \wedge n \longleftrightarrow a \leq b$
 ⟨proof⟩

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*: $0 < a \implies a < 1 \implies a * a \wedge n < a \wedge n$
 ⟨proof⟩

lemma *power-strict-decreasing*: $n < N \implies 0 < a \implies a < 1 \implies a \wedge N < a \wedge n$
 ⟨proof⟩

lemma *power-decreasing-iff* [simp]: $\llbracket 0 < b; b < 1 \rrbracket \implies b \wedge m \leq b \wedge n \longleftrightarrow n \leq m$
 ⟨proof⟩

lemma *power-strict-decreasing-iff* [simp]: $\llbracket 0 < b; b < 1 \rrbracket \implies b \wedge m < b \wedge n \longleftrightarrow n < m$
 ⟨proof⟩

lemma *power-Suc-less-one*: $0 < a \implies a < 1 \implies a \wedge \text{Suc } n < 1$
 ⟨proof⟩

Lemma for *power-strict-increasing*.

lemma *power-less-power-Suc*: $1 < a \implies a \wedge n < a * a \wedge n$
 ⟨proof⟩

lemma *power-strict-increasing*: $n < N \implies 1 < a \implies a \wedge n < a \wedge N$
 ⟨proof⟩

Surely we can strengthen this? It holds for $0 < a < 1$ too.

lemma *power-inject-exp* [simp]:
 $\langle a \wedge m = a \wedge n \longleftrightarrow m = n \rangle$ **if** $\langle 1 < a \rangle$
 ⟨proof⟩

lemma *power-inject-exp'*:
 $\langle a \wedge m = a \wedge n \longleftrightarrow m = n \rangle$ **if** $\langle a \neq 1 \rangle \langle a > 0 \rangle$
 ⟨proof⟩

Can relax the first premise to $0 < a$ in the case of the natural numbers.

lemma *power-less-imp-less-exp*: $1 < a \implies a \wedge m < a \wedge n \implies m < n$
 ⟨proof⟩

lemma *power-increasing-iff* [simp]: $1 < b \implies b \wedge x \leq b \wedge y \longleftrightarrow x \leq y$
 ⟨proof⟩

lemma *power-strict-increasing-iff* [simp]: $1 < b \implies b \wedge x < b \wedge y \longleftrightarrow x < y$
 ⟨proof⟩

lemma *power-le-imp-le-base*:
 assumes $le: a \wedge \text{Suc } n \leq b \wedge \text{Suc } n$
 and $0 \leq b$
 shows $a \leq b$
 ⟨proof⟩

lemma *power-less-imp-less-base*:

assumes *less*: $a \wedge n < b \wedge n$

assumes *nonneg*: $0 \leq b$

shows $a < b$

<proof>

lemma *power-inject-base*: $a \wedge \text{Suc } n = b \wedge \text{Suc } n \implies 0 \leq a \implies 0 \leq b \implies a = b$

<proof>

lemma *power-eq-imp-eq-base*: $a \wedge n = b \wedge n \implies 0 \leq a \implies 0 \leq b \implies 0 < n \implies a = b$

<proof>

lemma *power-eq-iff-eq-base*: $0 < n \implies 0 \leq a \implies 0 \leq b \implies a \wedge n = b \wedge n \longleftrightarrow a = b$

<proof>

lemma *power2-le-imp-le*: $x^2 \leq y^2 \implies 0 \leq y \implies x \leq y$

<proof>

lemma *power2-less-imp-less*: $x^2 < y^2 \implies 0 \leq y \implies x < y$

<proof>

lemma *power2-eq-imp-eq*: $x^2 = y^2 \implies 0 \leq x \implies 0 \leq y \implies x = y$

<proof>

lemma *power2-eq-iff-nonneg* [simp]:

assumes $0 \leq x$ $0 \leq y$

shows $(x \wedge 2 = y \wedge 2) \longleftrightarrow x = y$

<proof>

lemma *of-nat-less-numeral-power-cancel-iff*[simp]:

of-nat $x < \text{numeral } i \wedge n \longleftrightarrow x < \text{numeral } i \wedge n$

<proof>

lemma *of-nat-le-numeral-power-cancel-iff*[simp]:

of-nat $x \leq \text{numeral } i \wedge n \longleftrightarrow x \leq \text{numeral } i \wedge n$

<proof>

lemma *numeral-power-less-of-nat-cancel-iff*[simp]:

numeral $i \wedge n < \text{of-nat } x \longleftrightarrow \text{numeral } i \wedge n < x$

<proof>

lemma *numeral-power-le-of-nat-cancel-iff*[simp]:

numeral $i \wedge n \leq \text{of-nat } x \longleftrightarrow \text{numeral } i \wedge n \leq x$

<proof>

lemma *of-nat-le-of-nat-power-cancel-iff*[simp]: $(\text{of-nat } b) \wedge w \leq \text{of-nat } x \longleftrightarrow b \wedge w \leq x$

$\langle \text{proof} \rangle$

lemma *of-nat-power-le-of-nat-cancel-iff[simp]*: $\text{of-nat } x \leq (\text{of-nat } b) \wedge w \longleftrightarrow x \leq b \wedge w$
 $\langle \text{proof} \rangle$

lemma *of-nat-less-of-nat-power-cancel-iff[simp]*: $(\text{of-nat } b) \wedge w < \text{of-nat } x \longleftrightarrow b \wedge w < x$
 $\langle \text{proof} \rangle$

lemma *of-nat-power-less-of-nat-cancel-iff[simp]*: $\text{of-nat } x < (\text{of-nat } b) \wedge w \longleftrightarrow x < b \wedge w$
 $\langle \text{proof} \rangle$

lemma *power2-nonneg-ge-1-iff*:
assumes $x \geq 0$
shows $x \wedge 2 \geq 1 \longleftrightarrow x \geq 1$
 $\langle \text{proof} \rangle$

lemma *power2-nonneg-gt-1-iff*:
assumes $x \geq 0$
shows $x \wedge 2 > 1 \longleftrightarrow x > 1$
 $\langle \text{proof} \rangle$

end

Some *nat*-specific lemmas:

lemma *mono-ge2-power-minus-self*:
assumes $k \geq 2$ **shows** *mono* $(\lambda m. k \wedge m - m)$
 $\langle \text{proof} \rangle$

lemma *self-le-ge2-pow[simp]*:
assumes $k \geq 2$ **shows** $m \leq k \wedge m$
 $\langle \text{proof} \rangle$

lemma *diff-le-diff-pow[simp]*:
assumes $k \geq 2$ **shows** $m - n \leq k \wedge m - k \wedge n$
 $\langle \text{proof} \rangle$

context *linordered-ring-strict*
begin

lemma *sum-squares-eq-zero-iff*: $x * x + y * y = 0 \longleftrightarrow x = 0 \wedge y = 0$
 $\langle \text{proof} \rangle$

lemma *sum-squares-le-zero-iff*: $x * x + y * y \leq 0 \longleftrightarrow x = 0 \wedge y = 0$
 $\langle \text{proof} \rangle$

lemma *sum-squares-gt-zero-iff*: $0 < x * x + y * y \longleftrightarrow x \neq 0 \vee y \neq 0$
 ⟨proof⟩

end

context *linordered-idom*
begin

lemma *zero-le-power2* [*simp*]: $0 \leq a^2$
 ⟨proof⟩

lemma *zero-less-power2* [*simp*]: $0 < a^2 \longleftrightarrow a \neq 0$
 ⟨proof⟩

lemma *power2-less-0* [*simp*]: $\neg a^2 < 0$
 ⟨proof⟩

lemma *power-abs*: $|a \wedge n| = |a| \wedge n$ — FIXME simp?
 ⟨proof⟩

lemma *power-sgn* [*simp*]: $\text{sgn } (a \wedge n) = \text{sgn } a \wedge n$
 ⟨proof⟩

lemma *abs-power-minus* [*simp*]: $|(- a) \wedge n| = |a \wedge n|$
 ⟨proof⟩

lemma *zero-less-power-abs-iff* [*simp*]: $0 < |a| \wedge n \longleftrightarrow a \neq 0 \vee n = 0$
 ⟨proof⟩

lemma *zero-le-power-abs* [*simp*]: $0 \leq |a| \wedge n$
 ⟨proof⟩

lemma *power2-less-eq-zero-iff* [*simp*]: $a^2 \leq 0 \longleftrightarrow a = 0$
 ⟨proof⟩

lemma *abs-power2* [*simp*]: $|a^2| = a^2$
 ⟨proof⟩

lemma *power2-abs* [*simp*]: $|a|^2 = a^2$
 ⟨proof⟩

lemma *odd-power-less-zero*: $a < 0 \implies a \wedge \text{Suc } (2 * n) < 0$
 ⟨proof⟩

lemma *odd-0-le-power-imp-0-le*: $0 \leq a \wedge \text{Suc } (2 * n) \implies 0 \leq a$
 ⟨proof⟩

lemma *zero-le-even-power'* [*simp*]: $0 \leq a \wedge (2 * n)$
 ⟨proof⟩

lemma *sum-power2-ge-zero*: $0 \leq x^2 + y^2$
 ⟨proof⟩

lemma *not-sum-power2-lt-zero*: $\neg x^2 + y^2 < 0$
 ⟨proof⟩

lemma *sum-power2-eq-zero-iff*: $x^2 + y^2 = 0 \longleftrightarrow x = 0 \wedge y = 0$
 ⟨proof⟩

lemma *sum-power2-le-zero-iff*: $x^2 + y^2 \leq 0 \longleftrightarrow x = 0 \wedge y = 0$
 ⟨proof⟩

lemma *sum-power2-gt-zero-iff*: $0 < x^2 + y^2 \longleftrightarrow x \neq 0 \vee y \neq 0$
 ⟨proof⟩

lemma *abs-le-square-iff*: $|x| \leq |y| \longleftrightarrow x^2 \leq y^2$
 (is ?lhs \longleftrightarrow ?rhs)
 ⟨proof⟩

lemma *power2-le-iff-abs-le*:
 $y \geq 0 \implies x^2 \leq y^2 \longleftrightarrow |x| \leq y$
 ⟨proof⟩

lemma *abs-square-le-1*: $x^2 \leq 1 \longleftrightarrow |x| \leq 1$
 ⟨proof⟩

lemma *abs-square-eq-1*: $x^2 = 1 \longleftrightarrow |x| = 1$
 ⟨proof⟩

lemma *abs-square-less-1*: $x^2 < 1 \longleftrightarrow |x| < 1$
 ⟨proof⟩

lemma *square-le-1*:
 assumes $-1 \leq x \leq 1$
 shows $x^2 \leq 1$
 ⟨proof⟩

lemma *power2-mono*: $|x| \leq |y| \implies x^2 \leq y^2$
 ⟨proof⟩

lemma *power2-strict-mono*:
 assumes $|x| < |y|$
 shows $x^2 < y^2$
 ⟨proof⟩

end

47.3 Miscellaneous rules

context *linordered-semidom*

begin

lemma *self-le-power*: $1 \leq a \implies 0 < n \implies a \leq a \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-le-one-iff*: $0 \leq a \implies a \wedge n \leq 1 \longleftrightarrow (n = 0 \vee a \leq 1)$
 $\langle \text{proof} \rangle$

lemma *power-less1-D*: $a \wedge n < 1 \implies a < 1$
 $\langle \text{proof} \rangle$

lemma *power-less-one-iff*: $0 \leq a \implies a \wedge n < 1 \longleftrightarrow (n > 0 \wedge a < 1)$
 $\langle \text{proof} \rangle$

end

lemma *power2-ge-1-iff*: $x \wedge 2 \geq 1 \longleftrightarrow x \geq 1 \vee x \leq (-1 :: 'a :: \text{linordered-idom})$
 $\langle \text{proof} \rangle$

lemma *power2-less-1-iff*: $x^2 < 1 \longleftrightarrow (-1 :: 'a :: \text{linordered-idom}) < x \wedge x < 1$
 $\langle \text{proof} \rangle$

lemma *power2-gt-1-iff*: $x^2 > 1 \longleftrightarrow x < (-1 :: 'a :: \text{linordered-idom}) \vee x > 1$
 $\langle \text{proof} \rangle$

lemma (**in** *power*) *power-eq-if*: $p \wedge m = (\text{if } m=0 \text{ then } 1 \text{ else } p * (p \wedge (m - 1)))$
 $\langle \text{proof} \rangle$

lemma (**in** *comm-semiring-1*) *power2-sum*: $(x + y)^2 = x^2 + y^2 + 2 * x * y$
 $\langle \text{proof} \rangle$

context *comm-ring-1*

begin

lemma *power2-diff*: $(x - y)^2 = x^2 + y^2 - 2 * x * y$
 $\langle \text{proof} \rangle$

lemma *power2-commute*: $(x - y)^2 = (y - x)^2$
 $\langle \text{proof} \rangle$

lemma *minus-power-mult-self*: $(-a) \wedge n * (-a) \wedge n = a \wedge (2 * n)$
 $\langle \text{proof} \rangle$

lemma *minus-one-mult-self* [*simp*]: $(-1) \wedge n * (-1) \wedge n = 1$
 $\langle \text{proof} \rangle$

lemma *left-minus-one-mult-self* [*simp*]: $(-1) \wedge n * ((-1) \wedge n * a) = a$

$\langle \text{proof} \rangle$

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]: $\text{Suc } 0 \leq i \implies \text{Suc } 0 \leq i^n$
 $\langle \text{proof} \rangle$

lemma *nat-zero-less-power-iff* [simp]: $x^n > 0 \iff x > 0 \vee n = 0$
for $x :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *nat-power-eq-Suc-0-iff* [simp]: $x^m = \text{Suc } 0 \iff m = 0 \vee x = \text{Suc } 0$
 $\langle \text{proof} \rangle$

lemma *power-Suc-0* [simp]: $\text{Suc } 0^n = \text{Suc } 0$
 $\langle \text{proof} \rangle$

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 $i :: \text{nat}$
assumes *nonneg*: $0 < i$
assumes *less*: $i^m < i^n$
shows $m < n$
 $\langle \text{proof} \rangle$

lemma *power-gt-expt*: $n > \text{Suc } 0 \implies n^k > k$
 $\langle \text{proof} \rangle$

lemma *less-exp* [simp]:
 $\langle n < 2^n \rangle$
 $\langle \text{proof} \rangle$

lemma *power-dvd-imp-le*:
fixes $i :: \text{nat}$
assumes $i^m \text{ dvd } i^n$ $1 < i$
shows $m \leq n$
 $\langle \text{proof} \rangle$

lemma *dvd-power-iff-le*:

fixes *k::nat*

shows $2 \leq k \implies ((k \wedge m) \text{ dvd } (k \wedge n) \longleftrightarrow m \leq n)$

<proof>

lemma *power2-nat-le-eq-le*: $m^2 \leq n^2 \longleftrightarrow m \leq n$

for *m n :: nat*

<proof>

lemma *power2-nat-le-imp-le*:

fixes *m n :: nat*

assumes $m^2 \leq n$

shows $m \leq n$

<proof>

lemma *ex-power-ivl1*: **fixes** *b k :: nat* **assumes** $b \geq 2$

shows $k \geq 1 \implies \exists n. b^n \leq k \wedge k < b^{n+1}$ (**is** $\implies \exists n. ?P\ k\ n$)

<proof>

lemma *ex-power-ivl2*: **fixes** *b k :: nat* **assumes** $b \geq 2\ k \geq 2$

shows $\exists n. b^n < k \wedge k \leq b^{n+1}$

<proof>

47.4.1 Cardinality of the Powerset

lemma *card-UNIV-bool* [*simp*]: $\text{card } (\text{UNIV} :: \text{bool set}) = 2$

<proof>

lemma *card-Pow*: $\text{finite } A \implies \text{card } (\text{Pow } A) = 2^{\text{card } A}$

<proof>

47.5 Code generator tweak

code-identifier

code-module *Power* \mapsto (*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
 $\langle proof \rangle$

interpretation *comp?*: *comp-fun-commute* $f \circ g$
 $\langle proof \rangle$

definition $F :: ('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'a$
where *eq-fold*: $F \ g \ A = Finite\text{-}Set.fold \ (f \circ g) \ \mathbf{1} \ A$

lemma *infinite [simp]*: $\neg \text{finite } A \Longrightarrow F \ g \ A = \mathbf{1}$
 $\langle proof \rangle$

lemma *empty [simp]*: $F \ g \ \{\} = \mathbf{1}$
 $\langle proof \rangle$

lemma *insert [simp]*: $\text{finite } A \Longrightarrow x \notin A \Longrightarrow F \ g \ (\text{insert } x \ A) = g \ x * F \ g \ A$
 $\langle proof \rangle$

lemma *remove*:
assumes *finite* A **and** $x \in A$
shows $F \ g \ A = g \ x * F \ g \ (A - \{x\})$
 $\langle proof \rangle$

lemma *insert-remove*: $\text{finite } A \Longrightarrow F \ g \ (\text{insert } x \ A) = g \ x * F \ g \ (A - \{x\})$
 $\langle proof \rangle$

lemma *insert-if*: $\text{finite } A \Longrightarrow F \ g \ (\text{insert } x \ A) = (\text{if } x \in A \text{ then } F \ g \ A \text{ else } g \ x * F \ g \ A)$
 $\langle proof \rangle$

lemma *neutral*: $\forall x \in A. \ g \ x = \mathbf{1} \Longrightarrow F \ g \ A = \mathbf{1}$
 $\langle proof \rangle$

lemma *neutral-const [simp]*: $F \ (\lambda \cdot. \mathbf{1}) \ A = \mathbf{1}$
 $\langle proof \rangle$

lemma *union-inter*:
assumes *finite* A **and** *finite* B
shows $F \ g \ (A \cup B) * F \ g \ (A \cap B) = F \ g \ A * F \ g \ B$
— The reversed orientation looks more natural, but LOOPS as a simprule!
 $\langle proof \rangle$

corollary *union-inter-neutral*:
assumes *finite* A **and** *finite* B
and $\forall x \in A \cap B. \ g \ x = \mathbf{1}$

shows $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$
 $\langle proof \rangle$

corollary *union-disjoint:*

assumes *finite A and finite B*
assumes $A \cap B = \{\}$
shows $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$
 $\langle proof \rangle$

lemma *union-diff2:*

assumes *finite A and finite B*
shows $F\ g\ (A \cup B) = F\ g\ (A - B) * F\ g\ (B - A) * F\ g\ (A \cap B)$
 $\langle proof \rangle$

lemma *subset-diff:*

assumes $B \subseteq A$ **and** *finite A*
shows $F\ g\ A = F\ g\ (A - B) * F\ g\ B$
 $\langle proof \rangle$

lemma *Int-Diff:*

assumes *finite A*
shows $F\ g\ A = F\ g\ (A \cap B) * F\ g\ (A - B)$
 $\langle proof \rangle$

lemma *setdiff-irrelevant:*

assumes *finite A*
shows $F\ g\ (A - \{x.\ g\ x = z\}) = F\ g\ A$
 $\langle proof \rangle$

lemma *not-neutral-contains-not-neutral:*

assumes $F\ g\ A \neq 1$
obtains a **where** $a \in A$ **and** $g\ a \neq 1$
 $\langle proof \rangle$

lemma *reindex:*

assumes *inj-on h A*
shows $F\ g\ (h\ ' A) = F\ (g \circ h)\ A$
 $\langle proof \rangle$

lemma *cong [fundef-cong]:*

assumes $A = B$
assumes $g\text{-}h: \bigwedge x. x \in B \implies g\ x = h\ x$
shows $F\ g\ A = F\ h\ B$
 $\langle proof \rangle$

lemma *cong-simp [cong]:*

$\llbracket A = B; \bigwedge x. x \in B =_{simp} \implies g\ x = h\ x \rrbracket \implies F\ (\lambda x. g\ x)\ A = F\ (\lambda x. h\ x)\ B$
 $\langle proof \rangle$

lemma *reindex-cong*:

assumes *inj-on* l B

assumes $A = l \text{ ` } B$

assumes $\bigwedge x. x \in B \implies g (l \ x) = h \ x$

shows $F \ g \ A = F \ h \ B$

<proof>

lemma *image-eq*:

assumes *inj-on* g A

shows $F (\lambda x. x) (g \text{ ` } A) = F \ g \ A$

<proof>

lemma *UNION-disjoint*:

assumes *finite* I **and** $\forall i \in I. \text{finite } (A \ i)$

and $\forall i \in I. \forall j \in I. i \neq j \longrightarrow A \ i \cap A \ j = \{\}$

shows $F \ g \ (\bigcup (A \text{ ` } I)) = F (\lambda x. F \ g \ (A \ x)) \ I$

<proof>

lemma *Union-disjoint*:

assumes $\forall A \in C. \text{finite } A \ \forall A \in C. \forall B \in C. A \neq B \longrightarrow A \cap B = \{\}$

shows $F \ g \ (\bigcup C) = (F \circ F) \ g \ C$

<proof>

lemma *distrib*: $F (\lambda x. g \ x * h \ x) \ A = F \ g \ A * F \ h \ A$

<proof>

lemma *Sigma*:

assumes *finite* $A \ \forall x \in A. \text{finite } (B \ x)$

shows $F (\lambda x. F \ (g \ x) \ (B \ x)) \ A = F \ (\text{case-prod } g) \ (\text{SIGMA } x:A. B \ x)$

<proof>

lemma *related*:

assumes *Re*: $R \ 1 \ 1$

and *Rop*: $\forall x1 \ y1 \ x2 \ y2. R \ x1 \ x2 \wedge R \ y1 \ y2 \longrightarrow R \ (x1 * y1) \ (x2 * y2)$

and *fin*: *finite* S

and *R-h-g*: $\forall x \in S. R \ (h \ x) \ (g \ x)$

shows $R \ (F \ h \ S) \ (F \ g \ S)$

<proof>

lemma *mono-neutral-cong-left*:

assumes *finite* T

and $S \subseteq T$

and $\forall i \in T - S. h \ i = 1$

and $\bigwedge x. x \in S \implies g \ x = h \ x$

shows $F \ g \ S = F \ h \ T$

<proof>

lemma *mono-neutral-cong-right*:

finite $T \implies S \subseteq T \implies \forall i \in T - S. g \ i = 1 \implies (\bigwedge x. x \in S \implies g \ x = h \ x)$

\implies
 $F\ g\ T = F\ h\ S$
 $\langle proof \rangle$

lemma *mono-neutral-left*: $finite\ T \implies S \subseteq T \implies \forall i \in T - S. g\ i = \mathbf{1} \implies F\ g\ S = F\ g\ T$
 $\langle proof \rangle$

lemma *mono-neutral-right*: $finite\ T \implies S \subseteq T \implies \forall i \in T - S. g\ i = \mathbf{1} \implies F\ g\ T = F\ g\ S$
 $\langle proof \rangle$

lemma *mono-neutral-cong*:
assumes $[simp]$: $finite\ T\ finite\ S$
and $*$: $\bigwedge i. i \in T - S \implies h\ i = \mathbf{1} \bigwedge i. i \in S - T \implies g\ i = \mathbf{1}$
and gh : $\bigwedge x. x \in S \cap T \implies g\ x = h\ x$
shows $F\ g\ S = F\ h\ T$
 $\langle proof \rangle$

lemma *reindex-bij-betw*: $bij\ betw\ h\ S\ T \implies F\ (\lambda x. g\ (h\ x))\ S = F\ g\ T$
 $\langle proof \rangle$

lemma *reindex-bij-witness*:
assumes *witness*:
 $\bigwedge a. a \in S \implies i\ (j\ a) = a$
 $\bigwedge a. a \in S \implies j\ a \in T$
 $\bigwedge b. b \in T \implies j\ (i\ b) = b$
 $\bigwedge b. b \in T \implies i\ b \in S$
assumes *eq*:
 $\bigwedge a. a \in S \implies h\ (j\ a) = g\ a$
shows $F\ g\ S = F\ h\ T$
 $\langle proof \rangle$

lemma *reindex-bij-betw-not-neutral*:
assumes *fin*: $finite\ S'\ finite\ T'$
assumes *bij*: $bij\ betw\ h\ (S - S')\ (T - T')$
assumes *nn*:
 $\bigwedge a. a \in S' \implies g\ (h\ a) = z$
 $\bigwedge b. b \in T' \implies g\ b = z$
shows $F\ (\lambda x. g\ (h\ x))\ S = F\ g\ T$
 $\langle proof \rangle$

lemma *reindex-nontrivial*:
assumes *finite* A
and *nz*: $\bigwedge x\ y. x \in A \implies y \in A \implies x \neq y \implies h\ x = h\ y \implies g\ (h\ x) = \mathbf{1}$
shows $F\ g\ (h\ ` A) = F\ (g \circ h)\ A$
 $\langle proof \rangle$

lemma *reindex-bij-witness-not-neutral*:

assumes *fin*: *finite* S' *finite* T'
assumes *witness*:
 $\bigwedge a. a \in S - S' \implies i (j a) = a$
 $\bigwedge a. a \in S - S' \implies j a \in T - T'$
 $\bigwedge b. b \in T - T' \implies j (i b) = b$
 $\bigwedge b. b \in T - T' \implies i b \in S - S'$
assumes *nn*:
 $\bigwedge a. a \in S' \implies g a = z$
 $\bigwedge b. b \in T' \implies h b = z$
assumes *eq*:
 $\bigwedge a. a \in S \implies h (j a) = g a$
shows $F g S = F h T$
 $\langle proof \rangle$

lemma *delta-remove*:
assumes *fS*: *finite* S
shows $F (\lambda k. \text{if } k = a \text{ then } b k \text{ else } c k) S = (\text{if } a \in S \text{ then } b a * F c (S - \{a\})$
 $\text{else } F c (S - \{a\}))$
 $\langle proof \rangle$

lemma *delta [simp]*:
assumes *fS*: *finite* S
shows $F (\lambda k. \text{if } k = a \text{ then } b k \text{ else } 1) S = (\text{if } a \in S \text{ then } b a \text{ else } 1)$
 $\langle proof \rangle$

lemma *delta' [simp]*:
assumes *fin*: *finite* S
shows $F (\lambda k. \text{if } a = k \text{ then } b k \text{ else } 1) S = (\text{if } a \in S \text{ then } b a \text{ else } 1)$
 $\langle proof \rangle$

lemma *If-cases*:
fixes $P :: 'b \Rightarrow \text{bool}$ **and** $g h :: 'b \Rightarrow 'a$
assumes *fin*: *finite* A
shows $F (\lambda x. \text{if } P x \text{ then } h x \text{ else } g x) A = F h (A \cap \{x. P x\}) * F g (A \cap -$
 $\{x. P x\})$
 $\langle proof \rangle$

lemma *cartesian-product*: $F (\lambda x. F (g x) B) A = F (\text{case-prod } g) (A \times B)$
 $\langle proof \rangle$

lemma *cartesian-product'*:
 $F g (A \times B) = F (\lambda x. F (\lambda y. g (x,y)) B) A$
 $\langle proof \rangle$

lemma *inter-restrict*:
assumes *finite* A
shows $F g (A \cap B) = F (\lambda x. \text{if } x \in B \text{ then } g x \text{ else } 1) A$
 $\langle proof \rangle$

lemma *inter-filter*:

finite $A \implies F\ g\ \{x \in A.\ P\ x\} = F\ (\lambda x.\ \text{if } P\ x\ \text{then } g\ x\ \text{else } \mathbf{1})\ A$
 ⟨proof⟩

lemma *Union-comp*:

assumes $\forall A \in B.\ \text{finite } A$
and $\bigwedge A1\ A2\ x.\ A1 \in B \implies A2 \in B \implies A1 \neq A2 \implies x \in A1 \implies x \in A2$
 $\implies g\ x = \mathbf{1}$
shows $F\ g\ (\bigcup B) = (F \circ F)\ g\ B$
 ⟨proof⟩

lemma *swap*: $F\ (\lambda i.\ F\ (g\ i)\ B)\ A = F\ (\lambda j.\ F\ (\lambda i.\ g\ i\ j)\ A)\ B$
 ⟨proof⟩

lemma *swap-restrict*:

finite $A \implies \text{finite } B \implies$
 $F\ (\lambda x.\ F\ (g\ x)\ \{y.\ y \in B \wedge R\ x\ y\})\ A = F\ (\lambda y.\ F\ (\lambda x.\ g\ x\ y)\ \{x.\ x \in A \wedge R\ x\ y\})\ B$
 ⟨proof⟩

lemma *image-gen*:

assumes *fin*: *finite* S
shows $F\ h\ S = F\ (\lambda y.\ F\ h\ \{x.\ x \in S \wedge g\ x = y\})\ (g\ ` S)$
 ⟨proof⟩

lemma *group*:

assumes *fS*: *finite* S **and** *fT*: *finite* T **and** *fST*: $g\ ` S \subseteq T$
shows $F\ (\lambda y.\ F\ h\ \{x.\ x \in S \wedge g\ x = y\})\ T = F\ h\ S$
 ⟨proof⟩

lemma *Plus*:

fixes $A :: 'b\ \text{set}$ **and** $B :: 'c\ \text{set}$
assumes *fin*: *finite* A *finite* B
shows $F\ g\ (A\ <+>\ B) = F\ (g \circ \text{Inl})\ A * F\ (g \circ \text{Inr})\ B$
 ⟨proof⟩

lemma *same-carrier*:

assumes *finite* C
assumes *subset*: $A \subseteq C\ B \subseteq C$
assumes *trivial*: $\bigwedge a.\ a \in C - A \implies g\ a = \mathbf{1} \wedge \bigwedge b.\ b \in C - B \implies h\ b = \mathbf{1}$
shows $F\ g\ A = F\ h\ B \longleftrightarrow F\ g\ C = F\ h\ C$
 ⟨proof⟩

lemma *same-carrierI*:

assumes *finite* C
assumes *subset*: $A \subseteq C\ B \subseteq C$
assumes *trivial*: $\bigwedge a.\ a \in C - A \implies g\ a = \mathbf{1} \wedge \bigwedge b.\ b \in C - B \implies h\ b = \mathbf{1}$
assumes $F\ g\ C = F\ h\ C$

shows $F\ g\ A = F\ h\ B$
 $\langle proof \rangle$

lemma *eq-general*:

assumes $B: \bigwedge y. y \in B \implies \exists! x. x \in A \wedge h\ x = y$ **and** $A: \bigwedge x. x \in A \implies h\ x \in B \wedge \gamma(h\ x) = \varphi\ x$
shows $F\ \varphi\ A = F\ \gamma\ B$
 $\langle proof \rangle$

lemma *eq-general-inverses*:

assumes $B: \bigwedge y. y \in B \implies k\ y \in A \wedge h(k\ y) = y$ **and** $A: \bigwedge x. x \in A \implies h\ x \in B \wedge k(h\ x) = x \wedge \gamma(h\ x) = \varphi\ x$
shows $F\ \varphi\ A = F\ \gamma\ B$
 $\langle proof \rangle$

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 $G :: ['b \Rightarrow 'a, 'b\ set] \Rightarrow 'a$

where $G\ p\ I \equiv$ if finite $\{x \in I. p\ x \neq 1\}$ then $F\ p\ \{x \in I. p\ x \neq 1\}$ else 1

lemma *finite-Collect-op*:

shows $\llbracket \text{finite } \{i \in I. x\ i \neq 1\}; \text{finite } \{i \in I. y\ i \neq 1\} \rrbracket \implies \text{finite } \{i \in I. x\ i * y\ i \neq 1\}$
 $\langle proof \rangle$

lemma *empty'* [simp]: $G\ p\ \{\} = 1$
 $\langle proof \rangle$

lemma *eq-sum* [simp]: $\text{finite } I \implies G\ p\ I = F\ p\ I$
 $\langle proof \rangle$

lemma *insert'* [simp]:

assumes $\text{finite } \{x \in I. p\ x \neq 1\}$
shows $G\ p\ (\text{insert } i\ I) = (\text{if } i \in I \text{ then } G\ p\ I \text{ else } p\ i * G\ p\ I)$
 $\langle proof \rangle$

lemma *distrib-triv'*:

assumes $\text{finite } I$
shows $G\ (\lambda i. g\ i * h\ i)\ I = G\ g\ I * G\ h\ I$
 $\langle proof \rangle$

lemma *non-neutral'*: $G\ g\ \{x \in I. g\ x \neq 1\} = G\ g\ I$
 $\langle proof \rangle$

lemma *distrib'*:

assumes $\text{finite } \{x \in I. g\ x \neq 1\}$ $\text{finite } \{x \in I. h\ x \neq 1\}$
shows $G\ (\lambda i. g\ i * h\ i)\ I = G\ g\ I * G\ h\ I$

$\langle proof \rangle$

lemma *cong'*:

assumes $A = B$

assumes $g\text{-}h: \bigwedge x. x \in B \implies g\ x = h\ x$

shows $G\ g\ A = G\ h\ B$

$\langle proof \rangle$

lemma *mono-neutral-cong-left'*:

assumes $S \subseteq T$

and $\bigwedge i. i \in T - S \implies h\ i = \mathbf{1}$

and $\bigwedge x. x \in S \implies g\ x = h\ x$

shows $G\ g\ S = G\ h\ T$

$\langle proof \rangle$

lemma *mono-neutral-cong-right'*:

$S \subseteq T \implies \forall i \in T - S. g\ i = \mathbf{1} \implies (\bigwedge x. x \in S \implies g\ x = h\ x) \implies$
 $G\ g\ T = G\ h\ S$

$\langle proof \rangle$

lemma *mono-neutral-left'*: $S \subseteq T \implies \forall i \in T - S. g\ i = \mathbf{1} \implies G\ g\ S = G\ g\ T$

$\langle proof \rangle$

lemma *mono-neutral-right'*: $S \subseteq T \implies \forall i \in T - S. g\ i = \mathbf{1} \implies G\ g\ T = G\ g\ S$

$\langle proof \rangle$

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** $sum' = sum.G$ $\langle proof \rangle$

abbreviation *Sum* ($\langle \sum \rangle$)

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*)

$-sum :: pptrn \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow 'b::comm-monoid-add\ (\langle (\langle indent=3\ notation=\langle binder\ SUM \rangle \rangle SUM\ (-/:-). / -) \rangle [0, 51, 10] 10)$

syntax

$-sum :: pptrn \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow 'b::comm-monoid-add\ (\langle (\langle indent=2\ notation=\langle binder\ \sum \rangle \rangle \sum\ (-/\in-). / -) \rangle [0, 51, 10] 10)$

syntax-consts

$$-sum \Rightarrow sum$$

translations — Beware of argument permutation!

$$\sum_{i \in A}. b \Rightarrow CONST\ sum\ (\lambda i. b)\ A$$

Instead of $\sum x \in \{x. P\}. e$ we introduce the shorter $\sum x | P. e$.

syntax (ASCII)

$$-qsum :: pttm \Rightarrow bool \Rightarrow 'a \Rightarrow 'a\ (\langle (\langle indent=3\ notation=\langle binder\ SUM\ Collect \rangle \rangle SUM - | / - / -) \rangle [0, 0, 10]\ 10)$$
syntax

$$-qsum :: pttm \Rightarrow bool \Rightarrow 'a \Rightarrow 'a\ (\langle (\langle indent=2\ notation=\langle binder\ \sum\ Collect \rangle \rangle \sum - | (-) / -) \rangle [0, 0, 10]\ 10)$$
syntax-consts

$$-qsum == sum$$
translations

$$\sum x | P. t \Rightarrow CONST\ sum\ (\lambda x. t)\ \{x. P\}$$

(ML)

48.2.1 Properties in more restricted classes of structures**lemma sum-Un:**

$$finite\ A \Longrightarrow finite\ B \Longrightarrow sum\ f\ (A \cup B) = sum\ f\ A + sum\ f\ B - sum\ f\ (A \cap B)$$

for $f :: 'b \Rightarrow 'a::ab-group-add$

<proof>

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>

lemma sum-diff:

fixes $f :: 'b \Rightarrow 'a::ab-group-add$

assumes $finite\ A\ B \subseteq A$

shows $sum\ f\ (A - B) = sum\ f\ A - sum\ f\ B$

<proof>

lemma sum-diff1:

fixes $f :: 'b \Rightarrow 'a::ab-group-add$

assumes $finite\ A$

shows $sum\ f\ (A - \{a\}) = (if\ a \in A\ then\ sum\ f\ A - f\ a\ else\ sum\ f\ A)$

<proof>

lemma sum-diff1'-aux:

fixes $f :: 'a \Rightarrow 'b::ab-group-add$

assumes $finite\ F\ \{i \in I. f\ i \neq 0\} \subseteq F$

shows $\text{sum}' f (I - \{i\}) = (\text{if } i \in I \text{ then } \text{sum}' f I - f i \text{ else } \text{sum}' f I)$
 $\langle \text{proof} \rangle$

lemma *sum-diff1'*:

fixes $f :: 'a \Rightarrow 'b :: \text{ab-group-add}$

assumes $\text{finite } \{i \in I. f i \neq 0\}$

shows $\text{sum}' f (I - \{i\}) = (\text{if } i \in I \text{ then } \text{sum}' f I - f i \text{ else } \text{sum}' f I)$

$\langle \text{proof} \rangle$

lemma (**in** *ordered-comm-monoid-add*) *sum-mono*:

$(\bigwedge i. i \in K \implies f i \leq g i) \implies (\sum i \in K. f i) \leq (\sum i \in K. g i)$

$\langle \text{proof} \rangle$

lemma (**in** *ordered-cancel-comm-monoid-add*) *sum-strict-mono-strong*:

assumes $\text{finite } A \ a \in A \ f a < g a$

and $\bigwedge x. x \in A \implies f x \leq g x$

shows $\text{sum } f A < \text{sum } g A$

$\langle \text{proof} \rangle$

lemma (**in** *strict-ordered-comm-monoid-add*) *sum-strict-mono*:

assumes $\text{finite } A \ A \neq \{\}$

and $\bigwedge x. x \in A \implies f x < g x$

shows $\text{sum } f A < \text{sum } g A$

$\langle \text{proof} \rangle$

lemma *sum-strict-mono-ex1*:

fixes $f g :: 'i \Rightarrow 'a :: \text{ordered-cancel-comm-monoid-add}$

assumes $\text{finite } A$

and $\forall x \in A. f x \leq g x$

and $\exists a \in A. f a < g a$

shows $\text{sum } f A < \text{sum } g A$

$\langle \text{proof} \rangle$

lemma *sum-mono-inv*:

fixes $f g :: 'i \Rightarrow 'a :: \text{ordered-cancel-comm-monoid-add}$

assumes $\text{eq: } \text{sum } f I = \text{sum } g I$

assumes $\text{le: } \bigwedge i. i \in I \implies f i \leq g i$

assumes $i: i \in I$

assumes $I: \text{finite } I$

shows $f i = g i$

$\langle \text{proof} \rangle$

lemma *member-le-sum*:

fixes $f :: - \Rightarrow 'b :: \{\text{semiring-1}, \text{ordered-comm-monoid-add}\}$

assumes $i \in A$

and $\text{le: } \bigwedge x. x \in A - \{i\} \implies 0 \leq f x$

and $\text{finite } A$

shows $f i \leq \text{sum } f A$

$\langle \text{proof} \rangle$

lemma *sum-negf*: $(\sum x \in A. - f x) = - (\sum x \in A. f x)$
for $f :: 'b \Rightarrow 'a :: ab\text{-group-add}$
 $\langle proof \rangle$

lemma *sum-subtractf*: $(\sum x \in A. f x - g x) = (\sum x \in A. f x) - (\sum x \in A. g x)$
for $f g :: 'b \Rightarrow 'a :: ab\text{-group-add}$
 $\langle proof \rangle$

lemma *sum-subtractf-nat*:
 $(\bigwedge x. x \in A \implies g x \leq f x) \implies (\sum x \in A. f x - g x) = (\sum x \in A. f x) - (\sum x \in A. g x)$
for $f g :: 'a \Rightarrow nat$
 $\langle proof \rangle$

context *ordered-comm-monoid-add*
begin

lemma *sum-nonneg*: $(\bigwedge x. x \in A \implies 0 \leq f x) \implies 0 \leq \text{sum } f A$
 $\langle proof \rangle$

lemma *sum-nonpos*: $(\bigwedge x. x \in A \implies f x \leq 0) \implies \text{sum } f A \leq 0$
 $\langle proof \rangle$

lemma *sum-nonneg-eq-0-iff*:
 $\text{finite } A \implies (\bigwedge x. x \in A \implies 0 \leq f x) \implies \text{sum } f A = 0 \longleftrightarrow (\forall x \in A. f x = 0)$
 $\langle proof \rangle$

lemma *sum-nonneg-0*:
 $\text{finite } s \implies (\bigwedge i. i \in s \implies f i \geq 0) \implies (\sum i \in s. f i) = 0 \implies i \in s \implies f i = 0$
 $\langle proof \rangle$

lemma *sum-nonneg-leq-bound*:
assumes $\text{finite } s \bigwedge i. i \in s \implies f i \geq 0$ $(\sum i \in s. f i) = B$ $i \in s$
shows $f i \leq B$
 $\langle proof \rangle$

lemma *sum-mono2*:
assumes $\text{fin}: \text{finite } B$
and $\text{sub}: A \subseteq B$
and $\text{nn}: \bigwedge b. b \in B - A \implies 0 \leq f b$
shows $\text{sum } f A \leq \text{sum } f B$
 $\langle proof \rangle$

lemma *sum-le-included*:
assumes $\text{finite } s \text{ finite } t$
and $\forall y \in t. 0 \leq g y$ $(\forall x \in s. \exists y \in t. i y = x \wedge f x \leq g y)$
shows $\text{sum } f s \leq \text{sum } g t$
 $\langle proof \rangle$

end

lemma (in *canonically-ordered-monoid-add*) *sum-eq-0-iff* [simp]:
 $\text{finite } F \implies (\text{sum } f \ F = 0) = (\forall a \in F. f \ a = 0)$
 ⟨proof⟩

context *semiring-0*
begin

lemma *sum-distrib-left*: $r * \text{sum } f \ A = (\sum n \in A. r * f \ n)$
 ⟨proof⟩

lemma *sum-distrib-right*: $\text{sum } f \ A * r = (\sum n \in A. f \ n * r)$
 ⟨proof⟩

end

lemma *sum-divide-distrib*: $\text{sum } f \ A / r = (\sum n \in A. f \ n / r)$
for $r :: 'a::\text{field}$
 ⟨proof⟩

lemma *sum-abs[iff]*: $|\text{sum } f \ A| \leq \text{sum } (\lambda i. |f \ i|) \ A$
for $f :: 'a \Rightarrow 'b::\text{ordered-ab-group-add-abs}$
 ⟨proof⟩

lemma *sum-abs-ge-zero[iff]*: $0 \leq \text{sum } (\lambda i. |f \ i|) \ A$
for $f :: 'a \Rightarrow 'b::\text{ordered-ab-group-add-abs}$
 ⟨proof⟩

lemma *abs-sum-abs[simp]*: $|\sum a \in A. f \ a| = (\sum a \in A. |f \ a|)$
for $f :: 'a \Rightarrow 'b::\text{ordered-ab-group-add-abs}$
 ⟨proof⟩

lemma *sum-product*:
fixes $f :: 'a \Rightarrow 'b::\text{semiring-0}$
shows $\text{sum } f \ A * \text{sum } g \ B = (\sum i \in A. \sum j \in B. f \ i * g \ j)$
 ⟨proof⟩

lemma *sum-mult-sum-if-inj*:
fixes $f :: 'a \Rightarrow 'b::\text{semiring-0}$
shows *inj-on* $(\lambda(a, b). f \ a * g \ b) \ (A \times B) \implies$
 $\text{sum } f \ A * \text{sum } g \ B = \text{sum } \text{id} \ \{f \ a * g \ b \mid a \in A \wedge b \in B\}$
 ⟨proof⟩

lemma *sum-SucD*: $\text{sum } f \ A = \text{Suc } n \implies \exists a \in A. 0 < f \ a$
 ⟨proof⟩

lemma *sum-eq-Suc0-iff*:

$finite\ A \implies sum\ f\ A = Suc\ 0 \iff (\exists a \in A. f\ a = Suc\ 0 \wedge (\forall b \in A. a \neq b \implies f\ b = 0))$
 <proof>

lemmas $sum\text{-}eq\text{-}1\text{-}iff = sum\text{-}eq\text{-}Suc0\text{-}iff[simplified\ One\text{-}nat\text{-}def[symmetric]]$

lemma $sum\text{-}Un\text{-}nat$:

$finite\ A \implies finite\ B \implies sum\ f\ (A \cup B) = sum\ f\ A + sum\ f\ B - sum\ f\ (A \cap B)$

for $f :: 'a \Rightarrow nat$

— For the natural numbers, we have subtraction.

<proof>

lemma $sum\text{-}diff1\text{-}nat$: $sum\ f\ (A - \{a\}) = (if\ a \in A\ then\ sum\ f\ A - f\ a\ else\ sum\ f\ A)$

for $f :: 'a \Rightarrow nat$

<proof>

lemma $sum\text{-}diff\text{-}nat$:

fixes $f :: 'a \Rightarrow nat$

assumes $finite\ B$ **and** $B \subseteq A$

shows $sum\ f\ (A - B) = sum\ f\ A - sum\ f\ B$

<proof>

lemma $sum\text{-}comp\text{-}morphism$:

$h\ 0 = 0 \implies (\bigwedge x\ y. h\ (x + y) = h\ x + h\ y) \implies sum\ (h \circ g)\ A = h\ (sum\ g\ A)$

<proof>

lemma (**in** $comm\text{-}semiring\text{-}1$) $dvd\text{-}sum$: $(\bigwedge a. a \in A \implies d\ dvd\ f\ a) \implies d\ dvd\ sum\ f\ A$

<proof>

lemma (**in** $ordered\text{-}comm\text{-}monoid\text{-}add$) $sum\text{-}pos$:

$finite\ I \implies I \neq \{\} \implies (\bigwedge i. i \in I \implies 0 < f\ i) \implies 0 < sum\ f\ I$

<proof>

lemma (**in** $ordered\text{-}comm\text{-}monoid\text{-}add$) $sum\text{-}pos2$:

assumes I : $finite\ I\ i \in I\ 0 < f\ i \bigwedge i. i \in I \implies 0 \leq f\ i$

shows $0 < sum\ f\ I$

<proof>

lemma $sum\text{-}strict\text{-}mono2$:

fixes $f :: 'a \Rightarrow 'b::ordered\text{-}cancel\text{-}comm\text{-}monoid\text{-}add$

assumes $finite\ B\ A \subseteq B\ b \in B - A\ f\ b > 0$ **and** $\bigwedge x. x \in B \implies f\ x \geq 0$

shows $sum\ f\ A < sum\ f\ B$

<proof>

lemma $sum\text{-}cong\text{-}Suc$:

assumes $0 \notin A \bigwedge x. Suc\ x \in A \implies f\ (Suc\ x) = g\ (Suc\ x)$

shows $sum\ f\ A = sum\ g\ A$

⟨proof⟩

48.2.2 Cardinality as special case of *sum*

lemma *card-eq-sum*: $\text{card } A = \text{sum } (\lambda x. 1) A$

⟨proof⟩

context *semiring-1*

begin

lemma *sum-constant* [*simp*]:

$(\sum x \in A. y) = \text{of-nat } (\text{card } A) * y$

⟨proof⟩

context

fixes *A*

assumes ⟨*finite A*⟩

begin

lemma *sum-of-bool-eq* [*simp*]:

$\langle (\sum x \in A. \text{of-bool } (P x)) = \text{of-nat } (\text{card } (A \cap \{x. P x\})) \rangle$ **if** ⟨*finite A*⟩

⟨proof⟩

lemma *sum-mult-of-bool-eq* [*simp*]:

$\langle (\sum x \in A. f x * \text{of-bool } (P x)) = (\sum x \in (A \cap \{x. P x\}). f x) \rangle$

⟨proof⟩

lemma *sum-of-bool-mult-eq* [*simp*]:

$\langle (\sum x \in A. \text{of-bool } (P x) * f x) = (\sum x \in (A \cap \{x. P x\}). f x) \rangle$

⟨proof⟩

end

end

lemma *sum-Suc*: $\text{sum } (\lambda x. \text{Suc}(f x)) A = \text{sum } f A + \text{card } A$

⟨proof⟩

lemma *sum-bounded-above*:

fixes *K* :: 'a::{*semiring-1*,*ordered-comm-monoid-add*}

assumes *le*: $\bigwedge i. i \in A \implies f i \leq K$

shows $\text{sum } f A \leq \text{of-nat } (\text{card } A) * K$

⟨proof⟩

lemma *sum-bounded-above-divide*:

fixes *K* :: 'a::*linordered-field*

assumes *le*: $\bigwedge i. i \in A \implies f i \leq K$ / *of-nat* (*card A*) **and** *fin*: *finite A* *A* ≠ {}

shows $\text{sum } f A \leq K$

⟨proof⟩

lemma *sum-bounded-above-strict*:

fixes $K :: 'a::\{\text{ordered-cancel-comm-monoid-add}, \text{semiring-1}\}$
assumes $\bigwedge i. i \in A \implies f\ i < K \text{ card } A > 0$
shows $\text{sum } f\ A < \text{of-nat } (\text{card } A) * K$
 $\langle \text{proof} \rangle$

lemma *sum-bounded-below*:

fixes $K :: 'a::\{\text{semiring-1}, \text{ordered-comm-monoid-add}\}$
assumes $\text{le}: \bigwedge i. i \in A \implies K \leq f\ i$
shows $\text{of-nat } (\text{card } A) * K \leq \text{sum } f\ A$
 $\langle \text{proof} \rangle$

lemma *convex-sum-bound-le*:

fixes $x :: 'a \Rightarrow 'b::\text{linordered-idom}$
assumes $0: \bigwedge i. i \in I \implies 0 \leq x\ i$ **and** $1: \text{sum } x\ I = 1$
and $\delta: \bigwedge i. i \in I \implies |a\ i - b| \leq \delta$
shows $|(\sum i \in I. a\ i * x\ i) - b| \leq \delta$
 $\langle \text{proof} \rangle$

lemma *card-UN-disjoint*:

assumes *finite I* **and** $\forall i \in I. \text{finite } (A\ i)$
and $\forall i \in I. \forall j \in I. i \neq j \longrightarrow A\ i \cap A\ j = \{\}$
shows $\text{card } (\bigcup (A\ ` I)) = (\sum i \in I. \text{card}(A\ i))$
 $\langle \text{proof} \rangle$

lemma *card-Union-disjoint*:

assumes *pairwise disjnt C* **and** $\text{fin}: \bigwedge A. A \in C \implies \text{finite } A$
shows $\text{card } (\bigcup C) = \text{sum card } C$
 $\langle \text{proof} \rangle$

lemma *card-Union-le-sum-card-weak*:

fixes $U :: 'a \text{ set set}$
assumes $\forall u \in U. \text{finite } u$
shows $\text{card } (\bigcup U) \leq \text{sum card } U$
 $\langle \text{proof} \rangle$

lemma *card-Union-le-sum-card*:

fixes $U :: 'a \text{ set set}$
shows $\text{card } (\bigcup U) \leq \text{sum card } U$
 $\langle \text{proof} \rangle$

lemma *card-UN-le*:

assumes *finite I*
shows $\text{card}(\bigcup i \in I. A\ i) \leq (\sum i \in I. \text{card}(A\ i))$
 $\langle \text{proof} \rangle$

lemma *card-quotient-disjoint*:

assumes *finite A inj-on* $(\lambda x. \{x\} // r)\ A$

shows $\text{card } (A/r) = \text{card } A$
 $\langle \text{proof} \rangle$

lemma *sum-multicount-gen*:
assumes $\text{finite } s \text{ finite } t \ \forall j \in t. (\text{card } \{i \in s. R \ i \ j\} = k \ j)$
shows $\text{sum } (\lambda i. (\text{card } \{j \in t. R \ i \ j\})) \ s = \text{sum } k \ t$
(is ?l = ?r)
 $\langle \text{proof} \rangle$

lemma *sum-multicount*:
assumes $\text{finite } S \text{ finite } T \ \forall j \in T. (\text{card } \{i \in S. R \ i \ j\} = k)$
shows $\text{sum } (\lambda i. \text{card } \{j \in T. R \ i \ j\}) \ S = k * \text{card } T$ **(is ?l = ?r)**
 $\langle \text{proof} \rangle$

lemma *sum-card-image*:
assumes $\text{finite } A$
assumes $\text{pairwise } (\lambda s \ t. \text{disjnt } (f \ s) \ (f \ t)) \ A$
shows $\text{sum } \text{card } (f \ ` \ A) = \text{sum } (\lambda a. \text{card } (f \ a)) \ A$
 $\langle \text{proof} \rangle$

By Jakub Kdzioka:

lemma *sum-fun-comp*:
assumes $\text{finite } S \text{ finite } R \ g \ ` \ S \subseteq R$
shows $(\sum x \in S. f \ (g \ x)) = (\sum y \in R. \text{of-nat } (\text{card } \{x \in S. g \ x = y\}) * f \ y)$
 $\langle \text{proof} \rangle$

48.2.3 Cardinality of products

lemma *card-SigmaI [simp]*:
 $\text{finite } A \implies \forall a \in A. \text{finite } (B \ a) \implies \text{card } (\text{SIGMA } x: A. B \ x) = (\sum a \in A. \text{card } (B \ a))$
 $\langle \text{proof} \rangle$

lemma *card-cartesian-product*: $\text{card } (A \times B) = \text{card } A * \text{card } B$
 $\langle \text{proof} \rangle$

lemma *card-cartesian-product-singleton*: $\text{card } (\{x\} \times A) = \text{card } A$
 $\langle \text{proof} \rangle$

48.3 Generalized product over a set

context *comm-monoid-mult*
begin

sublocale *prod: comm-monoid-set times 1*
defines $\text{prod} = \text{prod}.F$ **and** $\text{prod}' = \text{prod}.G$ $\langle \text{proof} \rangle$

abbreviation *Prod* ($\langle \prod \rangle$)
 where $\prod \equiv \text{prod } (\lambda x. x)$

end

syntax (*ASCII*)
 $\text{-prod} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b :: \text{comm-monoid-mult} \ (\langle (\langle \text{indent}=4 \text{ notation}=\langle \text{binder PROD} \rangle \rangle \text{PROD } (-/:-)/ -) \rangle [0, 51, 10] 10)$

syntax
 $\text{-prod} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b :: \text{comm-monoid-mult} \ (\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{binder } \prod \rangle \rangle \prod (-/\in-)/ -) \rangle [0, 51, 10] 10)$

syntax-consts
 $\text{-prod} == \text{prod}$

translations — Beware of argument permutation!
 $\prod_{i \in A}. b == \text{CONST prod } (\lambda i. b) A$

Instead of $\prod x \in \{x. P\}. e$ we introduce the shorter $\prod x | P. e$.

syntax (*ASCII*)
 $\text{-qprod} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a \ (\langle (\langle \text{indent}=4 \text{ notation}=\langle \text{binder PROD Collect} \rangle \rangle \text{PROD } - | / - / -) \rangle [0, 0, 10] 10)$

syntax
 $\text{-qprod} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a \ (\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{binder } \prod \text{ Collect} \rangle \rangle \prod - | (-)/ -) \rangle [0, 0, 10] 10)$

syntax-consts
 $\text{-qprod} == \text{prod}$

translations
 $\prod x | P. t \Rightarrow \text{CONST prod } (\lambda x. t) \{x. P\}$
 $\langle \text{ML} \rangle$

context *comm-monoid-mult*
begin

lemma *prod-dvd-prod*: $(\bigwedge a. a \in A \implies f a \text{ dvd } g a) \implies \text{prod } f A \text{ dvd } \text{prod } g A$
 $\langle \text{proof} \rangle$

lemma *prod-dvd-prod-subset*: $\text{finite } B \implies A \subseteq B \implies \text{prod } f A \text{ dvd } \text{prod } f B$
 $\langle \text{proof} \rangle$

end

48.3.1 Properties in more restricted classes of structures

context *linordered-nonzero-semiring*
begin

lemma *prod-ge-1*: $(\bigwedge x. x \in A \implies 1 \leq f x) \implies 1 \leq \text{prod } f A$

$\langle proof \rangle$

lemma *prod-le-1*:

fixes $f :: 'b \Rightarrow 'a$

assumes $\bigwedge x. x \in A \implies 0 \leq f\ x \wedge f\ x \leq 1$

shows $prod\ f\ A \leq 1$

$\langle proof \rangle$

end

context *comm-semiring-1*

begin

lemma *dvd-prod-eqI* [*intro*]:

assumes *finite* A **and** $a \in A$ **and** $b = f\ a$

shows $b\ dvd\ prod\ f\ A$

$\langle proof \rangle$

lemma *dvd-prodI* [*intro*]: $finite\ A \implies a \in A \implies f\ a\ dvd\ prod\ f\ A$

$\langle proof \rangle$

lemma *prod-zero*:

assumes *finite* A **and** $\exists a \in A. f\ a = 0$

shows $prod\ f\ A = 0$

$\langle proof \rangle$

lemma *prod-dvd-prod-subset2*:

assumes *finite* B **and** $A \subseteq B$ **and** $\bigwedge a. a \in A \implies f\ a\ dvd\ g\ a$

shows $prod\ f\ A\ dvd\ prod\ g\ B$

$\langle proof \rangle$

end

lemma (*in semidom*) *prod-zero-iff* [*simp*]:

fixes $f :: 'b \Rightarrow 'a$

assumes *finite* A

shows $prod\ f\ A = 0 \iff (\exists a \in A. f\ a = 0)$

$\langle proof \rangle$

lemma (*in semidom-divide*) *prod-diff1*:

assumes *finite* A **and** $f\ a \neq 0$

shows $prod\ f\ (A - \{a\}) = (if\ a \in A\ then\ prod\ f\ A\ div\ f\ a\ else\ prod\ f\ A)$

$\langle proof \rangle$

lemma *prod-uminus*: $(\prod x \in A. -f\ x :: 'a :: comm-ring-1) = (-1) \wedge card\ A * (\prod x \in A. f\ x)$

$\langle proof \rangle$

lemma *prod-diff*:

fixes $f :: 'a \Rightarrow 'b :: \text{field}$
assumes $\text{finite } A \ B \subseteq A \ \bigwedge x. x \in B \implies f\ x \neq 0$
shows $\text{prod } f\ (A - B) = \text{prod } f\ A \ / \ \text{prod } f\ B$
 $\langle \text{proof} \rangle$

lemma *sum-zero-power* [simp]: $(\sum i \in A. c\ i * 0^{\widehat{i}}) = (\text{if } \text{finite } A \wedge 0 \in A \text{ then } c\ 0 \text{ else } 0)$
for $c :: \text{nat} \Rightarrow 'a :: \text{division-ring}$
 $\langle \text{proof} \rangle$

lemma *sum-zero-power'* [simp]:
 $(\sum i \in A. c\ i * 0^{\widehat{i}} \ / \ d\ i) = (\text{if } \text{finite } A \wedge 0 \in A \text{ then } c\ 0 \ / \ d\ 0 \text{ else } 0)$
for $c :: \text{nat} \Rightarrow 'a :: \text{field}$
 $\langle \text{proof} \rangle$

lemma (in *field*) *prod-inversef*: $\text{prod } (\text{inverse} \circ f)\ A = \text{inverse } (\text{prod } f\ A)$
 $\langle \text{proof} \rangle$

lemma (in *field*) *prod-dividef*: $(\prod x \in A. f\ x \ / \ g\ x) = \text{prod } f\ A \ / \ \text{prod } g\ A$
 $\langle \text{proof} \rangle$

lemma *prod-Un*:
fixes $f :: 'b \Rightarrow 'a :: \text{field}$
assumes $\text{finite } A$ **and** $\text{finite } B$
and $\forall x \in A \cap B. f\ x \neq 0$
shows $\text{prod } f\ (A \cup B) = \text{prod } f\ A * \text{prod } f\ B \ / \ \text{prod } f\ (A \cap B)$
 $\langle \text{proof} \rangle$

context *linordered-semidom*
begin

lemma *prod-nonneg*: $(\bigwedge a. a \in A \implies 0 \leq f\ a) \implies 0 \leq \text{prod } f\ A$
 $\langle \text{proof} \rangle$

lemma *prod-pos*: $(\bigwedge a. a \in A \implies 0 < f\ a) \implies 0 < \text{prod } f\ A$
 $\langle \text{proof} \rangle$

lemma *prod-mono*:
 $(\bigwedge i. i \in A \implies 0 \leq f\ i \wedge f\ i \leq g\ i) \implies \text{prod } f\ A \leq \text{prod } g\ A$
 $\langle \text{proof} \rangle$

Only one needs to be strict

lemma *prod-mono-strict*:
assumes $i \in A \ f\ i < g\ i$
assumes $\text{finite } A$
assumes $\bigwedge i. i \in A \implies 0 \leq f\ i \wedge f\ i \leq g\ i$
assumes $\bigwedge i. i \in A \implies 0 < g\ i$
shows $\text{prod } f\ A < \text{prod } g\ A$
 $\langle \text{proof} \rangle$

lemma *prod-le-power*:

assumes $A: \bigwedge i. i \in A \implies 0 \leq f\ i \wedge f\ i \leq n \text{ card } A \leq k$ **and** $n \geq 1$
shows $\text{prod } f\ A \leq n \wedge^k$
 $\langle \text{proof} \rangle$

end

lemma *prod-mono2*:

fixes $f :: 'a \Rightarrow 'b :: \text{linordered-idom}$
assumes $\text{fin}: \text{finite } B$
and $\text{sub}: A \subseteq B$
and $\text{nn}: \bigwedge b. b \in B - A \implies 1 \leq f\ b$
and $A: \bigwedge a. a \in A \implies 0 \leq f\ a$
shows $\text{prod } f\ A \leq \text{prod } f\ B$
 $\langle \text{proof} \rangle$

lemma *less-1-prod*:

fixes $f :: 'a \Rightarrow 'b :: \text{linordered-idom}$
shows $\text{finite } I \implies I \neq \{\} \implies (\bigwedge i. i \in I \implies 1 < f\ i) \implies 1 < \text{prod } f\ I$
 $\langle \text{proof} \rangle$

lemma *less-1-prod2*:

fixes $f :: 'a \Rightarrow 'b :: \text{linordered-idom}$
assumes $I: \text{finite } I \ i \in I \ 1 < f\ i \ \bigwedge i. i \in I \implies 1 \leq f\ i$
shows $1 < \text{prod } f\ I$
 $\langle \text{proof} \rangle$

lemma (**in** *linordered-field*) *abs-prod*: $|\text{prod } f\ A| = (\prod x \in A. |f\ x|)$
 $\langle \text{proof} \rangle$

lemma *prod-eq-1-iff* [*simp*]: $\text{finite } A \implies \text{prod } f\ A = 1 \longleftrightarrow (\forall a \in A. f\ a = 1)$
for $f :: 'a \Rightarrow \text{nat}$
 $\langle \text{proof} \rangle$

lemma *prod-pos-nat-iff* [*simp*]: $\text{finite } A \implies \text{prod } f\ A > 0 \longleftrightarrow (\forall a \in A. f\ a > 0)$
for $f :: 'a \Rightarrow \text{nat}$
 $\langle \text{proof} \rangle$

lemma *prod-constant* [*simp*]: $(\prod x \in A. y) = y \wedge^{\text{card } A}$
for $y :: 'a :: \text{comm-monoid-mult}$
 $\langle \text{proof} \rangle$

lemma *prod-diff-swap*:

fixes $f :: 'a \Rightarrow 'b :: \text{comm-ring-1}$
shows $\text{prod } (\lambda x. f\ x - g\ x)\ A = (-1) \wedge^{\text{card } A} * \text{prod } (\lambda x. g\ x - f\ x)\ A$
 $\langle \text{proof} \rangle$

lemma *prod-power-distrib*: $\text{prod } f\ A \wedge^n = \text{prod } (\lambda x. (f\ x) \wedge^n)\ A$

for $f :: 'a \Rightarrow 'b :: \text{comm-semiring-1}$
 $\langle \text{proof} \rangle$

lemma *power-inject-exp'*:
assumes $a \neq 1 \ a > (0 :: 'a :: \text{linordered-semidom})$
shows $a \wedge^m = a \wedge^n \longleftrightarrow m = n$
 $\langle \text{proof} \rangle$

lemma *power-sum*: $c \wedge (\sum_{a \in A} f\ a) = (\prod_{a \in A} c \wedge f\ a)$
 $\langle \text{proof} \rangle$

lemma *prod-gen-delta*:
fixes $b :: 'b \Rightarrow 'a :: \text{comm-monoid-mult}$
assumes $\text{fin: finite } S$
shows $\text{prod } (\lambda k. \text{if } k = a \text{ then } b\ k \text{ else } c)\ S =$
 $(\text{if } a \in S \text{ then } b\ a * c \wedge (\text{card } S - 1) \text{ else } c \wedge \text{card } S)$
 $\langle \text{proof} \rangle$

lemma *sum-image-le*:
fixes $g :: 'a \Rightarrow 'b :: \text{ordered-comm-monoid-add}$
assumes $\text{finite } I \ \bigwedge i. i \in I \implies 0 \leq g(f\ i)$
shows $\text{sum } g\ (f\ 'I) \leq \text{sum } (g \circ f)\ I$
 $\langle \text{proof} \rangle$

lemma *prod-add*:
fixes $f1\ f2 :: 'a \Rightarrow 'c :: \text{comm-semiring-1}$
assumes $\text{finite: finite } A$
shows $(\prod_{x \in A} f1\ x + f2\ x) = (\sum_{X \in \text{Pow } A} (\prod_{x \in X} f1\ x) * (\prod_{x \in A-X} f2\ x))$
 $\langle \text{proof} \rangle$

lemma *prod-diff-conv-sum*:
fixes $f1\ f2 :: 'a \Rightarrow 'c :: \text{comm-ring-1}$
assumes $\text{finite: finite } A$
shows $(\prod_{x \in A} f1\ x - f2\ x) = (\sum_{X \in \text{Pow } A} (-1) \wedge \text{card } X * (\prod_{x \in X} f2\ x) * (\prod_{x \in A-X} f1\ x))$
 $\langle \text{proof} \rangle$

lemma *prod-diff-conv-sum'*:
fixes $f1\ f2 :: 'a \Rightarrow 'c :: \text{comm-ring-1}$
assumes $\text{finite: finite } A$
shows $(\prod_{x \in A} f1\ x - f2\ x) = (\sum_{X \in \text{Pow } A} (-1) \wedge (\text{card } A - \text{card } X) * (\prod_{x \in X} f1\ x) * (\prod_{x \in A-X} f2\ x))$
 $\langle \text{proof} \rangle$

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  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a set  $\Rightarrow$  bool
  where chain ord S  $\longleftrightarrow$  ( $\forall x \in S. \forall y \in S. \text{ord } x \ y \vee \text{ord } y \ x$ )
```

```
lemma chainI:
  assumes  $\bigwedge x \ y. x \in S \implies y \in S \implies \text{ord } x \ y \vee \text{ord } y \ x$ 
  shows chain ord S
  <proof>
```

```
lemma chainD:
  assumes chain ord S and  $x \in S$  and  $y \in S$ 
  shows  $\text{ord } x \ y \vee \text{ord } y \ x$ 
  <proof>
```

```
lemma chainE:
  assumes chain ord S and  $x \in S$  and  $y \in S$ 
  obtains  $\text{ord } x \ y \mid \text{ord } y \ x$ 
  <proof>
```

```
lemma chain-empty: chain ord {}
  <proof>
```

```
lemma chain-equality: chain (=) A  $\longleftrightarrow$  ( $\forall x \in A. \forall y \in A. x = y$ )
  <proof>
```

```
lemma chain-subset: chain ord A  $\implies B \subseteq A \implies$  chain ord B
  <proof>
```

```
lemma chain-imageI:
  assumes chain: chain le-a Y
  and mono:  $\bigwedge x \ y. x \in Y \implies y \in Y \implies \text{le-a } x \ y \implies \text{le-b } (f \ x) \ (f \ y)$ 
  shows chain le-b (f ` Y)
  <proof>
```

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 ( $\leq$ ) A  $\implies x \in A \implies x \leq \text{Sup } A$ 
  assumes ccpo-Sup-least: chain ( $\leq$ ) A  $\implies (\bigwedge x. x \in A \implies x \leq z) \implies \text{Sup } A \leq$ 
 $z$ 
begin

```

```

lemma chain-singleton: Complete-Partial-Order.chain ( $\leq$ )  $\{x\}$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma ccpo-Sup-singleton [simp]:  $\bigsqcup \{x\} = x$ 
   $\langle \text{proof} \rangle$ 

```

49.3 Transfinite iteration of a function

```

context notes [inductive-internals]
begin

```

```

inductive-set iterates :: ( $'a \Rightarrow 'a$ )  $\Rightarrow 'a$  set
  for f ::  $'a \Rightarrow 'a$ 
  where
    step:  $x \in \text{iterates } f \implies f x \in \text{iterates } f$ 
    | Sup: chain ( $\leq$ ) M  $\implies \forall x \in M. x \in \text{iterates } f \implies \text{Sup } M \in \text{iterates } f$ 

```

```

end

```

```

lemma iterates-le-f:  $x \in \text{iterates } f \implies \text{monotone } (\leq) (\leq) f \implies x \leq f x$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma chain-iterates:
  assumes f: monotone ( $\leq$ ) ( $\leq$ ) f
  shows chain ( $\leq$ ) (iterates f) (is chain - ?C)
   $\langle \text{proof} \rangle$ 

```

```

lemma bot-in-iterates:  $\text{Sup } \{\} \in \text{iterates } f$ 
   $\langle \text{proof} \rangle$ 

```

49.4 Fixpoint combinator

```

definition fixp :: ( $'a \Rightarrow 'a$ )  $\Rightarrow 'a$ 
  where fixp f = Sup (iterates f)

```

```

lemma iterates-fixp:
  assumes f: monotone ( $\leq$ ) ( $\leq$ ) f
  shows fixp f  $\in \text{iterates } f$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma fixp-unfold:
  assumes f: monotone ( $\leq$ ) ( $\leq$ ) f
  shows fixp f = f (fixp f)
   $\langle \text{proof} \rangle$ 

```


lemma *fixp-lowerbound*:
assumes *f*: *monotone* (\leq) (\leq) *f*
and *z*: $f\ z \leq z$
shows $\text{fixp } f \leq z$
 $\langle \text{proof} \rangle$

end

49.5 Fixpoint induction

$\langle ML \rangle$

definition *admissible* :: $('a \text{ set} \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *admissible* *lub ord P* $\longleftrightarrow (\forall A. \text{chain ord } A \longrightarrow A \neq \{\} \longrightarrow (\forall x \in A. P\ x) \longrightarrow P\ (\text{lub } A))$

lemma *admissibleI*:
assumes $\bigwedge A. \text{chain ord } A \Longrightarrow A \neq \{\} \Longrightarrow \forall x \in A. P\ x \Longrightarrow P\ (\text{lub } A)$
shows *ccpo.admissible lub ord P*
 $\langle \text{proof} \rangle$

lemma *admissibleD*:
assumes *ccpo.admissible lub ord P*
assumes *chain ord A*
assumes $A \neq \{\}$
assumes $\bigwedge x. x \in A \Longrightarrow P\ x$
shows $P\ (\text{lub } A)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma (**in** *ccpo*) *fixp-induct*:
assumes *adm*: *ccpo.admissible Sup* (\leq) *P*
assumes *mono*: *monotone* (\leq) (\leq) *f*
assumes *bot*: $P\ (\text{Sup } \{\})$
assumes *step*: $\bigwedge x. P\ x \Longrightarrow P\ (f\ x)$
shows $P\ (\text{fixp } f)$
 $\langle \text{proof} \rangle$

lemma *admissible-True*: *ccpo.admissible lub ord* $(\lambda x. \text{True})$
 $\langle \text{proof} \rangle$

lemma *admissible-const*: *ccpo.admissible lub ord* $(\lambda x. t)$
 $\langle \text{proof} \rangle$

lemma *admissible-conj*:
assumes *ccpo.admissible lub ord* $(\lambda x. P\ x)$

```

assumes ccpo.admissible lub ord ( $\lambda x. Q\ x$ )
shows ccpo.admissible lub ord ( $\lambda x. P\ x \wedge Q\ x$ )
 $\langle proof \rangle$ 

lemma admissible-all:
assumes  $\bigwedge y. \textit{ccpo.admissible lub ord } (\lambda x. P\ x\ y)$ 
shows ccpo.admissible lub ord ( $\lambda x. \forall y. P\ x\ y$ )
 $\langle proof \rangle$ 

lemma admissible-ball:
assumes  $\bigwedge y. y \in A \implies \textit{ccpo.admissible lub ord } (\lambda x. P\ x\ y)$ 
shows ccpo.admissible lub ord ( $\lambda x. \forall y \in A. P\ x\ y$ )
 $\langle proof \rangle$ 

lemma chain-compr: chain ord A  $\implies$  chain ord  $\{x \in A. P\ x\}$ 
 $\langle proof \rangle$ 

context ccpo
begin

lemma admissible-disj:
fixes  $P\ Q :: 'a \Rightarrow bool$ 
assumes  $P: \textit{ccpo.admissible Sup } (\leq) (\lambda x. P\ x)$ 
assumes  $Q: \textit{ccpo.admissible Sup } (\leq) (\lambda x. Q\ x)$ 
shows ccpo.admissible Sup  $(\leq) (\lambda x. P\ x \vee Q\ x)$ 
 $\langle proof \rangle$ 

end

instance complete-lattice  $\subseteq$  ccpo
 $\langle proof \rangle$ 

lemma lfp-eq-fixp:
assumes mono: mono f
shows lfp f = fixp f
 $\langle proof \rangle$ 

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
 $(y = \text{None} \implies P) \implies (\bigwedge a. y = \text{Some } a \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma [*case-names None Some, induct type: option*]:
 — for backward compatibility – names of variables differ
 $P \text{ None} \implies (\bigwedge \text{option}. P (\text{Some } \text{option})) \implies P \text{ option}$
 $\langle \text{proof} \rangle$

Compatibility:

$\langle \text{ML} \rangle$
lemmas *inducts* = *option.induct*
lemmas *cases* = *option.case*
 $\langle \text{ML} \rangle$

lemma *not-None-eq [iff]*: $x \neq \text{None} \longleftrightarrow (\exists y. x = \text{Some } y)$
 $\langle \text{proof} \rangle$

lemma *not-Some-eq [iff]*: $(\forall y. x \neq \text{Some } y) \longleftrightarrow x = \text{None}$
 $\langle \text{proof} \rangle$

lemma *comp-the-Some[simp]*: *the o Some* = *id*
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *case-optionE*:
assumes *c*: (*case* *x* of *None* \Rightarrow *P* | *Some* *y* \Rightarrow *Q y*)
obtains
 (*None*) *x* = *None* **and** *P*
 | (*Some*) *y* **where** *x* = *Some y* **and** *Q y*
 $\langle \text{proof} \rangle$

lemma *split-option-all*: $(\forall x. P x) \longleftrightarrow P \text{ None} \wedge (\forall x. P (\text{Some } x))$
 $\langle \text{proof} \rangle$

lemma *split-option-ex*: $(\exists x. P x) \longleftrightarrow P \text{ None} \vee (\exists x. P (\text{Some } x))$
 $\langle \text{proof} \rangle$

lemma *UNIV-option-conv*: *UNIV* = *insert None (range Some)*

$\langle \text{proof} \rangle$

lemma *rel-option-None1* [simp]: $\text{rel-option } P \text{ None } x \longleftrightarrow x = \text{None}$
 $\langle \text{proof} \rangle$

lemma *rel-option-None2* [simp]: $\text{rel-option } P x \text{ None} \longleftrightarrow x = \text{None}$
 $\langle \text{proof} \rangle$

lemma *option-rel-Some1*: $\text{rel-option } A (\text{Some } x) y \longleftrightarrow (\exists y'. y = \text{Some } y' \wedge A x y')$
 $\langle \text{proof} \rangle$

lemma *option-rel-Some2*: $\text{rel-option } A x (\text{Some } y) \longleftrightarrow (\exists x'. x = \text{Some } x' \wedge A x' y)$
 $\langle \text{proof} \rangle$

lemma *rel-option-inf*: $\text{inf } (\text{rel-option } A) (\text{rel-option } B) = \text{rel-option } (\text{inf } A B)$
 (is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *rel-option-refl*:
 $(\bigwedge x. x \in \text{set-option } y \implies P x x) \implies \text{rel-option } P y y$
 $\langle \text{proof} \rangle$

50.0.1 Operations

lemma *ospec* [dest]: $(\forall x \in \text{set-option } A. P x) \implies A = \text{Some } x \implies P x$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *elem-set* [iff]: $(x \in \text{set-option } xo) = (xo = \text{Some } x)$
 $\langle \text{proof} \rangle$

lemma *set-empty-eq* [simp]: $(\text{set-option } xo = \{\}) = (xo = \text{None})$
 $\langle \text{proof} \rangle$

lemma *map-option-case*: $\text{map-option } f y = (\text{case } y \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow \text{Some } (f x))$
 $\langle \text{proof} \rangle$

lemma *map-option-is-None* [iff]: $(\text{map-option } f \text{ opt} = \text{None}) = (\text{opt} = \text{None})$
 $\langle \text{proof} \rangle$

lemma *None-eq-map-option-iff* [iff]: $\text{None} = \text{map-option } f x \longleftrightarrow x = \text{None}$
 $\langle \text{proof} \rangle$

lemma *map-option-eq-Some* [iff]: $(\text{map-option } f xo = \text{Some } y) = (\exists z. xo = \text{Some } z \wedge f z = y)$

$\langle \text{proof} \rangle$

lemma *map-option-o-case-sum* [simp]:

$\text{map-option } f \circ \text{case-sum } g \ h = \text{case-sum } (\text{map-option } f \circ g) (\text{map-option } f \circ h)$
 $\langle \text{proof} \rangle$

lemma *map-option-cong*: $x = y \implies (\bigwedge a. y = \text{Some } a \implies f \ a = g \ a) \implies \text{map-option } f \ x = \text{map-option } g \ y$
 $\langle \text{proof} \rangle$

lemma *map-option-idI*: $(\bigwedge y. y \in \text{set-option } x \implies f \ y = y) \implies \text{map-option } f \ x = x$
 $\langle \text{proof} \rangle$

functor *map-option*: *map-option*
 $\langle \text{proof} \rangle$

lemma *case-map-option* [simp]: $\text{case-option } g \ h (\text{map-option } f \ x) = \text{case-option } g \ (h \circ f) \ x$
 $\langle \text{proof} \rangle$

lemma *None-notin-image-Some* [simp]: $\text{None} \notin \text{Some } 'A$
 $\langle \text{proof} \rangle$

lemma *notin-range-Some*: $x \notin \text{range } \text{Some} \longleftrightarrow x = \text{None}$
 $\langle \text{proof} \rangle$

lemma *rel-option-iff*:

$\text{rel-option } R \ x \ y = (\text{case } (x, y) \text{ of } (\text{None}, \text{None}) \Rightarrow \text{True}$
 $\quad | (\text{Some } x, \text{Some } y) \Rightarrow R \ x \ y$
 $\quad | - \Rightarrow \text{False})$
 $\langle \text{proof} \rangle$

definition *combine-options* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \text{ option} \Rightarrow 'a \text{ option} \Rightarrow 'a \text{ option}$
where $\text{combine-options } f \ x \ y =$
 $(\text{case } x \text{ of } \text{None} \Rightarrow y \mid \text{Some } x \Rightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{Some } x \mid \text{Some } y \Rightarrow \text{Some } (f \ x \ y)))$

lemma *combine-options-simps* [simp]:

$\text{combine-options } f \ \text{None} \ y = y$
 $\text{combine-options } f \ x \ \text{None} = x$
 $\text{combine-options } f \ (\text{Some } a) \ (\text{Some } b) = \text{Some } (f \ a \ b)$
 $\langle \text{proof} \rangle$

lemma *combine-options-cases* [case-names None1 None2 Some]:

$(x = \text{None} \implies P \ x \ y) \implies (y = \text{None} \implies P \ x \ y) \implies$
 $(\bigwedge a \ b. x = \text{Some } a \implies y = \text{Some } b \implies P \ x \ y) \implies P \ x \ y$

$\langle \text{proof} \rangle$

lemma *combine-options-commute*:

$(\bigwedge x y. f x y = f y x) \implies \text{combine-options } f x y = \text{combine-options } f y x$
 $\langle \text{proof} \rangle$

lemma *combine-options-assoc*:

$(\bigwedge x y z. f (f x y) z = f x (f y z)) \implies$
 $\text{combine-options } f (\text{combine-options } f x y) z =$
 $\text{combine-options } f x (\text{combine-options } f y z)$
 $\langle \text{proof} \rangle$

lemma *combine-options-left-commute*:

$(\bigwedge x y. f x y = f y x) \implies (\bigwedge x y z. f (f x y) z = f x (f y z)) \implies$
 $\text{combine-options } f y (\text{combine-options } f x z) =$
 $\text{combine-options } f x (\text{combine-options } f y z)$
 $\langle \text{proof} \rangle$

lemmas *combine-options-ac* =

combine-options-commute combine-options-assoc combine-options-left-commute

context

begin

qualified definition *is-none* :: 'a option \Rightarrow bool

where [code-post]: *is-none* $x \longleftrightarrow x = \text{None}$

lemma *is-none-simps* [simp]:

is-none None
 $\neg \text{is-none } (\text{Some } x)$
 $\langle \text{proof} \rangle$

lemma *is-none-code* [code]:

is-none None \longleftrightarrow True
is-none (Some x) \longleftrightarrow False
 $\langle \text{proof} \rangle$

lemma *rel-option-unfold*:

rel-option $R x y \longleftrightarrow$
 $(\text{is-none } x \longleftrightarrow \text{is-none } y) \wedge (\neg \text{is-none } x \longrightarrow \neg \text{is-none } y \longrightarrow R (\text{the } x) (\text{the } y))$
 $\langle \text{proof} \rangle$

lemma *rel-optionI*:

$\llbracket \text{is-none } x \longleftrightarrow \text{is-none } y; \llbracket \neg \text{is-none } x; \neg \text{is-none } y \rrbracket \implies P (\text{the } x) (\text{the } y) \rrbracket$
 $\implies \text{rel-option } P x y$
 $\langle \text{proof} \rangle$

lemma *is-none-map-option* [simp]: $\text{is-none } (\text{map-option } f \ x) \longleftrightarrow \text{is-none } x$
 ⟨proof⟩

lemma *the-map-option*: $\neg \text{is-none } x \implies \text{the } (\text{map-option } f \ x) = f \ (\text{the } x)$
 ⟨proof⟩ **primrec** *bind* :: $'a \ \text{option} \Rightarrow ('a \Rightarrow 'b \ \text{option}) \Rightarrow 'b \ \text{option}$

where

bind-lzero: $\text{bind } \text{None } f = \text{None}$
 | *bind-lunit*: $\text{bind } (\text{Some } x) f = f \ x$

lemma *is-none-bind*: $\text{is-none } (\text{bind } f \ g) \longleftrightarrow \text{is-none } f \vee \text{is-none } (g \ (\text{the } f))$
 ⟨proof⟩

lemma *bind-runit*[simp]: $\text{bind } x \ \text{Some} = x$
 ⟨proof⟩

lemma *bind-assoc*[simp]: $\text{bind } (\text{bind } x \ f) \ g = \text{bind } x \ (\lambda y. \text{bind } (f \ y) \ g)$
 ⟨proof⟩

lemma *bind-rzero*[simp]: $\text{bind } x \ (\lambda x. \ \text{None}) = \text{None}$
 ⟨proof⟩ **lemma** *bind-cong*: $x = y \implies (\bigwedge a. y = \text{Some } a \implies f \ a = g \ a) \implies \text{bind } x \ f = \text{bind } y \ g$
 ⟨proof⟩

lemma *bind-split*: $P \ (\text{bind } m \ f) \longleftrightarrow (m = \text{None} \longrightarrow P \ \text{None}) \wedge (\forall v. m = \text{Some } v \longrightarrow P \ (f \ v))$
 ⟨proof⟩

lemma *bind-split-asm*: $P \ (\text{bind } m \ f) \longleftrightarrow \neg (m = \text{None} \wedge \neg P \ \text{None} \vee (\exists x. m = \text{Some } x \wedge \neg P \ (f \ x)))$
 ⟨proof⟩

lemmas *bind-splits* = *bind-split bind-split-asm*

lemma *bind-eq-Some-conv*: $\text{bind } f \ g = \text{Some } x \longleftrightarrow (\exists y. f = \text{Some } y \wedge g \ y = \text{Some } x)$
 ⟨proof⟩

lemma *bind-eq-None-conv*: $\text{Option.bind } a \ f = \text{None} \longleftrightarrow a = \text{None} \vee f \ (\text{the } a) = \text{None}$
 ⟨proof⟩

lemma *map-option-bind*: $\text{map-option } f \ (\text{bind } x \ g) = \text{bind } x \ (\text{map-option } f \ \circ \ g)$
 ⟨proof⟩

lemma *bind-option-cong*:
 $\llbracket x = y; \bigwedge z. z \in \text{set-option } y \implies f \ z = g \ z \rrbracket \implies \text{bind } x \ f = \text{bind } y \ g$
 ⟨proof⟩

lemma *bind-option-cong-simp*:

$\llbracket x = y; \bigwedge z. z \in \text{set-option } y = \text{simp} \Rightarrow f z = g z \rrbracket \Longrightarrow \text{bind } x f = \text{bind } y g$
 $\langle \text{proof} \rangle$

lemma *bind-option-cong-code*: $x = y \Longrightarrow \text{bind } x f = \text{bind } y f$
 $\langle \text{proof} \rangle$

lemma *bind-map-option*: $\text{bind } (\text{map-option } f x) g = \text{bind } x (g \circ f)$
 $\langle \text{proof} \rangle$

lemma *set-bind-option [simp]*: $\text{set-option } (\text{bind } x f) = (\bigcup ((\text{set-option} \circ f) \text{ ` } \text{set-option } x))$
 $\langle \text{proof} \rangle$

lemma *map-conv-bind-option*: $\text{map-option } f x = \text{Option.bind } x (\text{Some} \circ f)$
 $\langle \text{proof} \rangle$

end

$\langle \text{ML} \rangle$

context

begin

qualified definition *these* :: $'a \text{ option set} \Rightarrow 'a \text{ set}$
where *these* $A = \text{the } \{x \in A. x \neq \text{None}\}$

qualified lemma *these-eq [code]*:
 $\langle \text{these } A = \text{the } (\text{Set.filter } (\text{Not} \circ \text{Option.is-none}) A) \rangle$
 $\langle \text{proof} \rangle$ **lemma** *these-unfold*:
 $\langle \text{these } A = \{x. \exists y \in A. y = \text{Some } x\} \rangle$
 $\langle \text{proof} \rangle$

lemma *these-empty [simp]*: $\text{these } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *these-insert-None [simp]*: $\text{these } (\text{insert } \text{None } A) = \text{these } A$
 $\langle \text{proof} \rangle$

lemma *these-insert-Some [simp]*: $\text{these } (\text{insert } (\text{Some } x) A) = \text{insert } x (\text{these } A)$
 $\langle \text{proof} \rangle$

lemma *in-these-eq*: $x \in \text{these } A \longleftrightarrow \text{Some } x \in A$
 $\langle \text{proof} \rangle$

lemma *these-image-Some-eq [simp]*: $\text{these } (\text{Some } \text{ ` } A) = A$
 $\langle \text{proof} \rangle$

lemma *Some-image-these-eq*: $\text{Some } \text{ ` } \text{these } A = \{x \in A. x \neq \text{None}\}$

<proof>

lemma *these-empty-eq*: *these* $B = \{\}$ $\longleftrightarrow B = \{\} \vee B = \{None\}$
<proof>

lemma *these-not-empty-eq*: *these* $B \neq \{\} \longleftrightarrow B \neq \{\} \wedge B \neq \{None\}$
<proof> **definition** *image-filter* :: $('a \Rightarrow 'b \text{ option}) \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set}$
where *image-filter-eq*: *image-filter* $f A = \text{these } (f ` A)$

end

lemma *finite-range-Some*: *finite* (*range* (*Some* :: $'a \Rightarrow 'a \text{ option}$)) = *finite* (*UNIV* :: $'a \text{ set}$)
<proof>

50.1 Transfer rules for the Transfer package

context *includes* *lifting-syntax*
begin

lemma *option-bind-transfer* [*transfer-rule*]:
 $(\text{rel-option } A \text{ ===> } (A \text{ ===> rel-option } B) \text{ ===> rel-option } B)$
Option.bind Option.bind
<proof>

lemma *pred-option-parametric* [*transfer-rule*]:
 $((A \text{ ===> } (=)) \text{ ===> rel-option } A \text{ ===> } (=)) \text{ pred-option pred-option}$
<proof>

end

50.1.1 Interaction with finite sets

lemma *finite-option-UNIV* [*simp*]:
 $\text{finite } (\text{UNIV} :: 'a \text{ option set}) = \text{finite } (\text{UNIV} :: 'a \text{ set})$
<proof>

instance *option* :: (*finite*) *finite*
<proof>

50.1.2 Code generator setup

lemma *equal-None-code-unfold* [*code-unfold*]:
 $\text{HOL.equal } x \text{ None} \longleftrightarrow \text{Option.is-none } x$
 $\text{HOL.equal } \text{None} = \text{Option.is-none}$
<proof>

code-printing

type-constructor *option* \rightarrow
(SML) - option

```

    and (OCaml) - option
    and (Haskell) Maybe -
    and (Scala) !Option[(-)]
| constant None  $\mapsto$ 
  (SML) NONE
  and (OCaml) None
  and (Haskell) Nothing
  and (Scala) !None
| constant Some  $\mapsto$ 
  (SML) SOME
  and (OCaml) Some -
  and (Haskell) Just
  and (Scala) Some
| class-instance option :: equal  $\mapsto$ 
  (Haskell) -
| constant HOL.equal :: 'a option  $\Rightarrow$  'a option  $\Rightarrow$  bool  $\mapsto$ 
  (Haskell) infix 4 ==

code-reserved
  (SML) option NONE SOME
  and (OCaml) option None Some
  and (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>

lemma (in ccpo) in-chain-finite:
  assumes Complete-Partial-Order.chain ( $\leq$ ) A finite A  $A \neq \{\}$ 
  shows  $\bigsqcup A \in A$ 
<proof>

lemma (in ccpo) admissible-chfin:
  ( $\forall S. \text{Complete-Partial-Order.chain } (\leq) S \longrightarrow \text{finite } S$ )
 $\implies \text{ccpo.admissible Sup } (\leq) P$ 
<proof>

```

51.1 Axiomatic setup

This technical locale contains the requirements for function definitions with ccpo fixed points.

definition *fun-ord* $ord\ f\ g \longleftrightarrow (\forall x. ord\ (f\ x)\ (g\ x))$

definition *fun-lub* $L\ A = (\lambda x. L\ \{y. \exists f \in A. y = f\ x\})$

definition *img-ord* $f\ ord = (\lambda x\ y. ord\ (f\ x)\ (f\ y))$

definition *img-lub* $f\ g\ Lub = (\lambda A. g\ (Lub\ (f\ ` A)))$

lemma *chain-fun*:

assumes $A: chain\ (fun-ord\ ord)\ A$

shows $chain\ ord\ \{y. \exists f \in A. y = f\ a\}$ **(is** $chain\ ord\ ?C$)

<proof>

lemma *call-mono*[*partial-function-mono*]: $monotone\ (fun-ord\ ord)\ ord\ (\lambda f. f\ t)$

<proof>

lemma *let-mono*[*partial-function-mono*]:

$(\bigwedge x. monotone\ orda\ ordb\ (\lambda f. b\ f\ x))$

$\implies monotone\ orda\ ordb\ (\lambda f. Let\ t\ (b\ f))$

<proof>

lemma *if-mono*[*partial-function-mono*]: $monotone\ orda\ ordb\ F$

$\implies monotone\ orda\ ordb\ G$

$\implies monotone\ orda\ ordb\ (\lambda f. if\ c\ then\ F\ f\ else\ G\ f)$

<proof>

definition *mk-less* $R = (\lambda x\ y. R\ x\ y \wedge \neg R\ y\ x)$

locale *partial-function-definitions* =

fixes $leq :: 'a \Rightarrow 'a \Rightarrow bool$

fixes $lub :: 'a\ set \Rightarrow 'a$

assumes *leq-refl*: $leq\ x\ x$

assumes *leq-trans*: $leq\ x\ y \implies leq\ y\ z \implies leq\ x\ z$

assumes *leq-antisym*: $leq\ x\ y \implies leq\ y\ x \implies x = y$

assumes *lub-upper*: $chain\ leq\ A \implies x \in A \implies leq\ x\ (lub\ A)$

assumes *lub-least*: $chain\ leq\ A \implies (\bigwedge x. x \in A \implies leq\ x\ z) \implies leq\ (lub\ A)\ z$

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>

lemma *ccpo*: **assumes** *partial-function-definitions* $ord\ lb$

shows *class.ccpo* $lb\ ord\ (mk-less\ ord)$

<proof>

lemma *partial-function-image*:

```

assumes partial-function-definitions ord Lub
assumes inj:  $\bigwedge x y. f\ x = f\ y \implies x = y$ 
assumes inv:  $\bigwedge x. f\ (g\ x) = x$ 
shows partial-function-definitions (img-ord f ord) (img-lub f g Lub)
<proof>

```

```

context partial-function-definitions
begin

```

```

abbreviation le-fun  $\equiv$  fun-ord leq
abbreviation lub-fun  $\equiv$  fun-lub lub
abbreviation fixp-fun  $\equiv$  ccpo.fixp lub-fun le-fun
abbreviation mono-body  $\equiv$  monotone le-fun leq
abbreviation admissible  $\equiv$  ccpo.admissible lub-fun le-fun

```

Interpret manually, to avoid flooding everything with facts about orders

```

lemma ccpo: class.ccpo lub-fun le-fun (mk-less le-fun)
<proof>

```

The crucial fixed-point theorem

```

lemma mono-body-fixp:
  ( $\bigwedge x. \text{mono-body } (\lambda f. F\ f\ x)$ )  $\implies$  fixp-fun F = F (fixp-fun F)
<proof>

```

Version with curry/uncurry combinators, to be used by package

```

lemma fixp-rule-uc:
  fixes F :: 'c  $\Rightarrow$  'c and
    U :: 'c  $\Rightarrow$  'b  $\Rightarrow$  'a and
    C :: ('b  $\Rightarrow$  'a)  $\Rightarrow$  'c
  assumes mono:  $\bigwedge x. \text{mono-body } (\lambda f. U\ (F\ (C\ f))\ x)$ 
  assumes eq:  $f \equiv C\ (\text{fixp-fun } (\lambda f. U\ (F\ (C\ f))))$ 
  assumes inverse:  $\bigwedge f. C\ (U\ f) = f$ 
  shows  $f = F\ f$ 
<proof>

```

Fixpoint induction rule

```

lemma fixp-induct-uc:
  fixes F :: 'c  $\Rightarrow$  'c
  and U :: 'c  $\Rightarrow$  'b  $\Rightarrow$  'a
  and C :: ('b  $\Rightarrow$  'a)  $\Rightarrow$  'c
  and P :: ('b  $\Rightarrow$  'a)  $\Rightarrow$  bool
  assumes mono:  $\bigwedge x. \text{mono-body } (\lambda f. U\ (F\ (C\ f))\ x)$ 
  and eq:  $f \equiv C\ (\text{fixp-fun } (\lambda f. U\ (F\ (C\ f))))$ 
  and inverse:  $\bigwedge f. U\ (C\ f) = f$ 
  and adm: ccpo.admissible lub-fun le-fun P
  and bot:  $P\ (\lambda -. \text{lub } \{\})$ 
  and step:  $\bigwedge f. P\ (U\ f) \implies P\ (U\ (F\ f))$ 
  shows  $P\ (U\ f)$ 

```

$\langle proof \rangle$

Rules for *monotone le-fun leq*:

lemma *const-mono*[*partial-function-mono*]: *monotone ord leq* ($\lambda f. c$)
 $\langle proof \rangle$

end

51.2 Flat interpretation: tailrec and option

definition

flat-ord $b\ x\ y \longleftrightarrow x = b \vee x = y$

definition

flat-lub $b\ A = (\text{if } A \subseteq \{b\} \text{ then } b \text{ else } (THE\ x. x \in A - \{b\}))$

lemma *flat-interpretation*:

partial-function-definitions (*flat-ord* b) (*flat-lub* b)
 $\langle proof \rangle$

lemma *flat-ordI*: $(x \neq a \implies x = y) \implies \text{flat-ord } a\ x\ y$

$\langle proof \rangle$

lemma *flat-ord-antisym*: $\llbracket \text{flat-ord } a\ x\ y; \text{flat-ord } a\ y\ x \rrbracket \implies x = y$

$\langle proof \rangle$

lemma *antisym-flat-ord*: *antisym* (*flat-ord* a)

$\langle proof \rangle$

interpretation *tailrec*:

partial-function-definitions *flat-ord* *undefined* *flat-lub* *undefined*
rewrites *flat-lub* *undefined* $\{\} \equiv \text{undefined}$
 $\langle proof \rangle$

interpretation *option*:

partial-function-definitions *flat-ord* *None* *flat-lub* *None*
rewrites *flat-lub* *None* $\{\} \equiv \text{None}$
 $\langle proof \rangle$

abbreviation *tailrec-ord* $\equiv \text{flat-ord } \text{undefined}$

abbreviation *mono-tailrec* $\equiv \text{monotone } (\text{fun-ord } \text{tailrec-ord})\ \text{tailrec-ord}$

lemma *tailrec-admissible*:

ccpo.admissible (*fun-lub* (*flat-lub* c)) (*fun-ord* (*flat-ord* c))
 $(\lambda a. \forall x. a\ x \neq c \longrightarrow P\ x\ (a\ x))$
 $\langle proof \rangle$

lemma *fixp-induct-tailrec*:

fixes $F :: 'c \Rightarrow 'c$ **and**
 $U :: 'c \Rightarrow 'b \Rightarrow 'a$ **and**
 $C :: ('b \Rightarrow 'a) \Rightarrow 'c$ **and**
 $P :: 'b \Rightarrow 'a \Rightarrow \text{bool}$ **and**
 $x :: 'b$
assumes *mono*: $\bigwedge x. \text{monotone } (\text{fun-ord } (\text{flat-ord } c)) (\text{flat-ord } c) (\lambda f. U (F (C f)) x)$
assumes *eq*: $f \equiv C (\text{ccpo.fixp } (\text{fun-lub } (\text{flat-lub } c)) (\text{fun-ord } (\text{flat-ord } c)) (\lambda f. U (F (C f))))$
assumes *inverse2*: $\bigwedge f. U (C f) = f$
assumes *step*: $\bigwedge f x y. (\bigwedge x y. U f x = y \implies y \neq c \implies P x y) \implies U (F f) x = y \implies y \neq c \implies P x y$
assumes *result*: $U f x = y$
assumes *defined*: $y \neq c$
shows $P x y$
 $\langle \text{proof} \rangle$

lemma *admissible-image*:
assumes *pfun*: *partial-function-definitions* *le* *lub*
assumes *adm*: *ccpo.admissible* *lub* *le* $(P \circ g)$
assumes *inj*: $\bigwedge x y. f x = f y \implies x = y$
assumes *inv*: $\bigwedge x. f (g x) = x$
shows *ccpo.admissible* $(\text{img-lub } f g \text{ lub}) (\text{img-ord } f \text{ le}) P$
 $\langle \text{proof} \rangle$

lemma *admissible-fun*:
assumes *pfun*: *partial-function-definitions* *le* *lub*
assumes *adm*: $\bigwedge x. \text{ccpo.admissible } \text{lub } \text{le } (Q x)$
shows *ccpo.admissible* $(\text{fun-lub } \text{lub}) (\text{fun-ord } \text{le}) (\lambda f. \forall x. Q x (f x))$
 $\langle \text{proof} \rangle$

abbreviation *option-ord* $\equiv \text{flat-ord } \text{None}$
abbreviation *mono-option* $\equiv \text{monotone } (\text{fun-ord } \text{option-ord}) \text{ option-ord}$

lemma *bind-mono*[*partial-function-mono*]:
assumes *mf*: *mono-option* *B* **and** *mg*: $\bigwedge y. \text{mono-option } (\lambda f. C y f)$
shows *mono-option* $(\lambda f. \text{Option.bind } (B f) (\lambda y. C y f))$
 $\langle \text{proof} \rangle$

lemma *flat-lub-in-chain*:
assumes *ch*: *chain* $(\text{flat-ord } b) A$
assumes *lub*: *flat-lub* *b* $A = a$
shows $a = b \vee a \in A$
 $\langle \text{proof} \rangle$

lemma *option-admissible*: *option.admissible* $(\%(f :: 'a \Rightarrow 'b \text{ option}). (\forall x y. f x = \text{Some } y \longrightarrow P x y))$
 $\langle \text{proof} \rangle$

```

lemma fixp-induct-option:
  fixes  $F :: 'c \Rightarrow 'c$  and
     $U :: 'c \Rightarrow 'b \Rightarrow 'a \text{ option}$  and
     $C :: ('b \Rightarrow 'a \text{ option}) \Rightarrow 'c$  and
     $P :: 'b \Rightarrow 'a \Rightarrow \text{bool}$ 
  assumes mono:  $\bigwedge x. \text{mono-option } (\lambda f. U (F (C f)) x)$ 
  assumes eq:  $f \equiv C (\text{ccpo.fixp } (\text{fun-lub } (\text{flat-lub None})) (\text{fun-ord option-ord}) (\lambda f. U (F (C f))))$ 
  assumes inverse2:  $\bigwedge f. U (C f) = f$ 
  assumes step:  $\bigwedge f x y. (\bigwedge x y. U f x = \text{Some } y \implies P x y) \implies U (F f) x = \text{Some } y \implies P x y$ 
  assumes defined:  $U f x = \text{Some } y$ 
  shows  $P x y$ 
  <proof>

<ML>

hide-const (open) chain

end

```

```

theory Argo
imports HOL
begin

```

<ML>

end

52 Reconstructing external resolution proofs for propositional logic

```

theory SAT
imports Argo
begin

```

<ML>

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* :: $'a \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow 'a$
where *THE-default* $d\ P = (\text{if } (\exists!x. P\ x) \text{ then } (THE\ x. P\ x) \text{ else } d)$

lemma *THE-defaultI'*: $\exists!x. P\ x \Longrightarrow P\ (THE\text{-default}\ d\ P)$
 $\langle\text{proof}\rangle$

lemma *THE-default1-equality*: $\exists!x. P\ x \Longrightarrow P\ a \Longrightarrow THE\text{-default}\ d\ P = a$
 $\langle\text{proof}\rangle$

lemma *THE-default-none*: $\neg (\exists!x. P\ x) \Longrightarrow THE\text{-default}\ d\ P = d$
 $\langle\text{proof}\rangle$

lemma *fundef-ex1-existence*:
assumes *f-def*: $f \equiv (\lambda x::'a. THE\text{-default}\ (d\ x) (\lambda y. G\ x\ y))$
assumes *ex1*: $\exists!y. G\ x\ y$
shows $G\ x\ (f\ x)$
 $\langle\text{proof}\rangle$

lemma *fundef-ex1-uniqueness*:
assumes *f-def*: $f \equiv (\lambda x::'a. THE\text{-default}\ (d\ x) (\lambda y. G\ x\ y))$
assumes *ex1*: $\exists!y. G\ x\ y$
assumes *elm*: $G\ x\ (h\ x)$
shows $h\ x = f\ x$
 $\langle\text{proof}\rangle$

lemma *fundef-ex1-iff*:
assumes *f-def*: $f \equiv (\lambda x::'a. THE\text{-default}\ (d\ x) (\lambda y. G\ x\ y))$
assumes *ex1*: $\exists!y. G\ x\ y$
shows $(G\ x\ y) = (f\ x = y)$
 $\langle\text{proof}\rangle$

lemma *fundef-default-value*:
assumes *f-def*: $f \equiv (\lambda x::'a. THE\text{-default}\ (d\ x) (\lambda y. G\ x\ y))$
assumes *graph*: $\bigwedge x\ y. G\ x\ y \Longrightarrow D\ x$
assumes $\neg D\ x$
shows $f\ x = d\ x$
 $\langle\text{proof}\rangle$

definition *in-rel-def[simp]*: $\text{in-rel}\ R\ x\ y \equiv (x, y) \in R$

lemma *wf-in-rel*: $\text{wf}\ R \Longrightarrow \text{wfp}\ (\text{in-rel}\ R)$
 $\langle\text{proof}\rangle$

$\langle ML \rangle$

53.2 Measure functions

inductive *is-measure* :: (*a* \Rightarrow *nat*) \Rightarrow *bool*
where *is-measure-trivial*: *is-measure* *f*

named-theorems *measure-function* rules that guide the heuristic generation of measure functions

$\langle ML \rangle$

lemma *measure-size*[*measure-function*]: *is-measure* *size*
 $\langle proof \rangle$

lemma *measure-fst*[*measure-function*]: *is-measure* *f* \Longrightarrow *is-measure* ($\lambda p. f (fst\ p)$)
 $\langle proof \rangle$

lemma *measure-snd*[*measure-function*]: *is-measure* *f* \Longrightarrow *is-measure* ($\lambda p. f (snd\ p)$)
 $\langle proof \rangle$

$\langle ML \rangle$

53.3 Congruence rules

lemma *let-cong* [*fundef-cong*]: $M = N \Longrightarrow (\bigwedge x. x = N \Longrightarrow f\ x = g\ x) \Longrightarrow Let\ M\ f = Let\ N\ g$
 $\langle proof \rangle$

lemmas [*fundef-cong*] =
if-cong *image-cong*
be_x-cong *ball-cong* *imp-cong* *map-option-cong* *Option.bind-cong*

lemma *split-cong* [*fundef-cong*]:
 $(\bigwedge x\ y. (x, y) = q \Longrightarrow f\ x\ y = g\ x\ y) \Longrightarrow p = q \Longrightarrow case\ prod\ f\ p = case\ prod\ g\ q$
 $\langle proof \rangle$

lemma *comp-cong* [*fundef-cong*]: $f\ (g\ x) = f'\ (g'\ x') \Longrightarrow (f \circ g)\ x = (f' \circ g')\ x'$
 $\langle proof \rangle$

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*]: $\text{size-prod } f \ g \ p = f \ (\text{fst } p) + g \ (\text{snd } p) + \text{Suc } 0$
 $\langle \text{proof} \rangle$

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*: $R \ O \ R = \{\} \implies \text{wf } R$
 $\langle \text{proof} \rangle$

53.6 Reduction pairs

definition *reduction-pair* $P \longleftrightarrow \text{wf } (\text{fst } P) \wedge \text{fst } P \ O \ \text{snd } P \subseteq \text{fst } P$

lemma *reduction-pairI*[*intro*]: $\text{wf } R \implies R \ O \ S \subseteq R \implies \text{reduction-pair } (R, S)$
 $\langle \text{proof} \rangle$

lemma *reduction-pair-lemma*:
assumes *rp*: *reduction-pair* P
assumes $R \subseteq \text{fst } P$
assumes $S \subseteq \text{snd } P$
assumes $\text{wf } S$
shows $\text{wf } (R \cup S)$
 $\langle \text{proof} \rangle$

definition *rp-inv-image* $= (\lambda(R,S) f. (\text{inv-image } R \ f, \text{inv-image } S \ f))$

lemma *rp-inv-image-rp*: $\text{reduction-pair } P \implies \text{reduction-pair } (\text{rp-inv-image } P \ f)$
 $\langle \text{proof} \rangle$

53.7 Concrete orders for SCNP termination proofs

definition *pair-less* $= \text{less-than } <*\text{lex}*> \text{less-than}$

definition *pair-leq* $= \text{pair-less}^=$

definition *max-strict* $= \text{max-ext pair-less}$

definition *max-weak* $= \text{max-ext pair-leq} \cup \{(\{\}, \{\})\}$

definition *min-strict* $= \text{min-ext pair-less}$

definition *min-weak* $= \text{min-ext pair-leq} \cup \{(\{\}, \{\})\}$

lemma *wf-pair-less*[*simp*]: wf pair-less
 $\langle \text{proof} \rangle$

lemma *total-pair-less* [*iff*]: $\text{total-on } A \ \text{pair-less}$ **and** trans-pair-less [*iff*]: trans pair-less
 $\langle \text{proof} \rangle$

Introduction rules for *pair-less*/*pair-leq*

lemma *pair-leqI1*: $a < b \implies ((a, s), (b, t)) \in \text{pair-leq}$
and *pair-leqI2*: $a \leq b \implies s \leq t \implies ((a, s), (b, t)) \in \text{pair-leq}$
and *pair-lessI1*: $a < b \implies ((a, s), (b, t)) \in \text{pair-less}$
and *pair-lessI2*: $a \leq b \implies s < t \implies ((a, s), (b, t)) \in \text{pair-less}$
 ⟨proof⟩

lemma *pair-less-iff1* [simp]: $((x, y), (x, z)) \in \text{pair-less} \longleftrightarrow y < z$
 ⟨proof⟩

Introduction rules for max

lemma *smax-emptyI*: $\text{finite } Y \implies Y \neq \{\} \implies (\{\}, Y) \in \text{max-strict}$
and *smax-insertI*:
 $y \in Y \implies (x, y) \in \text{pair-less} \implies (X, Y) \in \text{max-strict} \implies (\text{insert } x \ X, Y) \in \text{max-strict}$
and *wmax-emptyI*: $\text{finite } X \implies (\{\}, X) \in \text{max-weak}$
and *wmax-insertI*:
 $y \in YS \implies (x, y) \in \text{pair-leq} \implies (XS, YS) \in \text{max-weak} \implies (\text{insert } x \ XS, YS) \in \text{max-weak}$
 ⟨proof⟩

Introduction rules for min

lemma *smin-emptyI*: $X \neq \{\} \implies (X, \{\}) \in \text{min-strict}$
and *smin-insertI*:
 $x \in XS \implies (x, y) \in \text{pair-less} \implies (XS, YS) \in \text{min-strict} \implies (XS, \text{insert } y \ YS) \in \text{min-strict}$
and *wmin-emptyI*: $(X, \{\}) \in \text{min-weak}$
and *wmin-insertI*:
 $x \in XS \implies (x, y) \in \text{pair-leq} \implies (XS, YS) \in \text{min-weak} \implies (XS, \text{insert } y \ YS) \in \text{min-weak}$
 ⟨proof⟩

Reduction Pairs.

lemma *max-ext-compat*:
assumes $R \ O \ S \subseteq R$
shows $\text{max-ext } R \ O \ (\text{max-ext } S \cup \{(\{\}, \{\})\}) \subseteq \text{max-ext } R$
 ⟨proof⟩

lemma *max-rpair-set*: *reduction-pair* (*max-strict*, *max-weak*)
 ⟨proof⟩

lemma *min-ext-compat*:
assumes $R \ O \ S \subseteq R$
shows $\text{min-ext } R \ O \ (\text{min-ext } S \cup \{(\{\}, \{\})\}) \subseteq \text{min-ext } R$
 ⟨proof⟩

lemma *min-rpair-set*: *reduction-pair* (*min-strict*, *min-weak*)
 ⟨proof⟩

53.8 Yet more induction principles on the natural numbers

lemma *nat-descend-induct* [*case-names base descend*]:
 fixes $P :: \text{nat} \Rightarrow \text{bool}$
 assumes $H1: \bigwedge k. k > n \implies P\ k$
 assumes $H2: \bigwedge k. k \leq n \implies (\bigwedge i. i > k \implies P\ i) \implies P\ k$
 shows $P\ m$
 $\langle \text{proof} \rangle$

lemma *induct-nat-012* [*case-names 0 1 ge2*]:
 $P\ 0 \implies P\ (\text{Suc}\ 0) \implies (\bigwedge n. P\ n \implies P\ (\text{Suc}\ n) \implies P\ (\text{Suc}\ (\text{Suc}\ n))) \implies P\ n$
 $\langle \text{proof} \rangle$

53.9 Tool setup

$\langle ML \rangle$

end

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* :: $(\text{nat} \times \text{nat}) \Rightarrow (\text{nat} \times \text{nat}) \Rightarrow \text{bool}$
 where $\text{intrel} = (\lambda(x, y)\ (u, v). x + v = u + y)$

lemma *intrel-iff* [*simp*]: $\text{intrel}\ (x, y)\ (u, v) \longleftrightarrow x + v = u + y$
 $\langle \text{proof} \rangle$

quotient-type $\text{int} = \text{nat} \times \text{nat} / \text{intrel}$
morphisms *Rep-Integ Abs-Integ*
 $\langle \text{proof} \rangle$

54.2 Integers form a commutative ring

instantiation *int* :: *comm-ring-1*
begin

lift-definition *zero-int* :: *int* **is** $(0, 0)$ $\langle \text{proof} \rangle$

lift-definition *one-int* :: *int* **is** $(1, 0)$ $\langle \text{proof} \rangle$

lift-definition *plus-int* :: *int* \Rightarrow *int* \Rightarrow *int*
is $\lambda(x, y)\ (u, v). (x + u, y + v)$

$\langle proof \rangle$

lift-definition *uminus-int* :: *int* \Rightarrow *int*
is $\lambda(x, y). (y, x)$
 $\langle proof \rangle$

lift-definition *minus-int* :: *int* \Rightarrow *int* \Rightarrow *int*
is $\lambda(x, y) (u, v). (x + v, y + u)$
 $\langle proof \rangle$

lift-definition *times-int* :: *int* \Rightarrow *int* \Rightarrow *int*
is $\lambda(x, y) (u, v). (x*u + y*v, x*v + y*u)$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

abbreviation *int* :: *nat* \Rightarrow *int*
where *int* \equiv *of-nat*

lemma *int-def*: *int* *n* = *Abs-Integ* (*n*, 0)
 $\langle proof \rangle$

lemma *int-transfer* [*transfer-rule*]:
includes *lifting-syntax*
shows *rel-fun* (=) *pcr-int* ($\lambda n. (n, 0)$) *int*
 $\langle proof \rangle$

lemma *int-diff-cases*: **obtains** (*diff*) *m n* **where** *z* = *int* *m* - *int* *n*
 $\langle proof \rangle$

54.3 Integers are totally ordered

instantiation *int* :: *linorder*
begin

lift-definition *less-eq-int* :: *int* \Rightarrow *int* \Rightarrow *bool*
is $\lambda(x, y) (u, v). x + v \leq u + y$
 $\langle proof \rangle$

lift-definition *less-int* :: *int* \Rightarrow *int* \Rightarrow *bool*
is $\lambda(x, y) (u, v). x + v < u + y$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

instantiation *int* :: *distrib-lattice*
begin

definition (*inf* :: *int* \Rightarrow *int* \Rightarrow *int*) = *min*

definition (*sup* :: *int* \Rightarrow *int* \Rightarrow *int*) = *max*

instance
 $\langle proof \rangle$

end

54.4 Ordering properties of arithmetic operations

instance *int* :: *ordered-cancel-ab-semigroup-add*
 $\langle proof \rangle$

Strict Monotonicity of Multiplication.

Strict, in 1st argument; proof is by induction on $k > 0$.

lemma *zmult-zless-mono2-lemma*: $i < j \implies 0 < k \implies \text{int } k * i < \text{int } k * j$
for $i\ j :: \text{int}$
 $\langle proof \rangle$

lemma *zero-le-imp-eq-int*:
assumes $k \geq (0 :: \text{int})$ **shows** $\exists n. k = \text{int } n$
 $\langle proof \rangle$

lemma *zero-less-imp-eq-int*:
assumes $k > (0 :: \text{int})$ **shows** $\exists n > 0. k = \text{int } n$
 $\langle proof \rangle$

lemma *zmult-zless-mono2*: $i < j \implies 0 < k \implies k * i < k * j$
for $i\ j\ k :: \text{int}$
 $\langle proof \rangle$

The integers form an ordered integral domain.

instantiation *int* :: *linordered-idom*
begin

definition *zabs-def*: $|i :: \text{int}| = (\text{if } i < 0 \text{ then } -i \text{ else } i)$

definition *zsgn-def*: $\text{sgn } (i :: \text{int}) = (\text{if } i = 0 \text{ then } 0 \text{ else if } 0 < i \text{ then } 1 \text{ else } -1)$

instance
 $\langle proof \rangle$

end

instance *int* :: *discrete-linordered-semidom*
 ⟨*proof*⟩

lemma *zless-imp-add1-zle*: $w < z \implies w + 1 \leq z$
for $w\ z :: \text{int}$
 ⟨*proof*⟩

lemma *zless-iff-Suc-zadd*: $w < z \longleftrightarrow (\exists n. z = w + \text{int } (\text{Suc } n))$
for $w\ z :: \text{int}$
 ⟨*proof*⟩

lemma *zabs-less-one-iff* [*simp*]: $|z| < 1 \longleftrightarrow z = 0$ (**is** $?lhs \longleftrightarrow ?rhs$)
for $z :: \text{int}$
 ⟨*proof*⟩

54.5 Embedding of the Integers into any *ring-1*: *of-int*

context *ring-1*
begin

lift-definition *of-int* :: $\text{int} \Rightarrow 'a$
is $\lambda(i, j). \text{of-nat } i - \text{of-nat } j$
 ⟨*proof*⟩

lemma *of-int-0* [*simp*]: $\text{of-int } 0 = 0$
 ⟨*proof*⟩

lemma *of-int-1* [*simp*]: $\text{of-int } 1 = 1$
 ⟨*proof*⟩

lemma *of-int-add* [*simp*]: $\text{of-int } (w + z) = \text{of-int } w + \text{of-int } z$
 ⟨*proof*⟩

lemma *of-int-minus* [*simp*]: $\text{of-int } (-z) = -(\text{of-int } z)$
 ⟨*proof*⟩

lemma *of-int-diff* [*simp*]: $\text{of-int } (w - z) = \text{of-int } w - \text{of-int } z$
 ⟨*proof*⟩

lemma *of-int-mult* [*simp*]: $\text{of-int } (w * z) = \text{of-int } w * \text{of-int } z$
 ⟨*proof*⟩

lemma *mult-of-int-commute*: $\text{of-int } x * y = y * \text{of-int } x$
 ⟨*proof*⟩

Collapse nested embeddings.

lemma *of-int-of-nat-eq* [*simp*]: $\text{of-int } (\text{int } n) = \text{of-nat } n$

$\langle \text{proof} \rangle$

lemma *of-int-numeral* [*simp*, *code-post*]: *of-int* (numeral *k*) = numeral *k*
 $\langle \text{proof} \rangle$

lemma *of-int-neg-numeral* [*code-post*]: *of-int* (– numeral *k*) = – numeral *k*
 $\langle \text{proof} \rangle$

lemma *of-int-power* [*simp*]: *of-int* (*z* ^ *n*) = *of-int* *z* ^ *n*
 $\langle \text{proof} \rangle$

lemma *of-int-of-bool* [*simp*]:
of-int (*of-bool* *P*) = *of-bool* *P*
 $\langle \text{proof} \rangle$

end

context *ring-char-0*
begin

lemma *of-int-eq-iff* [*simp*]: *of-int* *w* = *of-int* *z* \longleftrightarrow *w* = *z*
 $\langle \text{proof} \rangle$

Special cases where either operand is zero.

lemma *of-int-eq-0-iff* [*simp*]: *of-int* *z* = 0 \longleftrightarrow *z* = 0
 $\langle \text{proof} \rangle$

lemma *of-int-0-eq-iff* [*simp*]: 0 = *of-int* *z* \longleftrightarrow *z* = 0
 $\langle \text{proof} \rangle$

lemma *of-int-eq-1-iff* [*iff*]: *of-int* *z* = 1 \longleftrightarrow *z* = 1
 $\langle \text{proof} \rangle$

lemma *of-nat-of-int-iff*: *of-int* *i* = *of-nat* *n* \longleftrightarrow *i* = *of-nat* *n* *of-nat* *n* = *of-int* *i*
 \longleftrightarrow *i* = *of-nat* *n*
 $\langle \text{proof} \rangle$

lemma *numeral-power-eq-of-int-cancel-iff* [*simp*]:
numeral *x* ^ *n* = *of-int* *y* \longleftrightarrow numeral *x* ^ *n* = *y*
 $\langle \text{proof} \rangle$

lemma *of-int-eq-numeral-power-cancel-iff* [*simp*]:
of-int *y* = numeral *x* ^ *n* \longleftrightarrow *y* = numeral *x* ^ *n*
 $\langle \text{proof} \rangle$

lemma *neg-numeral-power-eq-of-int-cancel-iff* [*simp*]:
(– numeral *x*) ^ *n* = *of-int* *y* \longleftrightarrow (– numeral *x*) ^ *n* = *y*
 $\langle \text{proof} \rangle$

lemma *of-int-eq-neg-numeral-power-cancel-iff* [simp]:

$$\text{of-int } y = (- \text{ numeral } x) \wedge n \longleftrightarrow y = (- \text{ numeral } x) \wedge n$$
 ⟨proof⟩

lemma *of-int-eq-of-int-power-cancel-iff* [simp]: $(\text{of-int } b) \wedge w = \text{of-int } x \longleftrightarrow b \wedge w = x$
 ⟨proof⟩

lemma *of-int-power-eq-of-int-cancel-iff* [simp]: $\text{of-int } x = (\text{of-int } b) \wedge w \longleftrightarrow x = b \wedge w$
 ⟨proof⟩

end

context *linordered-idom*
begin

Every *linordered-idom* has characteristic zero.

subclass *ring-char-0* ⟨proof⟩

lemma *of-int-le-iff* [simp]: $\text{of-int } w \leq \text{of-int } z \longleftrightarrow w \leq z$
 ⟨proof⟩

lemma *of-int-less-iff* [simp]: $\text{of-int } w < \text{of-int } z \longleftrightarrow w < z$
 ⟨proof⟩

lemma *of-int-0-le-iff* [simp]: $0 \leq \text{of-int } z \longleftrightarrow 0 \leq z$
 ⟨proof⟩

lemma *of-int-le-0-iff* [simp]: $\text{of-int } z \leq 0 \longleftrightarrow z \leq 0$
 ⟨proof⟩

lemma *of-int-0-less-iff* [simp]: $0 < \text{of-int } z \longleftrightarrow 0 < z$
 ⟨proof⟩

lemma *of-int-less-0-iff* [simp]: $\text{of-int } z < 0 \longleftrightarrow z < 0$
 ⟨proof⟩

lemma *of-int-1-le-iff* [simp]: $1 \leq \text{of-int } z \longleftrightarrow 1 \leq z$
 ⟨proof⟩

lemma *of-int-le-1-iff* [simp]: $\text{of-int } z \leq 1 \longleftrightarrow z \leq 1$
 ⟨proof⟩

lemma *of-int-1-less-iff* [simp]: $1 < \text{of-int } z \longleftrightarrow 1 < z$
 ⟨proof⟩

lemma *of-int-less-1-iff* [simp]: $\text{of-int } z < 1 \longleftrightarrow z < 1$
 ⟨proof⟩

lemma *of-int-pos*: $z > 0 \implies \text{of-int } z > 0$
 $\langle \text{proof} \rangle$

lemma *of-int-nonneg*: $z \geq 0 \implies \text{of-int } z \geq 0$
 $\langle \text{proof} \rangle$

lemma *of-int-abs* [simp]: $\text{of-int } |x| = |\text{of-int } x|$
 $\langle \text{proof} \rangle$

lemma *of-int-lessD*:
 assumes $|\text{of-int } n| < x$
 shows $n = 0 \vee x > 1$
 $\langle \text{proof} \rangle$

lemma *of-int-leD*:
 assumes $|\text{of-int } n| \leq x$
 shows $n = 0 \vee 1 \leq x$
 $\langle \text{proof} \rangle$

lemma *numeral-power-le-of-int-cancel-iff* [simp]:
 $\text{numeral } x \wedge n \leq \text{of-int } a \longleftrightarrow \text{numeral } x \wedge n \leq a$
 $\langle \text{proof} \rangle$

lemma *of-int-le-numeral-power-cancel-iff* [simp]:
 $\text{of-int } a \leq \text{numeral } x \wedge n \longleftrightarrow a \leq \text{numeral } x \wedge n$
 $\langle \text{proof} \rangle$

lemma *numeral-power-less-of-int-cancel-iff* [simp]:
 $\text{numeral } x \wedge n < \text{of-int } a \longleftrightarrow \text{numeral } x \wedge n < a$
 $\langle \text{proof} \rangle$

lemma *of-int-less-numeral-power-cancel-iff* [simp]:
 $\text{of-int } a < \text{numeral } x \wedge n \longleftrightarrow a < \text{numeral } x \wedge n$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-power-le-of-int-cancel-iff* [simp]:
 $(- \text{numeral } x) \wedge n \leq \text{of-int } a \longleftrightarrow (- \text{numeral } x) \wedge n \leq a$
 $\langle \text{proof} \rangle$

lemma *of-int-le-neg-numeral-power-cancel-iff* [simp]:
 $\text{of-int } a \leq (- \text{numeral } x) \wedge n \longleftrightarrow a \leq (- \text{numeral } x) \wedge n$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-power-less-of-int-cancel-iff* [simp]:
 $(- \text{numeral } x) \wedge n < \text{of-int } a \longleftrightarrow (- \text{numeral } x) \wedge n < a$
 $\langle \text{proof} \rangle$

lemma *of-int-less-neg-numeral-power-cancel-iff* [simp]:

$of_int\ a < (-\ numeral\ x) \wedge n \longleftrightarrow a < (-\ numeral\ x::int) \wedge n$
 $\langle proof \rangle$

lemma *of-int-le-of-int-power-cancel-iff*[simp]: $(of_int\ b) \wedge w \leq of_int\ x \longleftrightarrow b \wedge w \leq x$
 $\langle proof \rangle$

lemma *of-int-power-le-of-int-cancel-iff*[simp]: $of_int\ x \leq (of_int\ b) \wedge w \longleftrightarrow x \leq b \wedge w$
 $\langle proof \rangle$

lemma *of-int-less-of-int-power-cancel-iff*[simp]: $(of_int\ b) \wedge w < of_int\ x \longleftrightarrow b \wedge w < x$
 $\langle proof \rangle$

lemma *of-int-power-less-of-int-cancel-iff*[simp]: $of_int\ x < (of_int\ b) \wedge w \longleftrightarrow x < b \wedge w$
 $\langle proof \rangle$

lemma *of-int-max*: $of_int\ (max\ x\ y) = max\ (of_int\ x)\ (of_int\ y)$
 $\langle proof \rangle$

lemma *of-int-min*: $of_int\ (min\ x\ y) = min\ (of_int\ x)\ (of_int\ y)$
 $\langle proof \rangle$

end

context *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 *of-int-eq-numeral-iff* [iff]: $of_int\ z = (numeral\ n :: 'a::ring-char-0) \longleftrightarrow z = numeral\ n$
 $\langle proof \rangle$

lemma *of-int-le-numeral-iff* [simp]:
 $of_int\ z \leq (numeral\ n :: 'a::linordered-idom) \longleftrightarrow z \leq numeral\ n$
 $\langle proof \rangle$

lemma *of-int-numeral-le-iff* [simp]:
 $(numeral\ n :: 'a::linordered-idom) \leq of_int\ z \longleftrightarrow numeral\ n \leq z$
 $\langle proof \rangle$

lemma *of-int-less-numeral-iff* [simp]:

$of_int\ z < (numeral\ n :: 'a::linordered-idom) \longleftrightarrow z < numeral\ n$
 $\langle proof \rangle$

lemma *of-int-numeral-less-iff* [simp]:
 $(numeral\ n :: 'a::linordered-idom) < of_int\ z \longleftrightarrow numeral\ n < z$
 $\langle proof \rangle$

lemma *of-nat-less-of-int-iff*: $(of_nat\ n :: 'a::linordered-idom) < of_int\ x \longleftrightarrow int\ n$
 $< x$
 $\langle proof \rangle$

lemma *of-int-eq-id* [simp]: $of_int = id$
 $\langle proof \rangle$

instance $int :: no_top$
 $\langle proof \rangle$

instance $int :: no_bot$
 $\langle proof \rangle$

54.6 Magnitude of an Integer, as a Natural Number: *nat*

lift-definition $nat :: int \Rightarrow nat$ is $\lambda(x, y). x - y$
 $\langle proof \rangle$

lemma *nat-int* [simp]: $nat\ (int\ n) = n$
 $\langle proof \rangle$

lemma *int-nat-eq* [simp]: $int\ (nat\ z) = (if\ 0 \leq z\ then\ z\ else\ 0)$
 $\langle proof \rangle$

lemma *nat-0-le*: $0 \leq z \implies int\ (nat\ z) = z$
 $\langle proof \rangle$

lemma *nat-le-0* [simp]: $z \leq 0 \implies nat\ z = 0$
 $\langle proof \rangle$

lemma *nat-le-eq-zle*: $0 < w \vee 0 \leq z \implies nat\ w \leq nat\ z \longleftrightarrow w \leq z$
 $\langle proof \rangle$

An alternative condition is $0 \leq w$.

lemma *nat-mono-iff*: $0 < z \implies nat\ w < nat\ z \longleftrightarrow w < z$
 $\langle proof \rangle$

lemma *nat-less-eq-zless*: $0 \leq w \implies nat\ w < nat\ z \longleftrightarrow w < z$
 $\langle proof \rangle$

lemma *zless-nat-conj* [simp]: $nat\ w < nat\ z \longleftrightarrow 0 < z \wedge w < z$
 $\langle proof \rangle$

lemma *nonneg-int-cases*:

assumes $0 \leq k$

obtains n where $k = \text{int } n$

$\langle \text{proof} \rangle$

lemma *pos-int-cases*:

assumes $0 < k$

obtains n where $k = \text{int } n$ and $n > 0$

$\langle \text{proof} \rangle$

lemma *nonpos-int-cases*:

assumes $k \leq 0$

obtains n where $k = - \text{int } n$

$\langle \text{proof} \rangle$

lemma *neg-int-cases*:

assumes $k < 0$

obtains n where $k = - \text{int } n$ and $n > 0$

$\langle \text{proof} \rangle$

lemma *nat-eq-iff*: $\text{nat } w = m \longleftrightarrow (\text{if } 0 \leq w \text{ then } w = \text{int } m \text{ else } m = 0)$

$\langle \text{proof} \rangle$

lemma *nat-eq-iff2*: $m = \text{nat } w \longleftrightarrow (\text{if } 0 \leq w \text{ then } w = \text{int } m \text{ else } m = 0)$

$\langle \text{proof} \rangle$

lemma *nat-0* [simp]: $\text{nat } 0 = 0$

$\langle \text{proof} \rangle$

lemma *nat-1* [simp]: $\text{nat } 1 = \text{Suc } 0$

$\langle \text{proof} \rangle$

lemma *nat-numeral* [simp]: $\text{nat } (\text{numeral } k) = \text{numeral } k$

$\langle \text{proof} \rangle$

lemma *nat-neg-numeral* [simp]: $\text{nat } (- \text{numeral } k) = 0$

$\langle \text{proof} \rangle$

lemma *nat-2*: $\text{nat } 2 = \text{Suc } (\text{Suc } 0)$

$\langle \text{proof} \rangle$

lemma *nat-less-iff*: $0 \leq w \implies \text{nat } w < m \longleftrightarrow w < \text{of-nat } m$

$\langle \text{proof} \rangle$

lemma *nat-le-iff*: $\text{nat } x \leq n \longleftrightarrow x \leq \text{int } n$

$\langle \text{proof} \rangle$

lemma *nat-mono*: $x \leq y \implies \text{nat } x \leq \text{nat } y$

$\langle proof \rangle$

lemma *nat-0-iff* [*simp*]: $nat\ i = 0 \longleftrightarrow i \leq 0$
for $i :: int$
 $\langle proof \rangle$

lemma *int-eq-iff*: $of\ nat\ m = z \longleftrightarrow m = nat\ z \wedge 0 \leq z$
 $\langle proof \rangle$

lemma *zero-less-nat-eq* [*simp*]: $0 < nat\ z \longleftrightarrow 0 < z$
 $\langle proof \rangle$

lemma *nat-add-distrib*: $0 \leq z \implies 0 \leq z' \implies nat\ (z + z') = nat\ z + nat\ z'$
 $\langle proof \rangle$

lemma *nat-diff-distrib'*: $0 \leq x \implies 0 \leq y \implies nat\ (x - y) = nat\ x - nat\ y$
 $\langle proof \rangle$

lemma *nat-diff-distrib*: $0 \leq z' \implies z' \leq z \implies nat\ (z - z') = nat\ z - nat\ z'$
 $\langle proof \rangle$

lemma *nat-zminus-int* [*simp*]: $nat\ (-\ int\ n) = 0$
 $\langle proof \rangle$

lemma *le-nat-iff*: $k \geq 0 \implies n \leq nat\ k \longleftrightarrow int\ n \leq k$
 $\langle proof \rangle$

lemma *zless-nat-eq-int-zless*: $m < nat\ z \longleftrightarrow int\ m < z$
 $\langle proof \rangle$

lemma (*in ring-1*) *of-nat-nat* [*simp*]: $0 \leq z \implies of\ nat\ (nat\ z) = of\ int\ z$
 $\langle proof \rangle$

lemma *diff-nat-numeral* [*simp*]: $(numeral\ v :: nat) - numeral\ v' = nat\ (numeral\ v - numeral\ v')$
 $\langle proof \rangle$

lemma *nat-abs-triangle-ineq*:
 $nat\ |k + l| \leq nat\ |k| + nat\ |l|$
 $\langle proof \rangle$

lemma *nat-of-bool* [*simp*]:
 $nat\ (of\ bool\ P) = of\ bool\ P$
 $\langle proof \rangle$

lemma *split-nat* [*linarith-split*]: $P\ (nat\ i) \longleftrightarrow ((\forall n. i = int\ n \longrightarrow P\ n) \wedge (i < 0 \longrightarrow P\ 0))$
 $(is\ ?P = (?L \wedge ?R))$
for $i :: int$

$\langle \text{proof} \rangle$

lemma *all-nat*: $(\forall x. P\ x) \longleftrightarrow (\forall x \geq 0. P\ (\text{nat } x))$
 $\langle \text{proof} \rangle$

lemma *ex-nat*: $(\exists x. P\ x) \longleftrightarrow (\exists x. 0 \leq x \wedge P\ (\text{nat } x))$
 $\langle \text{proof} \rangle$

For termination proofs:

lemma *measure-function-int*[*measure-function*]: *is-measure* (*nat* \circ *abs*) $\langle \text{proof} \rangle$

54.7 Lemmas about the Function *of-nat* and Orderings

lemma *negative-zless-0*: $-(\text{int } (\text{Suc } n)) < (0 :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *negative-zless [iff]*: $-(\text{int } (\text{Suc } n)) < \text{int } m$
 $\langle \text{proof} \rangle$

lemma *negative-zle-0*: $-\text{int } n \leq 0$
 $\langle \text{proof} \rangle$

lemma *negative-zle [iff]*: $-\text{int } n \leq \text{int } m$
 $\langle \text{proof} \rangle$

lemma *not-zle-0-negative [simp]*: $\neg 0 \leq -\text{int } (\text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *int-zle-neg*: $\text{int } n \leq -\text{int } m \longleftrightarrow n = 0 \wedge m = 0$
 $\langle \text{proof} \rangle$

lemma *not-int-zless-negative [simp]*: $\neg \text{int } n < -\text{int } m$
 $\langle \text{proof} \rangle$

lemma *negative-eq-positive [simp]*: $-\text{int } n = \text{of-nat } m \longleftrightarrow n = 0 \wedge m = 0$
 $\langle \text{proof} \rangle$

lemma *zle-iff-zadd*: $w \leq z \longleftrightarrow (\exists n. z = w + \text{int } n)$
 (*is ?lhs* \longleftrightarrow *?rhs*)
 $\langle \text{proof} \rangle$

lemma *zadd-int-left*: $\text{int } m + (\text{int } n + z) = \text{int } (m + n) + z$
 $\langle \text{proof} \rangle$

lemma *negD*:
 assumes $x < 0$ shows $\exists n. x = -(\text{int } (\text{Suc } n))$
 $\langle \text{proof} \rangle$

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*]:
 $(\bigwedge n. z = \text{int } n \implies P) \implies (\bigwedge n. z = -(\text{int } n) \implies P) \implies P$
 ⟨*proof*⟩

This is the default, with a negative case.

lemma *int-cases* [*case-names nonneg neg, cases type: int*]:
 assumes *pos*: $\bigwedge n. z = \text{int } n \implies P$ and *neg*: $\bigwedge n. z = -(\text{int } (\text{Suc } n)) \implies P$
 shows *P*
 ⟨*proof*⟩

lemma *int-cases3* [*case-names zero pos neg*]:
 fixes *k* :: *int*
 assumes $k = 0 \implies P$ and $\bigwedge n. k = \text{int } n \implies n > 0 \implies P$
 and $\bigwedge n. k = -\text{int } n \implies n > 0 \implies P$
 shows *P*
 ⟨*proof*⟩

lemma *int-of-nat-induct* [*case-names nonneg neg, induct type: int*]:
 $(\bigwedge n. P (\text{int } n)) \implies (\bigwedge n. P (- (\text{int } (\text{Suc } n)))) \implies P z$
 ⟨*proof*⟩

lemma *sgn-mult-dvd-iff* [*simp*]:
 $\text{sgn } r * l \text{ dvd } k \longleftrightarrow l \text{ dvd } k \wedge (r = 0 \longrightarrow k = 0)$ for *k l r* :: *int*
 ⟨*proof*⟩

lemma *mult-sgn-dvd-iff* [*simp*]:
 $l * \text{sgn } r \text{ dvd } k \longleftrightarrow l \text{ dvd } k \wedge (r = 0 \longrightarrow k = 0)$ for *k l r* :: *int*
 ⟨*proof*⟩

lemma *dvd-sgn-mult-iff* [*simp*]:
 $l \text{ dvd } \text{sgn } r * k \longleftrightarrow l \text{ dvd } k \vee r = 0$ for *k l r* :: *int*
 ⟨*proof*⟩

lemma *dvd-mult-sgn-iff* [*simp*]:
 $l \text{ dvd } k * \text{sgn } r \longleftrightarrow l \text{ dvd } k \vee r = 0$ for *k l r* :: *int*
 ⟨*proof*⟩

lemma *int-sgnE*:
 fixes *k* :: *int*
 obtains *n* and *l* where $k = \text{sgn } l * \text{int } n$
 ⟨*proof*⟩

54.8.1 Binary comparisons

Preliminaries

lemma *le-imp-0-less*:
 fixes $z :: \text{int}$
 assumes $le: 0 \leq z$
 shows $0 < 1 + z$
 $\langle \text{proof} \rangle$

lemma *odd-less-0-iff*: $1 + z + z < 0 \longleftrightarrow z < 0$
 for $z :: \text{int}$
 $\langle \text{proof} \rangle$

54.8.2 Comparisons, for Ordered Rings

lemma *odd-nonzero*: $1 + z + z \neq 0$
 for $z :: \text{int}$
 $\langle \text{proof} \rangle$

54.9 The Set of Integers

context *ring-1*
begin

definition *Ints* :: 'a set ($\langle \mathbb{Z} \rangle$)
 where $\mathbb{Z} = \text{range of-int}$

lemma *Ints-of-int [simp]*: $\text{of-int } z \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-of-nat [simp]*: $\text{of-nat } n \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-0 [simp]*: $0 \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-1 [simp]*: $1 \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-numeral [simp]*: $\text{numeral } n \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-add [simp]*: $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a + b \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-minus [simp]*: $a \in \mathbb{Z} \implies -a \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *minus-in-Ints-iff [simp]*: $-x \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$

$\langle \text{proof} \rangle$

lemma *Ints-diff* [*simp*]: $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a - b \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-mult* [*simp*]: $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a * b \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-power* [*simp*]: $a \in \mathbb{Z} \implies a \wedge n \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-cases* [*cases set: Ints*]:
 assumes $q \in \mathbb{Z}$
 obtains (*of-int*) z **where** $q = \text{of-int } z$
 $\langle \text{proof} \rangle$

lemma *Ints-induct* [*case-names of-int, induct set: Ints*]:
 $q \in \mathbb{Z} \implies (\bigwedge z. P (\text{of-int } z)) \implies P q$
 $\langle \text{proof} \rangle$

lemma *Nats-subset-Ints*: $\mathbb{N} \subseteq \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Nats-altdef1*: $\mathbb{N} = \{\text{of-int } n \mid n. n \geq 0\}$
 $\langle \text{proof} \rangle$

end

lemma *Ints-sum* [*intro*]: $(\bigwedge x. x \in A \implies f x \in \mathbb{Z}) \implies \text{sum } f A \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-prod* [*intro*]: $(\bigwedge x. x \in A \implies f x \in \mathbb{Z}) \implies \text{prod } f A \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma (*in linordered-idom*) *Ints-abs* [*simp*]:
 shows $a \in \mathbb{Z} \implies \text{abs } a \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma (*in linordered-idom*) *Nats-altdef2*: $\mathbb{N} = \{n \in \mathbb{Z}. n \geq 0\}$
 $\langle \text{proof} \rangle$

lemma (*in idom-divide*) *of-int-divide-in-Ints*:
 $\text{of-int } a \text{ div } \text{of-int } b \in \mathbb{Z}$ **if** $b \text{ dvd } a$
 $\langle \text{proof} \rangle$

The premise involving \mathbb{Z} prevents $a = 1 / (2::'a)$.

lemma *Ints-double-eq-0-iff*:
 fixes $a :: 'a::\text{ring-char-0}$
 assumes *in-Ints*: $a \in \mathbb{Z}$

shows $a + a = 0 \longleftrightarrow a = 0$
 (**is** $?lhs \longleftrightarrow ?rhs$)
 $\langle proof \rangle$

lemma *Ints-odd-nonzero*:
fixes $a :: 'a::ring-char-0$
assumes $in-Ints: a \in \mathbb{Z}$
shows $1 + a + a \neq 0$
 $\langle proof \rangle$

lemma *Nats-numeral [simp]*: $numeral\ w \in \mathbb{N}$
 $\langle proof \rangle$

lemma *Ints-odd-less-0*:
fixes $a :: 'a::linordered-idom$
assumes $in-Ints: a \in \mathbb{Z}$
shows $1 + a + a < 0 \longleftrightarrow a < 0$
 $\langle proof \rangle$

lemma *add-in-Ints-iff-left [simp]*: $x \in \mathbb{Z} \implies x + y \in \mathbb{Z} \longleftrightarrow y \in \mathbb{Z}$
 $\langle proof \rangle$

lemma *add-in-Ints-iff-right [simp]*: $y \in \mathbb{Z} \implies x + y \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$
 $\langle proof \rangle$

lemma *diff-in-Ints-iff-left [simp]*: $x \in \mathbb{Z} \implies x - y \in \mathbb{Z} \longleftrightarrow y \in \mathbb{Z}$
 $\langle proof \rangle$

lemma *diff-in-Ints-iff-right [simp]*: $y \in \mathbb{Z} \implies x - y \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$
 $\langle proof \rangle$

lemmas $[simp] = minus-in-Ints-iff$

lemma *fraction-not-in-Ints*:
assumes $\neg(n\ dvd\ m)\ n \neq 0$
shows $of-int\ m\ /\ of-int\ n \notin (\mathbb{Z} :: 'a :: \{division-ring, ring-char-0\}\ set)$
 $\langle proof \rangle$

lemma *of-int-div-of-int-in-Ints-iff*:
 $(of-int\ n\ /\ of-int\ m :: 'a :: \{division-ring, ring-char-0\}) \in \mathbb{Z} \longleftrightarrow m = 0 \vee m\ dvd\ n$
 $\langle proof \rangle$

lemma *fraction-numeral-not-in-Ints [simp]*:
assumes $\neg(numeral\ b :: int)\ dvd\ numeral\ a$
shows $numeral\ a\ /\ numeral\ b \notin (\mathbb{Z} :: 'a :: \{division-ring, ring-char-0\}\ set)$
 $\langle proof \rangle$

54.10 *sum and prod***context** *semiring-1***begin****lemma** *of-nat-sum* [*simp*]:
$$\text{of-nat } (\text{sum } f \ A) = (\sum_{x \in A}. \text{of-nat } (f \ x))$$

<proof>

end**context** *ring-1***begin****lemma** *of-int-sum* [*simp*]:
$$\text{of-int } (\text{sum } f \ A) = (\sum_{x \in A}. \text{of-int } (f \ x))$$

<proof>

end**context** *comm-semiring-1***begin****lemma** *of-nat-prod* [*simp*]:
$$\text{of-nat } (\text{prod } f \ A) = (\prod_{x \in A}. \text{of-nat } (f \ x))$$

<proof>

end**context** *comm-ring-1***begin****lemma** *of-int-prod* [*simp*]:
$$\text{of-int } (\text{prod } f \ A) = (\prod_{x \in A}. \text{of-int } (f \ x))$$

<proof>

end**54.11** *Setting up simplification procedures**<ML>***54.12** *More Inequality Reasoning***lemma** *zless-add1-eq*: $w < z + 1 \longleftrightarrow w < z \vee w = z$ **for** $w \ z :: \text{int}$ *<proof>***lemma** *add1-zle-eq*: $w + 1 \leq z \longleftrightarrow w < z$ **for** $w \ z :: \text{int}$

$\langle proof \rangle$

lemma *zle-diff1-eq* [simp]: $w \leq z - 1 \longleftrightarrow w < z$
for $w\ z :: int$
 $\langle proof \rangle$

lemma *zle-add1-eq-le* [simp]: $w < z + 1 \longleftrightarrow w \leq z$
for $w\ z :: int$
 $\langle proof \rangle$

lemma *int-one-le-iff-zero-less*: $1 \leq z \longleftrightarrow 0 < z$
for $z :: int$
 $\langle proof \rangle$

lemma *Ints-nonzero-abs-ge1*:
fixes $x :: 'a :: linordered-idom$
assumes $x \in Ints\ x \neq 0$
shows $1 \leq abs\ x$
 $\langle proof \rangle$

lemma *Ints-nonzero-abs-less1*:
fixes $x :: 'a :: linordered-idom$
shows $\llbracket x \in Ints; abs\ x < 1 \rrbracket \implies x = 0$
 $\langle proof \rangle$

lemma *Ints-eq-abs-less1*:
fixes $x :: 'a :: linordered-idom$
shows $\llbracket x \in Ints; y \in Ints \rrbracket \implies x = y \longleftrightarrow abs\ (x - y) < 1$
 $\langle proof \rangle$

54.13 The functions *nat* and *int*

Simplify the term $w + -\ z$.

lemma *one-less-nat-eq* [simp]: $Suc\ 0 < nat\ z \longleftrightarrow 1 < z$
 $\langle proof \rangle$

lemma *int-eq-iff-numeral* [simp]:
 $int\ m = numeral\ v \longleftrightarrow m = numeral\ v$
 $\langle proof \rangle$

lemma *nat-abs-int-diff*:
 $nat\ |int\ a - int\ b| = (if\ a \leq b\ then\ b - a\ else\ a - b)$
 $\langle proof \rangle$

lemma *nat-int-add*: $nat\ (int\ a + int\ b) = a + b$
 $\langle proof \rangle$

context *ring-1*
begin

lemma *of-int-of-nat* [*nitpick-simp*]:

of-int $k = (\text{if } k < 0 \text{ then } - \text{of-nat } (\text{nat } (- k)) \text{ else } \text{of-nat } (\text{nat } k))$
 $\langle \text{proof} \rangle$

end

lemma *transfer-rule-of-int*:

includes *lifting-syntax*
fixes $R :: 'a::\text{ring-1} \Rightarrow 'b::\text{ring-1} \Rightarrow \text{bool}$
assumes [*transfer-rule*]: $R \ 0 \ 0 \ R \ 1 \ 1$
 $(R \implies R \implies R) \ (+) \ (+)$
 $(R \implies R) \ \text{uminus} \ \text{uminus}$
shows $((=) \implies R) \ \text{of-int} \ \text{of-int}$
 $\langle \text{proof} \rangle$

lemma *nat-mult-distrib*:

fixes $z \ z' :: \text{int}$
assumes $0 \leq z$
shows $\text{nat } (z * z') = \text{nat } z * \text{nat } z'$
 $\langle \text{proof} \rangle$

lemma *nat-mult-distrib-neg*:

assumes $z \leq (0::\text{int})$ **shows** $\text{nat } (z * z') = \text{nat } (-z) * \text{nat } (-z')$ (**is** $?L = ?R$)
 $\langle \text{proof} \rangle$

lemma *nat-abs-mult-distrib*: $\text{nat } |w * z| = \text{nat } |w| * \text{nat } |z|$
 $\langle \text{proof} \rangle$

lemma *int-in-range-abs* [*simp*]: $\text{int } n \in \text{range } \text{abs}$
 $\langle \text{proof} \rangle$

lemma *range-abs-Nats* [*simp*]: $\text{range } \text{abs} = (\mathbb{N} :: \text{int set})$
 $\langle \text{proof} \rangle$

lemma *Suc-nat-eq-nat-zadd1*: $0 \leq z \implies \text{Suc } (\text{nat } z) = \text{nat } (1 + z)$
for $z :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *diff-nat-eq-if*:

$\text{nat } z - \text{nat } z' =$
 $(\text{if } z' < 0 \text{ then } \text{nat } z$
 else
 $\text{let } d = z - z'$
 $\text{in if } d < 0 \text{ then } 0 \text{ else } \text{nat } d)$
 $\langle \text{proof} \rangle$

lemma *nat-numeral-diff-1* [*simp*]: $\text{numeral } v - (1::\text{nat}) = \text{nat } (\text{numeral } v - 1)$
 $\langle \text{proof} \rangle$

54.14 Induction principles for int

Well-founded segments of the integers.

definition *int-ge-less-than* :: *int* \Rightarrow (*int* \times *int*) *set*
where *int-ge-less-than* *d* = {(*z'*, *z*). *d* \leq *z'* \wedge *z'* < *z*}

lemma *wf-int-ge-less-than*: *wf* (*int-ge-less-than* *d*)
 <proof>

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

definition *int-ge-less-than2* :: *int* \Rightarrow (*int* \times *int*) *set*
where *int-ge-less-than2* *d* = {(*z'*, *z*). *d* \leq *z* \wedge *z'* < *z*}

lemma *wf-int-ge-less-than2*: *wf* (*int-ge-less-than2* *d*)
 <proof>

theorem *int-ge-induct* [*case-names* *base step*, *induct set*: *int*]:
fixes *i* :: *int*
assumes *ge*: *k* \leq *i*
and *base*: *P* *k*
and *step*: $\bigwedge i. k \leq i \Rightarrow P\ i \Rightarrow P\ (i + 1)$
shows *P* *i*
 <proof>

theorem *int-gr-induct* [*case-names* *base step*, *induct set*: *int*]:
fixes *i k* :: *int*
assumes *k* < *i* *P* (*k* + 1) $\bigwedge i. k < i \Rightarrow P\ i \Rightarrow P\ (i + 1)$
shows *P* *i*
 <proof>

theorem *int-le-induct* [*consumes 1*, *case-names* *base step*]:
fixes *i k* :: *int*
assumes *le*: *i* \leq *k*
and *base*: *P* *k*
and *step*: $\bigwedge i. i \leq k \Rightarrow P\ i \Rightarrow P\ (i - 1)$
shows *P* *i*
 <proof>

theorem *int-less-induct* [*consumes 1*, *case-names* *base step*]:
fixes *i k* :: *int*
assumes *i* < *k* *P* (*k* - 1) $\bigwedge i. i < k \Rightarrow P\ i \Rightarrow P\ (i - 1)$
shows *P* *i*
 <proof>

theorem *int-induct* [*case-names* *base step1 step2*]:
fixes *k* :: *int*

assumes *base*: $P\ k$
and *step1*: $\bigwedge i. k \leq i \implies P\ i \implies P\ (i + 1)$
and *step2*: $\bigwedge i. k \geq i \implies P\ i \implies P\ (i - 1)$
shows $P\ i$
 $\langle \text{proof} \rangle$

54.15 Intermediate value theorems

lemma *nat-ivt-aux*:
 $\llbracket \forall i < n. |f\ (Suc\ i) - f\ i| \leq 1; f\ 0 \leq k; k \leq f\ n \rrbracket \implies \exists i \leq n. f\ i = k$
for $m\ n :: nat$ **and** $k :: int$
 $\langle \text{proof} \rangle$

lemma *nat-intermed-int-val*:
fixes $m\ n :: nat$ **and** $k :: int$
assumes $\forall i. m \leq i \wedge i < n \longrightarrow |f\ (Suc\ i) - f\ i| \leq 1\ m \leq n\ f\ m \leq k\ k \leq f\ n$
shows $\exists i. m \leq i \wedge i \leq n \wedge f\ i = k$
 $\langle \text{proof} \rangle$

lemma *nat0-intermed-int-val*:
 $\exists i \leq n. f\ i = k$
if $\forall i < n. |f\ (i + 1) - f\ i| \leq 1\ f\ 0 \leq k\ k \leq f\ n$
for $n :: nat$ **and** $k :: int$
 $\langle \text{proof} \rangle$

54.16 Products and 1, by T. M. Rasmussen

lemma *abs-zmult-eq-1*:
fixes $m\ n :: int$
assumes $mn: |m * n| = 1$
shows $|m| = 1$
 $\langle \text{proof} \rangle$

lemma *pos-zmult-eq-1-iff-lemma*: $m * n = 1 \implies m = 1 \vee m = -1$
for $m\ n :: int$
 $\langle \text{proof} \rangle$

lemma *pos-zmult-eq-1-iff*:
fixes $m\ n :: int$
assumes $0 < m$
shows $m * n = 1 \longleftrightarrow m = 1 \wedge n = 1$
 $\langle \text{proof} \rangle$

lemma *zmult-eq-1-iff*: $m * n = 1 \longleftrightarrow (m = 1 \wedge n = 1) \vee (m = -1 \wedge n = -1)$ (is ?L = ?R)
for $m\ n :: int$
 $\langle \text{proof} \rangle$

lemma *zmult-eq-neg1-iff*: $a * b = (-1 :: int) \longleftrightarrow a = 1 \wedge b = -1 \vee a = -1 \wedge b = 1$

$\langle proof \rangle$

lemma *infinite-UNIV-int* [simp]: $\neg finite (UNIV::int\ set)$
 $\langle proof \rangle$

54.17 The divides relation

lemma *zdvd-antisym-nonneg*: $0 \leq m \implies 0 \leq n \implies m\ dvd\ n \implies n\ dvd\ m \implies m = n$
for $m\ n :: int$
 $\langle proof \rangle$

lemma *zdvd-antisym-abs*:
fixes $a\ b :: int$
assumes $a\ dvd\ b$ **and** $b\ dvd\ a$
shows $|a| = |b|$
 $\langle proof \rangle$

lemma *zdvd-zdiffD*: $k\ dvd\ m - n \implies k\ dvd\ n \implies k\ dvd\ m$
for $k\ m\ n :: int$
 $\langle proof \rangle$

lemma *zdvd-reduce*: $k\ dvd\ n + k * m \longleftrightarrow k\ dvd\ n$
for $k\ m\ n :: int$
 $\langle proof \rangle$

lemma *dvd-imp-le-int*:
fixes $d\ i :: int$
assumes $i \neq 0$ **and** $d\ dvd\ i$
shows $|d| \leq |i|$
 $\langle proof \rangle$

lemma *zdvd-not-zless*:
fixes $m\ n :: int$
assumes $0 < m$ **and** $m < n$
shows $\neg n\ dvd\ m$
 $\langle proof \rangle$

lemma *zdvd-mult-cancel*:
fixes $k\ m\ n :: int$
assumes $d: k * m\ dvd\ k * n$
and $k \neq 0$
shows $m\ dvd\ n$
 $\langle proof \rangle$

lemma *int-dvd-int-iff* [simp]:
 $int\ m\ dvd\ int\ n \longleftrightarrow m\ dvd\ n$
 $\langle proof \rangle$

lemma *dvd-nat-abs-iff* [simp]:
 $n \text{ dvd nat } |k| \longleftrightarrow \text{int } n \text{ dvd } k$
 ⟨proof⟩

lemma *nat-abs-dvd-iff* [simp]:
 $\text{nat } |k| \text{ dvd } n \longleftrightarrow k \text{ dvd int } n$
 ⟨proof⟩

lemma *zdvd1-eq* [simp]: $x \text{ dvd } 1 \longleftrightarrow |x| = 1$ (is ?lhs \longleftrightarrow ?rhs)
 for $x :: \text{int}$
 ⟨proof⟩

lemma *zdvd-mult-cancel1*:
 fixes $m :: \text{int}$
 assumes $mp: m \neq 0$
 shows $m * n \text{ dvd } m \longleftrightarrow |n| = 1$
 (is ?lhs \longleftrightarrow ?rhs)
 ⟨proof⟩

lemma *nat-dvd-iff*: $\text{nat } z \text{ dvd } m \longleftrightarrow (\text{if } 0 \leq z \text{ then } z \text{ dvd int } m \text{ else } m = 0)$
 ⟨proof⟩

lemma *eq-nat-nat-iff*: $0 \leq z \implies 0 \leq z' \implies \text{nat } z = \text{nat } z' \longleftrightarrow z = z'$
 ⟨proof⟩

lemma *nat-power-eq*: $0 \leq z \implies \text{nat } (z \wedge n) = \text{nat } z \wedge n$
 ⟨proof⟩

lemma *numeral-power-eq-nat-cancel-iff* [simp]:
 $\text{numeral } x \wedge n = \text{nat } y \longleftrightarrow \text{numeral } x \wedge n = y$
 ⟨proof⟩

lemma *nat-eq-numeral-power-cancel-iff* [simp]:
 $\text{nat } y = \text{numeral } x \wedge n \longleftrightarrow y = \text{numeral } x \wedge n$
 ⟨proof⟩

lemma *numeral-power-le-nat-cancel-iff* [simp]:
 $\text{numeral } x \wedge n \leq \text{nat } a \longleftrightarrow \text{numeral } x \wedge n \leq a$
 ⟨proof⟩

lemma *nat-le-numeral-power-cancel-iff* [simp]:
 $\text{nat } a \leq \text{numeral } x \wedge n \longleftrightarrow a \leq \text{numeral } x \wedge n$
 ⟨proof⟩

lemma *numeral-power-less-nat-cancel-iff* [simp]:
 $\text{numeral } x \wedge n < \text{nat } a \longleftrightarrow \text{numeral } x \wedge n < a$
 ⟨proof⟩

lemma *nat-less-numeral-power-cancel-iff* [simp]:

$\text{nat } a < \text{numeral } x \wedge n \longleftrightarrow a < \text{numeral } x \wedge n$
 $\langle \text{proof} \rangle$

lemma *zdvd-imp-le*: $z \leq n$ **if** $z \text{ dvd } n$ $0 < n$ **for** $n z :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *zdvd-period*:
fixes $a d :: \text{int}$
assumes $a \text{ dvd } d$
shows $a \text{ dvd } (x + t) \longleftrightarrow a \text{ dvd } ((x + c * d) + t)$
(is ?lhs \longleftrightarrow ?rhs)
 $\langle \text{proof} \rangle$

lemma *fraction-numeral-not-in-Ints'* [simp]:
assumes $b \neq \text{Num.One}$
shows $1 / \text{numeral } b \notin (\mathbb{Z} :: 'a :: \{\text{division-ring}, \text{ring-char-0}\} \text{ set})$
 $\langle \text{proof} \rangle$

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 exponentiation.

definition *power-int* :: $'a :: \{\text{inverse}, \text{power}\} \Rightarrow \text{int} \Rightarrow 'a$ (**infixr** $\langle \text{powi} \rangle$ 80) **where**
 $\text{power-int } x \ n = (\text{if } n \geq 0 \text{ then } x \wedge \text{nat } n \text{ else inverse } x \wedge (\text{nat } (-n)))$

lemma *power-int-0-right* [simp]: $\text{power-int } x \ 0 = 1$
and *power-int-1-right* [simp]:
 $\text{power-int } (y :: 'a :: \{\text{power}, \text{inverse}, \text{monoid-mult}\}) \ 1 = y$
and *power-int-minus1-right* [simp]:
 $\text{power-int } (y :: 'a :: \{\text{power}, \text{inverse}, \text{monoid-mult}\}) \ (-1) = \text{inverse } y$
 $\langle \text{proof} \rangle$

lemma *power-int-of-nat* [simp]: $\text{power-int } x \ (\text{int } n) = x \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-int-numeral* [simp]: $\text{power-int } x \ (\text{numeral } n) = x \wedge \text{numeral } n$
 $\langle \text{proof} \rangle$

lemma *powi-numeral-reduce*: $x \text{ powi } \text{numeral } n = x * x \text{ powi } \text{int } (\text{pred-numeral } n)$
 $\langle \text{proof} \rangle$

lemma *powi-minus-numeral-reduce*: $x \text{ powi } - (\text{numeral } n) = \text{inverse } x * x \text{ powi } - \text{int } (\text{pred-numeral } n)$
 $\langle \text{proof} \rangle$

lemma *int-cases4* [case-names nonneg neg]:
 fixes $m :: \text{int}$
 obtains n where $m = \text{int } n \mid n$ where $n > 0$ $m = -\text{int } n$
 $\langle \text{proof} \rangle$

context
 assumes *SORT-CONSTRAINT*('a::division-ring)
begin

lemma *power-int-minus*: $\text{power-int } (x :: 'a) (-n) = \text{inverse } (\text{power-int } x \ n)$
 $\langle \text{proof} \rangle$

lemma *power-int-minus-divide*: $\text{power-int } (x :: 'a) (-n) = 1 / (\text{power-int } x \ n)$
 $\langle \text{proof} \rangle$

lemma *power-int-eq-0-iff* [simp]: $\text{power-int } (x :: 'a) \ n = 0 \iff x = 0 \wedge n \neq 0$
 $\langle \text{proof} \rangle$

lemma *power-int-0-left-if*: $\text{power-int } (0 :: 'a) \ m = (\text{if } m = 0 \text{ then } 1 \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *power-int-0-left* [simp]: $m \neq 0 \implies \text{power-int } (0 :: 'a) \ m = 0$
 $\langle \text{proof} \rangle$

lemma *power-int-1-left* [simp]: $\text{power-int } 1 \ n = (1 :: 'a :: \text{division-ring})$
 $\langle \text{proof} \rangle$

lemma *power-diff-conv-inverse*: $x \neq 0 \implies m \leq n \implies (x :: 'a) ^ (n - m) = x ^ n * \text{inverse } x ^ m$
 $\langle \text{proof} \rangle$

lemma *power-mult-inverse-distrib*: $x ^ m * \text{inverse } (x :: 'a) = \text{inverse } x * x ^ m$
 $\langle \text{proof} \rangle$

lemma *power-mult-power-inverse-commute*:
 $x ^ m * \text{inverse } (x :: 'a) ^ n = \text{inverse } x ^ n * x ^ m$
 $\langle \text{proof} \rangle$

lemma *power-int-add*:
 assumes $x \neq 0 \vee m + n \neq 0$
 shows $\text{power-int } (x :: 'a) \ (m + n) = \text{power-int } x \ m * \text{power-int } x \ n$
 $\langle \text{proof} \rangle$

lemma *power-int-add-1*:
 assumes $x \neq 0 \vee m \neq -1$
 shows $\text{power-int } (x :: 'a) \ (m + 1) = \text{power-int } x \ m * x$
 $\langle \text{proof} \rangle$

lemma *power-int-add-1'*:
assumes $x \neq 0 \vee m \neq -1$
shows $\text{power-int } (x :: 'a) (m + 1) = x * \text{power-int } x m$
 $\langle \text{proof} \rangle$

lemma *power-int-commutes*: $\text{power-int } (x :: 'a) n * x = x * \text{power-int } x n$
 $\langle \text{proof} \rangle$

lemma *power-int-inverse* [*field-simps*, *field-split-simps*, *divide-simps*]:
 $\text{power-int } (\text{inverse } (x :: 'a)) n = \text{inverse } (\text{power-int } x n)$
 $\langle \text{proof} \rangle$

lemma *power-int-mult*: $\text{power-int } (x :: 'a) (m * n) = \text{power-int } (\text{power-int } x m) n$
 $\langle \text{proof} \rangle$

lemma *power-int-power*: $(a \wedge b :: 'a :: \text{division-ring}) \text{ powi } c = a \text{ powi } (\text{int } b * c)$
 $\langle \text{proof} \rangle$

lemma *power-int-power'*: $(a \text{ powi } b :: 'a :: \text{division-ring}) \wedge c = a \text{ powi } (b * \text{int } c)$
 $\langle \text{proof} \rangle$

lemma *power-int-nonneg-exp*: $n \geq 0 \implies x \text{ powi } n = x \wedge \text{nat } n$
 $\langle \text{proof} \rangle$

end

context
assumes *SORT-CONSTRAINT*('a::field)
begin

lemma *power-int-diff*:
assumes $x \neq 0 \vee m \neq n$
shows $\text{power-int } (x :: 'a) (m - n) = \text{power-int } x m / \text{power-int } x n$
 $\langle \text{proof} \rangle$

lemma *power-int-minus-mult*: $x \neq 0 \vee n \neq 0 \implies \text{power-int } (x :: 'a) (n - 1) * x = \text{power-int } x n$
 $\langle \text{proof} \rangle$

lemma *power-int-mult-distrib*: $\text{power-int } (x * y :: 'a) m = \text{power-int } x m * \text{power-int } y m$
 $\langle \text{proof} \rangle$

lemmas *power-int-mult-distrib-numeral1* = *power-int-mult-distrib* [**where** $x = \text{numeral } w$ **for** w , *simp*]
lemmas *power-int-mult-distrib-numeral2* = *power-int-mult-distrib* [**where** $y = \text{numeral } w$ **for** w , *simp*]

lemma *power-int-divide-distrib*: $\text{power-int } (x / y :: 'a) m = \text{power-int } x m /$

power-int *y m*
 ⟨*proof*⟩

end

lemma *power-int-add-numeral* [*simp*]:
 $\text{power-int } x \text{ (numeral } m) * \text{power-int } x \text{ (numeral } n) = \text{power-int } x \text{ (numeral } (m + n))$
for $x :: 'a :: \text{division-ring}$
 ⟨*proof*⟩

lemma *power-int-add-numeral2* [*simp*]:
 $\text{power-int } x \text{ (numeral } m) * (\text{power-int } x \text{ (numeral } n) * b) = \text{power-int } x \text{ (numeral } (m + n)) * b$
for $x :: 'a :: \text{division-ring}$
 ⟨*proof*⟩

lemma *power-int-mult-numeral* [*simp*]:
 $\text{power-int } (\text{power-int } x \text{ (numeral } m)) \text{ (numeral } n) = \text{power-int } x \text{ (numeral } (m * n))$
for $x :: 'a :: \text{division-ring}$
 ⟨*proof*⟩

lemma *power-int-not-zero*: $(x :: 'a :: \text{division-ring}) \neq 0 \vee n = 0 \implies \text{power-int } x \text{ } n \neq 0$
 ⟨*proof*⟩

lemma *power-int-one-over* [*field-simps*, *field-split-simps*, *divide-simps*]:
 $\text{power-int } (1 / x :: 'a :: \text{division-ring}) \text{ } n = 1 / \text{power-int } x \text{ } n$
 ⟨*proof*⟩

context

assumes *SORT-CONSTRAINT*('a :: *linordered-field*)

begin

lemma *power-int-numeral-neg-numeral* [*simp*]:
 $\text{power-int } (\text{numeral } m) \text{ } (-\text{numeral } n) = (\text{inverse } (\text{numeral } (\text{Num.pow } m \text{ } n))) :: 'a$
 ⟨*proof*⟩

lemma *zero-less-power-int* [*simp*]: $0 < (x :: 'a) \implies 0 < \text{power-int } x \text{ } n$
 ⟨*proof*⟩

lemma *zero-le-power-int* [*simp*]: $0 \leq (x :: 'a) \implies 0 \leq \text{power-int } x \text{ } n$
 ⟨*proof*⟩

lemma *power-int-mono*: $(x :: 'a) \leq y \implies n \geq 0 \implies 0 \leq x \implies \text{power-int } x \text{ } n \leq$

power-int $y\ n$
 ⟨proof⟩

lemma *one-le-power-int* [simp]: $1 \leq (x :: 'a) \implies n \geq 0 \implies 1 \leq \text{power-int } x\ n$
 ⟨proof⟩

lemma *power-int-le-one*: $0 \leq (x :: 'a) \implies n \geq 0 \implies x \leq 1 \implies \text{power-int } x\ n \leq 1$
 ⟨proof⟩

lemma *power-int-le-imp-le-exp*:
 assumes *gt1*: $1 < (x :: 'a :: \text{linordered-field})$
 assumes *power-int* $x\ m \leq \text{power-int } x\ n\ n \geq 0$
 shows $m \leq n$
 ⟨proof⟩

lemma *power-int-le-imp-less-exp*:
 assumes *gt1*: $1 < (x :: 'a :: \text{linordered-field})$
 assumes *power-int* $x\ m < \text{power-int } x\ n\ n \geq 0$
 shows $m < n$
 ⟨proof⟩

lemma *power-int-strict-mono*:
 $(a :: 'a :: \text{linordered-field}) < b \implies 0 \leq a \implies 0 < n \implies \text{power-int } a\ n < \text{power-int } b\ n$
 ⟨proof⟩

lemma *power-int-mono-iff* [simp]:
 fixes $a\ b :: 'a :: \text{linordered-field}$
 shows $\llbracket a \geq 0; b \geq 0; n > 0 \rrbracket \implies \text{power-int } a\ n \leq \text{power-int } b\ n \longleftrightarrow a \leq b$
 ⟨proof⟩

lemma *power-int-strict-increasing*:
 fixes $a :: 'a :: \text{linordered-field}$
 assumes $n < N\ 1 < a$
 shows $\text{power-int } a\ N > \text{power-int } a\ n$
 ⟨proof⟩

lemma *power-int-increasing*:
 fixes $a :: 'a :: \text{linordered-field}$
 assumes $n \leq N\ a \geq 1$
 shows $\text{power-int } a\ N \geq \text{power-int } a\ n$
 ⟨proof⟩

lemma *power-int-strict-decreasing*:
 fixes $a :: 'a :: \text{linordered-field}$
 assumes $n < N\ 0 < a\ a < 1$
 shows $\text{power-int } a\ N < \text{power-int } a\ n$
 ⟨proof⟩

lemma *power-int-decreasing*:

fixes $a :: 'a :: \text{linordered-field}$

assumes $n \leq N \ 0 \leq a \ a \leq 1 \ a \neq 0 \vee N \neq 0 \vee n = 0$

shows $\text{power-int } a \ N \leq \text{power-int } a \ n$

<proof>

lemma *one-less-power-int*: $1 < (a :: 'a) \implies 0 < n \implies 1 < \text{power-int } a \ n$

<proof>

lemma *power-int-abs*: $|\text{power-int } a \ n :: 'a| = \text{power-int } |a| \ n$

<proof>

lemma *power-int-sgn* [simp]: $\text{sgn } (\text{power-int } a \ n :: 'a) = \text{power-int } (\text{sgn } a) \ n$

<proof>

lemma *abs-power-int-minus* [simp]: $|\text{power-int } (-a) \ n :: 'a| = |\text{power-int } a \ n|$

<proof>

lemma *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>

lemma *power-int-antimono*:

assumes $(a :: 'a :: \text{linordered-field}) \leq b \ 0 < a \ n < 0$

shows $\text{power-int } a \ n \geq \text{power-int } b \ n$

<proof>

end

54.19 Finiteness of intervals

lemma *finite-interval-int1* [iff]: $\text{finite } \{i :: \text{int}. a \leq i \wedge i \leq b\}$

<proof>

lemma *finite-interval-int2* [iff]: $\text{finite } \{i :: \text{int}. a \leq i \wedge i < b\}$

<proof>

lemma *finite-interval-int3* [iff]: $\text{finite } \{i :: \text{int}. a < i \wedge i \leq b\}$

<proof>

lemma *finite-interval-int4* [iff]: $\text{finite } \{i :: \text{int}. a < i \wedge i < b\}$

<proof>

54.20 Configuration of the code generator

Constructors

definition $\text{Pos} :: \text{num} \Rightarrow \text{int}$

where $[simp, code-abbrev]: Pos = numeral$

definition $Neg :: num \Rightarrow int$

where $[simp, code-abbrev]: Neg\ n = -\ (Pos\ n)$

code-datatype $0::int\ Pos\ Neg$

Auxiliary operations.

definition $dup :: int \Rightarrow int$

where $[simp]: dup\ k = k + k$

lemma $dup-code\ [code]:$

$dup\ 0 = 0$

$dup\ (Pos\ n) = Pos\ (Num.Bit0\ n)$

$dup\ (Neg\ n) = Neg\ (Num.Bit0\ n)$

$\langle proof \rangle$

definition $sub :: num \Rightarrow num \Rightarrow int$

where $[simp]: sub\ m\ n = numeral\ m - numeral\ n$

lemma $sub-code\ [code]:$

$sub\ Num.One\ Num.One = 0$

$sub\ (Num.Bit0\ m)\ Num.One = Pos\ (Num.BitM\ m)$

$sub\ (Num.Bit1\ m)\ Num.One = Pos\ (Num.Bit0\ m)$

$sub\ Num.One\ (Num.Bit0\ n) = Neg\ (Num.BitM\ n)$

$sub\ Num.One\ (Num.Bit1\ n) = Neg\ (Num.Bit0\ n)$

$sub\ (Num.Bit0\ m)\ (Num.Bit0\ n) = dup\ (sub\ m\ n)$

$sub\ (Num.Bit1\ m)\ (Num.Bit1\ n) = dup\ (sub\ m\ n)$

$sub\ (Num.Bit1\ m)\ (Num.Bit0\ n) = dup\ (sub\ m\ n) + 1$

$sub\ (Num.Bit0\ m)\ (Num.Bit1\ n) = dup\ (sub\ m\ n) - 1$

$\langle proof \rangle$

lemma $sub-BitM-One-eq:$

$\langle (Num.sub\ (Num.BitM\ n)\ num.One) = 2 * (Num.sub\ n\ Num.One :: int) \rangle$

$\langle proof \rangle$

Implementations.

lemma $one-int-code\ [code]: 1 = Pos\ Num.One$

$\langle proof \rangle$

lemma $plus-int-code\ [code]:$

$k + 0 = k$

$0 + l = l$

$Pos\ m + Pos\ n = Pos\ (m + n)$

$Pos\ m + Neg\ n = sub\ m\ n$

$Neg\ m + Pos\ n = sub\ n\ m$

$Neg\ m + Neg\ n = Neg\ (m + n)$

for $k\ l :: int$

$\langle proof \rangle$

lemma *uminus-int-code* [code]:

$uminus\ 0 = (0::int)$
 $uminus\ (Pos\ m) = Neg\ m$
 $uminus\ (Neg\ m) = Pos\ m$
 ⟨proof⟩

lemma *minus-int-code* [code]:

$k - 0 = k$
 $0 - l = uminus\ l$
 $Pos\ m - Pos\ n = sub\ m\ n$
 $Pos\ m - Neg\ n = Pos\ (m + n)$
 $Neg\ m - Pos\ n = Neg\ (m + n)$
 $Neg\ m - Neg\ n = sub\ n\ m$
for $k\ l :: int$
 ⟨proof⟩

lemma *times-int-code* [code]:

$k * 0 = 0$
 $0 * l = 0$
 $Pos\ m * Pos\ n = Pos\ (m * n)$
 $Pos\ m * Neg\ n = Neg\ (m * n)$
 $Neg\ m * Pos\ n = Neg\ (m * n)$
 $Neg\ m * Neg\ n = Pos\ (m * n)$
for $k\ l :: int$
 ⟨proof⟩

instantiation $int :: equal$

begin

definition $HOL.equal\ k\ l \longleftrightarrow k = (l::int)$

instance

⟨proof⟩

end

lemma *equal-int-code* [code]:

$HOL.equal\ 0\ (0::int) \longleftrightarrow True$
 $HOL.equal\ 0\ (Pos\ l) \longleftrightarrow False$
 $HOL.equal\ 0\ (Neg\ l) \longleftrightarrow False$
 $HOL.equal\ (Pos\ k)\ 0 \longleftrightarrow False$
 $HOL.equal\ (Pos\ k)\ (Pos\ l) \longleftrightarrow HOL.equal\ k\ l$
 $HOL.equal\ (Pos\ k)\ (Neg\ l) \longleftrightarrow False$
 $HOL.equal\ (Neg\ k)\ 0 \longleftrightarrow False$
 $HOL.equal\ (Neg\ k)\ (Pos\ l) \longleftrightarrow False$
 $HOL.equal\ (Neg\ k)\ (Neg\ l) \longleftrightarrow HOL.equal\ k\ l$
 ⟨proof⟩

lemma *equal-int-refl* [code nbe]: $HOL.equal\ k\ k \longleftrightarrow True$
for $k :: int$
 ⟨proof⟩

lemma *less-eq-int-code* [code]:
 $0 \leq (0 :: int) \longleftrightarrow True$
 $0 \leq Pos\ l \longleftrightarrow True$
 $0 \leq Neg\ l \longleftrightarrow False$
 $Pos\ k \leq 0 \longleftrightarrow False$
 $Pos\ k \leq Pos\ l \longleftrightarrow k \leq l$
 $Pos\ k \leq Neg\ l \longleftrightarrow False$
 $Neg\ k \leq 0 \longleftrightarrow True$
 $Neg\ k \leq Pos\ l \longleftrightarrow True$
 $Neg\ k \leq Neg\ l \longleftrightarrow l \leq k$
 ⟨proof⟩

lemma *less-int-code* [code]:
 $0 < (0 :: int) \longleftrightarrow False$
 $0 < Pos\ l \longleftrightarrow True$
 $0 < Neg\ l \longleftrightarrow False$
 $Pos\ k < 0 \longleftrightarrow False$
 $Pos\ k < Pos\ l \longleftrightarrow k < l$
 $Pos\ k < Neg\ l \longleftrightarrow False$
 $Neg\ k < 0 \longleftrightarrow True$
 $Neg\ k < Pos\ l \longleftrightarrow True$
 $Neg\ k < Neg\ l \longleftrightarrow l < k$
 ⟨proof⟩

lemma *nat-code* [code]:
 $nat\ (Int.Neg\ k) = 0$
 $nat\ 0 = 0$
 $nat\ (Int.Pos\ k) = nat-of-num\ k$
 ⟨proof⟩

lemma (in *ring-1*) *of-int-code* [code]:
 $of-int\ (Int.Neg\ k) = -\ numeral\ k$
 $of-int\ 0 = 0$
 $of-int\ (Int.Pos\ k) = numeral\ k$
 ⟨proof⟩

Serializer setup.

code-identifier

code-module $Int \rightarrow (SML)\ Arith\ \mathbf{and}\ (OCaml)\ Arith\ \mathbf{and}\ (Haskell)\ Arith$

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*
 ⟨*proof*⟩

definition *F :: 'a set ⇒ 'a*

where

eq-fold': *F A = the (Finite-Set.fold (λx y. Some (case y of None ⇒ x | Some z ⇒ f x z)) None A)*

lemma *eq-fold*:

assumes *finite A*

shows *F (insert x A) = Finite-Set.fold f x A*

⟨*proof*⟩

lemma *singleton [simp]*:

$F \{x\} = x$
 $\langle proof \rangle$

lemma *insert-not-elem*:
 assumes *finite* A and $x \notin A$ and $A \neq \{\}$
 shows $F (\text{insert } x \ A) = x * F \ A$
 $\langle proof \rangle$

lemma *in-idem*:
 assumes *finite* A and $x \in A$
 shows $x * F \ A = F \ A$
 $\langle proof \rangle$

lemma *insert [simp]*:
 assumes *finite* A and $A \neq \{\}$
 shows $F (\text{insert } x \ A) = x * F \ A$
 $\langle proof \rangle$

lemma *union*:
 assumes *finite* A $A \neq \{\}$ and *finite* B $B \neq \{\}$
 shows $F (A \cup B) = F \ A * F \ B$
 $\langle proof \rangle$

lemma *remove*:
 assumes *finite* A and $x \in A$
 shows $F \ A = (\text{if } A - \{x\} = \{\} \text{ then } x \text{ else } x * F (A - \{x\}))$
 $\langle proof \rangle$

lemma *insert-remove*:
 assumes *finite* A
 shows $F (\text{insert } x \ A) = (\text{if } A - \{x\} = \{\} \text{ then } x \text{ else } x * F (A - \{x\}))$
 $\langle proof \rangle$

lemma *subset*:
 assumes *finite* A $B \neq \{\}$ and $B \subseteq A$
 shows $F \ B * F \ A = F \ A$
 $\langle proof \rangle$

lemma *closed*:
 assumes *finite* A $A \neq \{\}$ and *elem*: $\bigwedge x \ y. x * y \in \{x, y\}$
 shows $F \ A \in A$
 $\langle proof \rangle$

lemma *hom-commute*:
 assumes *hom*: $\bigwedge x \ y. h (x * y) = h \ x * h \ y$
 and N : *finite* N $N \neq \{\}$
 shows $h (F \ N) = F (h \ ` \ N)$
 $\langle proof \rangle$

lemma *infinite*: $\neg \text{finite } A \implies F\ A = \text{the None}$
 $\langle \text{proof} \rangle$

end

locale *semilattice-order-set* = *binary?*: *semilattice-order* + *semilattice-set*
begin

lemma *bounded-iff*:
 assumes *finite* *A* and $A \neq \{\}$
 shows $x \leq F\ A \longleftrightarrow (\forall a \in A. x \leq a)$
 $\langle \text{proof} \rangle$

lemma *boundedI*:
 assumes *finite* *A*
 assumes $A \neq \{\}$
 assumes $\bigwedge a. a \in A \implies x \leq a$
 shows $x \leq F\ A$
 $\langle \text{proof} \rangle$

lemma *boundedE*:
 assumes *finite* *A* and $A \neq \{\}$ and $x \leq F\ A$
 obtains $\bigwedge a. a \in A \implies x \leq a$
 $\langle \text{proof} \rangle$

lemma *coboundedI*:
 assumes *finite* *A*
 and $a \in A$
 shows $F\ A \leq a$
 $\langle \text{proof} \rangle$

lemma *subset-imp*:
 assumes $A \subseteq B$ and $A \neq \{\}$ and *finite* *B*
 shows $F\ B \leq F\ A$
 $\langle \text{proof} \rangle$

end

55.1.2 With neutral element

locale *semilattice-neutr-set* = *semilattice-neutr*
begin

interpretation *comp-fun-idem* *f*
 $\langle \text{proof} \rangle$

definition $F :: 'a \text{ set} \Rightarrow 'a$

where

eq-fold: $F\ A = \text{Finite-Set.fold } f\ \mathbf{1}\ A$

lemma *infinite* [*simp*]:
 $\neg \text{finite } A \implies F A = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *empty* [*simp*]:
 $F \{\} = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *insert* [*simp*]:
 assumes *finite* *A*
 shows $F (\text{insert } x A) = x * F A$
 $\langle \text{proof} \rangle$

lemma *in-idem*:
 assumes *finite* *A* and $x \in A$
 shows $x * F A = F A$
 $\langle \text{proof} \rangle$

lemma *union*:
 assumes *finite* *A* and *finite* *B*
 shows $F (A \cup B) = F A * F B$
 $\langle \text{proof} \rangle$

lemma *remove*:
 assumes *finite* *A* and $x \in A$
 shows $F A = x * F (A - \{x\})$
 $\langle \text{proof} \rangle$

lemma *insert-remove*:
 assumes *finite* *A*
 shows $F (\text{insert } x A) = x * F (A - \{x\})$
 $\langle \text{proof} \rangle$

lemma *subset*:
 assumes *finite* *A* and $B \subseteq A$
 shows $F B * F A = F A$
 $\langle \text{proof} \rangle$

lemma *closed*:
 assumes *finite* *A* $A \neq \{\}$ and *elem*: $\bigwedge x y. x * y \in \{x, y\}$
 shows $F A \in A$
 $\langle \text{proof} \rangle$

end

locale *semilattice-order-neutr-set* = *binary?*: *semilattice-neutr-order* + *semilattice-neutr-set*
begin

lemma *bounded-iff*:
 assumes *finite A*
 shows $x \leq F A \longleftrightarrow (\forall a \in A. x \leq a)$
 $\langle proof \rangle$

lemma *boundedI*:
 assumes *finite A*
 assumes $\bigwedge a. a \in A \implies x \leq a$
 shows $x \leq F A$
 $\langle proof \rangle$

lemma *boundedE*:
 assumes *finite A* and $x \leq F A$
 obtains $\bigwedge a. a \in A \implies x \leq a$
 $\langle proof \rangle$

lemma *coboundedI*:
 assumes *finite A*
 and $a \in A$
 shows $F A \leq a$
 $\langle proof \rangle$

lemma *subset-imp*:
 assumes $A \subseteq B$ and *finite B*
 shows $F B \leq F A$
 $\langle proof \rangle$

end

55.2 Lattice operations on finite sets

context *semilattice-inf*
begin

sublocale *Inf-fin: semilattice-order-set inf less-eq less*
defines
 $Inf-fin (\lhd \sqcap_{fin} \rightarrow [900] 900) = Inf-fin.F \langle proof \rangle$

end

context *semilattice-sup*
begin

sublocale *Sup-fin: semilattice-order-set sup greater-eq greater*
defines
 $Sup-fin (\lhd \sqcup_{fin} \rightarrow [900] 900) = Sup-fin.F \langle proof \rangle$

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 $\bigcap_{fin} A \leq \bigcup_{fin} A$

<proof>

lemma *sup-Inf-absorb* [*simp*]:

finite A $\implies a \in A \implies \bigcap_{fin} A \sqcup a = a$

<proof>

lemma *inf-Sup-absorb* [*simp*]:

finite A $\implies a \in A \implies a \sqcap \bigcup_{fin} A = a$

<proof>

end

context *distrib-lattice*

begin

lemma *sup-Inf1-distrib*:

assumes *finite A*

and $A \neq \{\}$

shows $\sup x (\bigcap_{fin} A) = \bigcap_{fin} \{\sup x a \mid a. a \in A\}$

<proof>

lemma *sup-Inf2-distrib*:

assumes *A: finite A A* $\neq \{\}$ and *B: finite B B* $\neq \{\}$

shows $\sup (\bigcap_{fin} A) (\bigcap_{fin} B) = \bigcap_{fin} \{\sup a b \mid a b. a \in A \wedge b \in B\}$

<proof>

lemma *inf-Sup1-distrib*:

assumes *finite A* and $A \neq \{\}$

shows $\inf x (\bigcup_{fin} A) = \bigcup_{fin} \{\inf x a \mid a. a \in A\}$

<proof>

lemma *inf-Sup2-distrib*:

assumes *A: finite A A* $\neq \{\}$ and *B: finite B B* $\neq \{\}$

shows $\inf (\bigcup_{fin} A) (\bigcup_{fin} B) = \bigcup_{fin} \{\inf a b \mid a b. a \in A \wedge b \in B\}$

<proof>

end

context *complete-lattice*

begin

lemma *Inf-fin-Inf*:

assumes *finite A* and $A \neq \{\}$
 shows $\sqcap_{fin} A = \sqcap A$
 $\langle proof \rangle$

lemma *Sup-fin-Sup*:
 assumes *finite A* and $A \neq \{\}$
 shows $\sqcup_{fin} A = \sqcup A$
 $\langle proof \rangle$

end

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* $\langle proof \rangle$

end

syntax
 -*MIN1* :: *pttrns* \Rightarrow '*b* \Rightarrow '*b* ($\langle \langle indent=3 notation=\langle binder MIN \rangle \rangle MIN$
 -./ -) \rangle [*0*, *10*] *10*)
 -*MIN* :: *pttrn* \Rightarrow '*a set* \Rightarrow '*b* \Rightarrow '*b* ($\langle \langle indent=3 notation=\langle binder MIN \rangle \rangle MIN$
 - \in -./ -) \rangle [*0*, *0*, *10*] *10*)
 -*MAX1* :: *pttrns* \Rightarrow '*b* \Rightarrow '*b* ($\langle \langle indent=3 notation=\langle binder MAX \rangle \rangle MAX$
 -./ -) \rangle [*0*, *10*] *10*)
 -*MAX* :: *pttrn* \Rightarrow '*a set* \Rightarrow '*b* \Rightarrow '*b* ($\langle \langle indent=3 notation=\langle binder MAX \rangle \rangle MAX$
 - \in -./ -) \rangle [*0*, *0*, *10*] *10*)

syntax-consts
 -*MIN1* -*MIN* \Leftarrow *Min* and
 -*MAX1* -*MAX* \Leftarrow *Max*

translations
 $MIN\ x\ y.\ f \Leftarrow MIN\ x.\ MIN\ y.\ f$
 $MIN\ x.\ f \Leftarrow CONST\ Min\ (CONST\ range\ (\lambda x.\ f))$
 $MIN\ x \in A.\ f \Leftarrow CONST\ Min\ ((\lambda x.\ f)\ 'A)$
 $MAX\ x\ y.\ f \Leftarrow MAX\ x.\ MAX\ y.\ f$
 $MAX\ x.\ f \Leftarrow CONST\ Max\ (CONST\ range\ (\lambda x.\ f))$
 $MAX\ x \in A.\ f \Leftarrow CONST\ Max\ ((\lambda x.\ f)\ 'A)$

An aside: *Min/Max* on linear orders as special case of *Inf-fin/Sup-fin*

lemma *Inf-fin-Min*:
 $Inf-fin = (Min :: 'a :: \{semilattice-inf, linorder\}\ set \Rightarrow 'a)$
 $\langle proof \rangle$

lemma *Sup-fin-Max*:

Sup-fin = (*Max* :: 'a:: $\{semilattice-sup, linorder\}$ set \Rightarrow 'a)
 ⟨proof⟩

context *linorder*
begin

lemma *dual-min*:

ord.min greater-eq = *max*
 ⟨proof⟩

lemma *dual-max*:

ord.max greater-eq = *min*
 ⟨proof⟩

lemma *dual-Min*:

linorder.Min greater-eq = *Max*
 ⟨proof⟩

lemma *dual-Max*:

linorder.Max greater-eq = *Min*
 ⟨proof⟩

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** $A \neq \{\}$

shows *Min A* $\in A$

⟨proof⟩

lemma *Max-in [simp]*:

assumes *finite A* **and** $A \neq \{\}$

shows *Max A* $\in A$

⟨proof⟩

lemma *Min-insert2*:

assumes *finite A* **and** *min*: $\bigwedge b. b \in A \implies a \leq b$

shows *Min (insert a A)* = *a*

⟨proof⟩

lemma *Max-insert2*:

assumes *finite A* **and** *max*: $\bigwedge b. b \in A \implies b \leq a$
shows *Max* (*insert a A*) = *a*
 $\langle \text{proof} \rangle$

lemma *Max-const[simp]*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{Max } ((\lambda-. c) ` A) = c$
 $\langle \text{proof} \rangle$

lemma *Min-const[simp]*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{Min } ((\lambda-. c) ` A) = c$
 $\langle \text{proof} \rangle$

lemma *Min-le [simp]*:
assumes *finite A* **and** $x \in A$
shows *Min A* $\leq x$
 $\langle \text{proof} \rangle$

lemma *Max-ge [simp]*:
assumes *finite A* **and** $x \in A$
shows $x \leq \text{Max } A$
 $\langle \text{proof} \rangle$

lemma *Min-eqI*:
assumes *finite A*
assumes $\bigwedge y. y \in A \implies y \geq x$
and $x \in A$
shows *Min A* = *x*
 $\langle \text{proof} \rangle$

lemma *Max-eqI*:
assumes *finite A*
assumes $\bigwedge y. y \in A \implies y \leq x$
and $x \in A$
shows *Max A* = *x*
 $\langle \text{proof} \rangle$

lemma *eq-Min-iff*:
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies m = \text{Min } A \iff m \in A \wedge (\forall a \in A. m \leq a)$
 $\langle \text{proof} \rangle$

lemma *Min-eq-iff*:
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{Min } A = m \iff m \in A \wedge (\forall a \in A. m \leq a)$
 $\langle \text{proof} \rangle$

lemma *eq-Max-iff*:
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies m = \text{Max } A \iff m \in A \wedge (\forall a \in A. a \leq m)$
 $\langle \text{proof} \rangle$

lemma *Max-eq-iff*:
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{Max } A = m \iff m \in A \wedge (\forall a \in A. a \leq m)$
 $\langle \text{proof} \rangle$

context
fixes $A :: 'a \text{ set}$
assumes $\text{fin-nonempty: finite } A \ A \neq \{\}$
begin

lemma $\text{Min-ge-iff [simp]:}$
 $x \leq \text{Min } A \longleftrightarrow (\forall a \in A. x \leq a)$
 $\langle \text{proof} \rangle$

lemma $\text{Max-le-iff [simp]:}$
 $\text{Max } A \leq x \longleftrightarrow (\forall a \in A. a \leq x)$
 $\langle \text{proof} \rangle$

lemma $\text{Min-gr-iff [simp]:}$
 $x < \text{Min } A \longleftrightarrow (\forall a \in A. x < a)$
 $\langle \text{proof} \rangle$

lemma $\text{Max-less-iff [simp]:}$
 $\text{Max } A < x \longleftrightarrow (\forall a \in A. a < x)$
 $\langle \text{proof} \rangle$

lemma Min-le-iff:
 $\text{Min } A \leq x \longleftrightarrow (\exists a \in A. a \leq x)$
 $\langle \text{proof} \rangle$

lemma Max-ge-iff:
 $x \leq \text{Max } A \longleftrightarrow (\exists a \in A. x \leq a)$
 $\langle \text{proof} \rangle$

lemma Min-less-iff:
 $\text{Min } A < x \longleftrightarrow (\exists a \in A. a < x)$
 $\langle \text{proof} \rangle$

lemma Max-gr-iff:
 $x < \text{Max } A \longleftrightarrow (\exists a \in A. x < a)$
 $\langle \text{proof} \rangle$

end

Handy results about Max and Min by Chelsea Edmonds

lemma obtains-Max:
assumes $\text{finite } A \text{ and } A \neq \{\}$
obtains x **where** $x \in A$ **and** $\text{Max } A = x$
 $\langle \text{proof} \rangle$

lemma obtains-MAX:
assumes $\text{finite } A \text{ and } A \neq \{\}$
obtains x **where** $x \in A$ **and** $\text{Max } (f \text{ ` } A) = f x$

$\langle \text{proof} \rangle$

lemma *obtains-Min*:

assumes *finite* A and $A \neq \{\}$
 obtains x where $x \in A$ and $\text{Min } A = x$
 $\langle \text{proof} \rangle$

lemma *obtains-MIN*:

assumes *finite* A and $A \neq \{\}$
 obtains x where $x \in A$ and $\text{Min } (f \text{ ` } A) = f x$
 $\langle \text{proof} \rangle$

lemma *Max-eq-if*:

assumes *finite* A *finite* B $\forall a \in A. \exists b \in B. a \leq b$ $\forall b \in B. \exists a \in A. b \leq a$
 shows $\text{Max } A = \text{Max } B$
 $\langle \text{proof} \rangle$

lemma *Min-antimono*:

assumes $M \subseteq N$ and $M \neq \{\}$ and *finite* N
 shows $\text{Min } N \leq \text{Min } M$
 $\langle \text{proof} \rangle$

lemma *Max-mono*:

assumes $M \subseteq N$ and $M \neq \{\}$ and *finite* N
 shows $\text{Max } M \leq \text{Max } N$
 $\langle \text{proof} \rangle$

lemma *mono-Min-commute*:

assumes *mono* f
 assumes *finite* A and $A \neq \{\}$
 shows $f (\text{Min } A) = \text{Min } (f \text{ ` } A)$
 $\langle \text{proof} \rangle$

lemma *mono-Max-commute*:

assumes *mono* f
 assumes *finite* A and $A \neq \{\}$
 shows $f (\text{Max } A) = \text{Max } (f \text{ ` } A)$
 $\langle \text{proof} \rangle$

lemma *finite-linorder-max-induct* [consumes 1, case-names empty insert]:

assumes *fin*: *finite* A
 and *empty*: $P \{\}$
 and *insert*: $\bigwedge b A. \text{finite } A \implies \forall a \in A. a < b \implies P A \implies P (\text{insert } b A)$
 shows $P A$
 $\langle \text{proof} \rangle$

lemma *finite-linorder-min-induct* [consumes 1, case-names empty insert]:

$\llbracket \text{finite } A; P \{\}; \bigwedge b A. \llbracket \text{finite } A; \forall a \in A. b < a; P A \rrbracket \implies P (\text{insert } b A) \rrbracket \implies P A$

$\langle \text{proof} \rangle$

lemma *finite-ranking-induct*[*consumes 1, case-names empty insert*]:
fixes $f :: 'b \Rightarrow 'a$
assumes *finite S*
assumes $P \ \{\}$
assumes $\bigwedge x S. \text{finite } S \implies (\bigwedge y. y \in S \implies f \ y \leq f \ x) \implies P \ S \implies P \ (\text{insert } x \ S)$
shows $P \ S$
 $\langle \text{proof} \rangle$

lemma *Least-Min*:
assumes *finite {a. P a}* **and** $\exists a. P \ a$
shows $(\text{LEAST } a. P \ a) = \text{Min } \{a. P \ a\}$
 $\langle \text{proof} \rangle$

lemma *Greatest-Max*:
assumes *finite {a. P a}* **and** $\exists a. P \ a$
shows $(\text{GREATEST } a. P \ a) = \text{Max } \{a. P \ a\}$
 $\langle \text{proof} \rangle$

lemma *infinite-growing*:
assumes $X \neq \{\}$
assumes $*$: $\bigwedge x. x \in X \implies \exists y \in X. y > x$
shows $\neg \text{finite } X$
 $\langle \text{proof} \rangle$

end

lemma *sum-le-card-Max*: *finite A* $\implies \text{sum } f \ A \leq \text{card } A * \text{Max } (f \ ' A)$
 $\langle \text{proof} \rangle$

lemma *card-Min-le-sum*: *finite A* $\implies \text{card } A * \text{Min } (f \ ' A) \leq \text{sum } f \ A$
 $\langle \text{proof} \rangle$

context *linordered-ab-semigroup-add*
begin

lemma *Min-add-commute*:
fixes k
assumes *finite S* **and** $S \neq \{\}$
shows $\text{Min } ((\lambda x. f \ x + k) \ ' S) = \text{Min}(f \ ' S) + k$
 $\langle \text{proof} \rangle$

lemma *Max-add-commute*:
fixes k
assumes *finite S* **and** $S \neq \{\}$
shows $\text{Max } ((\lambda x. f \ x + k) \ ' S) = \text{Max}(f \ ' S) + k$
 $\langle \text{proof} \rangle$

end

context *linordered-ab-group-add*
begin

lemma *minus-Max-eq-Min* [*simp*]:
 $\text{finite } S \implies S \neq \{\} \implies - \text{Max } S = \text{Min } (\text{uminus } ' S)$
 $\langle \text{proof} \rangle$

lemma *minus-Min-eq-Max* [*simp*]:
 $\text{finite } S \implies S \neq \{\} \implies - \text{Min } S = \text{Max } (\text{uminus } ' S)$
 $\langle \text{proof} \rangle$

end

context *complete-linorder*
begin

lemma *Min-Inf*:
assumes *finite A* **and** $A \neq \{\}$
shows $\text{Min } A = \text{Inf } A$
 $\langle \text{proof} \rangle$

lemma *Max-Sup*:
assumes *finite A* **and** $A \neq \{\}$
shows $\text{Max } A = \text{Sup } A$
 $\langle \text{proof} \rangle$

end

lemma *disjnt-ge-max*:
 $\langle \text{disjnt } X \ Y \rangle \text{ if } \langle \text{finite } Y \rangle \langle \bigwedge x. x \in X \implies x > \text{Max } Y \rangle$
 $\langle \text{proof} \rangle$

55.5 An aside: code generation for *LEAST* and *GREATEST*

context
begin

qualified definition *Least* :: $\langle 'a::\text{linorder set} \Rightarrow 'a \rangle$ — only for code generation
where *Least-eq* [*code-abbrev, simp*]: $\langle \text{Least } S = (\text{LEAST } x. x \in S) \rangle$

qualified lemma *Least-filter-eq* [*code-abbrev*]:
 $\langle \text{Least } (\text{Set.filter } P \ S) = (\text{LEAST } x. x \in S \wedge P \ x) \rangle$
 $\langle \text{proof} \rangle$ **definition** *Least-abort* :: $\langle 'a \text{ set} \Rightarrow 'a::\text{linorder} \rangle$
where $\langle \text{Least-abort} = \text{Least} \rangle$

qualified lemma *Least-code* [*code abort: Lattices-Big.Least-abort, code*]:

$\langle \text{Least } A = (\text{if finite } A \longrightarrow \text{Set.is-empty } A \text{ then Least-abort } A \text{ else Min } A) \rangle$
 $\langle \text{proof} \rangle$ **definition** *Greatest* :: $\langle 'a::\text{linorder set} \Rightarrow 'a \rangle$ — only for code generation
where *Greatest-eq* [code-abbrev, simp]: $\langle \text{Greatest } S = (\text{GREATEST } x. x \in S) \rangle$

qualified lemma *Greatest-filter-eq* [code-abbrev]:
 $\langle \text{Greatest } (\text{Set.filter } P \ S) = (\text{GREATEST } x. x \in S \wedge P \ x) \rangle$
 $\langle \text{proof} \rangle$ **definition** *Greatest-abort* :: $\langle 'a \text{ set} \Rightarrow 'a::\text{linorder} \rangle$
where $\langle \text{Greatest-abort} = \text{Greatest} \rangle$

qualified lemma *Greatest-code* [code abort: Lattices-Big.Greatest-abort, code]:
 $\langle \text{Greatest } A = (\text{if finite } A \longrightarrow \text{Set.is-empty } A \text{ then Greatest-abort } A \text{ else Max } A) \rangle$
 $\langle \text{proof} \rangle$

end

55.6 Arg Min

context *ord*
begin

definition *is-arg-min* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'b \Rightarrow \text{bool}$ **where**
 $\text{is-arg-min } f \ P \ x = (P \ x \wedge \neg(\exists y. P \ y \wedge f \ y < f \ x))$

definition *arg-min* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'b$ **where**
 $\text{arg-min } f \ P = (\text{SOME } x. \text{is-arg-min } f \ P \ x)$

definition *arg-min-on* :: $('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'b$ **where**
 $\text{arg-min-on } f \ S = \text{arg-min } f \ (\lambda x. x \in S)$

end

syntax

$\text{-arg-min} :: ('b \Rightarrow 'a) \Rightarrow \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'b$
 $(\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder ARG-MIN} \rangle \rangle \text{ARG}'\text{-MIN} \text{ - ./ -} \rangle [1000, 0, 10]$
 $10)$

syntax-consts

$\text{-arg-min} \Rightarrow \text{arg-min}$

translations

$\text{ARG-MIN } f \ x. P \Rightarrow \text{CONST arg-min } f \ (\lambda x. P)$

lemma *is-arg-min-linorder*: **fixes** $f :: 'a \Rightarrow 'b :: \text{linorder}$
shows $\text{is-arg-min } f \ P \ x = (P \ x \wedge (\forall y. P \ y \longrightarrow f \ x \leq f \ y))$
 $\langle \text{proof} \rangle$

lemma *is-arg-min-antimono*: **fixes** $f :: 'a \Rightarrow ('b::\text{order})$
shows $\llbracket \text{is-arg-min } f \ P \ x; f \ y \leq f \ x; P \ y \rrbracket \Longrightarrow \text{is-arg-min } f \ P \ y$
 $\langle \text{proof} \rangle$

lemma *arg-minI*:

```

[[ P x;
  ∧ y. P y ⇒ ¬ f y < f x;
  ∧ x. [[ P x; ∀ y. P y ⇒ ¬ f y < f x ]] ⇒ Q x ]]
⇒ Q (arg-min f P)
⟨proof⟩

```

lemma *arg-min-equality*:

```

[[ P k; ∧ x. P x ⇒ f k ≤ f x ]] ⇒ f (arg-min f P) = f k
for f :: - ⇒ 'a::order
⟨proof⟩

```

lemma *wf-linord-ex-has-least*:

```

[[ wf r; ∀ x y. (x, y) ∈ r+ ⇔ (y, x) ∉ r*; P k ]]
⇒ ∃ x. P x ∧ (∀ y. P y ⇒ (m x, m y) ∈ r*)
⟨proof⟩

```

lemma *ex-has-least-nat*: $P k \Rightarrow \exists x. P x \wedge (\forall y. P y \Rightarrow m x \leq m y)$

```

for m :: 'a ⇒ nat
⟨proof⟩

```

lemma *arg-min-nat-lemma*:

```

P k ⇒ P(arg-min m P) ∧ (∀ y. P y ⇒ m (arg-min m P) ≤ m y)
for m :: 'a ⇒ nat
⟨proof⟩

```

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 ⇒ is-arg-min m P (arg-min m P)
⟨proof⟩

```

lemma *arg-min-nat-le*: $P x \Rightarrow m (\text{arg-min } m P) \leq m x$

```

for m :: 'a ⇒ nat
⟨proof⟩

```

lemma *ex-min-if-finite*:

```

[[ finite S; S ≠ {} ]] ⇒ ∃ m ∈ S. ¬(∃ x ∈ S. x < (m::'a::order))
⟨proof⟩

```

lemma *ex-is-arg-min-if-finite*: **fixes** $f :: 'a \Rightarrow 'b :: order$

```

shows [[ finite S; S ≠ {} ]] ⇒ ∃ x. is-arg-min f (λx. x ∈ S) x
⟨proof⟩

```

lemma *arg-min-SOME-Min*:

```

finite S ⇒ arg-min-on f S = (SOME y. y ∈ S ∧ f y = Min(f ` S))
⟨proof⟩

```

lemma *arg-min-if-finite*: **fixes** $f :: 'a \Rightarrow 'b :: order$

```

assumes finite S S ≠ {}

```

shows $\text{arg-min-on } f \ S \in S \text{ and } \neg(\exists x \in S. f \ x < f \ (\text{arg-min-on } f \ S))$
 $\langle \text{proof} \rangle$

lemma *arg-min-least*: **fixes** $f :: 'a \Rightarrow 'b :: \text{linorder}$
shows $\llbracket \text{finite } S; \ S \neq \{\}; \ y \in S \rrbracket \Longrightarrow f(\text{arg-min-on } f \ S) \leq f \ y$
 $\langle \text{proof} \rangle$

lemma *arg-min-inj-eq*: **fixes** $f :: 'a \Rightarrow 'b :: \text{order}$
shows $\llbracket \text{inj-on } f \ \{x. P \ x\}; \ P \ a; \ \forall y. P \ y \longrightarrow f \ a \leq f \ y \rrbracket \Longrightarrow \text{arg-min } f \ P = a$
 $\langle \text{proof} \rangle$

55.7 Arg Max

context *ord*
begin

definition *is-arg-max* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'b \Rightarrow \text{bool}$ **where**
 $\text{is-arg-max } f \ P \ x = (P \ x \wedge \neg(\exists y. P \ y \wedge f \ y > f \ x))$

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* :: $('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'b$ **where**
 $\text{arg-max-on } f \ S = \text{arg-max } f \ (\lambda x. x \in S)$

end

syntax
 $\text{-arg-max} :: ('b \Rightarrow 'a) \Rightarrow \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a$
 $(\langle (\langle \text{indent} = 3 \text{ notation} = \langle \text{binder ARG-MAX} \rangle \rangle \text{ARG'-MAX} - \text{./ } -) \rangle [1000, 0, 10]$
 $10)$

syntax-consts
 $\text{-arg-max} \equiv \text{arg-max}$

translations
 $\text{ARG-MAX } f \ x. P \equiv \text{CONST arg-max } f \ (\lambda x. P)$

lemma *is-arg-max-linorder*: **fixes** $f :: 'a \Rightarrow 'b :: \text{linorder}$
shows $\text{is-arg-max } f \ P \ x = (P \ x \wedge (\forall y. P \ y \longrightarrow f \ x \geq f \ y))$
 $\langle \text{proof} \rangle$

lemma *arg-maxI*:
 $P \ x \Longrightarrow$
 $(\bigwedge y. P \ y \Longrightarrow \neg f \ y > f \ x) \Longrightarrow$
 $(\bigwedge x. P \ x \Longrightarrow \forall y. P \ y \longrightarrow \neg f \ y > f \ x \Longrightarrow Q \ x) \Longrightarrow$
 $Q \ (\text{arg-max } f \ P)$
 $\langle \text{proof} \rangle$

lemma *arg-max-equality*:
 $\llbracket P \ k; \bigwedge x. P \ x \Longrightarrow f \ x \leq f \ k \rrbracket \Longrightarrow f \ (\text{arg-max } f \ P) = f \ k$

for $f :: - \Rightarrow 'a::order$
 $\langle proof \rangle$

lemma *ex-has-greatest-nat-lemma*:

$P\ k \Longrightarrow \forall x. P\ x \longrightarrow (\exists y. P\ y \wedge \neg f\ y \leq f\ x) \Longrightarrow \exists y. P\ y \wedge \neg f\ y < f\ k + n$
for $f :: 'a \Rightarrow nat$
 $\langle proof \rangle$

lemma *ex-has-greatest-nat*:

assumes $P\ k$
and $\forall y. P\ y \longrightarrow (f:: 'a \Rightarrow nat)\ y < b$
shows $\exists x. P\ x \wedge (\forall y. P\ y \longrightarrow f\ y \leq f\ x)$
 $\langle proof \rangle$

lemma *arg-max-nat-lemma*:

$\llbracket P\ k; \forall y. P\ y \longrightarrow f\ y < b \rrbracket$
 $\Longrightarrow P\ (arg-max\ f\ P) \wedge (\forall y. P\ y \longrightarrow f\ y \leq f\ (arg-max\ f\ P))$
for $f :: 'a \Rightarrow nat$
 $\langle proof \rangle$

lemmas *arg-max-natI* = *arg-max-nat-lemma* [THEN conjunct1]

lemma *arg-max-nat-le*: $P\ x \Longrightarrow \forall y. P\ y \longrightarrow f\ y < b \Longrightarrow f\ x \leq f\ (arg-max\ f\ P)$

for $f :: 'a \Rightarrow nat$
 $\langle proof \rangle$

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* :: $'a \Rightarrow nat$

assumes *size-0* [simp]: *euclidean-size* 0 = 0

assumes *mod-size-less*:

$b \neq 0 \Longrightarrow euclidean-size\ (a\ mod\ b) < euclidean-size\ b$

assumes *size-mult-mono*:

$b \neq 0 \Longrightarrow euclidean-size\ a \leq euclidean-size\ (a * b)$

begin

lemma *euclidean-size-eq-0-iff* [simp]:

$euclidean-size\ b = 0 \longleftrightarrow b = 0$

$\langle proof \rangle$

lemma *euclidean-size-greater-0-iff* [simp]:

euclidean-size $b > 0 \longleftrightarrow b \neq 0$

<proof>

lemma *size-mult-mono'*: $b \neq 0 \implies \text{euclidean-size } a \leq \text{euclidean-size } (b * a)$

<proof>

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>

lemma *euclidean-size-times-unit*:

assumes *is-unit* a

shows $\text{euclidean-size } (a * b) = \text{euclidean-size } b$

<proof>

lemma *euclidean-size-unit*:

is-unit $a \implies \text{euclidean-size } a = \text{euclidean-size } 1$

<proof>

lemma *unit-iff-euclidean-size*:

is-unit $a \longleftrightarrow \text{euclidean-size } a = \text{euclidean-size } 1 \wedge a \neq 0$

<proof>

lemma *euclidean-size-times-nonunit*:

assumes $a \neq 0$ $b \neq 0 \neg \text{is-unit } a$

shows $\text{euclidean-size } b < \text{euclidean-size } (a * b)$

<proof>

lemma *dvd-imp-size-le*:

assumes $a \text{ dvd } b$ $b \neq 0$

shows $\text{euclidean-size } a \leq \text{euclidean-size } b$

<proof>

lemma *dvd-proper-imp-size-less*:

assumes $a \text{ dvd } b \neg b \text{ dvd } a$ $b \neq 0$

shows $\text{euclidean-size } a < \text{euclidean-size } b$

<proof>

lemma *unit-imp-mod-eq-0*:

$a \bmod b = 0$ **if** *is-unit* b

<proof>

lemma *mod-eq-self-iff-div-eq-0*:

$a \bmod b = a \longleftrightarrow a \text{ div } b = 0$ (**is** $?P \longleftrightarrow ?Q$)

<proof>

lemma *coprime-mod-left-iff* [simp]:
 $\text{coprime } (a \bmod b) \ b \longleftrightarrow \text{coprime } a \ b \text{ if } b \neq 0$
 ⟨proof⟩

lemma *coprime-mod-right-iff* [simp]:
 $\text{coprime } a \ (b \bmod a) \longleftrightarrow \text{coprime } a \ b \text{ if } a \neq 0$
 ⟨proof⟩

end

class *euclidean-ring* = *idom-modulo* + *euclidean-semiring*
begin

lemma *dvd-diff-commute* [ac-simps]:
 $a \text{ dvd } c - b \longleftrightarrow a \text{ dvd } b - c$
 ⟨proof⟩

end

56.2 Euclidean (semi)rings with cancel rules

class *euclidean-semiring-cancel* = *euclidean-semiring* +
assumes *div-mult-self1* [simp]: $b \neq 0 \implies (a + c * b) \text{ div } b = c + a \text{ div } b$
and *div-mult-mult1* [simp]: $c \neq 0 \implies (c * a) \text{ div } (c * b) = a \text{ div } b$
begin

lemma *div-mult-self2* [simp]:
assumes $b \neq 0$
shows $(a + b * c) \text{ div } b = c + a \text{ div } b$
 ⟨proof⟩

lemma *div-mult-self3* [simp]:
assumes $b \neq 0$
shows $(c * b + a) \text{ div } b = c + a \text{ div } b$
 ⟨proof⟩

lemma *div-mult-self4* [simp]:
assumes $b \neq 0$
shows $(b * c + a) \text{ div } b = c + a \text{ div } b$
 ⟨proof⟩

lemma *mod-mult-self1* [simp]: $(a + c * b) \bmod b = a \bmod b$
 ⟨proof⟩

lemma *mod-mult-self2* [simp]:
 $(a + b * c) \bmod b = a \bmod b$
 ⟨proof⟩

lemma *mod-mult-self3* [simp]:
 $(c * b + a) \bmod b = a \bmod b$
 ⟨proof⟩

lemma *mod-mult-self4* [simp]:
 $(b * c + a) \bmod b = a \bmod b$
 ⟨proof⟩

lemma *mod-mult-self1-is-0* [simp]:
 $b * a \bmod b = 0$
 ⟨proof⟩

lemma *mod-mult-self2-is-0* [simp]:
 $a * b \bmod b = 0$
 ⟨proof⟩

lemma *div-add-self1*:
 assumes $b \neq 0$
 shows $(b + a) \operatorname{div} b = a \operatorname{div} b + 1$
 ⟨proof⟩

lemma *div-add-self2*:
 assumes $b \neq 0$
 shows $(a + b) \operatorname{div} b = a \operatorname{div} b + 1$
 ⟨proof⟩

lemma *mod-add-self1* [simp]:
 $(b + a) \bmod b = a \bmod b$
 ⟨proof⟩

lemma *mod-add-self2* [simp]:
 $(a + b) \bmod b = a \bmod b$
 ⟨proof⟩

lemma *mod-div-trivial* [simp]:
 $a \bmod b \operatorname{div} b = 0$
 ⟨proof⟩

lemma *mod-mod-trivial* [simp]:
 $a \bmod b \bmod b = a \bmod b$
 ⟨proof⟩

lemma *mod-mod-cancel*:
 assumes $c \operatorname{dvd} b$
 shows $a \bmod b \bmod c = a \bmod c$
 ⟨proof⟩

lemma *div-mult-mult2* [simp]:
 $c \neq 0 \implies (a * c) \operatorname{div} (b * c) = a \operatorname{div} b$

$\langle \text{proof} \rangle$

lemma *div-mult-mult1-if* [simp]:

$$(c * a) \text{ div } (c * b) = (\text{if } c = 0 \text{ then } 0 \text{ else } a \text{ div } b)$$

$\langle \text{proof} \rangle$

lemma *mod-mult-mult1*:

$$(c * a) \text{ mod } (c * b) = c * (a \text{ mod } b)$$

$\langle \text{proof} \rangle$

lemma *mod-mult-mult2*:

$$(a * c) \text{ mod } (b * c) = (a \text{ mod } b) * c$$

$\langle \text{proof} \rangle$

lemma *mult-mod-left*: $(a \text{ mod } b) * c = (a * c) \text{ mod } (b * c)$

$\langle \text{proof} \rangle$

lemma *mult-mod-right*: $c * (a \text{ mod } b) = (c * a) \text{ mod } (c * b)$

$\langle \text{proof} \rangle$

lemma *dvd-mod*: $k \text{ dvd } m \implies k \text{ dvd } n \implies k \text{ dvd } (m \text{ mod } n)$

$\langle \text{proof} \rangle$

lemma *div-plus-div-distrib-dvd-left*:

$$c \text{ dvd } a \implies (a + b) \text{ div } c = a \text{ div } c + b \text{ div } c$$

$\langle \text{proof} \rangle$

lemma *div-plus-div-distrib-dvd-right*:

$$c \text{ dvd } b \implies (a + b) \text{ div } c = a \text{ div } c + b \text{ div } c$$

$\langle \text{proof} \rangle$

lemma *sum-div-partition*:

$$\langle (\sum_{a \in A} a) \text{ div } b = (\sum_{a \in A \cap \{a. b \text{ dvd } f a\}} f a \text{ div } b) + (\sum_{a \in A \cap \{a. \neg b \text{ dvd } f a\}} f a) \text{ div } b \rangle$$

if $\langle \text{finite } A \rangle$

$\langle \text{proof} \rangle$

named-theorems *mod-simps*

Addition respects modular equivalence.

lemma *mod-add-left-eq* [mod-simps]:

$$(a \text{ mod } c + b) \text{ mod } c = (a + b) \text{ mod } c$$

$\langle \text{proof} \rangle$

lemma *mod-add-right-eq* [mod-simps]:

$$(a + b \text{ mod } c) \text{ mod } c = (a + b) \text{ mod } c$$

$\langle \text{proof} \rangle$

lemma *mod-add-eq*:

$$(a \bmod c + b \bmod c) \bmod c = (a + b) \bmod c$$

<proof>

lemma *mod-sum-eq* [*mod-simps*]:
 $(\sum_{i \in A} f \ i \bmod a) \bmod a = \text{sum } f \ A \bmod a$
<proof>

lemma *mod-add-cong*:
assumes $a \bmod c = a' \bmod c$
assumes $b \bmod c = b' \bmod c$
shows $(a + b) \bmod c = (a' + b') \bmod c$
<proof>

Multiplication respects modular equivalence.

lemma *mod-mult-left-eq* [*mod-simps*]:
 $((a \bmod c) * b) \bmod c = (a * b) \bmod c$
<proof>

lemma *mod-mult-right-eq* [*mod-simps*]:
 $(a * (b \bmod c)) \bmod c = (a * b) \bmod c$
<proof>

lemma *mod-mult-eq*:
 $((a \bmod c) * (b \bmod c)) \bmod c = (a * b) \bmod c$
<proof>

lemma *mod-prod-eq* [*mod-simps*]:
 $(\prod_{i \in A} f \ i \bmod a) \bmod a = \text{prod } f \ A \bmod a$
<proof>

lemma *mod-mult-cong*:
assumes $a \bmod c = a' \bmod c$
assumes $b \bmod c = b' \bmod c$
shows $(a * b) \bmod c = (a' * b') \bmod c$
<proof>

Exponentiation respects modular equivalence.

lemma *power-mod* [*mod-simps*]:
 $((a \bmod b) ^ n) \bmod b = (a ^ n) \bmod b$
<proof>

lemma *power-diff-power-eq*:
 $\langle a ^ m \text{ div } a ^ n = (\text{if } n \leq m \text{ then } a ^ (m - n) \text{ else } 1 \text{ div } a ^ (n - m)) \rangle$
if $\langle a \neq 0 \rangle$
<proof>

end

class *euclidean-ring-cancel* = *euclidean-ring* + *euclidean-semiring-cancel*
begin

subclass *idom-divide* $\langle \text{proof} \rangle$

lemma *div-minus-minus* [*simp*]: $(- a) \text{ div } (- b) = a \text{ div } b$
 $\langle \text{proof} \rangle$

lemma *mod-minus-minus* [*simp*]: $(- a) \text{ mod } (- b) = - (a \text{ mod } b)$
 $\langle \text{proof} \rangle$

lemma *div-minus-right*: $a \text{ div } (- b) = (- a) \text{ div } b$
 $\langle \text{proof} \rangle$

lemma *mod-minus-right*: $a \text{ mod } (- b) = - ((- a) \text{ mod } b)$
 $\langle \text{proof} \rangle$

lemma *div-minus1-right* [*simp*]: $a \text{ div } (- 1) = - a$
 $\langle \text{proof} \rangle$

lemma *mod-minus1-right* [*simp*]: $a \text{ mod } (- 1) = 0$
 $\langle \text{proof} \rangle$

Negation respects modular equivalence.

lemma *mod-minus-eq* [*mod-simps*]:
 $(- (a \text{ mod } b)) \text{ mod } b = (- a) \text{ mod } b$
 $\langle \text{proof} \rangle$

lemma *mod-minus-cong*:
assumes $a \text{ mod } b = a' \text{ mod } b$
shows $(- a) \text{ mod } b = (- a') \text{ mod } b$
 $\langle \text{proof} \rangle$

Subtraction respects modular equivalence.

lemma *mod-diff-left-eq* [*mod-simps*]:
 $(a \text{ mod } c - b) \text{ mod } c = (a - b) \text{ mod } c$
 $\langle \text{proof} \rangle$

lemma *mod-diff-right-eq* [*mod-simps*]:
 $(a - b \text{ mod } c) \text{ mod } c = (a - b) \text{ mod } c$
 $\langle \text{proof} \rangle$

lemma *mod-diff-eq*:
 $(a \text{ mod } c - b \text{ mod } c) \text{ mod } c = (a - b) \text{ mod } c$
 $\langle \text{proof} \rangle$

lemma *mod-diff-cong*:
assumes $a \text{ mod } c = a' \text{ mod } c$
assumes $b \text{ mod } c = b' \text{ mod } c$

shows $(a - b) \bmod c = (a' - b') \bmod c$
 $\langle \text{proof} \rangle$

lemma *minus-mod-self2* [simp]:
 $(a - b) \bmod b = a \bmod b$
 $\langle \text{proof} \rangle$

lemma *minus-mod-self1* [simp]:
 $(b - a) \bmod b = -a \bmod b$
 $\langle \text{proof} \rangle$

lemma *mod-eq-dvd-iff*:
 $a \bmod c = b \bmod c \longleftrightarrow c \text{ dvd } a - b$ (is $?P \longleftrightarrow ?Q$)
 $\langle \text{proof} \rangle$

lemma *mod-eqE*:
assumes $a \bmod c = b \bmod c$
obtains d **where** $b = a + c * d$
 $\langle \text{proof} \rangle$

lemma *invertible-coprime*:
 $\text{coprime } a \ c$ **if** $a * b \bmod c = 1$
 $\langle \text{proof} \rangle$

end

56.3 Uniquely determined division

class *unique-euclidean-semiring* = *euclidean-semiring* +
assumes *euclidean-size-mult*: $\langle \text{euclidean-size } (a * b) = \text{euclidean-size } a * \text{euclidean-size } b \rangle$
fixes *division-segment* :: $\langle 'a \Rightarrow 'a \rangle$
assumes *is-unit-division-segment* [simp]: $\langle \text{is-unit } (\text{division-segment } a) \rangle$
and *division-segment-mult*:
 $\langle a \neq 0 \implies b \neq 0 \implies \text{division-segment } (a * b) = \text{division-segment } a * \text{division-segment } b \rangle$
and *division-segment-mod*:
 $\langle b \neq 0 \implies \neg b \text{ dvd } a \implies \text{division-segment } (a \bmod b) = \text{division-segment } b \rangle$
assumes *div-bounded*:
 $\langle b \neq 0 \implies \text{division-segment } r = \text{division-segment } b$
 $\implies \text{euclidean-size } r < \text{euclidean-size } b$
 $\implies (q * b + r) \text{ div } b = q \rangle$
begin

lemma *division-segment-not-0* [simp]:
 $\langle \text{division-segment } a \neq 0 \rangle$
 $\langle \text{proof} \rangle$

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⟩

```

```

subclass euclidean-semiring-cancel
  ⟨proof⟩

```

```

lemma div-eq-0-iff:
  ⟨a div b = 0 ⟷ euclidean-size a < euclidean-size b ∨ b = 0⟩ (is - ⟷ ?P)
  if ⟨division-segment a = division-segment b⟩
  ⟨proof⟩

```

```

lemma div-mult1-eq:
  ⟨(a * b) div c = a * (b div c) + a * (b mod c) div c⟩
  ⟨proof⟩

```

```

lemma div-add1-eq:
  ⟨(a + b) div c = a div c + b div c + (a mod c + b mod c) div c⟩
  ⟨proof⟩

```

```

end

```

```

class unique-euclidean-ring = euclidean-ring + unique-euclidean-semiring
begin

```

```

subclass euclidean-ring-cancel ⟨proof⟩

```

```

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⟩
  ⟨proof⟩

```

definition *divide-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$

where $\langle m \text{ div } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \text{Max } \{k. k * n \leq m\}) \rangle$ **for** $m \ n :: \text{nat}$

instance

$\langle \text{proof} \rangle$

end

lemma *coprime-Suc-0-left* [simp]:

$\text{coprime } (\text{Suc } 0) \ n$

$\langle \text{proof} \rangle$

lemma *coprime-Suc-0-right* [simp]:

$\text{coprime } n \ (\text{Suc } 0)$

$\langle \text{proof} \rangle$

lemma *coprime-common-divisor-nat*: $\text{coprime } a \ b \Longrightarrow x \text{ dvd } a \Longrightarrow x \text{ dvd } b \Longrightarrow x = 1$

for $a \ b :: \text{nat}$

$\langle \text{proof} \rangle$

instantiation *nat* :: *unique-euclidean-semiring*

begin

definition *euclidean-size-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \rangle$

where [simp]: $\langle \text{euclidean-size-nat} = \text{id} \rangle$

definition *division-segment-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \rangle$

where [simp]: $\langle \text{division-segment } n = 1 \rangle$ **for** $n :: \text{nat}$

definition *modulo-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$

where $\langle m \text{ mod } n = m - (m \text{ div } n * n) \rangle$ **for** $m \ n :: \text{nat}$

instance $\langle \text{proof} \rangle$

end

lemma *euclidean-relation-natI* [case-names by0 divides euclidean-relation]:

$\langle (m \text{ div } n, m \text{ mod } n) = (q, r) \rangle$

if by0: $\langle n = 0 \Longrightarrow q = 0 \wedge r = m \rangle$

and divides: $\langle n > 0 \Longrightarrow n \text{ dvd } m \Longrightarrow r = 0 \wedge m = q * n \rangle$

and euclidean-relation: $\langle n > 0 \Longrightarrow \neg n \text{ dvd } m \Longrightarrow r < n \wedge m = q * n + r \rangle$

for $m \ n \ q \ r :: \text{nat}$

$\langle \text{proof} \rangle$

lemma *div-nat-eqI*:

$\langle m \text{ div } n = q \rangle$ **if** $\langle n * q \leq m \rangle$ **and** $\langle m < n * \text{Suc } q \rangle$ **for** $m \ n \ q :: \text{nat}$

$\langle \text{proof} \rangle$

lemma *mod-nat-eqI*:

$\langle m \bmod n = r \rangle$ **if** $\langle r < n \rangle$ **and** $\langle r \leq m \rangle$ **and** $\langle n \text{ dvd } m - r \rangle$ **for** $m \ n \ r :: \text{nat}$
 $\langle \text{proof} \rangle$

Tool support

$\langle ML \rangle$

lemma *div-mult-self-is-m* [simp]:

$m * n \text{ div } n = m$ **if** $n > 0$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *div-mult-self1-is-m* [simp]:

$n * m \text{ div } n = m$ **if** $n > 0$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *mod-less-divisor* [simp]:

$m \bmod n < n$ **if** $n > 0$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *mod-le-divisor* [simp]:

$m \bmod n \leq n$ **if** $n > 0$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *div-times-less-eq-dividend* [simp]:

$m \text{ div } n * n \leq m$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *times-div-less-eq-dividend* [simp]:

$n * (m \text{ div } n) \leq m$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *dividend-less-div-times*:

$m < n + (m \text{ div } n) * n$ **if** $0 < n$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *dividend-less-times-div*:

$m < n + n * (m \text{ div } n)$ **if** $0 < n$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *mod-Suc-le-divisor* [simp]:

$m \bmod \text{Suc } n \leq n$
 $\langle \text{proof} \rangle$

lemma *mod-less-eq-dividend* [simp]:

$m \bmod n \leq m$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma

div-less [simp]: $m \text{ div } n = 0$

and *mod-less* [*simp*]: $m \bmod n = m$
if $m < n$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *split-div*:

$\langle P \ (m \ \text{div} \ n) \longleftrightarrow$
 $(n = 0 \longrightarrow P \ 0) \wedge$
 $(n \neq 0 \longrightarrow (\forall i \ j. \ j < n \wedge m = n * i + j \longrightarrow P \ i)) \rangle$ (**is** *?div*)
and *split-mod*:
 $\langle Q \ (m \bmod n) \longleftrightarrow$
 $(n = 0 \longrightarrow Q \ m) \wedge$
 $(n \neq 0 \longrightarrow (\forall i \ j. \ j < n \wedge m = n * i + j \longrightarrow Q \ j)) \rangle$ (**is** *?mod*)
for $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

declare *split-div* [*of* - - $\langle \text{numeral } n \rangle$, *linarith-split*] **for** n

declare *split-mod* [*of* - - $\langle \text{numeral } n \rangle$, *linarith-split*] **for** n

lemma *split-div'*:

$P \ (m \ \text{div} \ n) \longleftrightarrow n = 0 \wedge P \ 0 \vee (\exists q. \ (n * q \leq m \wedge m < n * \text{Suc } q) \wedge P \ q)$
 $\langle \text{proof} \rangle$

lemma *le-div-geq*:

$m \ \text{div} \ n = \text{Suc} \ ((m - n) \ \text{div} \ n)$ **if** $0 < n$ **and** $n \leq m$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *le-mod-geq*:

$m \bmod n = (m - n) \bmod n$ **if** $n \leq m$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *div-if*:

$m \ \text{div} \ n = (\text{if } m < n \vee n = 0 \text{ then } 0 \text{ else } \text{Suc} \ ((m - n) \ \text{div} \ n))$
 $\langle \text{proof} \rangle$

lemma *mod-if*:

$m \bmod n = (\text{if } m < n \text{ then } m \text{ else } (m - n) \bmod n)$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *div-eq-0-iff*:

$m \ \text{div} \ n = 0 \longleftrightarrow m < n \vee n = 0$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *div-greater-zero-iff*:

$m \ \text{div} \ n > 0 \longleftrightarrow n \leq m \wedge n > 0$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *mod-greater-zero-iff-not-dvd*:

$m \bmod n > 0 \longleftrightarrow \neg n \ \text{dvd} \ m$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *div-by-Suc-0* [simp]:

$$m \operatorname{div} \operatorname{Suc} 0 = m$$

⟨proof⟩

lemma *mod-by-Suc-0* [simp]:

$$m \operatorname{mod} \operatorname{Suc} 0 = 0$$

⟨proof⟩

lemma *div2-Suc-Suc* [simp]:

$$\operatorname{Suc} (\operatorname{Suc} m) \operatorname{div} 2 = \operatorname{Suc} (m \operatorname{div} 2)$$

⟨proof⟩

lemma *Suc-n-div-2-gt-zero* [simp]:

$$0 < \operatorname{Suc} n \operatorname{div} 2 \text{ if } n > 0 \text{ for } n :: \text{nat}$$

⟨proof⟩

lemma *div-2-gt-zero* [simp]:

$$0 < n \operatorname{div} 2 \text{ if } \operatorname{Suc} 0 < n \text{ for } n :: \text{nat}$$

⟨proof⟩

lemma *mod2-Suc-Suc* [simp]:

$$\operatorname{Suc} (\operatorname{Suc} m) \operatorname{mod} 2 = m \operatorname{mod} 2$$

⟨proof⟩

lemma *add-self-div-2* [simp]:

$$(m + m) \operatorname{div} 2 = m \text{ for } m :: \text{nat}$$

⟨proof⟩

lemma *add-self-mod-2* [simp]:

$$(m + m) \operatorname{mod} 2 = 0 \text{ for } m :: \text{nat}$$

⟨proof⟩

lemma *mod2-gr-0* [simp]:

$$0 < m \operatorname{mod} 2 \longleftrightarrow m \operatorname{mod} 2 = 1 \text{ for } m :: \text{nat}$$

⟨proof⟩

lemma *mod-Suc-eq* [mod-simps]:

$$\operatorname{Suc} (m \operatorname{mod} n) \operatorname{mod} n = \operatorname{Suc} m \operatorname{mod} n$$

⟨proof⟩

lemma *mod-Suc-Suc-eq* [mod-simps]:

$$\operatorname{Suc} (\operatorname{Suc} (m \operatorname{mod} n)) \operatorname{mod} n = \operatorname{Suc} (\operatorname{Suc} m) \operatorname{mod} n$$

⟨proof⟩

lemma

$$\operatorname{Suc}\text{-mod-mult-self1} \text{ [simp]: } \operatorname{Suc} (m + k * n) \operatorname{mod} n = \operatorname{Suc} m \operatorname{mod} n$$

$$\text{and } \operatorname{Suc}\text{-mod-mult-self2} \text{ [simp]: } \operatorname{Suc} (m + n * k) \operatorname{mod} n = \operatorname{Suc} m \operatorname{mod} n$$

$$\text{and } \operatorname{Suc}\text{-mod-mult-self3} \text{ [simp]: } \operatorname{Suc} (k * n + m) \operatorname{mod} n = \operatorname{Suc} m \operatorname{mod} n$$

and *Suc-mod-mult-self4* [simp]: $\text{Suc } (n * k + m) \bmod n = \text{Suc } m \bmod n$
 ⟨proof⟩

lemma *Suc-0-mod-eq* [simp]:
 $\text{Suc } 0 \bmod n = \text{of_bool } (n \neq \text{Suc } 0)$
 ⟨proof⟩

lemma *div-mult2-eq*:
 $\langle m \bmod (n * q) = (m \bmod n) \bmod q \rangle$ (is ?Q)
and *mod-mult2-eq*:
 $\langle m \bmod (n * q) = n * (m \bmod n \bmod q) + m \bmod n \rangle$ (is ?R)
for $m \ n \ q :: \text{nat}$
 ⟨proof⟩

lemma *div-le-mono*:
 $m \bmod k \leq n \bmod k$ **if** $m \leq n$ **for** $m \ n \ k :: \text{nat}$
 ⟨proof⟩

Antimonotonicity of (*div*) in second argument

lemma *div-le-mono2*:
 $k \bmod n \leq k \bmod m$ **if** $0 < m$ **and** $m \leq n$ **for** $m \ n \ k :: \text{nat}$
 ⟨proof⟩

lemma *div-le-dividend* [simp]:
 $m \bmod n \leq m$ **for** $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *div-less-dividend* [simp]:
 $m \bmod n < m$ **if** $1 < n$ **and** $0 < m$ **for** $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *div-eq-dividend-iff*:
 $m \bmod n = m \iff n = 1$ **if** $m > 0$ **for** $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *less-mult-imp-div-less*:
 $m \bmod n < i$ **if** $m < i * n$ **for** $m \ n \ i :: \text{nat}$
 ⟨proof⟩

lemma *div-less-iff-less-mult*:
 $\langle m \bmod q < n \iff m < n * q \rangle$ (is $\langle ?P \iff ?Q \rangle$)
if $\langle q > 0 \rangle$ **for** $m \ n \ q :: \text{nat}$
 ⟨proof⟩

lemma *less-eq-div-iff-mult-less-eq*:
 $\langle m \leq n \bmod q \iff m * q \leq n \rangle$ **if** $\langle q > 0 \rangle$ **for** $m \ n \ q :: \text{nat}$
 ⟨proof⟩

lemma *div-Suc*:

$\langle \text{Suc } m \text{ div } n = (\text{if } \text{Suc } m \text{ mod } n = 0 \text{ then } \text{Suc } (m \text{ div } n) \text{ else } m \text{ div } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *mod-Suc*:

$\langle \text{Suc } m \text{ mod } n = (\text{if } \text{Suc } (m \text{ mod } n) = n \text{ then } 0 \text{ else } \text{Suc } (m \text{ mod } n)) \rangle$
 $\langle \text{proof} \rangle$

lemma *Suc-times-mod-eq*:

$\text{Suc } (m * n) \text{ mod } m = 1 \text{ if } \text{Suc } 0 < m$
 $\langle \text{proof} \rangle$

lemma *Suc-times-numeral-mod-eq [simp]*:

$\text{Suc } (\text{numeral } k * n) \text{ mod numeral } k = 1 \text{ if numeral } k \neq (1::\text{nat})$
 $\langle \text{proof} \rangle$

lemma *Suc-div-le-mono [simp]*:

$m \text{ div } n \leq \text{Suc } m \text{ div } n$
 $\langle \text{proof} \rangle$

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]*:

$m \text{ div } \text{Suc } (\text{Suc } (\text{Suc } n)) = m \text{ div } (3 + n)$
 $\langle \text{proof} \rangle$

lemma *mod-Suc-eq-mod-add3 [simp]*:

$m \text{ mod } \text{Suc } (\text{Suc } (\text{Suc } n)) = m \text{ mod } (3 + n)$
 $\langle \text{proof} \rangle$

lemma *Suc-div-eq-add3-div*:

$\text{Suc } (\text{Suc } (\text{Suc } m)) \text{ div } n = (3 + m) \text{ div } n$
 $\langle \text{proof} \rangle$

lemma *Suc-mod-eq-add3-mod*:

$\text{Suc } (\text{Suc } (\text{Suc } m)) \text{ mod } n = (3 + m) \text{ mod } n$
 $\langle \text{proof} \rangle$

lemmas *Suc-div-eq-add3-div-numeral [simp]* =

Suc-div-eq-add3-div [of - numeral v] for v

lemmas *Suc-mod-eq-add3-mod-numeral [simp]* =

Suc-mod-eq-add3-mod [of - numeral v] for v

lemma *(in field-char-0) of-nat-div*:

$\text{of-nat } (m \text{ div } n) = ((\text{of-nat } m - \text{of-nat } (m \text{ mod } n)) / \text{of-nat } n)$
 $\langle \text{proof} \rangle$

An “induction” law for modulus arithmetic.

lemma *mod-induct* [*consumes 3, case-names step*]:

$P\ m$ **if** $P\ n$ **and** $n < p$ **and** $m < p$
and *step*: $\bigwedge n. n < p \implies P\ n \implies P\ (Suc\ n\ mod\ p)$
 $\langle proof \rangle$

lemma *funpow-mod-eq*:

$\langle f\ \sim (m\ mod\ n) \rangle\ x = (f\ \sim m)\ x$ **if** $\langle f\ \sim n \rangle\ x = x$
 $\langle proof \rangle$

lemma *mod-eq-dvd-iff-nat*:

$\langle m\ mod\ q = n\ mod\ q \longleftrightarrow q\ dvd\ m - n \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)
if $\langle m \geq n \rangle$ **for** $m\ n\ q :: nat$
 $\langle proof \rangle$

lemma *mod-eq-iff-dvd-symdiff-nat*:

$\langle m\ mod\ q = n\ mod\ q \longleftrightarrow q\ dvd\ nat\ |int\ m - int\ n| \rangle$
 $\langle proof \rangle$

lemma *mod-eq-nat1E*:

fixes $m\ n\ q :: nat$
assumes $m\ mod\ q = n\ mod\ q$ **and** $m \geq n$
obtains s **where** $m = n + q * s$
 $\langle proof \rangle$

lemma *mod-eq-nat2E*:

fixes $m\ n\ q :: nat$
assumes $m\ mod\ q = n\ mod\ q$ **and** $n \geq m$
obtains s **where** $n = m + q * s$
 $\langle proof \rangle$

lemma *nat-mod-eq-iff*:

$(x :: nat)\ mod\ n = y\ mod\ n \longleftrightarrow (\exists\ q1\ q2. x + n * q1 = y + n * q2)$ (**is** $?lhs = ?rhs$)
 $\langle proof \rangle$

56.5 Division on *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 *int* :: $\{normalization-semidom, idom-modulo\}$
begin

definition *normalize-int* :: $\langle int \Rightarrow int \rangle$

where [*simp*]: $\langle normalize = (abs :: int \Rightarrow int) \rangle$

definition *unit-factor-int* :: $\langle \text{int} \Rightarrow \text{int} \rangle$
where [simp]: $\langle \text{unit-factor} = (\text{sgn} :: \text{int} \Rightarrow \text{int}) \rangle$

definition *divide-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$
where $\langle k \text{ div } l = (\text{sgn } k * \text{sgn } l * \text{int } (\text{nat } |k| \text{ div } \text{nat } |l|))$
 $- \text{of-bool } (l \neq 0 \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$

lemma *divide-int-unfold*:
 $\langle (\text{sgn } k * \text{int } m) \text{ div } (\text{sgn } l * \text{int } n) = (\text{sgn } k * \text{sgn } l * \text{int } (m \text{ div } n))$
 $- \text{of-bool } ((k = 0 \longleftrightarrow m = 0) \wedge l \neq 0 \wedge n \neq 0 \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg n \text{ dvd } m) \rangle$
 $\langle \text{proof} \rangle$

definition *modulo-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$
where $\langle k \text{ mod } l = \text{sgn } k * \text{int } (\text{nat } |k| \text{ mod } \text{nat } |l|) + l * \text{of-bool } (\text{sgn } k \neq \text{sgn } l$
 $\wedge \neg l \text{ dvd } k) \rangle$

lemma *modulo-int-unfold*:
 $\langle (\text{sgn } k * \text{int } m) \text{ mod } (\text{sgn } l * \text{int } n) =$
 $\text{sgn } k * \text{int } (m \text{ mod } (\text{of-bool } (l \neq 0) * n)) + (\text{sgn } l * \text{int } n) * \text{of-bool } ((k = 0$
 $\longleftrightarrow m = 0) \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg n \text{ dvd } m) \rangle$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *of-int-div*: $b \text{ dvd } a \implies \text{of-int } (a \text{ div } b) = (\text{of-int } a / \text{of-int } b :: 'a ::$
 $\text{field-char-0})$
 $\langle \text{proof} \rangle$

56.5.2 Algebraic foundations

lemma *coprime-int-iff* [simp]:
 $\text{coprime } (\text{int } m) (\text{int } n) \longleftrightarrow \text{coprime } m \ n \ (\text{is } ?P \longleftrightarrow ?Q)$
 $\langle \text{proof} \rangle$

lemma *coprime-abs-left-iff* [simp]:
 $\text{coprime } |k| \ l \longleftrightarrow \text{coprime } k \ l \ \text{for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *coprime-abs-right-iff* [simp]:
 $\text{coprime } k \ |l| \longleftrightarrow \text{coprime } k \ l \ \text{for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *coprime-nat-abs-left-iff* [simp]:
 $\text{coprime } (\text{nat } |k|) \ n \longleftrightarrow \text{coprime } k \ (\text{int } n)$
 $\langle \text{proof} \rangle$

lemma *coprime-nat-abs-right-iff* [simp]:
 $\text{coprime } n \ (\text{nat } |k|) \longleftrightarrow \text{coprime } (\text{int } n) \ k$
 ⟨proof⟩

lemma *coprime-common-divisor-int*: $\text{coprime } a \ b \implies x \ \text{dvd } a \implies x \ \text{dvd } b \implies |x| = 1$
for $a \ b :: \text{int}$
 ⟨proof⟩

56.5.3 Basic conversions

lemma *div-abs-eq-div-nat*:
 $|k| \ \text{div } |l| = \text{int } (\text{nat } |k| \ \text{div } \text{nat } |l|)$
 ⟨proof⟩

lemma *div-eq-div-abs*:
 $\langle k \ \text{div } l = \text{sgn } k * \text{sgn } l * (|k| \ \text{div } |l|)$
 $\quad - \text{of_bool } (l \neq 0 \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg l \ \text{dvd } k) \rangle$
for $k \ l :: \text{int}$
 ⟨proof⟩

lemma *div-abs-eq*:
 $\langle |k| \ \text{div } |l| = \text{sgn } k * \text{sgn } l * (k \ \text{div } l + \text{of_bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \ \text{dvd } k)) \rangle$
for $k \ l :: \text{int}$
 ⟨proof⟩

lemma *mod-abs-eq-div-nat*:
 $|k| \ \text{mod } |l| = \text{int } (\text{nat } |k| \ \text{mod } \text{nat } |l|)$
 ⟨proof⟩

lemma *mod-eq-mod-abs*:
 $\langle k \ \text{mod } l = \text{sgn } k * (|k| \ \text{mod } |l|) + l * \text{of_bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \ \text{dvd } k) \rangle$
for $k \ l :: \text{int}$
 ⟨proof⟩

lemma *mod-abs-eq*:
 $\langle |k| \ \text{mod } |l| = \text{sgn } k * (k \ \text{mod } l - l * \text{of_bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \ \text{dvd } k)) \rangle$
for $k \ l :: \text{int}$
 ⟨proof⟩

lemma *div-sgn-abs-cancel*:
fixes $k \ l \ v :: \text{int}$
assumes $v \neq 0$
shows $(\text{sgn } v * |k|) \ \text{div } (\text{sgn } v * |l|) = |k| \ \text{div } |l|$
 ⟨proof⟩

lemma *div-eq-sgn-abs*:
fixes $k \ l \ v :: \text{int}$

assumes $\text{sgn } k = \text{sgn } l$
shows $k \text{ div } l = |k| \text{ div } |l|$
 $\langle \text{proof} \rangle$

lemma *div-dvd-sgn-abs*:
fixes $k \ l :: \text{int}$
assumes $l \text{ dvd } k$
shows $k \text{ div } l = (\text{sgn } k * \text{sgn } l) * (|k| \text{ div } |l|)$
 $\langle \text{proof} \rangle$

lemma *div-noneq-sgn-abs*:
fixes $k \ l :: \text{int}$
assumes $l \neq 0$
assumes $\text{sgn } k \neq \text{sgn } l$
shows $k \text{ div } l = - (|k| \text{ div } |l|) - \text{of_bool } (\neg l \text{ dvd } k)$
 $\langle \text{proof} \rangle$

56.5.4 Euclidean division

instantiation $\text{int} :: \text{unique-euclidean-ring}$
begin

definition *euclidean-size-int* $:: \text{int} \Rightarrow \text{nat}$
where $[\text{simp}]: \text{euclidean-size-int} = (\text{nat} \circ \text{abs} :: \text{int} \Rightarrow \text{nat})$

definition *division-segment-int* $:: \text{int} \Rightarrow \text{int}$
where $\text{division-segment-int } k = (\text{if } k \geq 0 \text{ then } 1 \text{ else } -1)$

lemma *division-segment-eq-sgn*:
 $\text{division-segment } k = \text{sgn } k \text{ if } k \neq 0 \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *abs-division-segment* $[\text{simp}]$:
 $|\text{division-segment } k| = 1 \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *abs-mod-less*:
 $|k \bmod l| < |l| \text{ if } l \neq 0 \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *sgn-mod*:
 $\text{sgn } (k \bmod l) = \text{sgn } l \text{ if } l \neq 0 \neg l \text{ dvd } k \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *euclidean-relation-intI* $[\text{case-names by0 divides euclidean-relation}]$:

$\langle (k \text{ div } l, k \text{ mod } l) = (q, r) \rangle$
if $by0'$: $\langle l = 0 \implies q = 0 \wedge r = k \rangle$
and $divides'$: $\langle l \neq 0 \implies l \text{ dvd } k \implies r = 0 \wedge k = q * l \rangle$
and $euclidean-relation'$: $\langle l \neq 0 \implies \neg l \text{ dvd } k \implies \text{sgn } r = \text{sgn } l$
 $\wedge |r| < |l| \wedge k = q * l + r \rangle$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

56.5.5 Trivial reduction steps

lemma *div-pos-pos-trivial* [simp]:
 $k \text{ div } l = 0$ **if** $k \geq 0$ **and** $k < l$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *mod-pos-pos-trivial* [simp]:
 $k \text{ mod } l = k$ **if** $k \geq 0$ **and** $k < l$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *div-neg-neg-trivial* [simp]:
 $k \text{ div } l = 0$ **if** $k \leq 0$ **and** $l < k$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *mod-neg-neg-trivial* [simp]:
 $k \text{ mod } l = k$ **if** $k \leq 0$ **and** $l < k$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma
div-pos-neg-trivial: $\langle k \text{ div } l = -1 \rangle$ (**is** ?Q)
and *mod-pos-neg-trivial*: $\langle k \text{ mod } l = k + l \rangle$ (**is** ?R)
if $\langle 0 < k \rangle$ **and** $\langle k + l \leq 0 \rangle$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

There is neither *div-neg-pos-trivial* nor *mod-neg-pos-trivial* because $0 \text{ div } l = 0$ would supersede it.

56.5.6 More uniqueness rules

lemma
fixes $a \ b \ q \ r :: \text{int}$
assumes $\langle a = b * q + r \rangle \langle 0 \leq r \rangle \langle r < b \rangle$
shows *int-div-pos-eq*:
 $\langle a \text{ div } b = q \rangle$ (**is** ?Q)
and *int-mod-pos-eq*:
 $\langle a \text{ mod } b = r \rangle$ (**is** ?R)
 $\langle \text{proof} \rangle$

lemma *int-div-neg-eq*:
 $\langle a \text{ div } b = q \rangle$ **if** $\langle a = b * q + r \rangle \langle r \leq 0 \rangle \langle b < r \rangle$ **for** $a \ b \ q \ r :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *int-mod-neg-eq*:

$\langle a \bmod b = r \rangle$ **if** $\langle a = b * q + r \rangle \langle r \leq 0 \rangle \langle b < r \rangle$ **for** $a \ b \ q \ r :: \text{int}$
 $\langle \text{proof} \rangle$

56.5.7 Laws for unary minus

lemma *zmod-zminus1-not-zero*:

fixes $k \ l :: \text{int}$
shows $-k \bmod l \neq 0 \implies k \bmod l \neq 0$
 $\langle \text{proof} \rangle$

lemma *zmod-zminus2-not-zero*:

fixes $k \ l :: \text{int}$
shows $k \bmod -l \neq 0 \implies k \bmod l \neq 0$
 $\langle \text{proof} \rangle$

lemma *zdiv-zminus1-eq-if*:

$\langle (-a) \text{ div } b = (\text{if } a \bmod b = 0 \text{ then } -(a \text{ div } b) \text{ else } -(a \text{ div } b) - 1) \rangle$
if $\langle b \neq 0 \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *zdiv-zminus2-eq-if*:

$\langle a \text{ div } (-b) = (\text{if } a \bmod b = 0 \text{ then } -(a \text{ div } b) \text{ else } -(a \text{ div } b) - 1) \rangle$
if $\langle b \neq 0 \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *zmod-zminus1-eq-if*:

$\langle (-a) \bmod b = (\text{if } a \bmod b = 0 \text{ then } 0 \text{ else } b - (a \bmod b)) \rangle$
for $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *zmod-zminus2-eq-if*:

$\langle a \bmod (-b) = (\text{if } a \bmod b = 0 \text{ then } 0 \text{ else } (a \bmod b) - b) \rangle$
for $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

56.5.8 Borders

lemma *pos-mod-bound* [simp]:

$k \bmod l < l$ **if** $l > 0$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *neg-mod-bound* [simp]:

$l < k \bmod l$ **if** $l < 0$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *pos-mod-sign* [simp]:

$0 \leq k \bmod l$ **if** $l > 0$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *neg-mod-sign* [*simp*]:
 $k \bmod l \leq 0$ if $l < 0$ for $k \ l :: \text{int}$
 ⟨*proof*⟩

56.5.9 Splitting Rules for div and mod

lemma *split-zdiv*:
 $\langle P \ (n \ \text{div} \ k) \longleftrightarrow$
 $(k = 0 \longrightarrow P \ 0) \wedge$
 $(0 < k \longrightarrow (\forall i \ j. \ 0 \leq j \wedge j < k \wedge n = k * i + j \longrightarrow P \ i)) \wedge$
 $(k < 0 \longrightarrow (\forall i \ j. \ k < j \wedge j \leq 0 \wedge n = k * i + j \longrightarrow P \ i)) \rangle$ (is ?div)
and *split-zmod*:
 $\langle Q \ (n \ \text{mod} \ k) \longleftrightarrow$
 $(k = 0 \longrightarrow Q \ n) \wedge$
 $(0 < k \longrightarrow (\forall i \ j. \ 0 \leq j \wedge j < k \wedge n = k * i + j \longrightarrow Q \ j)) \wedge$
 $(k < 0 \longrightarrow (\forall i \ j. \ k < j \wedge j \leq 0 \wedge n = k * i + j \longrightarrow Q \ j)) \rangle$ (is ?mod)
 for $n \ k :: \text{int}$
 ⟨*proof*⟩

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*:
 $i \ \text{div} \ k = 0 \longleftrightarrow k = 0 \vee 0 \leq i \wedge i < k \vee i \leq 0 \wedge k < i$ (is ?L = ?R)
 for $i \ k :: \text{int}$
 ⟨*proof*⟩

lemma *zmod-trivial-iff*:
 fixes $i \ k :: \text{int}$
 shows $i \ \text{mod} \ k = i \longleftrightarrow k = 0 \vee 0 \leq i \wedge i < k \vee i \leq 0 \wedge k < i$
 ⟨*proof*⟩

56.5.10 Algebraic rewrites

lemma *zdiv-zmult2-eq*: $\langle a \ \text{div} \ (b * c) = (a \ \text{div} \ b) \ \text{div} \ c \rangle$ (is ?Q)
and *zmod-zmult2-eq*: $\langle a \ \text{mod} \ (b * c) = b * (a \ \text{div} \ b \ \text{mod} \ c) + a \ \text{mod} \ b \rangle$ (is ?P)
 if $\langle c \geq 0 \rangle$ for $a \ b \ c :: \text{int}$
 ⟨*proof*⟩

lemma *zdiv-zmult2-eq'*:
 $\langle k \ \text{div} \ (l * j) = ((\text{sgn} \ j * k) \ \text{div} \ l) \ \text{div} \ |j| \rangle$ for $k \ l \ j :: \text{int}$
 ⟨*proof*⟩

lemma *half-nonnegative-int-iff* [*simp*]:
 $\langle k \ \text{div} \ 2 \geq 0 \longleftrightarrow k \geq 0 \rangle$ for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *half-negative-int-iff* [*simp*]:
 $\langle k \text{ div } 2 < 0 \iff k < 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

56.5.11 Distributive laws for conversions.

lemma *zdiv-int*:
 $\langle \text{int } (m \text{ div } n) = \text{int } m \text{ div int } n \rangle$
 $\langle \text{proof} \rangle$

lemma *zmod-int*:
 $\langle \text{int } (m \text{ mod } n) = \text{int } m \text{ mod int } n \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-div-distrib*:
 $\langle \text{nat } (x \text{ div } y) = \text{nat } x \text{ div nat } y \rangle$ **if** $\langle 0 \leq x \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-div-distrib'*:
 $\langle \text{nat } (x \text{ div } y) = \text{nat } x \text{ div nat } y \rangle$ **if** $\langle 0 \leq y \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-mod-distrib*: — Fails if $y < 0$: the LHS collapses to $(\text{nat } z)$ but the RHS doesn't
 $\langle \text{nat } (x \text{ mod } y) = \text{nat } x \text{ mod nat } y \rangle$ **if** $\langle 0 \leq x \rangle \langle 0 \leq y \rangle$
 $\langle \text{proof} \rangle$

56.5.12 Monotonicity in the First Argument (Dividend)

lemma *zdiv-mono1*:
 $\langle a \text{ div } b \leq a' \text{ div } b \rangle$
if $\langle a \leq a' \rangle \langle 0 < b \rangle$
for $a \ b \ b' :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *zdiv-mono1-neg*:
 $\langle a' \text{ div } b \leq a \text{ div } b \rangle$
if $\langle a \leq a' \rangle \langle b < 0 \rangle$
for $a \ a' \ b :: \text{int}$
 $\langle \text{proof} \rangle$

56.5.13 Monotonicity in the Second Argument (Divisor)

lemma *zdiv-mono2*:
 $\langle a \text{ div } b \leq a \text{ div } b' \rangle$ **if** $\langle 0 \leq a \rangle \langle 0 < b' \rangle \langle b' \leq b \rangle$ **for** $a \ b \ b' :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *zdiv-mono2-neg*:

$\langle a \text{ div } b' \leq a \text{ div } b \rangle$ **if** $\langle a < 0 \rangle \langle 0 < b' \rangle \langle b' \leq b \rangle$ **for** $a \ b \ b' :: \text{int}$
 $\langle \text{proof} \rangle$

56.5.14 Quotients of Signs

lemma *div-eq-minus1*:

$\langle 0 < b \implies -1 \text{ div } b = -1 \rangle$ **for** $b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *zmod-minus1*:

$\langle 0 < b \implies -1 \text{ mod } b = b - 1 \rangle$ **for** $b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *minus-mod-int-eq*:

$\langle -k \text{ mod } l = l - 1 - (k - 1) \text{ mod } l \rangle$ **if** $\langle l \geq 0 \rangle$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *div-neg-pos-less0*:

$\langle a \text{ div } b < 0 \rangle$ **if** $\langle a < 0 \rangle \langle 0 < b \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *div-nonneg-neg-le0*:

$\langle a \text{ div } b \leq 0 \rangle$ **if** $\langle 0 \leq a \rangle \langle b < 0 \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *div-nonpos-pos-le0*:

$\langle a \text{ div } b \leq 0 \rangle$ **if** $\langle a \leq 0 \rangle \langle 0 < b \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

Now for some equivalences of the form $a \text{ div } b \geq 0 \iff \dots$ conditional upon the sign of a or b . There are many more. They should all be simple rules unless that causes too much search.

lemma *pos-imp-zdiv-nonneg-iff*:

$\langle 0 \leq a \text{ div } b \iff 0 \leq a \rangle$
if $\langle 0 < b \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *neg-imp-zdiv-nonneg-iff*:

$\langle 0 \leq a \text{ div } b \iff a \leq 0 \rangle$ **if** $\langle b < 0 \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *pos-imp-zdiv-pos-iff*:

$\langle 0 < (i :: \text{int}) \text{ div } k \iff k \leq i \rangle$ **if** $\langle 0 < k \rangle$ **for** $i \ k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *pos-imp-zdiv-neg-iff*:

$\langle a \text{ div } b < 0 \iff a < 0 \rangle$ **if** $\langle 0 < b \rangle$ **for** $a \ b :: \text{int}$
 — But not $(a \text{ div } b \leq 0) = (a \leq 0)$; consider $a = 1, b = 2$ when $a \text{ div } b = 0$.
 $\langle \text{proof} \rangle$

lemma *neg-imp-zdiv-neg-iff*:

— But not $(a \text{ div } b \leq 0) = (0 \leq a)$; consider $a = -1$, $b = -2$ when $a \text{ div } b = 0$.

$\langle a \text{ div } b < 0 \iff 0 < a \rangle$ **if** $\langle b < 0 \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *nonneg1-imp-zdiv-pos-iff*:

$\langle a \text{ div } b > 0 \iff a \geq b \wedge b > 0 \rangle$ **if** $\langle 0 \leq a \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *zmod-le-nonneg-dividend*:

$\langle m \bmod k \leq m \rangle$ **if** $\langle (m :: \text{int}) \geq 0 \rangle$ **for** $m \ k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *sgn-div-eq-sgn-mult*:

$\langle \text{sgn } (k \text{ div } l) = \text{of_bool } (k \text{ div } l \neq 0) * \text{sgn } (k * l) \rangle$
for $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

56.5.15 Further properties

lemma *div-int-pos-iff*:

$k \text{ div } l \geq 0 \iff k = 0 \vee l = 0 \vee k \geq 0 \wedge l \geq 0$
 $\vee k < 0 \wedge l < 0$
for $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *mod-int-pos-iff*:

$\langle k \bmod l \geq 0 \iff l \text{ dvd } k \vee l = 0 \wedge k \geq 0 \vee l > 0 \rangle$
for $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *abs-div*:

$\langle |x \text{ div } y| = |x| \text{ div } |y| \rangle$ **if** $\langle y \text{ dvd } x \rangle$ **for** $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *int-power-div-base*:

$\langle k \wedge^m \text{ div } k = k \wedge^{(m - \text{Suc } 0)} \rangle$ **if** $\langle 0 < m \rangle \langle 0 < k \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *int-div-less-self*:

$\langle x \text{ div } k < x \rangle$ **if** $\langle 0 < x \rangle \langle 1 < k \rangle$ **for** $x \ k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *int-div-le-self*:

$\langle x \text{ div } k \leq x \rangle$ **if** $\langle 0 < x \rangle$ **for** $x \ k :: \text{int}$
 $\langle \text{proof} \rangle$

56.5.16 Computing div and mod by shifting**lemma** *div-pos-geq*:
$$\langle k \text{ div } l = (k - l) \text{ div } l + 1 \rangle \text{ if } \langle 0 < l \rangle \langle l \leq k \rangle \text{ for } k \ l :: \text{int}$$

<proof>

lemma *mod-pos-geq*:
$$\langle k \text{ mod } l = (k - l) \text{ mod } l \rangle \text{ if } \langle 0 < l \rangle \langle l \leq k \rangle \text{ for } k \ l :: \text{int}$$

<proof>

lemma *pos-zdiv-mult-2*: $\langle (1 + 2 * b) \text{ div } (2 * a) = b \text{ div } a \rangle$ (**is** ?*Q*)

and *pos-zmod-mult-2*: $\langle (1 + 2 * b) \text{ mod } (2 * a) = 1 + 2 * (b \text{ mod } a) \rangle$ (**is** ?*R*)

if $\langle 0 \leq a \rangle$ **for** $a \ b :: \text{int}$

<proof>

lemma *neg-zdiv-mult-2*: $\langle (1 + 2 * b) \text{ div } (2 * a) = (b + 1) \text{ div } a \rangle$ (**is** ?*Q*)

and *neg-zmod-mult-2*: $\langle (1 + 2 * b) \text{ mod } (2 * a) = 2 * ((b + 1) \text{ mod } a) - 1 \rangle$

(**is** ?*R*)

if $\langle a \leq 0 \rangle$ **for** $a \ b :: \text{int}$

<proof>

lemma *zdiv-numeral-Bit0* [*simp*]:
$$\langle \text{numeral } (\text{Num.Bit0 } v) \text{ div numeral } (\text{Num.Bit0 } w) =$$

$$\text{numeral } v \text{ div (numeral } w :: \text{int}) \rangle$$

<proof>

lemma *zdiv-numeral-Bit1* [*simp*]:
$$\langle \text{numeral } (\text{Num.Bit1 } v) \text{ div numeral } (\text{Num.Bit0 } w) =$$

$$(\text{numeral } v \text{ div (numeral } w :: \text{int})) \rangle$$

<proof>

lemma *zmod-numeral-Bit0* [*simp*]:
$$\langle \text{numeral } (\text{Num.Bit0 } v) \text{ mod numeral } (\text{Num.Bit0 } w) =$$

$$(2 :: \text{int}) * (\text{numeral } v \text{ mod numeral } w) \rangle$$

<proof>

lemma *zmod-numeral-Bit1* [*simp*]:
$$\langle \text{numeral } (\text{Num.Bit1 } v) \text{ mod numeral } (\text{Num.Bit0 } w) =$$

$$2 * (\text{numeral } v \text{ mod numeral } w) + (1 :: \text{int}) \rangle$$

<proof>

56.6 Code generation**context****begin****qualified definition** *divmod-nat* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \times \text{nat}$ **where** *divmod-nat* $m \ n = (m \text{ div } n, m \text{ mod } n)$ **qualified lemma** *divmod-nat-if* [*code*]:

```

    divmod-nat m n = (if n = 0  $\vee$  m < n then (0, m) else
      let (q, r) = divmod-nat (m - n) n in (Suc q, r))
  <proof> lemma [code]:
    m div n = fst (divmod-nat m n)
    m mod n = snd (divmod-nat m n)
  <proof>

end

code-identifier
  code-module Euclidean-Rings  $\rightarrow$  (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:  $\langle a \bmod 2 = \text{of\_bool } (\neg 2 \text{ dvd } a) \rangle$ 
  and odd-one [simp]:  $\langle \neg 2 \text{ dvd } 1 \rangle$ 
  and even-half-succ-eq [simp]:  $\langle 2 \text{ dvd } a \implies (1 + a) \text{ div } 2 = a \text{ div } 2 \rangle$ 
begin

abbreviation even :: 'a  $\Rightarrow$  bool
  where  $\langle \text{even } a \equiv 2 \text{ dvd } a \rangle$ 

abbreviation odd :: 'a  $\Rightarrow$  bool
  where  $\langle \text{odd } a \equiv \neg 2 \text{ dvd } a \rangle$ 

end

class ring-parity = ring + semiring-parity
begin

subclass comm-ring-1 <proof>

end

instance nat :: semiring-parity
  <proof>

instance int :: ring-parity
  <proof>

```

context *semiring-parity*
begin

lemma *evenE* [*elim?*]:
assumes $\langle \text{even } a \rangle$
obtains *b* **where** $\langle a = 2 * b \rangle$
 $\langle \text{proof} \rangle$

lemma *oddE* [*elim?*]:
assumes $\langle \text{odd } a \rangle$
obtains *b* **where** $\langle a = 2 * b + 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *of-bool-odd-eq-mod-2*:
 $\langle \text{of-bool } (\text{odd } a) = a \bmod 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *odd-of-bool-self* [*simp*]:
 $\langle \text{odd } (\text{of-bool } p) \longleftrightarrow p \rangle$
 $\langle \text{proof} \rangle$

lemma *not-mod-2-eq-0-eq-1* [*simp*]:
 $\langle a \bmod 2 \neq 0 \longleftrightarrow a \bmod 2 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *not-mod-2-eq-1-eq-0* [*simp*]:
 $\langle a \bmod 2 \neq 1 \longleftrightarrow a \bmod 2 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *even-iff-mod-2-eq-zero*:
 $\langle 2 \text{ dvd } a \longleftrightarrow a \bmod 2 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *odd-iff-mod-2-eq-one*:
 $\langle \neg 2 \text{ dvd } a \longleftrightarrow a \bmod 2 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *even-mod-2-iff* [*simp*]:
 $\langle \text{even } (a \bmod 2) \longleftrightarrow \text{even } a \rangle$
 $\langle \text{proof} \rangle$

lemma *mod2-eq-if*:
 $a \bmod 2 = (\text{if even } a \text{ then } 0 \text{ else } 1)$
 $\langle \text{proof} \rangle$

lemma *zero-mod-two-eq-zero* [*simp*]:
 $\langle 0 \bmod 2 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *one-mod-two-eq-one* [simp]:

$\langle 1 \bmod 2 = 1 \rangle$

$\langle \text{proof} \rangle$

lemma *parity-cases* [case-names even odd]:

assumes $\langle \text{even } a \implies a \bmod 2 = 0 \implies P \rangle$

assumes $\langle \text{odd } a \implies a \bmod 2 = 1 \implies P \rangle$

shows P

$\langle \text{proof} \rangle$

lemma *even-zero* [simp]:

$\langle \text{even } 0 \rangle$

$\langle \text{proof} \rangle$

lemma *odd-even-add*:

$\text{even } (a + b) \text{ if odd } a \text{ and odd } b$

$\langle \text{proof} \rangle$

lemma *even-add* [simp]:

$\text{even } (a + b) \longleftrightarrow (\text{even } a \longleftrightarrow \text{even } b)$

$\langle \text{proof} \rangle$

lemma *odd-add* [simp]:

$\text{odd } (a + b) \longleftrightarrow \neg (\text{odd } a \longleftrightarrow \text{odd } b)$

$\langle \text{proof} \rangle$

lemma *even-plus-one-iff* [simp]:

$\text{even } (a + 1) \longleftrightarrow \text{odd } a$

$\langle \text{proof} \rangle$

lemma *even-mult-iff* [simp]:

$\text{even } (a * b) \longleftrightarrow \text{even } a \vee \text{even } b \text{ (is } ?P \longleftrightarrow ?Q)$

$\langle \text{proof} \rangle$

lemma *even-numeral* [simp]: $\text{even } (\text{numeral } (\text{Num.Bit0 } n))$

$\langle \text{proof} \rangle$

lemma *odd-numeral* [simp]: $\text{odd } (\text{numeral } (\text{Num.Bit1 } n))$

$\langle \text{proof} \rangle$

lemma *odd-numeral-BitM* [simp]:

$\langle \text{odd } (\text{numeral } (\text{Num.BitM } w)) \rangle$

$\langle \text{proof} \rangle$

lemma *even-power* [simp]: $\text{even } (a \wedge n) \longleftrightarrow \text{even } a \wedge n > 0$

$\langle \text{proof} \rangle$

lemma *even-prod-iff*:

$\langle \text{even } (\text{prod } f \ A) \longleftrightarrow (\exists a \in A. \text{ even } (f \ a)) \rangle \text{ if } \langle \text{finite } A \rangle$
 $\langle \text{proof} \rangle$

lemma *even-half-maybe-succ-eq* [simp]:
 $\langle \text{even } a \implies (\text{of-bool } b + a) \text{ div } 2 = a \text{ div } 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *even-half-maybe-succ'-eq* [simp]:
 $\langle \text{even } a \implies (b \text{ mod } 2 + a) \text{ div } 2 = a \text{ div } 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-eq-sum-exp*:
 $\langle 2^n - 1 = (\sum m \in \{q. q < n\}. 2^m) \rangle$
 $\langle \text{proof} \rangle$

lemma (*in -*) *mask-eq-sum-exp-nat*:
 $\langle 2^n - \text{Suc } 0 = (\sum m \in \{q. q < n\}. 2^m) \rangle$
 $\langle \text{proof} \rangle$

end

context *ring-parity*
begin

lemma *even-minus*:
 $\text{even } (- \ a) \longleftrightarrow \text{even } a$
 $\langle \text{proof} \rangle$

lemma *even-diff* [simp]:
 $\text{even } (a - b) \longleftrightarrow \text{even } (a + b)$
 $\langle \text{proof} \rangle$

end

57.2 Instance for *nat*

lemma *even-Suc-Suc-iff* [simp]:
 $\text{even } (\text{Suc } (\text{Suc } n)) \longleftrightarrow \text{even } n$
 $\langle \text{proof} \rangle$

lemma *even-Suc* [simp]: $\text{even } (\text{Suc } n) \longleftrightarrow \text{odd } n$
 $\langle \text{proof} \rangle$

lemma *even-diff-nat* [simp]:
 $\text{even } (m - n) \longleftrightarrow m < n \vee \text{even } (m + n) \text{ for } m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *odd-pos*:
 $\text{odd } n \implies 0 < n \text{ for } n :: \text{nat}$

⟨proof⟩

lemma *Suc-double-not-eq-double*:

$Suc (2 * m) \neq 2 * n$

⟨proof⟩

lemma *double-not-eq-Suc-double*:

$2 * m \neq Suc (2 * n)$

⟨proof⟩

lemma *odd-Suc-minus-one* [simp]: $odd\ n \implies Suc\ (n - Suc\ 0) = n$

⟨proof⟩

lemma *even-Suc-div-two* [simp]:

$even\ n \implies Suc\ n\ div\ 2 = n\ div\ 2$

⟨proof⟩

lemma *odd-Suc-div-two* [simp]:

$odd\ n \implies Suc\ n\ div\ 2 = Suc\ (n\ div\ 2)$

⟨proof⟩

lemma *odd-two-times-div-two-nat* [simp]:

assumes $odd\ n$

shows $2 * (n\ div\ 2) = n - (1 :: nat)$

⟨proof⟩

lemma *not-mod2-eq-Suc-0-eq-0* [simp]:

$n\ mod\ 2 \neq Suc\ 0 \longleftrightarrow n\ mod\ 2 = 0$

⟨proof⟩

lemma *odd-card-imp-not-empty*:

$\langle A \neq \{\} \rangle$ **if** $\langle odd\ (card\ A) \rangle$

⟨proof⟩

lemma *nat-induct2* [case-names 0 1 step]:

assumes $P\ 0\ P\ 1$ **and** *step*: $\bigwedge n::nat. P\ n \implies P\ (n + 2)$

shows $P\ n$

⟨proof⟩

lemma *mod-double-nat*:

$\langle n\ mod\ (2 * m) = n\ mod\ m \vee n\ mod\ (2 * m) = n\ mod\ m + m \rangle$

for $m\ n :: nat$

⟨proof⟩

context *semiring-parity*

begin

lemma *even-sum-iff*:

$\langle even\ (sum\ f\ A) \longleftrightarrow even\ (card\ \{a \in A. odd\ (f\ a)\}) \rangle$ **if** $\langle finite\ A \rangle$

⟨proof⟩

lemma *even-mask-iff* [simp]:

⟨even ($2 \wedge n - 1$) $\longleftrightarrow n = 0$ ⟩

⟨proof⟩

lemma *even-of-nat-iff* [simp]:

even (of-nat n) \longleftrightarrow even n

⟨proof⟩

end

57.3 Parity and powers

context *ring-1*

begin

lemma *power-minus-even* [simp]: even $n \implies (-a) \wedge n = a \wedge n$

⟨proof⟩

lemma *power-minus-odd* [simp]: odd $n \implies (-a) \wedge n = -(a \wedge n)$

⟨proof⟩

lemma *uminus-power-if*:

$(-a) \wedge n = (\text{if even } n \text{ then } a \wedge n \text{ else } -(a \wedge n))$

⟨proof⟩

lemma *neg-one-even-power* [simp]: even $n \implies (-1) \wedge n = 1$

⟨proof⟩

lemma *neg-one-odd-power* [simp]: odd $n \implies (-1) \wedge n = -1$

⟨proof⟩

lemma *neg-one-power-add-eq-neg-one-power-diff*: $k \leq n \implies (-1) \wedge (n + k) = (-1) \wedge (n - k)$

⟨proof⟩

lemma *minus-one-power-iff*: $(-1) \wedge n = (\text{if even } n \text{ then } 1 \text{ else } -1)$

⟨proof⟩

end

context *linordered-idom*

begin

lemma *zero-le-even-power*: even $n \implies 0 \leq a \wedge n$

⟨proof⟩

lemma *zero-le-odd-power*: odd $n \implies 0 \leq a \wedge n \longleftrightarrow 0 \leq a$

$\langle \text{proof} \rangle$

lemma *zero-le-power-eq*: $0 \leq a \wedge n \longleftrightarrow \text{even } n \vee \text{odd } n \wedge 0 \leq a$
 $\langle \text{proof} \rangle$

lemma *zero-less-power-eq*: $0 < a \wedge n \longleftrightarrow n = 0 \vee \text{even } n \wedge a \neq 0 \vee \text{odd } n \wedge 0 < a$
 $\langle \text{proof} \rangle$

lemma *power-less-zero-eq* [simp]: $a \wedge n < 0 \longleftrightarrow \text{odd } n \wedge a < 0$
 $\langle \text{proof} \rangle$

lemma *power-le-zero-eq*: $a \wedge n \leq 0 \longleftrightarrow n > 0 \wedge (\text{odd } n \wedge a \leq 0 \vee \text{even } n \wedge a = 0)$
 $\langle \text{proof} \rangle$

lemma *power-even-abs*: $\text{even } n \implies |a| \wedge n = a \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-mono-even*:
 assumes *even* n and $|a| \leq |b|$
 shows $a \wedge n \leq b \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-mono-odd*:
 assumes *odd* n and $a \leq b$
 shows $a \wedge n \leq b \wedge n$
 $\langle \text{proof} \rangle$

Simplify, when the exponent is a numeral

lemma *zero-le-power-eq-numeral* [simp]:
 $0 \leq a \wedge \text{numeral } w \longleftrightarrow \text{even } (\text{numeral } w :: \text{nat}) \vee \text{odd } (\text{numeral } w :: \text{nat}) \wedge 0 \leq a$
 $\langle \text{proof} \rangle$

lemma *zero-less-power-eq-numeral* [simp]:
 $0 < a \wedge \text{numeral } w \longleftrightarrow$
 $\text{numeral } w = (0 :: \text{nat}) \vee$
 $\text{even } (\text{numeral } w :: \text{nat}) \wedge a \neq 0 \vee$
 $\text{odd } (\text{numeral } w :: \text{nat}) \wedge 0 < a$
 $\langle \text{proof} \rangle$

lemma *power-le-zero-eq-numeral* [simp]:
 $a \wedge \text{numeral } w \leq 0 \longleftrightarrow$
 $(0 :: \text{nat}) < \text{numeral } w \wedge$
 $(\text{odd } (\text{numeral } w :: \text{nat}) \wedge a \leq 0 \vee \text{even } (\text{numeral } w :: \text{nat}) \wedge a = 0)$
 $\langle \text{proof} \rangle$

lemma *power-less-zero-eq-numeral* [simp]:

$a \wedge \text{numeral } w < 0 \longleftrightarrow \text{odd } (\text{numeral } w :: \text{nat}) \wedge a < 0$
 $\langle \text{proof} \rangle$

lemma *power-even-abs-numeral* [simp]:
 $\text{even } (\text{numeral } w :: \text{nat}) \implies |a| \wedge \text{numeral } w = a \wedge \text{numeral } w$
 $\langle \text{proof} \rangle$

end

57.4 Instance for *int*

lemma *even-diff-iff*:
 $\text{even } (k - l) \longleftrightarrow \text{even } (k + l) \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *even-abs-add-iff*:
 $\text{even } (|k| + l) \longleftrightarrow \text{even } (k + l) \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *even-add-abs-iff*:
 $\text{even } (k + |l|) \longleftrightarrow \text{even } (k + l) \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *even-nat-iff*: $0 \leq k \implies \text{even } (\text{nat } k) \longleftrightarrow \text{even } k$
 $\langle \text{proof} \rangle$

context
assumes *SORT-CONSTRAINT*('a::division-ring)
begin

lemma *power-int-minus-left*:
 $\text{power-int } (-a :: 'a) \ n = (\text{if even } n \text{ then power-int } a \ n \text{ else } -\text{power-int } a \ n)$
 $\langle \text{proof} \rangle$

lemma *power-int-minus-left-even* [simp]: $\text{even } n \implies \text{power-int } (-a :: 'a) \ n = \text{power-int } a \ n$
 $\langle \text{proof} \rangle$

lemma *power-int-minus-left-odd* [simp]: $\text{odd } n \implies \text{power-int } (-a :: 'a) \ n = -\text{power-int } a \ n$
 $\langle \text{proof} \rangle$

lemma *power-int-minus-left-distrib*:
 $\text{NO-MATCH } (-1) \ x \implies \text{power-int } (-a :: 'a) \ n = \text{power-int } (-1) \ n * \text{power-int } a \ n$
 $\langle \text{proof} \rangle$

lemma *power-int-minus-one-minus*: $\text{power-int } (-1 :: 'a) \ (-n) = \text{power-int } (-1) \ n$

$\langle \text{proof} \rangle$

lemma *power-int-minus-one-diff-commute*: $\text{power-int } (-1 :: 'a) (a - b) = \text{power-int } (-1) (b - a)$
 $\langle \text{proof} \rangle$

lemma *power-int-minus-one-mult-self* [simp]:
 $\text{power-int } (-1 :: 'a) m * \text{power-int } (-1) m = 1$
 $\langle \text{proof} \rangle$

lemma *power-int-minus-one-mult-self'* [simp]:
 $\text{power-int } (-1 :: 'a) m * (\text{power-int } (-1) m * b) = b$
 $\langle \text{proof} \rangle$

end

57.5 Special case: euclidean rings structurally containing the natural numbers

class *linordered-euclidean-semiring* = *discrete-linordered-semidom* + *unique-euclidean-semiring*
 +
assumes *of-nat-div*: $\text{of-nat } (m \text{ div } n) = \text{of-nat } m \text{ div } \text{of-nat } n$
and *division-segment-of-nat* [simp]: $\text{division-segment } (\text{of-nat } n) = 1$
and *division-segment-euclidean-size* [simp]: $\text{division-segment } a * \text{of-nat } (\text{euclidean-size } a) = a$
begin

lemma *division-segment-eq-iff*:
 $a = b$ **if** $\text{division-segment } a = \text{division-segment } b$
and $\text{euclidean-size } a = \text{euclidean-size } b$
 $\langle \text{proof} \rangle$

lemma *euclidean-size-of-nat* [simp]:
 $\text{euclidean-size } (\text{of-nat } n) = n$
 $\langle \text{proof} \rangle$

lemma *of-nat-euclidean-size*:
 $\text{of-nat } (\text{euclidean-size } a) = a \text{ div } \text{division-segment } a$
 $\langle \text{proof} \rangle$

lemma *division-segment-1* [simp]:
 $\text{division-segment } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *division-segment-numeral* [simp]:
 $\text{division-segment } (\text{numeral } k) = 1$
 $\langle \text{proof} \rangle$

lemma *euclidean-size-1* [simp]:

euclidean-size 1 = 1
 ⟨proof⟩

lemma *euclidean-size-numeral* [simp]:
euclidean-size (numeral *k*) = numeral *k*
 ⟨proof⟩

lemma *of-nat-dvd-iff*:
of-nat m dvd of-nat n \longleftrightarrow *m dvd n* (is ?*P* \longleftrightarrow ?*Q*)
 ⟨proof⟩

lemma *of-nat-mod*:
of-nat (m mod n) = *of-nat m mod of-nat n*
 ⟨proof⟩

lemma *one-div-two-eq-zero* [simp]:
 1 div 2 = 0
 ⟨proof⟩

lemma *one-mod-2-pow-eq* [simp]:
 1 mod (2 \wedge *n*) = of-bool (*n* > 0)
 ⟨proof⟩

lemma *one-div-2-pow-eq* [simp]:
 1 div (2 \wedge *n*) = of-bool (*n* = 0)
 ⟨proof⟩

lemma *div-mult2-eq'*:
 ⟨*a div (of-nat m * of-nat n)* = *a div of-nat m div of-nat n*⟩
 ⟨proof⟩

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⟩

lemma *div-mult2-numeral-eq*:
a div numeral k div numeral l = *a div numeral (k * l)* (is ?*A* = ?*B*)
 ⟨proof⟩

lemma *numeral-Bit0-div-2*:
numeral (num.Bit0 n) div 2 = numeral *n*
 ⟨proof⟩

lemma *numeral-Bit1-div-2*:
numeral (num.Bit1 n) div 2 = numeral *n*
 ⟨proof⟩

lemma *exp-mod-exp*:

$\langle 2^{\wedge m} \bmod 2^{\wedge n} = \text{of-bool } (m < n) * 2^{\wedge m} \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-mod-exp*:
 $\langle (2^{\wedge n} - 1) \bmod 2^{\wedge m} = 2^{\wedge \min m n} - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *of-bool-half-eq-0* [simp]:
 $\langle \text{of-bool } b \text{ div } 2 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-mod-double*:
 $\langle \text{of-nat } n \bmod (2 * \text{of-nat } m) = \text{of-nat } n \bmod \text{of-nat } m \vee \text{of-nat } n \bmod (2 * \text{of-nat } m) = \text{of-nat } n \bmod \text{of-nat } m + \text{of-nat } m \rangle$
 $\langle \text{proof} \rangle$

end

instance *nat* :: *linordered-euclidean-semiring*
 $\langle \text{proof} \rangle$

instance *int* :: *linordered-euclidean-semiring*
 $\langle \text{proof} \rangle$

context *linordered-euclidean-semiring*
begin

subclass *semiring-parity*
 $\langle \text{proof} \rangle$

lemma *even-succ-div-two* [simp]:
 $\text{even } a \implies (a + 1) \text{ div } 2 = a \text{ div } 2$
 $\langle \text{proof} \rangle$

lemma *odd-succ-div-two* [simp]:
 $\text{odd } a \implies (a + 1) \text{ div } 2 = a \text{ div } 2 + 1$
 $\langle \text{proof} \rangle$

lemma *even-two-times-div-two*:
 $\text{even } a \implies 2 * (a \text{ div } 2) = a$
 $\langle \text{proof} \rangle$

lemma *odd-two-times-div-two-succ* [simp]:
 $\text{odd } a \implies 2 * (a \text{ div } 2) + 1 = a$
 $\langle \text{proof} \rangle$

lemma *coprime-left-2-iff-odd* [simp]:
 $\text{coprime } 2 a \iff \text{odd } a$
 $\langle \text{proof} \rangle$

lemma *coprime-right-2-iff-odd* [simp]:

$\text{coprime } a \ 2 \longleftrightarrow \text{odd } a$

$\langle \text{proof} \rangle$

end

lemma *minus-1-mod-2-eq* [simp]:

$\langle -1 \bmod 2 = (1::\text{int}) \rangle$

$\langle \text{proof} \rangle$

lemma *minus-1-div-2-eq* [simp]:

$-1 \text{ div } 2 = -(1::\text{int})$

$\langle \text{proof} \rangle$

context *linordered-euclidean-semiring*

begin

lemma *even-decr-exp-div-exp-iff'*:

$\langle \text{even } ((2^m - 1) \text{ div } 2^n) \longleftrightarrow m \leq n \rangle$

$\langle \text{proof} \rangle$

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* :: $\langle \text{num} \Rightarrow \text{num} \Rightarrow 'a \times 'a \rangle$

and *divmod-step* :: $\langle 'a \Rightarrow 'a \times 'a \Rightarrow 'a \times 'a \rangle$ — These are conceptual definitions but force generated code to be monomorphic wrt. particular instances of this class which yields a significant speedup.

assumes *divmod-def*: $\langle \text{divmod } m \ n = (\text{numeral } m \text{ div numeral } n, \text{numeral } m \bmod \text{numeral } n) \rangle$

and *divmod-step-def* [simp]: $\langle \text{divmod-step } l \ (q, r) =$

$(\text{if } \text{euclidean-size } l \leq \text{euclidean-size } r \text{ then } (2 * q + 1, r - l)$

$\text{else } (2 * q, r)) \rangle$ — 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*:

$\langle \text{fst } (\text{divmod } m \ n) = \text{numeral } m \text{ div numeral } n \rangle$

$\langle \text{proof} \rangle$

lemma *snd-divmod*:

$\langle \text{snd } (\text{divmod } m \ n) = \text{numeral } m \bmod \text{numeral } n \rangle$

$\langle \text{proof} \rangle$

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*:

$\langle \text{divmod } m \ n = (\text{if } m < n \text{ then } (0, \text{numeral } m) \\ \text{else divmod-step (numeral } n) (\text{divmod } m (\text{Num.Bit0 } n))) \rangle$
 $\langle \text{proof} \rangle$

The division rewrite proper – first, trivial results involving 1

lemma *divmod-trivial [simp]*:

$\text{divmod } m \ \text{Num.One} = (\text{numeral } m, 0)$
 $\text{divmod } \text{num.One} (\text{num.Bit0 } n) = (0, \text{Numeral1})$
 $\text{divmod } \text{num.One} (\text{num.Bit1 } n) = (0, \text{Numeral1})$
 $\langle \text{proof} \rangle$

Division by an even number is a right-shift

lemma *divmod-cancel [simp]*:

$\langle \text{divmod } (\text{Num.Bit0 } m) (\text{Num.Bit0 } n) = (\text{case divmod } m \ n \text{ of } (q, r) \Rightarrow (q, 2 * r)) \rangle$ (is ?P)
 $\langle \text{divmod } (\text{Num.Bit1 } m) (\text{Num.Bit0 } n) = (\text{case divmod } m \ n \text{ of } (q, r) \Rightarrow (q, 2 * r + 1)) \rangle$ (is ?Q)
 $\langle \text{proof} \rangle$

The really hard work

lemma *divmod-steps [simp]*:

$\text{divmod } (\text{num.Bit0 } m) (\text{num.Bit1 } n) =$
 $(\text{if } m \leq n \text{ then } (0, \text{numeral } (\text{num.Bit0 } m))$
 $\text{else divmod-step (numeral } (\text{num.Bit1 } n))$
 $(\text{divmod } (\text{num.Bit0 } m)$
 $(\text{num.Bit0 } (\text{num.Bit1 } n))))$
 $\text{divmod } (\text{num.Bit1 } m) (\text{num.Bit1 } n) =$
 $(\text{if } m < n \text{ then } (0, \text{numeral } (\text{num.Bit1 } m))$
 $\text{else divmod-step (numeral } (\text{num.Bit1 } n))$
 $(\text{divmod } (\text{num.Bit1 } m)$
 $(\text{num.Bit0 } (\text{num.Bit1 } n))))$
 $\langle \text{proof} \rangle$

lemmas *divmod-algorithm-code = divmod-trivial divmod-cancel divmod-steps*

Special case: divisibility

definition *divides-aux* :: $'a \times 'a \Rightarrow \text{bool}$

where

$\text{divides-aux } qr \longleftrightarrow \text{snd } qr = 0$

lemma *divides-aux-eq [simp]*:

divides-aux (q, r) $\longleftrightarrow r = 0$
 $\langle \text{proof} \rangle$

lemma *dvd-numeral-simp* [*simp*]:
numeral m *dvd* *numeral* $n \longleftrightarrow \text{divides-aux} (\text{divmod } n \ m)$
 $\langle \text{proof} \rangle$

Generic computation of quotient and remainder

lemma *numeral-div-numeral* [*simp*]:
numeral k *div* *numeral* $l = \text{fst} (\text{divmod } k \ l)$
 $\langle \text{proof} \rangle$

lemma *numeral-mod-numeral* [*simp*]:
numeral k *mod* *numeral* $l = \text{snd} (\text{divmod } k \ l)$
 $\langle \text{proof} \rangle$

lemma *one-div-numeral* [*simp*]:
 1 *div* *numeral* $n = \text{fst} (\text{divmod } \text{num.One } n)$
 $\langle \text{proof} \rangle$

lemma *one-mod-numeral* [*simp*]:
 1 *mod* *numeral* $n = \text{snd} (\text{divmod } \text{num.One } n)$
 $\langle \text{proof} \rangle$

end

instantiation *nat* :: *linordered-euclidean-semiring-division*
begin

definition *divmod-nat* :: *num* \Rightarrow *num* \Rightarrow *nat* \times *nat*
where
divmod'-nat-def: *divmod-nat* $m \ n = (\text{numeral } m \ \text{div} \ \text{numeral } n, \text{numeral } m \ \text{mod} \ \text{numeral } n)$

definition *divmod-step-nat* :: *nat* \Rightarrow *nat* \times *nat* \Rightarrow *nat* \times *nat*
where
divmod-step-nat $l \ qr = (\text{let } (q, r) = qr$
in if $r \geq l$ *then* $(2 * q + 1, r - l)$
else $(2 * q, r))$

instance
 $\langle \text{proof} \rangle$

end

declare *divmod-algorithm-code* [**where** $?a = \text{nat}, \text{code}$]

lemma *Suc-0-div-numeral* [*simp*]:
 $\langle \text{Suc } 0 \ \text{div} \ \text{numeral } \text{Num.One} = 1 \rangle$

$\langle \text{Suc } 0 \text{ div numeral } (\text{Num.Bit0 } n) = 0 \rangle$
 $\langle \text{Suc } 0 \text{ div numeral } (\text{Num.Bit1 } n) = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *Suc-0-mod-numeral* [simp]:
 $\langle \text{Suc } 0 \text{ mod numeral } \text{Num.One} = 0 \rangle$
 $\langle \text{Suc } 0 \text{ mod numeral } (\text{Num.Bit0 } n) = 1 \rangle$
 $\langle \text{Suc } 0 \text{ mod numeral } (\text{Num.Bit1 } n) = 1 \rangle$
 $\langle \text{proof} \rangle$

instantiation *int* :: *linordered-euclidean-semiring-division*
begin

definition *divmod-int* :: *num* \Rightarrow *num* \Rightarrow *int* \times *int*
where
 $\text{divmod-int } m \ n = (\text{numeral } m \text{ div numeral } n, \text{numeral } m \text{ mod numeral } n)$

definition *divmod-step-int* :: *int* \Rightarrow *int* \times *int* \Rightarrow *int* \times *int*
where
 $\text{divmod-step-int } l \ qr = (\text{let } (q, r) = qr$
 $\text{in if } |l| \leq |r| \text{ then } (2 * q + 1, r - l)$
 $\text{else } (2 * q, r))$

instance
 $\langle \text{proof} \rangle$

end

declare *divmod-algorithm-code* [where ?'a = int, code]

context
begin

qualified definition *adjust-div* :: *int* \times *int* \Rightarrow *int*
where
 $\text{adjust-div } qr = (\text{let } (q, r) = qr \text{ in } q + \text{of-bool } (r \neq 0))$

qualified lemma *adjust-div-eq* [simp, code]:
 $\text{adjust-div } (q, r) = q + \text{of-bool } (r \neq 0)$
 $\langle \text{proof} \rangle$ **definition** *adjust-mod* :: *num* \Rightarrow *int* \Rightarrow *int*

where
[simp]: $\text{adjust-mod } l \ r = (\text{if } r = 0 \text{ then } 0 \text{ else numeral } l - r)$

lemma *minus-numeral-div-numeral* [simp]:
 $-\text{numeral } m \text{ div numeral } n = -(\text{adjust-div } (\text{divmod } m \ n) :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *minus-numeral-mod-numeral* [simp]:
 $-\text{numeral } m \text{ mod numeral } n = \text{adjust-mod } n \ (\text{snd } (\text{divmod } m \ n) :: \text{int})$

<proof>

lemma *numeral-div-minus-numeral* [simp]:
 $\text{numeral } m \text{ div } - \text{numeral } n = - (\text{adjust-div } (\text{divmod } m \ n) :: \text{int})$
<proof>

lemma *numeral-mod-minus-numeral* [simp]:
 $\text{numeral } m \text{ mod } - \text{numeral } n = - \text{adjust-mod } n \ (\text{snd } (\text{divmod } m \ n) :: \text{int})$
<proof>

lemma *minus-one-div-numeral* [simp]:
 $- 1 \text{ div numeral } n = - (\text{adjust-div } (\text{divmod } \text{Num.One } n) :: \text{int})$
<proof>

lemma *minus-one-mod-numeral* [simp]:
 $- 1 \text{ mod numeral } n = \text{adjust-mod } n \ (\text{snd } (\text{divmod } \text{Num.One } n) :: \text{int})$
<proof>

lemma *one-div-minus-numeral* [simp]:
 $1 \text{ div } - \text{numeral } n = - (\text{adjust-div } (\text{divmod } \text{Num.One } n) :: \text{int})$
<proof>

lemma *one-mod-minus-numeral* [simp]:
 $1 \text{ mod } - \text{numeral } n = - \text{adjust-mod } n \ (\text{snd } (\text{divmod } \text{Num.One } n) :: \text{int})$
<proof>

lemma [code]:
fixes $k :: \text{int}$
shows
 $k \text{ div } 0 = 0$
 $k \text{ mod } 0 = k$
 $0 \text{ div } k = 0$
 $0 \text{ mod } k = 0$
 $k \text{ div Int.Pos Num.One} = k$
 $k \text{ mod Int.Pos Num.One} = 0$
 $k \text{ div Int.Neg Num.One} = - k$
 $k \text{ mod Int.Neg Num.One} = 0$
 $\text{Int.Pos } m \text{ div Int.Pos } n = (\text{fst } (\text{divmod } m \ n) :: \text{int})$
 $\text{Int.Pos } m \text{ mod Int.Pos } n = (\text{snd } (\text{divmod } m \ n) :: \text{int})$
 $\text{Int.Neg } m \text{ div Int.Pos } n = - (\text{adjust-div } (\text{divmod } m \ n) :: \text{int})$
 $\text{Int.Neg } m \text{ mod Int.Pos } n = \text{adjust-mod } n \ (\text{snd } (\text{divmod } m \ n) :: \text{int})$
 $\text{Int.Pos } m \text{ div Int.Neg } n = - (\text{adjust-div } (\text{divmod } m \ n) :: \text{int})$
 $\text{Int.Pos } m \text{ mod Int.Neg } n = - \text{adjust-mod } n \ (\text{snd } (\text{divmod } m \ n) :: \text{int})$
 $\text{Int.Neg } m \text{ div Int.Neg } n = (\text{fst } (\text{divmod } m \ n) :: \text{int})$
 $\text{Int.Neg } m \text{ mod Int.Neg } n = - (\text{snd } (\text{divmod } m \ n) :: \text{int})$
<proof>

end

lemma *divmod-BitM-2-eq [simp]:*

$\langle \text{divmod } (\text{Num.BitM } m) (\text{Num.Bit0 Num.One}) = (\text{numeral } m - 1, (1 :: \text{int})) \rangle$
 $\langle \text{proof} \rangle$

57.6.1 Computation by simplification

lemma *euclidean-size-nat-less-eq-iff:*

$\langle \text{euclidean-size } m \leq \text{euclidean-size } n \iff m \leq n \rangle$ **for** $m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *euclidean-size-int-less-eq-iff:*

$\langle \text{euclidean-size } k \leq \text{euclidean-size } l \iff |k| \leq |l| \rangle$ **for** $k\ l :: \text{int}$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

57.7 Computing congruences modulo 2^q

context *linordered-euclidean-semiring-division*

begin

lemma *cong-exp-iff-simps:*

$\text{numeral } n \bmod \text{numeral Num.One} = 0$
 $\iff \text{True}$
 $\text{numeral } (\text{Num.Bit0 } n) \bmod \text{numeral } (\text{Num.Bit0 } q) = 0$
 $\iff \text{numeral } n \bmod \text{numeral } q = 0$
 $\text{numeral } (\text{Num.Bit1 } n) \bmod \text{numeral } (\text{Num.Bit0 } q) = 0$
 $\iff \text{False}$
 $\text{numeral } m \bmod \text{numeral Num.One} = (\text{numeral } n \bmod \text{numeral Num.One})$
 $\iff \text{True}$
 $\text{numeral Num.One} \bmod \text{numeral } (\text{Num.Bit0 } q) = (\text{numeral Num.One} \bmod \text{numeral } (\text{Num.Bit0 } q))$
 $\iff \text{True}$
 $\text{numeral Num.One} \bmod \text{numeral } (\text{Num.Bit0 } q) = (\text{numeral } (\text{Num.Bit0 } n) \bmod \text{numeral } (\text{Num.Bit0 } q))$
 $\iff \text{False}$
 $\text{numeral Num.One} \bmod \text{numeral } (\text{Num.Bit0 } q) = (\text{numeral } (\text{Num.Bit1 } n) \bmod \text{numeral } (\text{Num.Bit0 } q))$
 $\iff (\text{numeral } n \bmod \text{numeral } q) = 0$
 $\text{numeral } (\text{Num.Bit0 } m) \bmod \text{numeral } (\text{Num.Bit0 } q) = (\text{numeral Num.One} \bmod \text{numeral } (\text{Num.Bit0 } q))$
 $\iff \text{False}$
 $\text{numeral } (\text{Num.Bit0 } m) \bmod \text{numeral } (\text{Num.Bit0 } q) = (\text{numeral } (\text{Num.Bit0 } n) \bmod \text{numeral } (\text{Num.Bit0 } q))$
 $\iff \text{numeral } m \bmod \text{numeral } q = (\text{numeral } n \bmod \text{numeral } q)$
 $\text{numeral } (\text{Num.Bit0 } m) \bmod \text{numeral } (\text{Num.Bit0 } q) = (\text{numeral } (\text{Num.Bit1 } n) \bmod \text{numeral } (\text{Num.Bit0 } q))$
 $\iff \text{False}$
 $\text{numeral } (\text{Num.Bit1 } m) \bmod \text{numeral } (\text{Num.Bit0 } q) = (\text{numeral Num.One} \bmod \text{numeral } (\text{Num.Bit0 } q))$

```

     $\longleftrightarrow$  (numeral  $m$  mod numeral  $q$ ) = 0
    numeral (Num.Bit1  $m$ ) mod numeral (Num.Bit0  $q$ ) = (numeral (Num.Bit0  $n$ )
mod numeral (Num.Bit0  $q$ ))
     $\longleftrightarrow$  False
    numeral (Num.Bit1  $m$ ) mod numeral (Num.Bit0  $q$ ) = (numeral (Num.Bit1  $n$ )
mod numeral (Num.Bit0  $q$ ))
     $\longleftrightarrow$  numeral  $m$  mod numeral  $q$  = (numeral  $n$  mod numeral  $q$ )
    <proof>

end

```

code-identifier

code-module *Parity* \rightarrow (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>

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*: $i*u + (j*u + k) = (i+j)*u + (k::nat)$
 <proof>

For *cancel-numerals*

lemma *nat-diff-add-eq1*:
 $j <= (i::nat) \implies ((i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)$
 <proof>

lemma *nat-diff-add-eq2*:
 $i <= (j::nat) \implies ((i*u + m) - (j*u + n)) = (m - ((j-i)*u + n))$
 <proof>

lemma *nat-eq-add-iff1*:

$$j \leq (i::nat) \implies (i*u + m = j*u + n) = ((i-j)*u + m = n)$$

<proof>

lemma *nat-eq-add-iff2*:

$$i \leq (j::nat) \implies (i*u + m = j*u + n) = (m = (j-i)*u + n)$$

<proof>

lemma *nat-less-add-iff1*:

$$j < (i::nat) \implies (i*u + m < j*u + n) = ((i-j)*u + m < n)$$

<proof>

lemma *nat-less-add-iff2*:

$$i < (j::nat) \implies (i*u + m < j*u + n) = (m < (j-i)*u + n)$$

<proof>

lemma *nat-le-add-iff1*:

$$j \leq (i::nat) \implies (i*u + m \leq j*u + n) = ((i-j)*u + m \leq n)$$

<proof>

lemma *nat-le-add-iff2*:

$$i \leq (j::nat) \implies (i*u + m \leq j*u + n) = (m \leq (j-i)*u + n)$$

<proof>

For *cancel-numeral-factors*

lemma *nat-mult-le-cancel1*: $(0::nat) < k \implies (k*m \leq k*n) = (m \leq n)$

<proof>

lemma *nat-mult-less-cancel1*: $(0::nat) < k \implies (k*m < k*n) = (m < n)$

<proof>

lemma *nat-mult-eq-cancel1*: $(0::nat) < k \implies (k*m = k*n) = (m = n)$

<proof>

lemma *nat-mult-div-cancel1*: $(0::nat) < k \implies (k*m) \text{ div } (k*n) = (m \text{ div } n)$

<proof>

lemma *nat-mult-dvd-cancel-disj[simp]*:

$$(k*m) \text{ dvd } (k*n) = (k=0 \vee m \text{ dvd } (n::nat))$$

<proof>

lemma *nat-mult-dvd-cancel1*: $0 < k \implies (k*m) \text{ dvd } (k*n::nat) = (m \text{ dvd } n)$

<proof>

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 $k\ m\ n :: \text{nat}$

shows $k * m = k * n \longleftrightarrow k = 0 \vee m = n$

$\langle \text{proof} \rangle$

lemma *nat-mult-div-cancel-disj*:

fixes $k\ m\ n :: \text{nat}$

shows $(k * m) \text{ div } (k * n) = (\text{if } k = 0 \text{ then } 0 \text{ else } m \text{ div } n)$

$\langle \text{proof} \rangle$

lemma *numeral-times-minus-swap*:

fixes $x :: 'a :: \text{comm-ring-1}$ **shows** $\text{numeral } w * -x = x * - \text{numeral } w$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

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*: $w * y + x * z = w * z + x * y \longleftrightarrow w = x \vee y = z$

begin

lemma *crossproduct-noteq*:

$a \neq b \wedge c \neq d \longleftrightarrow a * c + b * d \neq a * d + b * c$

$\langle \text{proof} \rangle$

lemma *add-scale-eq-noteq*:

$r \neq 0 \implies a = b \wedge c \neq d \implies a + r * c \neq b + r * d$

$\langle \text{proof} \rangle$

lemma *add-0-iff*:

$b = b + a \longleftrightarrow a = 0$

$\langle \text{proof} \rangle$

end

subclass (**in** *idom*) *comm-semiring-1-cancel-crossproduct*

$\langle \text{proof} \rangle$

instance *nat* :: *comm-semiring-1-cancel-crossproduct*

$\langle \text{proof} \rangle$

Semiring normalization proper

$\langle ML \rangle$

context *comm-semiring-1*
begin

lemma *semiring-normalization-rules* [*no-atp*]:

$$\begin{aligned} (a * m) + (b * m) &= (a + b) * m \\ (a * m) + m &= (a + 1) * m \\ m + (a * m) &= (a + 1) * m \\ m + m &= (1 + 1) * m \\ 0 + a &= a \\ a + 0 &= a \\ a * b &= b * a \\ (a + b) * c &= (a * c) + (b * c) \\ 0 * a &= 0 \\ a * 0 &= 0 \\ 1 * a &= a \\ a * 1 &= a \\ (lx * ly) * (rx * ry) &= (lx * rx) * (ly * ry) \\ (lx * ly) * (rx * ry) &= lx * (ly * (rx * ry)) \\ (lx * ly) * (rx * ry) &= rx * ((lx * ly) * ry) \\ (lx * ly) * rx &= (lx * rx) * ly \\ (lx * ly) * rx &= lx * (ly * rx) \\ lx * (rx * ry) &= (lx * rx) * ry \\ lx * (rx * ry) &= rx * (lx * ry) \\ (a + b) + (c + d) &= (a + c) + (b + d) \\ (a + b) + c &= a + (b + c) \\ a + (c + d) &= c + (a + d) \\ (a + b) + c &= (a + c) + b \\ a + c &= c + a \\ a + (c + d) &= (a + c) + d \\ (x \hat{ } p) * (x \hat{ } q) &= x \hat{ } (p + q) \\ x * (x \hat{ } q) &= x \hat{ } (Suc\ q) \\ (x \hat{ } q) * x &= x \hat{ } (Suc\ q) \\ x * x &= x^2 \\ (x * y) \hat{ } q &= (x \hat{ } q) * (y \hat{ } q) \\ (x \hat{ } p) \hat{ } q &= x \hat{ } (p * q) \\ x \hat{ } 0 &= 1 \\ x \hat{ } 1 &= x \\ x * (y + z) &= (x * y) + (x * z) \\ x \hat{ } (Suc\ q) &= x * (x \hat{ } q) \\ x \hat{ } (2*n) &= (x \hat{ } n) * (x \hat{ } n) \\ \langle proof \rangle \end{aligned}$$

$\langle ML \rangle$

end

context *comm-ring-1*
begin

lemma *ring-normalization-rules* [*no-atp*]:
 $- x = (- 1) * x$
 $x - y = x + (- y)$
 $\langle proof \rangle$

$\langle ML \rangle$

end

context *comm-semiring-1-cancel-crossproduct*
begin

$\langle ML \rangle$

end

context *idom*
begin

$\langle ML \rangle$

end

context *field*
begin

$\langle ML \rangle$

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*

$(\neg(P \wedge Q)) = (\neg P \vee \neg Q)$ $(\neg(P \vee Q)) = (\neg P \wedge \neg Q)$
 $(P \longrightarrow Q) = (\neg P \vee Q)$
 $(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q))$ $(\neg \neg(P)) = P$
 $\langle proof \rangle$

lemma *dnf*:

$(P \wedge (Q \vee R)) = ((P \wedge Q) \vee (P \wedge R))$
 $((Q \vee R) \wedge P) = ((Q \wedge P) \vee (R \wedge P))$
 $(P \wedge Q) = (Q \wedge P)$
 $(P \vee Q) = (Q \vee P)$
 $\langle proof \rangle$

lemmas *weak-dnf-simps* = *dnf bool-simps*

lemma *PFalse*:

$P \equiv \text{False} \implies \neg P$
 $\neg P \implies (P \equiv \text{False})$
 $\langle proof \rangle$

named-theorems *algebra pre-simplification rules for algebraic methods*
 $\langle ML \rangle$

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*]:

$\langle m \bmod d = 0 \longleftrightarrow (\exists q. m = d * q) \rangle$ **for** $m \ d :: \text{int}$
 $\langle proof \rangle$

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

```

```

lemma card-2-iff:  $\text{card } S = 2 \longleftrightarrow (\exists x\ y. S = \{x, y\} \wedge x \neq y)$ 
  <proof>

```

```

lemma card-2-iff':  $\text{card } S = 2 \longleftrightarrow (\exists x \in S. \exists y \in S. x \neq y \wedge (\forall z \in S. z = x \vee z = y))$ 
  <proof>

```

```

lemma card-3-iff:  $\text{card } S = 3 \longleftrightarrow (\exists x\ y\ z. S = \{x, y, z\} \wedge x \neq y \wedge y \neq z \wedge x \neq z)$ 
  <proof>

```

```

context ord
begin

```

```

definition

```

```

  lessThan :: 'a  $\Rightarrow$  'a set (<(indent=1 notation=<mixfix set interval>>{..<-}>>)

```

```

where

```

```

  {..u} == {x. x < u}

```

definition

$$atMost :: 'a \Rightarrow 'a \text{ set } (\langle (\langle indent=1 \text{ notation}=\langle \text{mixfix set interval} \rangle \{.. \}) \rangle)$$
where

$$\{..u\} == \{x. x \leq u\}$$
definition

$$greaterThan :: 'a \Rightarrow 'a \text{ set } (\langle (\langle indent=1 \text{ notation}=\langle \text{mixfix set interval} \rangle \{<.. \}) \rangle)$$
where

$$\{l<..\} == \{x. l < x\}$$
definition

$$atLeast :: 'a \Rightarrow 'a \text{ set } (\langle (\langle indent=1 \text{ notation}=\langle \text{mixfix set interval} \rangle \{.. \}) \rangle)$$
where

$$\{l..\} == \{x. l \leq x\}$$
definition

$$greaterThanLessThan :: 'a \Rightarrow 'a \Rightarrow 'a \text{ set } (\langle (\langle indent=1 \text{ notation}=\langle \text{mixfix set interval} \rangle \{<./<..<- \}) \rangle) \text{ where}$$

$$\{l<..\} == \{l<..\} \cap \{..\}$$
definition

$$atLeastLessThan :: 'a \Rightarrow 'a \Rightarrow 'a \text{ set } (\langle (\langle indent=1 \text{ notation}=\langle \text{mixfix set interval} \rangle \{<./<..<- \}) \rangle) \text{ where}$$

$$\{l..\} == \{l..\} \cap \{..\}$$
definition

$$greaterThanAtMost :: 'a \Rightarrow 'a \Rightarrow 'a \text{ set } (\langle (\langle indent=1 \text{ notation}=\langle \text{mixfix set interval} \rangle \{<./<.. \}) \rangle) \text{ where}$$

$$\{l<..\} == \{l<..\} \cap \{..\}$$
definition

$$atLeastAtMost :: 'a \Rightarrow 'a \Rightarrow 'a \text{ set } (\langle (\langle indent=1 \text{ notation}=\langle \text{mixfix set interval} \rangle \{<./<.. \}) \rangle) \text{ where}$$

$$\{l..\} == \{l..\} \cap \{..\}$$
end

A note of warning when using $\{..\}$ on type *nat*: it is equivalent to $\{0..\}$ but some lemmas involving $\{m..\}$ may not exist in $\{..\}$ -form as well.

syntax (ASCII)

$$\text{-UNION-le} :: 'a \Rightarrow 'a \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set} \quad (\langle (\langle indent=3 \text{ notation}=\langle \text{binder UN} \rangle \rangle \text{UN} \text{ } \langle \text{<= } \text{<./ } \text{>} \rangle [0, 0, 10] 10)$$

$$\text{-UNION-less} :: 'a \Rightarrow 'a \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set} \quad (\langle (\langle indent=3 \text{ notation}=\langle \text{binder UN} \rangle \rangle \text{UN} \text{ } \langle \text{< } \text{<./ } \text{>} \rangle [0, 0, 10] 10)$$

$$\text{-INTER-le} :: 'a \Rightarrow 'a \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set} \quad (\langle (\langle indent=3 \text{ notation}=\langle \text{binder INT} \rangle \rangle \text{INT} \text{ } \langle \text{<= } \text{<./ } \text{>} \rangle [0, 0, 10] 10)$$

$$\text{-INTER-less} :: 'a \Rightarrow 'a \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set} \quad (\langle (\langle indent=3 \text{ notation}=\langle \text{binder INT} \rangle \rangle \text{INT} \text{ } \langle \text{< } \text{<./ } \text{>} \rangle [0, 0, 10] 10)$$

syntax (*latex output*)

-UNION-le :: 'a \Rightarrow 'a \Rightarrow 'b set \Rightarrow 'b set ($\langle (\exists \cup (\langle \text{unbreakable} \rangle - \leq -) / -) \rangle$
 $[0, 0, 10] 10$)
 -UNION-less :: 'a \Rightarrow 'a \Rightarrow 'b set \Rightarrow 'b set ($\langle (\exists \cup (\langle \text{unbreakable} \rangle - < -) / -) \rangle$
 $[0, 0, 10] 10$)
 -INTER-le :: 'a \Rightarrow 'a \Rightarrow 'b set \Rightarrow 'b set ($\langle (\exists \cap (\langle \text{unbreakable} \rangle - \leq -) / -) \rangle$ $[0,$
 $0, 10] 10$)
 -INTER-less :: 'a \Rightarrow 'a \Rightarrow 'b set \Rightarrow 'b set ($\langle (\exists \cap (\langle \text{unbreakable} \rangle - < -) / -) \rangle$ $[0,$
 $0, 10] 10$)

syntax

-UNION-le :: 'a \Rightarrow 'a \Rightarrow 'b set \Rightarrow 'b set ($\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder} \rangle$
 $\cup \rangle \cup -\leq - / -) \rangle$ $[0, 0, 10] 10$)
 -UNION-less :: 'a \Rightarrow 'a \Rightarrow 'b set \Rightarrow 'b set ($\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder} \rangle$
 $\cup \rangle \cup -< - / -) \rangle$ $[0, 0, 10] 10$)
 -INTER-le :: 'a \Rightarrow 'a \Rightarrow 'b set \Rightarrow 'b set ($\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder} \rangle$
 $\cap \rangle \cap -\leq - / -) \rangle$ $[0, 0, 10] 10$)
 -INTER-less :: 'a \Rightarrow 'a \Rightarrow 'b set \Rightarrow 'b set ($\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder} \rangle$
 $\cap \rangle \cap -< - / -) \rangle$ $[0, 0, 10] 10$)

syntax-consts

-UNION-le -UNION-less \Rightarrow Union and
 -INTER-le -INTER-less \Rightarrow Inter

translations

$\bigcup_{i \leq n}. A \Rightarrow \bigcup_{i \in \{..n\}}. A$
 $\bigcup_{i < n}. A \Rightarrow \bigcup_{i \in \{..<n\}}. A$
 $\bigcap_{i \leq n}. A \Rightarrow \bigcap_{i \in \{..n\}}. A$
 $\bigcap_{i < n}. A \Rightarrow \bigcap_{i \in \{..<n\}}. A$

61.1 Various equivalences

lemma (in ord) lessThan-iff [iff]: $(i \in \text{lessThan } k) = (i < k)$
 $\langle \text{proof} \rangle$

lemma Compl-lessThan [simp]:
 $!!k:: 'a::\text{linorder}. -\text{lessThan } k = \text{atLeast } k$
 $\langle \text{proof} \rangle$

lemma single-Diff-lessThan [simp]: $!!k:: 'a::\text{preorder}. \{k\} - \text{lessThan } k = \{k\}$
 $\langle \text{proof} \rangle$

lemma (in ord) greaterThan-iff [iff]: $(i \in \text{greaterThan } k) = (k < i)$
 $\langle \text{proof} \rangle$

lemma Compl-greaterThan [simp]:
 $!!k:: 'a::\text{linorder}. -\text{greaterThan } k = \text{atMost } k$
 $\langle \text{proof} \rangle$

lemma *Compl-atMost [simp]:* $!!k:: 'a::linorder. \neg atMost\ k = greaterThan\ k$
 $\langle proof \rangle$

lemma *(in ord) atLeast-iff [iff]:* $(i \in atLeast\ k) = (k \leq i)$
 $\langle proof \rangle$

lemma *Compl-atLeast [simp]:* $!!k:: 'a::linorder. \neg atLeast\ k = lessThan\ k$
 $\langle proof \rangle$

lemma *(in ord) atMost-iff [iff]:* $(i \in atMost\ k) = (i \leq k)$
 $\langle proof \rangle$

lemma *atMost-Int-atLeast:* $!!n:: 'a::order. atMost\ n \cap atLeast\ n = \{n\}$
 $\langle proof \rangle$

lemma *(in linorder) lessThan-Int-lessThan:* $\{a <..\} \cap \{b <..\} = \{max\ a\ b <..\}$
 $\langle proof \rangle$

lemma *(in linorder) greaterThan-Int-greaterThan:* $\{..< a\} \cap \{..< b\} = \{..< min\ a\ b\}$
 $\langle proof \rangle$

61.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-empty-triv [simp]:* $\{\{\}\} = UNIV$
 $\langle proof \rangle$

lemma *atMost-UNIV-triv [simp]:* $\{..UNIV\} = UNIV$
 $\langle proof \rangle$

lemma *atLeast-subset-iff [iff]:*
 $(atLeast\ x \subseteq atLeast\ y) = (y \leq (x::'a::preorder))$
 $\langle proof \rangle$

lemma *atLeast-eq-iff [iff]:*
 $(atLeast\ x = atLeast\ y) = (x = (y::'a::order))$
 $\langle proof \rangle$

lemma *greaterThan-subset-iff [iff]:*
 $(greaterThan\ x \subseteq greaterThan\ y) = (y \leq (x::'a::linorder))$
 $\langle proof \rangle$

lemma *greaterThan-eq-iff [iff]:*
 $(greaterThan\ x = greaterThan\ y) = (x = (y::'a::linorder))$
 $\langle proof \rangle$

lemma *atMost-subset-iff [iff]:* $(atMost\ x \subseteq atMost\ y) = (x \leq (y::'a::preorder))$
 $\langle proof \rangle$

lemma *atMost-eq-iff* [iff]: $(atMost\ x = atMost\ y) = (x = (y::'a::order))$
 ⟨proof⟩

lemma *lessThan-subset-iff* [iff]:
 $(lessThan\ x \subseteq lessThan\ y) = (x \leq (y::'a::linorder))$
 ⟨proof⟩

lemma *lessThan-eq-iff* [iff]:
 $(lessThan\ x = lessThan\ y) = (x = (y::'a::linorder))$
 ⟨proof⟩

lemma *lessThan-strict-subset-iff*:
 fixes $m\ n :: 'a::linorder$
 shows $\{.. m \} < \{.. n \} \longleftrightarrow m < n$
 ⟨proof⟩

lemma (in *linorder*) *Ici-subset-Ioi-iff*: $\{a\ ..\} \subseteq \{b <..\} \longleftrightarrow b < a$
 ⟨proof⟩

lemma (in *linorder*) *Iic-subset-Iio-iff*: $\{.. a \} \subseteq \{.. b \} \longleftrightarrow a < b$
 ⟨proof⟩

lemma (in *preorder*) *Ioi-le-Ico*: $\{a <..\} \subseteq \{a\ ..\}$
 ⟨proof⟩

61.3 Two-sided intervals

context *ord*
begin

lemma *greaterThanLessThan-iff* [simp]: $(i \in \{l <.. u \}) = (l < i \wedge i < u)$
 ⟨proof⟩

lemma *atLeastLessThan-iff* [simp]: $(i \in \{l..<u\}) = (l \leq i \wedge i < u)$
 ⟨proof⟩

lemma *greaterThanAtMost-iff* [simp]: $(i \in \{l <.. u \}) = (l < i \wedge i \leq u)$
 ⟨proof⟩

lemma *atLeastAtMost-iff* [simp]: $(i \in \{l..u\}) = (l \leq i \wedge i \leq u)$
 ⟨proof⟩

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*: $\{a <.. b \} = \{a <..\} \cap \{.. b \}$
 ⟨proof⟩

lemma (in *order*) *atLeastLessThan-eq-atLeastAtMost-diff*:
 $\{a..<b\} = \{a..b\} - \{b\}$

$\langle proof \rangle$

lemma (in order) *greaterThanAtMost-eq-atLeastAtMost-diff*:

$$\{a <..b\} = \{a..b\} - \{a\}$$

$\langle proof \rangle$

end

61.3.1 Emptyness, singletons, subset

context *preorder*

begin

lemma *atLeastatMost-empty-iff[simp]*:

$$\{a..b\} = \{\} \longleftrightarrow (\neg a \leq b)$$

$\langle proof \rangle$

lemma *atLeastatMost-empty-iff2[simp]*:

$$\{\} = \{a..b\} \longleftrightarrow (\neg a \leq b)$$

$\langle proof \rangle$

lemma *atLeastLessThan-empty-iff[simp]*:

$$\{a..<b\} = \{\} \longleftrightarrow (\neg a < b)$$

$\langle proof \rangle$

lemma *atLeastLessThan-empty-iff2[simp]*:

$$\{\} = \{a..<b\} \longleftrightarrow (\neg a < b)$$

$\langle proof \rangle$

lemma *greaterThanAtMost-empty-iff[simp]*: $\{k <..l\} = \{\} \longleftrightarrow \neg k < l$

$\langle proof \rangle$

lemma *greaterThanAtMost-empty-iff2[simp]*: $\{\} = \{k <..l\} \longleftrightarrow \neg k < l$

$\langle proof \rangle$

lemma *atLeastatMost-subset-iff[simp]*:

$$\{a..b\} \leq \{c..d\} \longleftrightarrow (\neg a \leq b) \vee c \leq a \wedge b \leq d$$

$\langle proof \rangle$

lemma *atLeastatMost-psubset-iff*:

$$\{a..b\} < \{c..d\} \longleftrightarrow$$

$$((\neg a \leq b) \vee c \leq a \wedge b \leq d \wedge (c < a \vee b < d)) \wedge c \leq d$$

$\langle proof \rangle$

lemma *atLeastAtMost-subseteq-atLeastLessThan-iff*:

$$\{a..b\} \subseteq \{c <..d\} \longleftrightarrow (a \leq b \longrightarrow c \leq a \wedge b < d)$$

$\langle proof \rangle$

lemma *Icc-subset-Ici-iff[simp]*:

$$\{l..h\} \subseteq \{l'.. \} = (\neg l \leq h \vee l \geq l')$$

<proof>

lemma *Icc-subset-Iic-iff[simp]*:
 $\{l..h\} \subseteq \{..h'\} = (\neg l \leq h \vee h \leq h')$
<proof>

lemma *not-Ici-eq-empty[simp]*: $\{l.. \} \neq \{\}$
<proof>

lemma *not-Iic-eq-empty[simp]*: $\{..h\} \neq \{\}$
<proof>

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]*: $b < a \implies \{a..b\} = \{\}$
and *atLeastatMost-empty'[simp]*: $\neg a \leq b \implies \{a..b\} = \{\}$
<proof>

lemma *atLeastLessThan-empty[simp]*:
 $b \leq a \implies \{a..<b\} = \{\}$
<proof>

lemma *greaterThanAtMost-empty[simp]*: $l \leq k \implies \{k<..l\} = \{\}$
<proof>

lemma *greaterThanLessThan-empty[simp]*: $l \leq k \implies \{k<..
<proof>$

lemma *atLeastAtMost-singleton [simp]*: $\{a..a\} = \{a\}$
<proof>

lemma *atLeastAtMost-singleton'*: $a = b \implies \{a .. b\} = \{a\}$ *<proof>*

lemma *Icc-eq-Icc[simp]*:
 $\{l..h\} = \{l'..h'\} = (l=l' \wedge h=h' \vee \neg l \leq h \wedge \neg l' \leq h')$
<proof>

lemma (**in** *linorder*) *Ico-eq-Ico*:
 $\{l..
<proof>$

lemma *atLeastAtMost-singleton-iff[simp]*:

$\{a \dots b\} = \{c\} \longleftrightarrow a = b \wedge b = c$
 $\langle \text{proof} \rangle$

Quantifiers

lemma *ex-interval-simps*:

$(\exists x \in \{..<u\}. P x) \longleftrightarrow (\exists x < u. P x)$
 $(\exists x \in \{..u\}. P x) \longleftrightarrow (\exists x \leq u. P x)$
 $(\exists x \in \{l<..\}. P x) \longleftrightarrow (\exists x > l. P x)$
 $(\exists x \in \{l..\}. P x) \longleftrightarrow (\exists x \geq l. P x)$
 $(\exists x \in \{l<..
 $(\exists x \in \{l..
 $(\exists x \in \{l<..u\}. P x) \longleftrightarrow (\exists x. l < x \wedge x \leq u \wedge P x)$
 $(\exists x \in \{l..u\}. P x) \longleftrightarrow (\exists x. l \leq x \wedge x \leq u \wedge P x)$
 $\langle \text{proof} \rangle$$$

lemma *all-interval-simps*:

$(\forall x \in \{..<u\}. P x) \longleftrightarrow (\forall x < u. P x)$
 $(\forall x \in \{..u\}. P x) \longleftrightarrow (\forall x \leq u. P x)$
 $(\forall x \in \{l<..\}. P x) \longleftrightarrow (\forall x > l. P x)$
 $(\forall x \in \{l..\}. P x) \longleftrightarrow (\forall x \geq l. P x)$
 $(\forall x \in \{l<..
 $(\forall x \in \{l..
 $(\forall x \in \{l<..u\}. P x) \longleftrightarrow (\forall x. l < x \longrightarrow x \leq u \longrightarrow P x)$
 $(\forall x \in \{l..u\}. P x) \longleftrightarrow (\forall x. l \leq x \longrightarrow x \leq u \longrightarrow P x)$
 $\langle \text{proof} \rangle$$$

The following results generalise their namesakes in *HOL.Nat* to intervals

lemma *lift-Suc-mono-le-ivl*:

assumes *mono*: $\bigwedge n. n \in N \implies f n \leq f (Suc n)$
and $n \leq n'$ **and** *subN*: $\{n..<n'\} \subseteq N$
shows $f n \leq f n'$
 $\langle \text{proof} \rangle$

lemma *lift-Suc-antimono-le-ivl*:

assumes *mono*: $\bigwedge n. n \in N \implies f n \geq f (Suc n)$
and $n \leq n'$ **and** *subN*: $\{n..<n'\} \subseteq N$
shows $f n \geq f n'$
 $\langle \text{proof} \rangle$

lemma *lift-Suc-mono-less-ivl*:

assumes *mono*: $\bigwedge n. n \in N \implies f n < f (Suc n)$
and $n < n'$ **and** *subN*: $\{n..<n'\} \subseteq N$
shows $f n < f n'$
 $\langle \text{proof} \rangle$

end

context *no-top*

begin

lemma *not-UNIV-le-Icc[simp]*: $\neg UNIV \subseteq \{l..h\}$
 $\langle proof \rangle$

lemma *not-UNIV-le-Iic[simp]*: $\neg UNIV \subseteq \{..h\}$
 $\langle proof \rangle$

lemma *not-Ici-le-Icc[simp]*: $\neg \{l..\} \subseteq \{l'..h'\}$
 $\langle proof \rangle$

lemma *not-Ici-le-Iic[simp]*: $\neg \{l..\} \subseteq \{..h'\}$
 $\langle proof \rangle$

end

context *no-bot*
begin

lemma *not-UNIV-le-Ici[simp]*: $\neg UNIV \subseteq \{l..\}$
 $\langle proof \rangle$

lemma *not-Iic-le-Icc[simp]*: $\neg \{..h\} \subseteq \{l'..h'\}$
 $\langle proof \rangle$

lemma *not-Iic-le-Ici[simp]*: $\neg \{..h\} \subseteq \{l'..\}$
 $\langle proof \rangle$

end

context *no-top*
begin

lemma *not-UNIV-eq-Icc[simp]*: $\neg UNIV = \{l'..h'\}$
 $\langle proof \rangle$

lemmas *not-Icc-eq-UNIV[simp]* = *not-UNIV-eq-Icc[symmetric]*

lemma *not-UNIV-eq-Iic[simp]*: $\neg UNIV = \{..h'\}$
 $\langle proof \rangle$

lemmas *not-Iic-eq-UNIV[simp]* = *not-UNIV-eq-Iic[symmetric]*

lemma *not-Icc-eq-Ici[simp]*: $\neg \{l..h\} = \{l'..\}$
 $\langle proof \rangle$

lemmas *not-Ici-eq-Icc[simp]* = *not-Icc-eq-Ici[symmetric]*

lemma *not-Ici-eq-Ici[simp]*: $\neg \{..h\} = \{l'.. \}$
 $\langle proof \rangle$

lemmas *not-Ici-eq-Ici[simp]* = *not-Ici-eq-Ici[symmetric]*

end

context *no-bot*
begin

lemma *not-UNIV-eq-Ici[simp]*: $\neg UNIV = \{l'.. \}$
 $\langle proof \rangle$

lemmas *not-Ici-eq-UNIV[simp]* = *not-UNIV-eq-Ici[symmetric]*

lemma *not-Icc-eq-Ici[simp]*: $\neg \{l..h\} = \{..h\}$
 $\langle proof \rangle$

lemmas *not-Ici-eq-Icc[simp]* = *not-Icc-eq-Ici[symmetric]*

end

context *dense-linorder*
begin

lemma *greaterThanLessThan-empty-iff[simp]*:
 $\{ a <..< b \} = \{ \} \longleftrightarrow b \leq a$
 $\langle proof \rangle$

lemma *greaterThanLessThan-empty-iff2[simp]*:
 $\{ \} = \{ a <..< b \} \longleftrightarrow b \leq a$
 $\langle proof \rangle$

lemma *atLeastLessThan-subseteq-atLeastAtMost-iff*:
 $\{ a ..< b \} \subseteq \{ c .. d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
 $\langle proof \rangle$

lemma *greaterThanAtMost-subseteq-atLeastAtMost-iff*:
 $\{ a <.. b \} \subseteq \{ c .. d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
 $\langle proof \rangle$

lemma *greaterThanLessThan-subseteq-atLeastAtMost-iff*:
 $\{ a <..< b \} \subseteq \{ c .. d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
 $\langle proof \rangle$

lemma *greaterThanLessThan-subseteq-greaterThanLessThan*:

$\{a <..< b\} \subseteq \{c <..< d\} \longleftrightarrow (a < b \longrightarrow a \geq c \wedge b \leq d)$
 $\langle proof \rangle$

lemma *greaterThanAtMost-subseteq-atLeastLessThan-iff*:
 $\{a <..< b\} \subseteq \{c ..< d\} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b < d)$
 $\langle proof \rangle$

lemma *greaterThanLessThan-subseteq-atLeastLessThan-iff*:
 $\{a <..< b\} \subseteq \{c ..< d\} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
 $\langle proof \rangle$

lemma *greaterThanLessThan-subseteq-greaterThanAtMost-iff*:
 $\{a <..< b\} \subseteq \{c <..< d\} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
 $\langle proof \rangle$

end

context *no-top*
begin

lemma *greaterThan-non-empty[simp]*: $\{x <..< \} \neq \{\}$
 $\langle proof \rangle$

end

context *no-bot*
begin

lemma *lessThan-non-empty[simp]*: $\{..< x\} \neq \{\}$
 $\langle proof \rangle$

end

lemma (**in** *linorder*) *atLeastLessThan-subset-iff*:
 $\{a..< b\} \subseteq \{c..< d\} \implies b \leq a \vee c \leq a \wedge b \leq d$
 $\langle proof \rangle$

lemma *atLeastLessThan-inj*:
fixes $a\ b\ c\ d :: 'a::linorder$
assumes $eq: \{a ..< b\} = \{c ..< d\}$ **and** $a < b\ c < d$
shows $a = c\ b = d$
 $\langle proof \rangle$

lemma *atLeastLessThan-eq-iff*:
fixes $a\ b\ c\ d :: 'a::linorder$
assumes $a < b\ c < d$
shows $\{a ..< b\} = \{c ..< d\} \longleftrightarrow a = c \wedge b = d$
 $\langle proof \rangle$

lemma (in *linorder*) *Ioc-inj*:

$\langle \{a <.. b\} = \{c <.. d\} \longleftrightarrow (b \leq a \wedge d \leq c) \vee a = c \wedge b = d \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$)
 $\langle proof \rangle$

lemma (in *order*) *Iio-Int-singleton*: $\{..<k\} \cap \{x\} = (if\ x < k\ then\ \{x\}\ else\ \{\})$
 $\langle proof \rangle$

lemma (in *linorder*) *Ioc-subset-iff*: $\{a <.. b\} \subseteq \{c <.. d\} \longleftrightarrow (b \leq a \vee c \leq a \wedge b \leq d)$
 $\langle proof \rangle$

lemma (in *order-bot*) *atLeast-eq-UNIV-iff*: $\{x.. \} = UNIV \longleftrightarrow x = bot$
 $\langle proof \rangle$

lemma (in *order-top*) *atMost-eq-UNIV-iff*: $\{..x\} = UNIV \longleftrightarrow x = top$
 $\langle proof \rangle$

lemma (in *bounded-lattice*) *atLeastAtMost-eq-UNIV-iff*:
 $\{x..y\} = UNIV \longleftrightarrow (x = bot \wedge y = top)$
 $\langle proof \rangle$

lemma *Iio-eq-empty-iff*: $\{..< n::'a::\{linorder, order-bot\}\} = \{\} \longleftrightarrow n = bot$
 $\langle proof \rangle$

lemma *lessThan-empty-iff*: $\{..< n::nat\} = \{\} \longleftrightarrow n = 0$
 $\langle proof \rangle$

lemma *mono-image-least*:

assumes *f-mono*: *mono f* **and** *f-img*: $f ' \{m ..< n\} = \{m' ..< n'\}$ $m < n$
shows $f\ m = m'$
 $\langle proof \rangle$

61.4 Infinite intervals

context *dense-linorder*

begin

lemma *infinite-Ioo*:

assumes $a < b$
shows $\neg finite\ \{a <..< b\}$
 $\langle proof \rangle$

lemma *infinite-Icc*: $a < b \implies \neg finite\ \{a .. b\}$
 $\langle proof \rangle$

lemma *infinite-Ico*: $a < b \implies \neg finite\ \{a ..< b\}$
 $\langle proof \rangle$

lemma *infinite-Ioc*: $a < b \implies \neg finite\ \{a <.. b\}$

$\langle proof \rangle$

lemma *infinite-Ioo-iff* [simp]: *infinite* $\{a < .. < b\} \longleftrightarrow a < b$
 $\langle proof \rangle$

lemma *infinite-Icc-iff* [simp]: *infinite* $\{a .. b\} \longleftrightarrow a < b$
 $\langle proof \rangle$

lemma *infinite-Ico-iff* [simp]: *infinite* $\{a .. < b\} \longleftrightarrow a < b$
 $\langle proof \rangle$

lemma *infinite-Ioc-iff* [simp]: *infinite* $\{a < .. b\} \longleftrightarrow a < b$
 $\langle proof \rangle$

end

lemma *infinite-Iio*: \neg *finite* $\{.. < a :: 'a :: \{no-bot, linorder\}\}$
 $\langle proof \rangle$

lemma *infinite-Iic*: \neg *finite* $\{.. a :: 'a :: \{no-bot, linorder\}\}$
 $\langle proof \rangle$

lemma *infinite-Ioi*: \neg *finite* $\{a :: 'a :: \{no-top, linorder\} < ..\}$
 $\langle proof \rangle$

lemma *infinite-Ici*: \neg *finite* $\{a :: 'a :: \{no-top, linorder\} ..\}$
 $\langle proof \rangle$

61.4.1 Intersection

context *linorder*

begin

lemma *Icc-minus-Ico* [simp]:
assumes $a \leq b$
shows $\{a .. b'\} - \{a .. < b\} = \{b .. b'\}$
 $\langle proof \rangle$

lemma *Icc-minus-Ioc* [simp]:
assumes $a \leq b$
shows $\{a' .. b\} - \{a' < .. b\} = \{a' .. a\}$
 $\langle proof \rangle$

lemma *Icc-minus-Ioo* [simp]:
assumes $a \leq b$
shows $\{a .. b\} - \{a < .. < b\} = \{a, b\}$
 $\langle proof \rangle$

lemma *Int-atLeastAtMost*[simp]: $\{a .. b\} \cap \{c .. d\} = \{\max a c .. \min b d\}$

<proof>

lemma *Int-atLeastAtMostR1[simp]*: $\{..b\} \cap \{c..d\} = \{c .. \min b d\}$
<proof>

lemma *Int-atLeastAtMostR2[simp]*: $\{a.. \} \cap \{c..d\} = \{\max a c .. d\}$
<proof>

lemma *Int-atLeastAtMostL1[simp]*: $\{a..b\} \cap \{..d\} = \{a .. \min b d\}$
<proof>

lemma *Int-atLeastAtMostL2[simp]*: $\{a..b\} \cap \{c.. \} = \{\max a c .. b\}$
<proof>

lemma *Int-atLeastLessThan[simp]*: $\{a..<b\} \cap \{c..<d\} = \{\max a c ..< \min b d\}$
<proof>

lemma *Int-greaterThanAtMost[simp]*: $\{a<..b\} \cap \{c<..d\} = \{\max a c <.. \min b d\}$
<proof>

lemma *Int-greaterThanLessThan[simp]*: $\{a<..<b\} \cap \{c<..<d\} = \{\max a c <..< \min b d\}$
<proof>

lemma *Int-atMost[simp]*: $\{..a\} \cap \{..b\} = \{.. \min a b\}$
<proof>

lemma *Ioc-disjoint*: $\{a<..b\} \cap \{c<..d\} = \{\} \longleftrightarrow b \leq a \vee d \leq c \vee b \leq c \vee d \leq a$
<proof>

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]*: $x < y \implies \text{Sup } \{x <.. y\} = y$
<proof>

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-atLeastLessThan*[simp]: $x < y \implies \text{Inf } \{ x ..< y \} = x$
 <proof>
end

lemma
fixes $x \ y :: 'a :: \{\text{complete-lattice, dense-linorder}\}$
shows *Sup-lessThan*[simp]: $\text{Sup } \{ ..< y \} = y$
and *Sup-atLeastLessThan*[simp]: $x < y \implies \text{Sup } \{ x ..< y \} = y$
and *Sup-greaterThanLessThan*[simp]: $x < y \implies \text{Sup } \{ x <..< y \} = y$
and *Inf-greaterThan*[simp]: $\text{Inf } \{ x <..< \} = x$
and *Inf-greaterThanAtMost*[simp]: $x < y \implies \text{Inf } \{ x <..< y \} = x$
and *Inf-greaterThanLessThan*[simp]: $x < y \implies \text{Inf } \{ x <..<< y \} = x$
 <proof>

61.5 Intervals of natural numbers

61.5.1 The Constant *lessThan*

lemma *lessThan-0* [simp]: $\text{lessThan } (0::\text{nat}) = \{\}$
 <proof>

lemma *lessThan-Suc*: $\text{lessThan } (\text{Suc } k) = \text{insert } k (\text{lessThan } k)$
 <proof>

The following proof is convenient in induction proofs where new elements get indices at the beginning. So it is used to transform $\{ ..< \text{Suc } n \}$ to \emptyset and $\{ ..< n \}$.

lemma *zero-notin-Suc-image* [simp]: $0 \notin \text{Suc } ` A$
 <proof>

lemma *lessThan-Suc-eq-insert-0*: $\{ ..< \text{Suc } n \} = \text{insert } 0 (\text{Suc } ` \{ ..< n \})$
 <proof>

lemma *lessThan-Suc-atMost*: $\text{lessThan } (\text{Suc } k) = \text{atMost } k$
 <proof>

lemma *atMost-Suc-eq-insert-0*: $\{ .. \text{Suc } n \} = \text{insert } 0 (\text{Suc } ` \{ .. n \})$
 <proof>

lemma *UN-lessThan-UNIV*: $(\bigcup m::\text{nat}. \text{lessThan } m) = \text{UNIV}$
 <proof>

61.5.2 The Constant *greaterThan*

lemma *greaterThan-0*: $\text{greaterThan } 0 = \text{range } \text{Suc}$
 <proof>

lemma *greaterThan-Suc*: $\text{greaterThan } (\text{Suc } k) = \text{greaterThan } k - \{ \text{Suc } k \}$

$\langle proof \rangle$

lemma *INT-greaterThan-UNIV*: $(\bigcap m::nat. greaterThan\ m) = \{\}$
 $\langle proof \rangle$

61.5.3 The Constant *atLeast*

lemma *atLeast-0 [simp]*: $atLeast\ (0::nat) = UNIV$
 $\langle proof \rangle$

lemma *atLeast-Suc*: $atLeast\ (Suc\ k) = atLeast\ k - \{k\}$
 $\langle proof \rangle$

lemma *atLeast-Suc-greaterThan*: $atLeast\ (Suc\ k) = greaterThan\ k$
 $\langle proof \rangle$

lemma *UN-atLeast-UNIV*: $(\bigcup m::nat. atLeast\ m) = UNIV$
 $\langle proof \rangle$

61.5.4 The Constant *atMost*

lemma *atMost-0 [simp]*: $atMost\ (0::nat) = \{0\}$
 $\langle proof \rangle$

lemma *atMost-Suc*: $atMost\ (Suc\ k) = insert\ (Suc\ k)\ (atMost\ k)$
 $\langle proof \rangle$

lemma *UN-atMost-UNIV*: $(\bigcup m::nat. atMost\ m) = UNIV$
 $\langle proof \rangle$

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]*: $\{0::nat..<n\} = \{..<n\}$
 $\langle proof \rangle$

lemma *atLeast0AtMost [code-abbrev]*: $\{0..n::nat\} = \{..n\}$
 $\langle proof \rangle$

lemma *lessThan-atLeast0*: $\{..<n\} = \{0::nat..<n\}$
 $\langle proof \rangle$

lemma *atMost-atLeast0*: $\{..n\} = \{0::nat..n\}$
 $\langle proof \rangle$

lemma *atLeastLessThan0*: $\{m..<0::nat\} = \{\}$
 ⟨proof⟩

lemma *atLeast0-lessThan-Suc*: $\{0..<Suc\ n\} = insert\ n\ \{0..<n\}$
 ⟨proof⟩

lemma *atLeast0-lessThan-Suc-eq-insert-0*: $\{0..<Suc\ n\} = insert\ 0\ (Suc\ '\{0..<n\})$
 ⟨proof⟩

61.5.6 The Constant *atLeastAtMost*

lemma *Icc-eq-insert-lb-nat*: $m \leq n \implies \{m..n\} = insert\ m\ \{Suc\ m..n\}$
 ⟨proof⟩

lemma *atLeast0-atMost-Suc*:
 $\{0..Suc\ n\} = insert\ (Suc\ n)\ \{0..n\}$
 ⟨proof⟩

lemma *atLeast0-atMost-Suc-eq-insert-0*:
 $\{0..Suc\ n\} = insert\ 0\ (Suc\ '\{0..n\})$
 ⟨proof⟩

61.5.7 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

lemma *atLeastLessThanSuc*:
 $\{m..<Suc\ n\} = (if\ m \leq n\ then\ insert\ n\ \{m..<n\}\ else\ \{\})$
 ⟨proof⟩

lemma *atLeastLessThan-singleton [simp]*: $\{m..<Suc\ m\} = \{m\}$
 ⟨proof⟩

lemma *atLeastLessThanSuc-atLeastAtMost*: $\{l..<Suc\ u\} = \{l..u\}$
 ⟨proof⟩

lemma *atLeastSucAtMost-greaterThanAtMost*: $\{Suc\ l..u\} = \{l<..u\}$
 ⟨proof⟩

lemma *atLeastSucLessThan-greaterThanLessThan*: $\{Suc\ l..<u\} = \{l<..<u\}$
 ⟨proof⟩

lemma *atLeastAtMostSuc-conv*: $m \leq Suc\ n \implies \{m..Suc\ n\} = insert\ (Suc\ n)\ \{m..n\}$
 ⟨proof⟩

lemma *atLeastAtMost-insertL*: $m \leq n \implies insert\ m\ \{Suc\ m..n\} = \{m..n\}$
 ⟨proof⟩

The analogous result is useful on *int*:

lemma *atLeastAtMostPlus1-int-conv*:

$m \leq 1+n \implies \{m..1+n\} = \text{insert } (1+n) \{m..n::\text{int}\}$
 $\langle \text{proof} \rangle$

lemma *atLeastLessThan-add-Un*: $i \leq j \implies \{i..<j+k\} = \{i..<j\} \cup \{j..<j+k::\text{nat}\}$
 $\langle \text{proof} \rangle$

61.5.8 Intervals and numerals

lemma *lessThan-nat-numeral*: — Evaluation for specific numerals

$\text{lessThan } (\text{numeral } k :: \text{nat}) = \text{insert } (\text{pred-numeral } k) (\text{lessThan } (\text{pred-numeral } k))$
 $\langle \text{proof} \rangle$

lemma *atMost-nat-numeral*: — Evaluation for specific numerals

$\text{atMost } (\text{numeral } k :: \text{nat}) = \text{insert } (\text{numeral } k) (\text{atMost } (\text{pred-numeral } k))$
 $\langle \text{proof} \rangle$

lemma *atLeastLessThan-nat-numeral*: — Evaluation for specific numerals

$\text{atLeastLessThan } m (\text{numeral } k :: \text{nat}) =$
 $(\text{if } m \leq (\text{pred-numeral } k) \text{ then } \text{insert } (\text{pred-numeral } k) (\text{atLeastLessThan } m$
 $(\text{pred-numeral } k))$
 $\text{else } \{\})$
 $\langle \text{proof} \rangle$

61.5.9 Image

context *linordered-semidom*

begin

lemma *image-add-atLeast[simp]*: $\text{plus } k \text{ ‘ } \{i..\} = \{k + i..\}$
 $\langle \text{proof} \rangle$

lemma *image-add-atLeastAtMost [simp]*:
 $\text{plus } k \text{ ‘ } \{i..j\} = \{i + k..j + k\} \text{ (is ?A = ?B)}$
 $\langle \text{proof} \rangle$

lemma *image-add-atLeastAtMost' [simp]*:
 $(\lambda n. n + k) \text{ ‘ } \{i..j\} = \{i + k..j + k\}$
 $\langle \text{proof} \rangle$

lemma *image-add-atLeastLessThan [simp]*:
 $\text{plus } k \text{ ‘ } \{i..<j\} = \{i + k..<j + k\}$
 $\langle \text{proof} \rangle$

lemma *image-add-atLeastLessThan' [simp]*:
 $(\lambda n. n + k) \text{ ‘ } \{i..<j\} = \{i + k..<j + k\}$
 $\langle \text{proof} \rangle$

lemma *image-add-greaterThanAtMost[simp]*: $(+) \text{ } c \text{ ‘ } \{a<..b\} = \{c + a<..c + b\}$

```

    <proof>

end

context ordered-ab-group-add
begin

lemma
  fixes x :: 'a
  shows image-uminus-greaterThan[simp]: uminus ' {x<..} = {.. $-x$ }
  and image-uminus-atLeast[simp]: uminus ' {x..} = {.. $-x$ }
  <proof>

lemma
  fixes x :: 'a
  shows image-uminus-lessThan[simp]: uminus ' {.. $x$ } = { $-x$ .. $\phantom{x}$ }
  and image-uminus-atMost[simp]: uminus ' {..x} = { $-x$ ..}
  <proof>

lemma
  fixes x :: 'a
  shows image-uminus-atLeastAtMost[simp]: uminus ' {x..y} = { $-y$ .. $-x$ }
  and image-uminus-greaterThanAtMost[simp]: uminus ' {x<..y} = { $-y$ .. $-x$ }
  and image-uminus-atLeastLessThan[simp]: uminus ' {x.. $y$ } = { $-y$ <.. $-x$ }
  and image-uminus-greaterThanLessThan[simp]: uminus ' {x<.. $y$ } = { $-y$ <.. $-x$ }
  <proof>

lemma image-add-atMost[simp]: (+) c ' {..a} = {.. $c + a$ }
  <proof>

end

lemma image-Suc-atLeastAtMost [simp]:
  Suc ' {i..j} = {Suc i..Suc j}
  <proof>

lemma image-Suc-atLeastLessThan [simp]:
  Suc ' {i.. $j$ } = {Suc i.. $Suc j$ }
  <proof>

corollary image-Suc-atMost:
  Suc ' {..n} = {1..Suc n}
  <proof>

corollary image-Suc-lessThan:
  Suc ' {.. $n$ } = {1..n}
  <proof>

lemma image-diff-atLeastAtMost [simp]:

```

fixes $d::'a::\text{linordered-idom}$ **shows** $((-) \ d \ ' \ \{a..b\}) = \{d-b..d-a\}$
 $\langle \text{proof} \rangle$

lemma *image-diff-atLeastLessThan* [simp]:
fixes $a \ b \ c::'a::\text{linordered-idom}$
shows $(-) \ c \ ' \ \{a..<b\} = \{c - b <.. c - a\}$
 $\langle \text{proof} \rangle$

lemma *image-minus-const-greaterThanAtMost* [simp]:
fixes $a \ b \ c::'a::\text{linordered-idom}$
shows $(-) \ c \ ' \ \{a <.. b\} = \{c - b.. < c - a\}$
 $\langle \text{proof} \rangle$

lemma *image-minus-const-atLeast* [simp]:
fixes $a \ c::'a::\text{linordered-idom}$
shows $(-) \ c \ ' \ \{a.. \} = \{.. c - a\}$
 $\langle \text{proof} \rangle$

lemma *image-minus-const-AtMost* [simp]:
fixes $b \ c::'a::\text{linordered-idom}$
shows $(-) \ c \ ' \ \{.. b\} = \{c - b.. \}$
 $\langle \text{proof} \rangle$

lemma *image-minus-const-atLeastAtMost'* [simp]:
 $(\lambda t. \ t-d) \ ' \ \{a..b\} = \{a-d..b-d\}$ **for** $d::'a::\text{linordered-idom}$
 $\langle \text{proof} \rangle$

context *linordered-field*
begin

lemma *image-mult-atLeastAtMost* [simp]:
 $((*) \ d \ ' \ \{a..b\}) = \{d*a..d*b\}$ **if** $d>0$
 $\langle \text{proof} \rangle$

lemma *image-divide-atLeastAtMost* [simp]:
 $((\lambda c. \ c / d) \ ' \ \{a..b\}) = \{a/d..b/d\}$ **if** $d>0$
 $\langle \text{proof} \rangle$

lemma *image-mult-atLeastAtMost-if*:
 $(*) \ c \ ' \ \{x .. y\} =$
 $(\text{if } c > 0 \text{ then } \{c * x .. c * y\} \text{ else if } x \leq y \text{ then } \{c * y .. c * x\} \text{ else } \{\})$
 $\langle \text{proof} \rangle$

lemma *image-mult-atLeastAtMost-if'*:
 $(\lambda x. \ x * c) \ ' \ \{x..y\} =$
 $(\text{if } x \leq y \text{ then if } c > 0 \text{ then } \{x * c .. y * c\} \text{ else } \{y * c .. x * c\} \text{ else } \{\})$
 $\langle \text{proof} \rangle$

lemma *image-affinity-atLeastAtMost*:

$((\lambda x. m * x + c) \text{ ‘ } \{a..b\}) = (\text{if } \{a..b\} = \{\} \text{ then } \{\}$
 $\quad \text{else if } 0 \leq m \text{ then } \{m * a + c .. m * b + c\}$
 $\quad \text{else } \{m * b + c .. m * a + c\})$
 <proof>

lemma *image-affinity-atLeastAtMost-diff:*

$((\lambda x. m * x - c) \text{ ‘ } \{a..b\}) = (\text{if } \{a..b\} = \{\} \text{ then } \{\}$
 $\quad \text{else if } 0 \leq m \text{ then } \{m * a - c .. m * b - c\}$
 $\quad \text{else } \{m * b - c .. m * a - c\})$
 <proof>

lemma *image-affinity-atLeastAtMost-div:*

$((\lambda x. x / m + c) \text{ ‘ } \{a..b\}) = (\text{if } \{a..b\} = \{\} \text{ then } \{\}$
 $\quad \text{else if } 0 \leq m \text{ then } \{a / m + c .. b / m + c\}$
 $\quad \text{else } \{b / m + c .. a / m + c\})$
 <proof>

lemma *image-affinity-atLeastAtMost-div-diff:*

$((\lambda x. x / m - c) \text{ ‘ } \{a..b\}) = (\text{if } \{a..b\} = \{\} \text{ then } \{\}$
 $\quad \text{else if } 0 \leq m \text{ then } \{a / m - c .. b / m - c\}$
 $\quad \text{else } \{b / m - c .. a / m - c\})$
 <proof>

end

lemma *atLeast1-lessThan-eq-remove0:*

$\{Suc\ 0..<n\} = \{..<n\} - \{0\}$
 <proof>

lemma *atLeast1-atMost-eq-remove0:*

$\{Suc\ 0..n\} = \{..n\} - \{0\}$
 <proof>

lemma *image-add-int-atLeastLessThan:*

$(\lambda x. x + (l::int)) \text{ ‘ } \{0..<u-l\} = \{l..<u\}$
 <proof>

lemma *image-minus-const-atLeastLessThan-nat:*

fixes $c :: nat$
shows $(\lambda i. i - c) \text{ ‘ } \{x ..< y\} =$
 $(\text{if } c < y \text{ then } \{x - c ..< y - c\} \text{ else if } x < y \text{ then } \{0\} \text{ else } \{\})$
(is - = ?right)
 <proof>

lemma *image-int-atLeastLessThan:*

$int \text{ ‘ } \{a..<b\} = \{int\ a..<int\ b\}$
 <proof>

lemma *image-int-atLeastAtMost:*

int ‘ $\{a..b\} = \{\text{int } a.. \text{int } b\}$
 $\langle \text{proof} \rangle$

61.5.10 Finiteness

lemma *finite-lessThan* [iff]: **fixes** $k :: \text{nat}$ **shows** *finite* $\{..<k\}$
 $\langle \text{proof} \rangle$

lemma *finite-atMost* [iff]: **fixes** $k :: \text{nat}$ **shows** *finite* $\{..k\}$
 $\langle \text{proof} \rangle$

lemma *finite-greaterThanLessThan* [iff]:
fixes $l :: \text{nat}$ **shows** *finite* $\{l<.. $u\}$
 $\langle \text{proof} \rangle$$

lemma *finite-atLeastLessThan* [iff]:
fixes $l :: \text{nat}$ **shows** *finite* $\{l.. $u\}$
 $\langle \text{proof} \rangle$$

lemma *finite-greaterThanAtMost* [iff]:
fixes $l :: \text{nat}$ **shows** *finite* $\{l<.. $u\}$
 $\langle \text{proof} \rangle$$

lemma *finite-atLeastAtMost* [iff]:
fixes $l :: \text{nat}$ **shows** *finite* $\{l.. $u\}$
 $\langle \text{proof} \rangle$$

A bounded set of natural numbers is finite.

lemma *bounded-nat-set-is-finite*: $(\forall i \in N. i < (n :: \text{nat})) \implies \text{finite } N$
 $\langle \text{proof} \rangle$

A set of natural numbers is finite iff it is bounded.

lemma *finite-nat-set-iff-bounded*:
 $\text{finite}(N :: \text{nat set}) = (\exists m. \forall n \in N. n < m)$ (**is** ?F = ?B)
 $\langle \text{proof} \rangle$

lemma *finite-nat-set-iff-bounded-le*: $\text{finite}(N :: \text{nat set}) = (\exists m. \forall n \in N. n \leq m)$
 $\langle \text{proof} \rangle$

lemma *finite-less-ub*:
 $\bigwedge f :: \text{nat} \Rightarrow \text{nat}. (!n. n \leq f n) \implies \text{finite } \{n. f n \leq u\}$
 $\langle \text{proof} \rangle$

lemma *bounded-Max-nat*:
fixes $P :: \text{nat} \Rightarrow \text{bool}$
assumes $x: P x$ **and** $M: \bigwedge x. P x \implies x \leq M$
obtains m **where** $P m \bigwedge x. P x \implies x \leq m$
 $\langle \text{proof} \rangle$

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

lemma *subset-card-intvl-is-intvl*:
assumes $A \subseteq \{k..<k + \text{card } A\}$
shows $A = \{k..<k + \text{card } A\}$
 $\langle \text{proof} \rangle$

61.5.11 Proving Inclusions and Equalities between Unions

lemma *UN-le-eq-UN0*:
 $(\bigcup i \leq n::\text{nat}. M\ i) = (\bigcup i \in \{1..n\}. M\ i) \cup M\ 0$ (**is** $?A = ?B$)
 $\langle \text{proof} \rangle$

lemma *UN-le-add-shift*:
 $(\bigcup i \leq n::\text{nat}. M(i+k)) = (\bigcup i \in \{k..n+k\}. M\ i)$ (**is** $?A = ?B$)
 $\langle \text{proof} \rangle$

lemma *UN-le-add-shift-strict*:
 $(\bigcup i < n::\text{nat}. M(i+k)) = (\bigcup i \in \{k..<n+k\}. M\ i)$ (**is** $?A = ?B$)
 $\langle \text{proof} \rangle$

lemma *UN-UN-finite-eq*: $(\bigcup n::\text{nat}. \bigcup i \in \{0..<n\}. A\ i) = (\bigcup n. A\ n)$
 $\langle \text{proof} \rangle$

lemma *UN-finite-subset*:
 $(\bigwedge n::\text{nat}. (\bigcup i \in \{0..<n\}. A\ i) \subseteq C) \implies (\bigcup n. A\ n) \subseteq C$
 $\langle \text{proof} \rangle$

lemma *UN-finite2-subset*:
assumes $\bigwedge n::\text{nat}. (\bigcup i \in \{0..<n\}. A\ i) \subseteq (\bigcup i \in \{0..<n+k\}. B\ i)$
shows $(\bigcup n. A\ n) \subseteq (\bigcup n. B\ n)$
 $\langle \text{proof} \rangle$

lemma *UN-finite2-eq*:
assumes $(\bigwedge n::\text{nat}. (\bigcup i \in \{0..<n\}. A\ i) = (\bigcup i \in \{0..<n+k\}. B\ i))$
shows $(\bigcup n. A\ n) = (\bigcup n. B\ n)$
 $\langle \text{proof} \rangle$

61.5.12 Cardinality

lemma *card-lessThan [simp]*: $\text{card } \{..<u\} = u$
 $\langle \text{proof} \rangle$

lemma *card-atMost [simp]*: $\text{card } \{..u\} = \text{Suc } u$
 $\langle \text{proof} \rangle$

lemma *card-atLeastLessThan [simp]*: $\text{card } \{l..<u\} = u - l$
 $\langle \text{proof} \rangle$

lemma *card-atLeastAtMost* [simp]: $\text{card } \{l..u\} = \text{Suc } u - l$
 ⟨proof⟩

lemma *card-greaterThanAtMost* [simp]: $\text{card } \{l<..u\} = u - l$
 ⟨proof⟩

lemma *card-greaterThanLessThan* [simp]: $\text{card } \{l<..\} = u - \text{Suc } l$
 ⟨proof⟩

lemma *subset-eq-atLeast0-lessThan-finite*:
 fixes $n :: \text{nat}$
 assumes $N \subseteq \{0..\}$
 shows *finite* N
 ⟨proof⟩

lemma *subset-eq-atLeast0-atMost-finite*:
 fixes $n :: \text{nat}$
 assumes $N \subseteq \{0..n\}$
 shows *finite* N
 ⟨proof⟩

lemma *ex-bij-betw-nat-finite*:
 $\text{finite } M \implies \exists h. \text{bij-betw } h \{0..\} M$
 ⟨proof⟩

lemma *ex-bij-betw-finite-nat*:
 $\text{finite } M \implies \exists h. \text{bij-betw } h M \{0..\}$
 ⟨proof⟩

lemma *finite-same-card-bij*:
 $\text{finite } A \implies \text{finite } B \implies \text{card } A = \text{card } B \implies \exists h. \text{bij-betw } h A B$
 ⟨proof⟩

lemma *ex-bij-betw-nat-finite-1*:
 $\text{finite } M \implies \exists h. \text{bij-betw } h \{1 .. \text{card } M\} M$
 ⟨proof⟩

lemma *bij-betw-iff-card*:
 assumes *finite* A *finite* B
 shows $(\exists f. \text{bij-betw } f A B) \longleftrightarrow (\text{card } A = \text{card } B)$
 ⟨proof⟩

lemma *subset-eq-atLeast0-lessThan-card*:
 fixes $n :: \text{nat}$
 assumes $N \subseteq \{0..\}$
 shows $\text{card } N \leq n$
 ⟨proof⟩

Relational version of *card-inj-on-le*:

lemma *card-le-if-inj-on-rel*:

assumes *finite B*

$\bigwedge a. a \in A \implies \exists b. b \in B \wedge r\ a\ b$

$\bigwedge a1\ a2\ b. \llbracket a1 \in A; a2 \in A; b \in B; r\ a1\ b; r\ a2\ b \rrbracket \implies a1 = a2$

shows $\text{card } A \leq \text{card } B$

<proof>

lemma *inj-on-funpow-least*:

$\langle \text{inj-on } (\lambda k. (f \rightsquigarrow k) s) \{0..<n\} \rangle$

if $\langle (f \rightsquigarrow n) s = s \rangle \langle \bigwedge m. 0 < m \implies m < n \implies (f \rightsquigarrow m) s \neq s \rangle$

<proof>

61.6 Intervals of integers

lemma *atLeastLessThanPlusOne-atLeastAtMost-int*: $\{l..<u+1\} = \{l..(u::\text{int})\}$

<proof>

lemma *atLeastPlusOneAtMost-greaterThanAtMost-int*: $\{l+1..u\} = \{l<..(u::\text{int})\}$

<proof>

lemma *atLeastPlusOneLessThan-greaterThanLessThan-int*:

$\{l+1..<u\} = \{l<..<u::\text{int}\}$

<proof>

61.6.1 Finiteness

lemma *image-atLeastZeroLessThan-int*:

assumes $0 \leq u$

shows $\{(0::\text{int})..<u\} = \text{int } ' \{..<\text{nat } u\}$

<proof>

lemma *finite-atLeastZeroLessThan-int*: *finite* $\{(0::\text{int})..<u\}$

<proof>

lemma *finite-atLeastLessThan-int [iff]*: *finite* $\{l..<u::\text{int}\}$

<proof>

lemma *finite-atLeastAtMost-int [iff]*: *finite* $\{l..(u::\text{int})\}$

<proof>

lemma *finite-greaterThanAtMost-int [iff]*: *finite* $\{l<..(u::\text{int})\}$

<proof>

lemma *finite-greaterThanLessThan-int [iff]*: *finite* $\{l<..<u::\text{int}\}$

<proof>

61.6.2 Cardinality

lemma *card-atLeastZeroLessThan-int*: $\text{card } \{(0::\text{int})..<u\} = \text{nat } u$

<proof>

lemma *card-atLeastLessThan-int* [simp]: $\text{card } \{l..<u\} = \text{nat } (u - l)$
<proof>

lemma *card-atLeastAtMost-int* [simp]: $\text{card } \{l..u\} = \text{nat } (u - l + 1)$
<proof>

lemma *card-greaterThanAtMost-int* [simp]: $\text{card } \{l<..u\} = \text{nat } (u - l)$
<proof>

lemma *card-greaterThanLessThan-int* [simp]: $\text{card } \{l<..
<proof>$

lemma *finite-M-bounded-by-nat*: $\text{finite } \{k. P\ k \wedge k < (i::\text{nat})\}$
<proof>

lemma *card-less*:
 assumes *zero-in-M*: $0 \in M$
 shows $\text{card } \{k \in M. k < \text{Suc } i\} \neq 0$
<proof>

lemma *card-less-Suc2*:
 assumes $0 \notin M$ shows $\text{card } \{k. \text{Suc } k \in M \wedge k < i\} = \text{card } \{k \in M. k < \text{Suc } i\}$
<proof>

lemma *card-less-Suc*:
 assumes $0 \in M$
 shows $\text{Suc } (\text{card } \{k. \text{Suc } k \in M \wedge k < i\}) = \text{card } \{k \in M. k < \text{Suc } i\}$
<proof>

lemma *card-le-Suc-Max*: $\text{finite } S \implies \text{card } S \leq \text{Suc } (\text{Max } S)$
<proof>

lemma *finite-countable-subset*:
 assumes *finite A* and $A: A \subseteq (\bigcup i::\text{nat}. B\ i)$
 obtains n where $A \subseteq (\bigcup i < n. B\ i)$
<proof>

lemma *finite-countable-equals*:
 assumes *finite A* $A = (\bigcup i::\text{nat}. B\ i)$
 obtains n where $A = (\bigcup i < n. B\ i)$
<proof>

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 *ivl-disj-un-singleton*:

$$\begin{aligned}
& \{l::'a::\text{linorder}\} \cup \{l<..\} = \{l..\} \\
& \{..

<proof>$$

One- and two-sided intervals

lemma *ivl-disj-un-one*:

$$\begin{aligned}
& (l::'a::\text{linorder}) < u \implies \{..l\} \cup \{l<..

<proof>$$

Two- and two-sided intervals

lemma *ivl-disj-un-two*:

$$\begin{aligned}
& \llbracket (l::'a::\text{linorder}) < m; m \leq u \rrbracket \implies \{l<..

<proof>$$

lemma *ivl-disj-un-two-touch*:

$$\begin{aligned}
& \llbracket (l::'a::\text{linorder}) < m; m < u \rrbracket \implies \{l<..m\} \cup \{m..

<proof>$$

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*:

$$\begin{aligned} \{..l::'a::order\} \cap \{l<..

<proof>$$

Two- and two-sided intervals

lemma *ivl-disj-int-two*:

$$\begin{aligned} \{l::'a::order<..

<proof>$$

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..

<proof>$$

lemma *(in linorder) lessThan-minus-lessThan [simp]*:

$$\{..

<proof>$$

lemma *(in linorder) atLeastAtMost-diff-ends*:

$$\{a..

<proof>$$

61.7.4 Some Subset Conditions

lemma *ivl-subset [simp]*: $(\{i..$

<proof>

61.8 Generic big monoid operation over intervals

context *semiring-char-0*

begin

lemma *inj-on-of-nat [simp]*:

inj-on of-nat N
 $\langle \text{proof} \rangle$

lemma *bij-betw-of-nat [simp]:*
bij-betw of-nat N A \longleftrightarrow of-nat ‘ N = A
 $\langle \text{proof} \rangle$

lemma *Nats-infinite: infinite (N :: 'a set)*
 $\langle \text{proof} \rangle$

end

context *comm-monoid-set*
begin

lemma *atLeastLessThan-reindex:*
 $F\ g\ \{h\ m..<h\ n\} = F\ (g \circ h)\ \{m..<n\}$
if *bij-betw h {m..<n} {h m..<h n} for m n :: nat*
 $\langle \text{proof} \rangle$

lemma *atLeastAtMost-reindex:*
 $F\ g\ \{h\ m..h\ n\} = F\ (g \circ h)\ \{m..n\}$
if *bij-betw h {m..n} {h m..h n} for m n :: nat*
 $\langle \text{proof} \rangle$

lemma *atLeastLessThan-shift-bounds:*
 $F\ g\ \{m + k..<n + k\} = F\ (g \circ \text{plus } k)\ \{m..<n\}$
for *m n k :: nat*
 $\langle \text{proof} \rangle$

lemma *atLeastAtMost-shift-bounds:*
 $F\ g\ \{m + k..n + k\} = F\ (g \circ \text{plus } k)\ \{m..n\}$
for *m n k :: nat*
 $\langle \text{proof} \rangle$

lemma *atLeast-Suc-lessThan-Suc-shift:*
 $F\ g\ \{\text{Suc } m..<\text{Suc } n\} = F\ (g \circ \text{Suc})\ \{m..<n\}$
 $\langle \text{proof} \rangle$

lemma *atLeast-Suc-atMost-Suc-shift:*
 $F\ g\ \{\text{Suc } m..\text{Suc } n\} = F\ (g \circ \text{Suc})\ \{m..n\}$
 $\langle \text{proof} \rangle$

lemma *atLeast-atMost-pred-shift:*
 $F\ (g \circ (\lambda n. n - \text{Suc } 0))\ \{\text{Suc } m..\text{Suc } n\} = F\ g\ \{m..n\}$
 $\langle \text{proof} \rangle$

lemma *atLeast-lessThan-pred-shift:*
 $F\ (g \circ (\lambda n. n - \text{Suc } 0))\ \{\text{Suc } m..<\text{Suc } n\} = F\ g\ \{m..<n\}$

$\langle proof \rangle$

lemma *atLeast-int-lessThan-int-shift:*

$F\ g\ \{int\ m..<int\ n\} = F\ (g \circ int)\ \{m..<n\}$
 $\langle proof \rangle$

lemma *atLeast-int-atMost-int-shift:*

$F\ g\ \{int\ m..int\ n\} = F\ (g \circ int)\ \{m..n\}$
 $\langle proof \rangle$

lemma *atLeast0-lessThan-Suc:*

$F\ g\ \{0..<Suc\ n\} = F\ g\ \{0..<n\} * g\ n$
 $\langle proof \rangle$

lemma *atLeast0-atMost-Suc:*

$F\ g\ \{0..Suc\ n\} = F\ g\ \{0..n\} * g\ (Suc\ n)$
 $\langle proof \rangle$

lemma *atLeast0-lessThan-Suc-shift:*

$F\ g\ \{0..<Suc\ n\} = g\ 0 * F\ (g \circ Suc)\ \{0..<n\}$
 $\langle proof \rangle$

lemma *atLeast0-atMost-Suc-shift:*

$F\ g\ \{0..Suc\ n\} = g\ 0 * F\ (g \circ Suc)\ \{0..n\}$
 $\langle proof \rangle$

lemma *atLeast-Suc-lessThan:*

$F\ g\ \{m..<n\} = g\ m * F\ g\ \{Suc\ m..<n\}$ **if** $m < n$
 $\langle proof \rangle$

lemma *atLeast-Suc-atMost:*

$F\ g\ \{m..n\} = g\ m * F\ g\ \{Suc\ m..n\}$ **if** $m \leq n$
 $\langle proof \rangle$

lemma *ivl-cong:*

$a = c \implies b = d \implies (\bigwedge x. c \leq x \implies x < d \implies g\ x = h\ x)$
 $\implies F\ g\ \{a..<b\} = F\ h\ \{c..<d\}$
 $\langle proof \rangle$

lemma *atLeastLessThan-shift-0:*

fixes $m\ n\ p :: nat$

shows $F\ g\ \{m..<n\} = F\ (g \circ plus\ m)\ \{0..<n - m\}$
 $\langle proof \rangle$

lemma *atLeastAtMost-shift-0:*

fixes $m\ n\ p :: nat$

assumes $m \leq n$

shows $F\ g\ \{m..n\} = F\ (g \circ plus\ m)\ \{0..n - m\}$
 $\langle proof \rangle$

lemma *atLeastLessThan-concat*:

fixes $m\ n\ p :: \text{nat}$

shows $m \leq n \implies n \leq p \implies F\ g\ \{m..<n\} * F\ g\ \{n..<p\} = F\ g\ \{m..<p\}$

<proof>

lemma *atLeastLessThan-rev*:

$F\ g\ \{n..<m\} = F\ (\lambda i. g\ (m + n - \text{Suc } i))\ \{n..<m\}$

<proof>

lemma *atLeastAtMost-rev*:

fixes $n\ m :: \text{nat}$

shows $F\ g\ \{n..m\} = F\ (\lambda i. g\ (m + n - i))\ \{n..m\}$

<proof>

lemma *atLeastLessThan-rev-at-least-Suc-atMost*:

$F\ g\ \{n..<m\} = F\ (\lambda i. g\ (m + n - i))\ \{\text{Suc } n..m\}$

<proof>

end

61.9 Summation indexed over intervals

syntax (*ASCII*)

-from-to-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{notation} = \langle \text{binder } \text{SUM} \rangle \rangle \text{SUM} - = \dots / - \rangle [0,0,0,10] 10$)

-from-upto-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{notation} = \langle \text{binder } \text{SUM} \rangle \rangle \text{SUM} - = \dots < - / - \rangle [0,0,0,10] 10$)

-upt-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{notation} = \langle \text{binder } \text{SUM} \rangle \rangle \text{SUM} - < - / - \rangle [0,0,10] 10$)

-upto-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{notation} = \langle \text{binder } \text{SUM} \rangle \rangle \text{SUM} - < = - / - \rangle [0,0,10] 10$)

syntax (*latex-sum output*)

-from-to-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$

($\langle \langle \mathcal{S} \sum - = - \rangle \rangle [0,0,0,10] 10$)

-from-upto-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$

($\langle \langle \mathcal{S} \sum - < - \rangle \rangle [0,0,0,10] 10$)

-upt-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$

($\langle \langle \mathcal{S} \sum - < - \rangle \rangle [0,0,10] 10$)

-upto-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$

($\langle \langle \mathcal{S} \sum - \leq - \rangle \rangle [0,0,10] 10$)

syntax

-from-to-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \sum \rangle \rangle \sum - = \dots / - \rangle [0,0,0,10] 10$)

-from-upto-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \sum \rangle \rangle \sum - = \dots < - / - \rangle [0,0,0,10] 10$)

-upt-sum :: $\text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \sum \rangle \rangle \sum - < - / - \rangle$)

```
[0,0,10] 10)
  -upto-sum :: idt => 'a => 'b => 'b  (⟨⟨indent=3 notation=binder ∑⟩⟩∑ -≤-./
-)> [0,0,10] 10)
```

syntax-consts

```
-from-to-sum -from-upto-sum -upt-sum -upto-sum == sum
```

translations

```
∑ x=a..b. t == CONST sum (λx. t) {a..b}
∑ x=a..<b. t == CONST sum (λx. t) {a..<b}
∑ i≤n. t == CONST sum (λi. t) {..n}
∑ i<n. t == CONST sum (λi. t) {..<n}
```

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	L ^A T _E X
$\sum_{x \in \{a..b\}}. e$	$\sum x = a..b. e$	$\sum_{x=a}^b e$
$\sum_{x \in \{a..<b\}}. e$	$\sum x = a..<b. e$	$\sum_{x=a}^{<b} e$
$\sum_{x \in \{..b\}}. e$	$\sum x \leq b. e$	$\sum_{x \leq b} e$
$\sum_{x \in \{..<b\}}. e$	$\sum x < b. e$	$\sum_{x < b} e$

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 L^AT_EX output because it only works well with italic-style formulae, not tt-style.

Note that for uniformity on *nat* it is better to use $\sum x = 0..<n. e$ rather than $\sum x < n. e$: *sum* may not provide all lemmas available for $\{m..<n\}$ also in the special form for $\{..<n\}$.

This congruence rule should be used for sums over intervals as the standard theorem *sum.cong* does not work well with the simplifier who adds the unsimplified premise $x \in B$ to the context.

context *comm-monoid-set*

begin

lemma *zero-middle*:

assumes $1 \leq p \ k \leq p$

shows $F (\lambda j. \text{if } j < k \text{ then } g \ j \text{ else if } j = k \text{ then } \mathbf{1} \text{ else } h \ (j - \text{Suc } 0)) \ \{..p\}$

$= F (\lambda j. \text{if } j < k \text{ then } g \ j \text{ else } h \ j) \ \{..p - \text{Suc } 0\} \ (\text{is } ?lhs = ?rhs)$

<proof>

lemma *atMost-Suc [simp]*:

$F \ g \ \{.. \text{Suc } n\} = F \ g \ \{..n\} * g \ (\text{Suc } n)$

<proof>

lemma *lessThan-Suc* [simp]:

$F\ g\ \{.. $\text{Suc}\ n\} = F\ g\ \{.. $n\} * g\ n$$
 $\langle\text{proof}\rangle$$

lemma *cl-ivl-Suc* [simp]:

$F\ g\ \{m.. $\text{Suc}\ n\} = (\text{if}\ \text{Suc}\ n < m\ \text{then}\ \mathbf{1}\ \text{else}\ F\ g\ \{m.. $n\} * g(\text{Suc}\ n))$$
 $\langle\text{proof}\rangle$$

lemma *op-ivl-Suc* [simp]:

$F\ g\ \{m.. $\text{Suc}\ n\} = (\text{if}\ n < m\ \text{then}\ \mathbf{1}\ \text{else}\ F\ g\ \{m.. $n\} * g(n))$$
 $\langle\text{proof}\rangle$$

lemma *head*:

fixes $n :: \text{nat}$
assumes $mn: m \leq n$
shows $F\ g\ \{m.. $n\} = g\ m * F\ g\ \{m <.. $n\}$ (is ?lhs = ?rhs)$
 $\langle\text{proof}\rangle$$

lemma *last-plus*:

fixes $n :: \text{nat}$ **shows** $m \leq n \implies F\ g\ \{m.. $n\} = g\ n * F\ g\ \{m <.. $n\}$$
 $\langle\text{proof}\rangle$$

lemma *head-if*:

fixes $n :: \text{nat}$
shows $F\ g\ \{m.. $n\} = (\text{if}\ n < m\ \text{then}\ \mathbf{1}\ \text{else}\ F\ g\ \{m <.. $n\} * g(n))$$
 $\langle\text{proof}\rangle$$

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\}$$
 $\langle\text{proof}\rangle$$$

lemma *nat-group*:

fixes $k :: \text{nat}$ **shows** $F\ (\lambda m. F\ g\ \{m * k ..< m * k + k\})\ \{.. $n\} = F\ g\ \{.. $n * k\}$$
 $\langle\text{proof}\rangle$$

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\}$$
 $\langle\text{proof}\rangle$$

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\}$$
 $\langle\text{proof}\rangle$$

lemma *nat-diff-reindex*: $F\ (\lambda i. g\ (n - \text{Suc}\ i))\ \{.. $n\} = F\ g\ \{.. $n\}$$$

$\langle \text{proof} \rangle$

lemma *shift-bounds-nat-ivl*:

$F\ g\ \{m+k..<n+k\} = F\ (\lambda i.\ g(i+k))\{m..<n::nat\}$
 $\langle \text{proof} \rangle$

lemma *shift-bounds-cl-nat-ivl*:

$F\ g\ \{m+k..n+k\} = F\ (\lambda i.\ g(i+k))\{m..n::nat\}$
 $\langle \text{proof} \rangle$

corollary *shift-bounds-cl-Suc-ivl*:

$F\ g\ \{Suc\ m..Suc\ n\} = F\ (\lambda i.\ g(Suc\ i))\{m..n\}$
 $\langle \text{proof} \rangle$

corollary *Suc-reindex-ivl*: $m \leq n \implies F\ g\ \{m..n\} * g\ (Suc\ n) = g\ m * F\ (\lambda i.\ g\ (Suc\ i))\ \{m..n\}$
 $\langle \text{proof} \rangle$

corollary *shift-bounds-Suc-ivl*:

$F\ g\ \{Suc\ m..<Suc\ n\} = F\ (\lambda i.\ g(Suc\ i))\{m..<n\}$
 $\langle \text{proof} \rangle$

lemma *atMost-Suc-shift*:

shows $F\ g\ \{..Suc\ n\} = g\ 0 * F\ (\lambda i.\ g\ (Suc\ i))\ \{..n\}$
 $\langle \text{proof} \rangle$

lemma *lessThan-Suc-shift*:

$F\ g\ \{..<Suc\ n\} = g\ 0 * F\ (\lambda i.\ g\ (Suc\ i))\ \{..<n\}$
 $\langle \text{proof} \rangle$

lemma *atMost-shift*:

$F\ g\ \{..n\} = g\ 0 * F\ (\lambda i.\ g\ (Suc\ i))\ \{..<n\}$
 $\langle \text{proof} \rangle$

lemma *nested-swap*:

$F\ (\lambda i.\ F\ (\lambda j.\ a\ i\ j)\ \{0..<i\})\ \{0..n\} = F\ (\lambda j.\ F\ (\lambda i.\ a\ i\ j)\ \{Suc\ j..n\})\ \{0..<n\}$
 $\langle \text{proof} \rangle$

lemma *nested-swap'*:

$F\ (\lambda i.\ F\ (\lambda j.\ a\ i\ j)\ \{..<i\})\ \{..n\} = F\ (\lambda j.\ F\ (\lambda i.\ a\ i\ j)\ \{Suc\ j..n\})\ \{..<n\}$
 $\langle \text{proof} \rangle$

lemma *atLeast1-atMost-eq*:

$F\ g\ \{Suc\ 0..n\} = F\ (\lambda k.\ g\ (Suc\ k))\ \{..<n\}$
 $\langle \text{proof} \rangle$

lemma *atLeastLessThan-Suc*: $a \leq b \implies F\ g\ \{a..<Suc\ b\} = F\ g\ \{a..<b\} * g\ b$
 $\langle \text{proof} \rangle$

lemma *nat-ivl-Suc'*:

assumes $m \leq \text{Suc } n$

shows $F\ g\ \{m..\text{Suc } n\} = g\ (\text{Suc } n) * F\ g\ \{m..n\}$

<proof>

lemma *in-pairs*: $F\ g\ \{2*m..\text{Suc}(2*n)\} = F\ (\lambda i. g(2*i) * g(\text{Suc}(2*i)))\ \{m..n\}$

<proof>

lemma *in-pairs-0*: $F\ g\ \{..\text{Suc}(2*n)\} = F\ (\lambda i. g(2*i) * g(\text{Suc}(2*i)))\ \{..n\}$

<proof>

end

lemma *card-sum-le-nat-sum*: $\sum \{0..<\text{card } S\} \leq \sum S$

<proof>

lemma *sum-natinterval-diff*:

fixes $f :: \text{nat} \Rightarrow ('a :: \text{ab-group-add})$

shows $\text{sum } (\lambda k. f\ k - f(k + 1))\ \{(m :: \text{nat}) .. n\} =$
 $(\text{if } m \leq n \text{ then } f\ m - f(n + 1) \text{ else } 0)$

<proof>

lemma *sum-diff-nat-ivl*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{ab-group-add}$

shows $\llbracket m \leq n; n \leq p \rrbracket \implies \text{sum } f\ \{m..<p\} - \text{sum } f\ \{m..<n\} = \text{sum } f\ \{n..<p\}$

<proof>

lemma *sum-diff-distrib*: $\forall x. Q\ x \leq P\ x \implies (\sum x < n. P\ x) - (\sum x < n. Q\ x) =$
 $(\sum x < n. P\ x - Q\ x :: \text{nat})$

<proof>

61.9.1 Shifting bounds

context *comm-monoid-add*

begin

context

fixes $f :: \text{nat} \Rightarrow 'a$

assumes $f\ 0 = 0$

begin

lemma *sum-shift-lb-Suc0-0-upt*:

$\text{sum } f\ \{\text{Suc } 0..<k\} = \text{sum } f\ \{0..<k\}$

<proof>

lemma *sum-shift-lb-Suc0-0*: $\text{sum } f\ \{\text{Suc } 0..k\} = \text{sum } f\ \{0..k\}$

<proof>

end

end

lemma *sum-Suc-diff*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{ab-group-add}$
 assumes $m \leq \text{Suc } n$
 shows $(\sum i = m..n. f(\text{Suc } i) - f i) = f (\text{Suc } n) - f m$
 $\langle \text{proof} \rangle$

lemma *sum-Suc-diff'*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{ab-group-add}$
 assumes $m \leq n$
 shows $(\sum i = m..<n. f (\text{Suc } i) - f i) = f n - f m$
 $\langle \text{proof} \rangle$

lemma *sum-diff-split*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{ab-group-add}$
 assumes $m \leq n$
 shows $(\sum i \leq n. f i) - (\sum i < m. f i) = (\sum i \leq n - m. f(n - i))$
 $\langle \text{proof} \rangle$

lemma *prod-divide-nat-ivl*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{idom-divide}$
 shows $\llbracket m \leq n; n \leq p; \text{prod } f \{m..<n\} \neq 0 \rrbracket \implies \text{prod } f \{m..<p\} \text{ div } \text{prod } f \{m..<n\} = \text{prod } f \{n..<p\}$
 $\langle \text{proof} \rangle$

lemma *prod-divide-split*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{idom-divide}$
 assumes $m \leq n$ $\text{prod } f \{..<m\} \neq 0$
 shows $(\text{prod } f \{..n\}) \text{ div } (\text{prod } f \{..<m\}) = \text{prod } (\lambda i. f(n - i)) \{..n - m\}$
 $\langle \text{proof} \rangle$

61.9.2 Telescoping sums

lemma *sum-telescope*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{ab-group-add}$
 shows $\text{sum } (\lambda i. f i - f (\text{Suc } i)) \{.. i\} = f 0 - f (\text{Suc } i)$
 $\langle \text{proof} \rangle$

lemma *sum-telescope''*:

assumes $m \leq n$
 shows $(\sum k \in \{\text{Suc } m..n\}. f k - f (k - 1)) = f n - (f m :: 'a :: \text{ab-group-add})$
 $\langle \text{proof} \rangle$

lemma *sum-lessThan-telescope*:

$(\sum n < m. f (\text{Suc } n) - f n :: 'a :: \text{ab-group-add}) = f m - f 0$
 $\langle \text{proof} \rangle$

lemma *sum-lessThan-telescope'*:
 $(\sum n < m. f\ n - f\ (Suc\ n) :: 'a :: ab-group-add) = f\ 0 - f\ m$
 ⟨proof⟩

61.9.3 The formula for geometric sums

lemma *sum-power2*: $(\sum i=0..<k. (2::nat)^i) = 2^k - 1$
 ⟨proof⟩

lemma *geometric-sum*:
assumes $x \neq 1$
shows $(\sum i < n. x^i) = (x^n - 1) / (x - 1 :: 'a :: field)$
 ⟨proof⟩

lemma *geometric-sum-less*:
assumes $0 < x < 1$ *finite S*
shows $(\sum i \in S. x^i) < 1 / (1 - x :: 'a :: linordered-field)$
 ⟨proof⟩

lemma *diff-power-eq-sum*:
fixes $y :: 'a :: \{comm-ring, monoid-mult\}$
shows

$$x^n (Suc\ n) - y^n (Suc\ n) =$$

$$(x - y) * (\sum p < Suc\ n. (x^p) * y^n (n - p))$$

 ⟨proof⟩

corollary *power-diff-sumr2*: — *COMPLEX-POLYFUN* in HOL Light
fixes $x :: 'a :: \{comm-ring, monoid-mult\}$
shows $x^n - y^n = (x - y) * (\sum i < n. y^i (n - Suc\ i) * x^i)$
 ⟨proof⟩

lemma *power-diff-1-eq*:
fixes $x :: 'a :: \{comm-ring, monoid-mult\}$
shows $x^n - 1 = (x - 1) * (\sum i < n. (x^i))$
 ⟨proof⟩

lemma *one-diff-power-eq'*:
fixes $x :: 'a :: \{comm-ring, monoid-mult\}$
shows $1 - x^n = (1 - x) * (\sum i < n. x^i (n - Suc\ i))$
 ⟨proof⟩

lemma *one-diff-power-eq*:
fixes $x :: 'a :: \{comm-ring, monoid-mult\}$
shows $1 - x^n = (1 - x) * (\sum i < n. x^i)$
 ⟨proof⟩

lemma *sum-gp-basic*:
fixes $x :: 'a :: \{comm-ring, monoid-mult\}$
shows $(1 - x) * (\sum i \leq n. x^i) = 1 - x^{Suc\ n}$

$\langle proof \rangle$

lemma *sum-power-shift*:

fixes $x :: 'a::\{comm-ring, monoid-mult\}$

assumes $m \leq n$

shows $(\sum_{i=m..n}. x^i) = x^m * (\sum_{i \leq n-m}. x^i)$

$\langle proof \rangle$

lemma *sum-gp-multiplied*:

fixes $x :: 'a::\{comm-ring, monoid-mult\}$

assumes $m \leq n$

shows $(1 - x) * (\sum_{i=m..n}. x^i) = x^m - x^{Suc\ n}$

$\langle proof \rangle$

lemma *sum-gp*:

fixes $x :: 'a::\{comm-ring, division-ring\}$

shows $(\sum_{i=m..n}. x^i) =$

$(if\ n < m\ then\ 0$

$else\ if\ x = 1\ then\ of_nat((n + 1) - m)$

$else\ (x^m - x^{Suc\ n}) / (1 - x))$

$\langle proof \rangle$

61.9.4 Geometric progressions

lemma *sum-gp0*:

fixes $x :: 'a::\{comm-ring, division-ring\}$

shows $(\sum_{i \leq n}. x^i) = (if\ x = 1\ then\ of_nat(n + 1)\ else\ (1 - x^{Suc\ n}) / (1 - x))$

$\langle proof \rangle$

lemma *sum-power-add*:

fixes $x :: 'a::\{comm-ring, monoid-mult\}$

shows $(\sum_{i \in I}. x^{m+i}) = x^m * (\sum_{i \in I}. x^i)$

$\langle proof \rangle$

lemma *sum-gp-offset*:

fixes $x :: 'a::\{comm-ring, division-ring\}$

shows $(\sum_{i=m..m+n}. x^i) =$

$(if\ x = 1\ then\ of_nat\ n + 1\ else\ x^m * (1 - x^{Suc\ n}) / (1 - x))$

$\langle proof \rangle$

lemma *sum-gp-strict*:

fixes $x :: 'a::\{comm-ring, division-ring\}$

shows $(\sum_{i < n}. x^i) = (if\ x = 1\ then\ of_nat\ n\ else\ (1 - x^n) / (1 - x))$

$\langle proof \rangle$

61.9.5 The formulae for arithmetic sums

context *comm-semiring-1*

begin

lemma *double-gauss-sum:*

$$2 * (\sum i = 0..n. \text{of-nat } i) = \text{of-nat } n * (\text{of-nat } n + 1)$$

<proof>

lemma *double-gauss-sum-from-Suc-0:*

$$2 * (\sum i = \text{Suc } 0..n. \text{of-nat } i) = \text{of-nat } n * (\text{of-nat } n + 1)$$

<proof>

lemma *double-arith-series:*

$$2 * (\sum i = 0..n. a + \text{of-nat } i * d) = (\text{of-nat } n + 1) * (2 * a + \text{of-nat } n * d)$$

<proof>

end

context *linordered-euclidean-semiring*

begin

lemma *gauss-sum:*

$$(\sum i = 0..n. \text{of-nat } i) = \text{of-nat } n * (\text{of-nat } n + 1) \text{ div } 2$$

<proof>

lemma *gauss-sum-from-Suc-0:*

$$(\sum i = \text{Suc } 0..n. \text{of-nat } i) = \text{of-nat } n * (\text{of-nat } n + 1) \text{ div } 2$$

<proof>

lemma *arith-series:*

$$(\sum i = 0..n. a + \text{of-nat } i * d) = (\text{of-nat } n + 1) * (2 * a + \text{of-nat } n * d) \text{ div } 2$$

<proof>

end

lemma *gauss-sum-nat:*

$$\sum \{0..n\} = (n * \text{Suc } n) \text{ div } 2$$

<proof>

lemma *arith-series-nat:*

$$(\sum i = 0..n. a + i * d) = \text{Suc } n * (2 * a + n * d) \text{ div } 2$$

<proof>

lemma *Sum-Icc-int:*

$$\sum \{m..n\} = (n * (n + 1) - m * (m - 1)) \text{ div } 2$$

if $m \leq n$ **for** $m \ n :: \text{int}$

<proof>

lemma *Sum-Icc-nat:*

$$\sum \{m..n\} = (n * (n + 1) - m * (m - 1)) \text{ div } 2 \text{ for } m \ n :: \text{nat}$$

<proof>

lemma *Sum-Ico-nat*:

$\sum \{m..<n\} = (n * (n - 1) - m * (m - 1)) \text{ div } 2$ **for** $m\ n :: \text{nat}$
 ⟨proof⟩

61.9.6 Division remainder

lemma *range-mod*:

fixes $n :: \text{nat}$
assumes $n > 0$
shows $\text{range } (\lambda m. m \bmod n) = \{0..<n\}$ (**is** $?A = ?B$)
 ⟨proof⟩

61.10 Products indexed over intervals

syntax (*ASCII*)

$\text{-from-to-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{notation} = \langle \text{binder } \text{PROD} \rangle \rangle \text{PROD}$
 $= \text{-} \dots \text{-} / \text{-} \rangle [0,0,0,10] 10$)
 $\text{-from-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{notation} = \langle \text{binder } \text{PROD} \rangle \rangle \text{PROD}$
 $= \text{-} \dots < \text{-} / \text{-} \rangle [0,0,0,10] 10$)
 $\text{-upt-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{notation} = \langle \text{binder } \text{PROD} \rangle \rangle \text{PROD} \text{-} < \text{-} / \text{-} \rangle$
 $[0,0,10] 10$)
 $\text{-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{notation} = \langle \text{binder } \text{PROD} \rangle \rangle \text{PROD} \text{-} < = \text{-} / \text{-} \rangle$
 $[0,0,10] 10$)

syntax (*latex-prod output*)

$\text{-from-to-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $\langle \langle \mathcal{P} \prod \text{-} = \text{-} \rangle [0,0,0,10] 10$)
 $\text{-from-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $\langle \langle \mathcal{P} \prod \text{-} \leq \text{-} \rangle [0,0,0,10] 10$)
 $\text{-upt-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $\langle \langle \mathcal{P} \prod \text{-} < \text{-} \rangle [0,0,10] 10$)
 $\text{-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $\langle \langle \mathcal{P} \prod \text{-} \leq \text{-} \rangle [0,0,10] 10$)

syntax

$\text{-from-to-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \prod \rangle \rangle \prod \text{-}$
 $= \text{-} \dots \text{-} / \text{-} \rangle [0,0,0,10] 10$)
 $\text{-from-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \prod \rangle \rangle \prod \text{-}$
 $= \text{-} \dots < \text{-} / \text{-} \rangle [0,0,0,10] 10$)
 $\text{-upt-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \prod \rangle \rangle \prod \text{-} < \text{-} / \text{-} \rangle$
 $[0,0,10] 10$)
 $\text{-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ($\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \prod \rangle \rangle \prod \text{-} \leq \text{-} / \text{-} \rangle$
 $[0,0,10] 10$)

syntax-consts

$\text{-from-to-prod} \text{-from-upto-prod} \text{-upt-prod} \text{-upto-prod} \equiv \text{prod}$

translations

$\prod x=a..b. t \equiv \text{CONST prod } (\lambda x. t) \{a..b\}$
 $\prod x=a..<b. t \equiv \text{CONST prod } (\lambda x. t) \{a..<b\}$

$$\prod_{i \leq n}. t \equiv \text{CONST prod } (\lambda i. t) \{..n\}$$

$$\prod_{i < n}. t \equiv \text{CONST prod } (\lambda i. t) \{..<n\}$$

lemma *prod-int-plus-eq*: $\text{prod int } \{i..i+j\} = \prod \{\text{int } i.. \text{int } (i+j)\}$
 $\langle \text{proof} \rangle$

lemma *prod-int-eq*: $\text{prod int } \{i..j\} = \prod \{\text{int } i.. \text{int } j\}$
 $\langle \text{proof} \rangle$

61.10.1 Telescoping products

lemma *prod-telescope*:
fixes $f::\text{nat} \Rightarrow 'a::\text{field}$
assumes $\bigwedge i. i \leq n \implies f (\text{Suc } i) \neq 0$
shows $(\prod_{i \leq n}. f \ i / f (\text{Suc } i)) = f \ 0 / f (\text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *prod-telescope''*:
fixes $f::\text{nat} \Rightarrow 'a::\text{field}$
assumes $m \leq n$
assumes $\bigwedge i. i \in \{m..n\} \implies f \ i \neq 0$
shows $(\prod_{i = \text{Suc } m..n}. f \ i / f (i - 1)) = f \ n / f \ m$
 $\langle \text{proof} \rangle$

lemma *prod-lessThan-telescope*:
fixes $f::\text{nat} \Rightarrow 'a::\text{field}$
assumes $\bigwedge i. i \leq n \implies f \ i \neq 0$
shows $(\prod_{i < n}. f (\text{Suc } i) / f \ i) = f \ n / f \ 0$
 $\langle \text{proof} \rangle$

lemma *prod-lessThan-telescope'*:
fixes $f::\text{nat} \Rightarrow 'a::\text{field}$
assumes $\bigwedge i. i \leq n \implies f \ i \neq 0$
shows $(\prod_{i < n}. f \ i / f (\text{Suc } i)) = f \ 0 / f \ n$
 $\langle \text{proof} \rangle$

61.11 Efficient folding over intervals

function *fold-atLeastAtMost-nat* **where**
 $[\text{simp del}]: \text{fold-atLeastAtMost-nat } f \ a \ (b::\text{nat}) \ \text{acc} =$
 $(\text{if } a > b \text{ then } \text{acc} \text{ else } \text{fold-atLeastAtMost-nat } f \ (a+1) \ b \ (f \ a \ \text{acc}))$
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$

lemma *fold-atLeastAtMost-nat*:
assumes *comp-fun-commute* f
shows $\text{fold-atLeastAtMost-nat } f \ a \ b \ \text{acc} = \text{Finite-Set.fold } f \ \text{acc } \{a..b\}$
 $\langle \text{proof} \rangle$

lemma *sum-atLeastAtMost-code*:

$sum\ f\ \{a..b\} = fold_atLeastAtMost_nat\ (\lambda a\ acc.\ f\ a + acc)\ a\ b\ 0$
 $\langle proof \rangle$

lemma *prod-atLeastAtMost-code*:

$prod\ f\ \{a..b\} = fold_atLeastAtMost_nat\ (\lambda a\ acc.\ f\ a * acc)\ a\ b\ 1$
 $\langle proof \rangle$

lemma *pairs-le-eq-Sigma*: $\{(i, j). i + j \leq m\} = Sigma\ (atMost\ m)\ (\lambda r.\ atMost\ (m - r))$
for $m :: nat$
 $\langle proof \rangle$

lemma *sum-up-index-split*: $(\sum k \leq m + n. f\ k) = (\sum k \leq m. f\ k) + (\sum k = Suc\ m..m + n. f\ k)$
 $\langle proof \rangle$

lemma *Sigma-interval-disjoint*: $(SIGMA\ i:A.\ \{..v\ i\}) \cap (SIGMA\ i:A.\ \{v\ i <..w\}) = \{\}$
for $w :: 'a::order$
 $\langle proof \rangle$

lemma *product-atMost-eq-Un*: $A \times \{..m\} = (SIGMA\ i:A.\ \{..m - i\}) \cup (SIGMA\ i:A.\ \{m - i <..m\})$
for $m :: nat$
 $\langle proof \rangle$

lemma *polynomial-product*:

fixes $x :: 'a::idom$
assumes $m: \bigwedge i. i > m \implies a\ i = 0$
and $n: \bigwedge j. j > n \implies b\ j = 0$
shows $(\sum i \leq m. (a\ i) * x^i) * (\sum j \leq n. (b\ j) * x^j) =$
 $(\sum r \leq m + n. (\sum k \leq r. (a\ k) * (b\ (r - k)))) * x^r$
 $\langle proof \rangle$

end

62 Decision Procedure for Presburger Arithmetic

theory *Presburger*

imports *Groebner-Basis Set-Interval*

begin

$\langle ML \rangle$

62.1 The $-\infty$ and $+\infty$ Properties

lemma *minf*:

$$\begin{aligned}
& \llbracket \exists (z :: 'a :: \text{linorder}). \forall x < z. P\ x = P'\ x; \exists z. \forall x < z. Q\ x = Q'\ x \rrbracket \\
& \implies \exists z. \forall x < z. (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x) \\
& \llbracket \exists (z :: 'a :: \text{linorder}). \forall x < z. P\ x = P'\ x; \exists z. \forall x < z. Q\ x = Q'\ x \rrbracket \\
& \implies \exists z. \forall x < z. (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x) \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x = t) = \text{False} \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x \neq t) = \text{True} \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x < t) = \text{True} \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x \leq t) = \text{True} \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x > t) = \text{False} \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x \geq t) = \text{False} \\
& \exists z. \forall (x :: 'b :: \{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x < z. (d\ \text{dvd}\ x + s) = (d\ \text{dvd}\ x + s) \\
& \exists z. \forall (x :: 'b :: \{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x < z. (\neg d\ \text{dvd}\ x + s) = (\neg d\ \text{dvd}\ x + s) \\
& \exists z. \forall x < z. F = F \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *pinf*:

$$\begin{aligned}
& \llbracket \exists (z :: 'a :: \text{linorder}). \forall x > z. P\ x = P'\ x; \exists z. \forall x > z. Q\ x = Q'\ x \rrbracket \\
& \implies \exists z. \forall x > z. (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x) \\
& \llbracket \exists (z :: 'a :: \text{linorder}). \forall x > z. P\ x = P'\ x; \exists z. \forall x > z. Q\ x = Q'\ x \rrbracket \\
& \implies \exists z. \forall x > z. (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x) \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x = t) = \text{False} \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x \neq t) = \text{True} \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x < t) = \text{False} \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x \leq t) = \text{False} \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x > t) = \text{True} \\
& \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x \geq t) = \text{True} \\
& \exists z. \forall (x :: 'b :: \{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x > z. (d\ \text{dvd}\ x + s) = (d\ \text{dvd}\ x + s) \\
& \exists z. \forall (x :: 'b :: \{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x > z. (\neg d\ \text{dvd}\ x + s) = (\neg d\ \text{dvd}\ x + s) \\
& \exists z. \forall x > z. F = F \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *inf-period*:

$$\begin{aligned}
& \llbracket \forall x\ k. P\ x = P\ (x - k * D); \forall x\ k. Q\ x = Q\ (x - k * D) \rrbracket \\
& \implies \forall x\ k. (P\ x \wedge Q\ x) = (P\ (x - k * D) \wedge Q\ (x - k * D)) \\
& \llbracket \forall x\ k. P\ x = P\ (x - k * D); \forall x\ k. Q\ x = Q\ (x - k * D) \rrbracket \\
& \implies \forall x\ k. (P\ x \vee Q\ x) = (P\ (x - k * D) \vee Q\ (x - k * D)) \\
& (d :: 'a :: \{\text{comm-ring}, \text{Rings.dvd}\})\ \text{dvd}\ D \implies \forall x\ k. (d\ \text{dvd}\ x + t) = (d\ \text{dvd}\ (x - k * D) + t) \\
& (d :: 'a :: \{\text{comm-ring}, \text{Rings.dvd}\})\ \text{dvd}\ D \implies \forall x\ k. (\neg d\ \text{dvd}\ x + t) = (\neg d\ \text{dvd}\ (x - k * D) + t) \\
& \forall x\ k. F = F \\
& \langle \text{proof} \rangle
\end{aligned}$$

62.2 The A and B sets

lemma *bset*:

$$\begin{aligned}
& \llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow P\ x \longrightarrow P\ (x - D) ; \\
& \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow Q\ x \longrightarrow Q\ (x - D) \rrbracket \implies \\
& \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P\ (x - D) \wedge Q\ (x - D))
\end{aligned}$$

$$\begin{aligned}
& D)) \\
& \llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow P x \longrightarrow P(x - D) ; \\
& \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow Q x \longrightarrow Q(x - D) \rrbracket \Longrightarrow \\
& \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (P x \vee Q x) \longrightarrow (P(x - D) \vee Q(x - D)) \\
& \llbracket D > 0 ; t - 1 \in B \rrbracket \Longrightarrow (\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t)) \\
& \llbracket D > 0 ; t \in B \rrbracket \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t)) \\
& D > 0 \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t)) \\
& D > 0 \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t)) \\
& \llbracket D > 0 ; t \in B \rrbracket \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t)) \\
& \llbracket D > 0 ; t - 1 \in B \rrbracket \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t)) \\
& d \text{ dvd } D \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (d \text{ dvd } x + t) \longrightarrow (d \text{ dvd } (x - D) + t)) \\
& d \text{ dvd } D \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (\neg d \text{ dvd } x + t) \longrightarrow (\neg d \text{ dvd } (x - D) + t)) \\
& \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow F \longrightarrow F \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma aset:

$$\begin{aligned}
& \llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow P x \longrightarrow P(x + D) ; \\
& \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow Q x \longrightarrow Q(x + D) \rrbracket \Longrightarrow \\
& \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (P x \wedge Q x) \longrightarrow (P(x + D) \wedge Q(x + D)) \\
& \llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow P x \longrightarrow P(x + D) ; \\
& \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow Q x \longrightarrow Q(x + D) \rrbracket \Longrightarrow \\
& \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (P x \vee Q x) \longrightarrow (P(x + D) \vee Q(x + D)) \\
& \llbracket D > 0 ; t + 1 \in A \rrbracket \Longrightarrow (\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x = t) \longrightarrow (x + D = t)) \\
& \llbracket D > 0 ; t \in A \rrbracket \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \neq t) \longrightarrow (x + D \neq t)) \\
& \llbracket D > 0 ; t \in A \rrbracket \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x < t) \longrightarrow (x + D < t)) \\
& \llbracket D > 0 ; t + 1 \in A \rrbracket \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \leq t) \longrightarrow (x + D \leq t)) \\
& D > 0 \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x > t) \longrightarrow (x + D > t)) \\
& D > 0 \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \geq t) \longrightarrow (x + D \geq t)) \\
& d \text{ dvd } D \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (d \text{ dvd } x + t) \longrightarrow (d \text{ dvd } (x + D) + t)) \\
& d \text{ dvd } D \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (\neg d \text{ dvd } x + t) \longrightarrow (\neg d \text{ dvd } (x + D) + t))
\end{aligned}$$

$\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow F \longrightarrow F$
 $\langle proof \rangle$

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*: $(0::int) < d$ **and** *modd*: $\forall x k. P\ x = P(x - k*d)$

shows $(\exists x. P\ x) = (\exists j \in \{1..d\}. P\ j)$

(**is** *?LHS* = *?RHS*)

$\langle proof \rangle$

62.3.2 The $-\infty$ Version

lemma *decr-lemma*: $0 < (d::int) \implies x - (|x - z| + 1) * d < z$

$\langle proof \rangle$

lemma *incr-lemma*: $0 < (d::int) \implies z < x + (|x - z| + 1) * d$

$\langle proof \rangle$

lemma *decr-mult-lemma*:

assumes *dpos*: $(0::int) < d$ **and** *minus*: $\forall x. P\ x \longrightarrow P(x - d)$ **and** *knneg*: $0 \leq k$

shows $\forall x. P\ x \longrightarrow P(x - k*d)$

$\langle proof \rangle$

lemma *minusinfinity*:

assumes *dpos*: $0 < d$ **and**

P1eqP1: $\forall x k. P1\ x = P1(x - k*d)$ **and** *ePeqP1*: $\exists z::int. \forall x. x < z \longrightarrow (P\ x = P1\ x)$

shows $(\exists x. P1\ x) \longrightarrow (\exists x. P\ x)$

$\langle proof \rangle$

lemma *cpmi*:

assumes *dp*: $0 < D$ **and** *p1*: $\exists z. \forall x < z. P\ x = P'\ x$

and *nb*: $\forall x. (\forall j \in \{1..D\}. \forall (b::int) \in B. x \neq b+j) \longrightarrow P\ (x) \longrightarrow P\ (x - D)$

and *pd*: $\forall x k. P'\ x = P'\ (x - k*D)$

shows $(\exists x. P\ x) = ((\exists j \in \{1..D\}. P'\ j) \vee (\exists j \in \{1..D\}. \exists b \in B. P\ (b+j)))$

(**is** *?L* = (*?R1* \vee *?R2*))

$\langle proof \rangle$

62.3.3 The $+\infty$ Version

lemma *plusinfinity*:

assumes *dpos*: $(0::int) < d$ **and**

P1eqP1: $\forall x k. P'\ x = P'\ (x - k*d)$ **and** *ePeqP1*: $\exists z. \forall x > z. P\ x = P'\ x$

shows $(\exists x. P'\ x) \longrightarrow (\exists x. P\ x)$

$\langle proof \rangle$

lemma *incr-mult-lemma*:

assumes *dpos*: $(0::int) < d$ **and** *plus*: $\forall x::int. P\ x \longrightarrow P(x + d)$ **and** *knneg*: $0 \leq k$
shows $\forall x. P\ x \longrightarrow P(x + k*d)$
 $\langle proof \rangle$

lemma *cpai*:

assumes *dp*: $0 < D$ **and** *p1*: $\exists z. \forall x > z. P\ x = P'\ x$
and *nb*: $\forall x. (\forall j \in \{1..D\}. \forall (b::int) \in A. x \neq b - j) \longrightarrow P\ (x) \longrightarrow P\ (x + D)$
and *pd*: $\forall x\ k. P'\ x = P'\ (x - k*D)$
shows $(\exists x. P\ x) = ((\exists j \in \{1..D\}. P'\ j) \vee (\exists j \in \{1..D\}. \exists b \in A. P\ (b - j)))$
(is *?L* = (*?R1* \vee *?R2*))
 $\langle proof \rangle$

lemma *simp-from-to*: $\{i..j::int\} = (\text{if } j < i \text{ then } \{\} \text{ else insert } i \{i+1..j\})$
 $\langle proof \rangle$

theorem *unity-coeff-ex*: $(\exists (x::'a::\{semiring-0, Rings.dvd\}). P\ (l * x)) \equiv (\exists x. l\ dvd\ (x + 0) \wedge P\ x)$
 $\langle proof \rangle$

lemma *zdvd-mono*:

fixes *k m t* :: *int*
assumes $k \neq 0$
shows $m\ dvd\ t \equiv k * m\ dvd\ k * t$
 $\langle proof \rangle$

lemma *uminus-dvd-conv*:

fixes *d t* :: *int*
shows $d\ dvd\ t \equiv -\ d\ dvd\ t$ **and** $d\ dvd\ t \equiv d\ dvd\ -\ t$
 $\langle proof \rangle$

Theorems for transforming predicates on *nat* to predicates on *int*

lemma *zdiff-int-split*: $P\ (int\ (x - y)) = ((y \leq x \longrightarrow P\ (int\ x - int\ y)) \wedge (x < y \longrightarrow P\ 0))$
 $\langle proof \rangle$

Specific instances of congruence rules, to prevent simplifier from looping.

theorem *imp-le-cong*:

$\llbracket x = x'; 0 \leq x' \implies P = P' \rrbracket \implies (0 \leq (x::int) \longrightarrow P) = (0 \leq x' \longrightarrow P')$
 $\langle proof \rangle$

theorem *conj-le-cong*:

$\llbracket x = x'; 0 \leq x' \implies P = P' \rrbracket \implies (0 \leq (x::int) \wedge P) = (0 \leq x' \wedge P')$
 $\langle proof \rangle$

$\langle ML \rangle$

```

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]:  $m \bmod 2 = (1::nat) \longleftrightarrow \neg 2 \text{ dvd } m$  <proof>
lemma [presburger, algebra]:  $m \bmod 2 = \text{Suc } 0 \longleftrightarrow \neg 2 \text{ dvd } m$  <proof>
lemma [presburger, algebra]:  $m \bmod (\text{Suc } (\text{Suc } 0)) = (1::nat) \longleftrightarrow \neg 2 \text{ dvd } m$ 
<proof>
lemma [presburger, algebra]:  $m \bmod (\text{Suc } (\text{Suc } 0)) = \text{Suc } 0 \longleftrightarrow \neg 2 \text{ dvd } m$  <proof>
lemma [presburger, algebra]:  $m \bmod 2 = (1::int) \longleftrightarrow \neg 2 \text{ dvd } m$  <proof>

context semiring-parity
begin

declare even-mult-iff [presburger]

declare even-power [presburger]

lemma [presburger]:
   $\text{even } (a + b) \longleftrightarrow \text{even } a \wedge \text{even } b \vee \text{odd } a \wedge \text{odd } b$ 
  <proof>

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 div Suc (Suc 0) = n div Suc (Suc 0)  $\longleftrightarrow$  even n
  <proof>

declare even-diff-nat [presburger]

lemma [presburger]:
  fixes k :: int
  shows (k + 1) div 2 = k div 2  $\longleftrightarrow$  even k
  <proof>

lemma [presburger]:
  fixes k :: int
  shows (k + 1) div 2 = k div 2 + 1  $\longleftrightarrow$  odd k
  <proof>

lemma [presburger]:
  even n  $\longleftrightarrow$  even (int n)
  <proof>

```

62.4 Nice facts about division by $4 :: 'a$

```

lemma even-even-mod-4-iff:
  even (n::nat)  $\longleftrightarrow$  even (n mod 4)
  <proof>

lemma odd-mod-4-div-2:
  n mod 4 = (3::nat)  $\implies$  odd ((n - Suc 0) div 2)
  <proof>

lemma even-mod-4-div-2:
  n mod 4 = Suc 0  $\implies$  even ((n - Suc 0) div 2)
  <proof>

end

```

```

theory Try0-HOL
  imports Try0 Presburger
begin

```

$\langle ML \rangle$

```
declare [[try0-schedule =
  satxmetis
  orderpresburgerlinarithalgebraargo
  simpautoblastfastfastforceforcemeson
]]

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:  $(\exists y. P \longleftrightarrow Q y) \longleftrightarrow (P \longrightarrow (\exists y. Q y)) \wedge ((\forall y. Q y) \longrightarrow P)$ 
   $\langle proof \rangle$ 
```

$\langle ML \rangle$

```
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.

```
typedecl 'a symb-list
```

```
consts
```

```
  Symb-Nil :: 'a symb-list
  Symb-Cons :: 'a  $\Rightarrow$  'a symb-list  $\Rightarrow$  'a symb-list
```

```
typedecl pattern
```

consts

pat :: 'a \Rightarrow pattern
nopat :: 'a \Rightarrow pattern

definition *trigger* :: pattern symb-list symb-list \Rightarrow bool \Rightarrow 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 \Rightarrow '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*)
 ⟨*proof*⟩

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
 ⟨*proof*⟩

lemma *nat-one-as-int*:
 1 = nat 1
 ⟨*proof*⟩

lemma *nat-numeral-as-int*: *numeral* = ($\lambda i. \text{nat } (\text{numeral } i)$) ⟨*proof*⟩
lemma *nat-less-as-int*: (<) = ($\lambda a b. \text{int } a < \text{int } b$) ⟨*proof*⟩
lemma *nat-leq-as-int*: (\leq) = ($\lambda a b. \text{int } a \leq \text{int } b$) ⟨*proof*⟩
lemma *Suc-as-int*: *Suc* = ($\lambda a. \text{nat } (\text{int } a + 1)$) ⟨*proof*⟩
lemma *nat-plus-as-int*: (+) = ($\lambda a b. \text{nat } (\text{int } a + \text{int } b)$) ⟨*proof*⟩
lemma *nat-minus-as-int*: (−) = ($\lambda a b. \text{nat } (\text{int } a - \text{int } b)$) ⟨*proof*⟩
lemma *nat-times-as-int*: (*) = ($\lambda a b. \text{nat } (\text{int } a * \text{int } b)$) ⟨*proof*⟩
lemma *nat-div-as-int*: (div) = ($\lambda a b. \text{nat } (\text{int } a \text{ div } \text{int } b)$) ⟨*proof*⟩

lemma *nat-mod-as-int*: $(\text{mod}) = (\lambda a b. \text{nat } (\text{int } a \text{ mod int } b)) \langle \text{proof} \rangle$

lemma *int-Suc*: $\text{int } (\text{Suc } n) = \text{int } n + 1 \langle \text{proof} \rangle$

lemma *int-plus*: $\text{int } (n + m) = \text{int } n + \text{int } m \langle \text{proof} \rangle$

lemma *int-minus*: $\text{int } (n - m) = \text{int } (\text{nat } (\text{int } n - \text{int } m)) \langle \text{proof} \rangle$

lemma *nat-int-comparison*:

fixes $a b :: \text{nat}$

shows $(a = b) = (\text{int } a = \text{int } b)$

and $(a < b) = (\text{int } a < \text{int } b)$

and $(a \leq b) = (\text{int } a \leq \text{int } b)$

$\langle \text{proof} \rangle$

lemma *int-ops*:

fixes $a b :: \text{nat}$

shows $\text{int } 0 = 0$

and $\text{int } 1 = 1$

and $\text{int } (\text{numeral } n) = \text{numeral } n$

and $\text{int } (\text{Suc } a) = \text{int } a + 1$

and $\text{int } (a + b) = \text{int } a + \text{int } b$

and $\text{int } (a - b) = (\text{if } \text{int } a < \text{int } b \text{ then } 0 \text{ else } \text{int } a - \text{int } b)$

and $\text{int } (a * b) = \text{int } a * \text{int } b$

and $\text{int } (a \text{ div } b) = \text{int } a \text{ div int } b$

and $\text{int } (a \text{ mod } b) = \text{int } a \text{ mod int } b$

$\langle \text{proof} \rangle$

lemma *int-if*:

fixes $a b :: \text{nat}$

shows $\text{int } (\text{if } P \text{ then } a \text{ else } b) = (\text{if } P \text{ then int } a \text{ else int } b)$

$\langle \text{proof} \rangle$

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 *div-as-z3div* and *mod-as-z3mod* theorems.

definition *z3div* :: $\text{int} \Rightarrow \text{int} \Rightarrow \text{int}$ **where**

$\text{z3div } k l = (\text{if } l \geq 0 \text{ then } k \text{ div } l \text{ else } -(k \text{ div } -l))$

definition *z3mod* :: $\text{int} \Rightarrow \text{int} \Rightarrow \text{int}$ **where**

$\text{z3mod } k l = k \text{ mod } (\text{if } l \geq 0 \text{ then } l \text{ else } -l)$

lemma *div-as-z3div*:

$\forall k l. k \text{ div } l = (\text{if } l = 0 \text{ then } 0 \text{ else if } l > 0 \text{ then } \text{z3div } k l \text{ else } \text{z3div } (-k) (-l))$

$\langle \text{proof} \rangle$

lemma *mod-as-z3mod*:

$\forall k l. k \text{ mod } l = (\text{if } l = 0 \text{ then } k \text{ else if } l > 0 \text{ then } \text{z3mod } k l \text{ else } -\text{z3mod } (-k)$

$(-l))$

$\langle proof \rangle$

63.6 Extra theorems for veriT reconstruction

lemma *verit-sko-forall*: $\langle (\forall x. P\ x) \longleftrightarrow P\ (SOME\ x. \neg P\ x) \rangle$
 $\langle proof \rangle$

lemma *verit-sko-forall'*: $\langle P\ (SOME\ x. \neg P\ x) = A \implies (\forall x. P\ x) = A \rangle$
 $\langle proof \rangle$

lemma *verit-sko-forall''*: $\langle B = A \implies (SOME\ x. P\ x) = A \equiv (SOME\ x. P\ x) = B \rangle$
 $\langle proof \rangle$

lemma *verit-sko-forall-indirect*: $\langle x = (SOME\ x. \neg P\ x) \implies (\forall x. P\ x) \longleftrightarrow P\ x \rangle$
 $\langle proof \rangle$

lemma *verit-sko-forall-indirect2*:
 $\langle x = (SOME\ x. \neg P\ x) \implies (\bigwedge x :: 'a. (P\ x = P'\ x)) \implies (\forall x. P'\ x) \longleftrightarrow P\ x \rangle$
 $\langle proof \rangle$

lemma *verit-sko-ex*: $\langle (\exists x. P\ x) \longleftrightarrow P\ (SOME\ x. P\ x) \rangle$
 $\langle proof \rangle$

lemma *verit-sko-ex'*: $\langle P\ (SOME\ x. P\ x) = A \implies (\exists x. P\ x) = A \rangle$
 $\langle proof \rangle$

lemma *verit-sko-ex-indirect*: $\langle x = (SOME\ x. P\ x) \implies (\exists x. P\ x) \longleftrightarrow P\ x \rangle$
 $\langle proof \rangle$

lemma *verit-sko-ex-indirect2*: $\langle x = (SOME\ x. P\ x) \implies (\bigwedge x. P\ x = P'\ x) \implies (\exists x. P'\ x) \longleftrightarrow P\ x \rangle$
 $\langle proof \rangle$

lemma *verit-Pure-trans*:
 $\langle P \equiv Q \implies Q \implies P \rangle$
 $\langle proof \rangle$

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 $\langle (if\ b\ then\ x\ else\ y) \equiv (if\ c\ then\ u\ else\ v) \rangle$
 $\langle proof \rangle$

lemma *verit-if-weak-cong'*:
 $\langle b \equiv c \implies (if\ b\ then\ x\ else\ y) \equiv (if\ c\ then\ x\ else\ y) \rangle$
 $\langle proof \rangle$

lemma *verit-or-neg*:

$$\begin{aligned} &\langle (A \Longrightarrow B) \Longrightarrow B \vee \neg A \rangle \\ &\langle (\neg A \Longrightarrow B) \Longrightarrow B \vee A \rangle \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *verit-subst-bool*: $\langle P \Longrightarrow f \text{ True} \Longrightarrow f P \rangle$

$\langle \text{proof} \rangle$

lemma *verit-and-pos*:

$$\begin{aligned} &\langle (a \Longrightarrow \neg(b \wedge c) \vee A) \Longrightarrow \neg(a \wedge b \wedge c) \vee A \rangle \\ &\langle (a \Longrightarrow b \Longrightarrow A) \Longrightarrow \neg(a \wedge b) \vee A \rangle \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *verit-farkas*:

$$\begin{aligned} &\langle (a \Longrightarrow A) \Longrightarrow \neg a \vee A \rangle \\ &\langle (\neg a \Longrightarrow A) \Longrightarrow a \vee A \rangle \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *verit-or-pos*:

$$\begin{aligned} &\langle A \wedge A' \Longrightarrow (c \wedge A) \vee (\neg c \wedge A') \rangle \\ &\langle A \wedge A' \Longrightarrow (\neg c \wedge A) \vee (c \wedge A') \rangle \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *verit-la-generic*:

$$\begin{aligned} &\langle (a::\text{int}) \leq x \vee a = x \vee a \geq x \rangle \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *verit-bfun-elim*:

$$\begin{aligned} &\langle (\text{if } b \text{ then } P \text{ True else } P \text{ False}) = P \ b \rangle \\ &\langle (\forall b. P' \ b) = (P' \text{ False} \wedge P' \text{ True}) \rangle \\ &\langle (\exists b. P' \ b) = (P' \text{ False} \vee P' \text{ True}) \rangle \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *verit-eq-true-simplify*:

$$\begin{aligned} &\langle (P = \text{True}) \equiv P \rangle \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *verit-and-neg*:

$$\begin{aligned} &\langle (a \Longrightarrow \neg b \vee A) \Longrightarrow \neg(a \wedge b) \vee A \rangle \\ &\langle (a \Longrightarrow A) \Longrightarrow \neg a \vee A \rangle \\ &\langle (\neg a \Longrightarrow A) \Longrightarrow a \vee A \rangle \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *verit-forall-inst*:

$$\begin{aligned} &\langle A \longleftrightarrow B \Longrightarrow \neg A \vee B \rangle \\ &\langle \neg A \longleftrightarrow B \Longrightarrow A \vee B \rangle \\ &\langle A \longleftrightarrow B \Longrightarrow \neg B \vee A \rangle \\ &\langle A \longleftrightarrow \neg B \Longrightarrow B \vee A \rangle \end{aligned}$$

$\langle A \longrightarrow B \Longrightarrow \neg A \vee B \rangle$
 $\langle \neg A \longrightarrow B \Longrightarrow A \vee B \rangle$
 $\langle \text{proof} \rangle$

lemma *verit-eq-transitive*:

$\langle A = B \Longrightarrow B = C \Longrightarrow A = C \rangle$
 $\langle A = B \Longrightarrow C = B \Longrightarrow A = C \rangle$
 $\langle B = A \Longrightarrow B = C \Longrightarrow A = C \rangle$
 $\langle B = A \Longrightarrow C = B \Longrightarrow A = C \rangle$
 $\langle \text{proof} \rangle$

lemma *verit-bool-simplify*:

$\langle \neg(P \longrightarrow Q) \longleftrightarrow P \wedge \neg Q \rangle$
 $\langle \neg(P \vee Q) \longleftrightarrow \neg P \wedge \neg Q \rangle$
 $\langle \neg(P \wedge Q) \longleftrightarrow \neg P \vee \neg Q \rangle$
 $\langle (P \longrightarrow (Q \longrightarrow R)) \longleftrightarrow ((P \wedge Q) \longrightarrow R) \rangle$
 $\langle ((P \longrightarrow Q) \longrightarrow Q) \longleftrightarrow P \vee Q \rangle$
 $\langle (Q \longleftrightarrow (P \vee Q)) \longleftrightarrow (P \longrightarrow Q) \rangle$ — This rule was inverted
 $\langle P \wedge (P \longrightarrow Q) \longleftrightarrow P \wedge Q \rangle$
 $\langle (P \longrightarrow Q) \wedge P \longleftrightarrow P \wedge Q \rangle$

 $\langle \text{proof} \rangle$

We need the last equation for $\neg (\forall a\ b. \neg P\ a\ b)$

lemma *verit-connective-def*: — the definition of XOR is missing as the operator is not generated by Isabelle

$\langle (A = B) \longleftrightarrow ((A \longrightarrow B) \wedge (B \longrightarrow A)) \rangle$
 $\langle (\text{If } A\ B\ C) = ((A \longrightarrow B) \wedge (\neg A \longrightarrow C)) \rangle$
 $\langle (\exists x. P\ x) \longleftrightarrow \neg(\forall x. \neg P\ x) \rangle$
 $\langle \neg(\exists x. P\ x) \longleftrightarrow (\forall x. \neg P\ x) \rangle$
 $\langle \text{proof} \rangle$

lemma *verit-ite-simplify*:

$\langle (\text{If } \text{True}\ B\ C) = B \rangle$
 $\langle (\text{If } \text{False}\ B\ C) = C \rangle$
 $\langle (\text{If } A'\ B\ B) = B \rangle$
 $\langle (\text{If } (\neg A')\ B\ C) = (\text{If } A'\ C\ B) \rangle$
 $\langle (\text{If } c\ (\text{If } c\ A\ B)\ C) = (\text{If } c\ A\ C) \rangle$
 $\langle (\text{If } c\ C\ (\text{If } c\ A\ B)) = (\text{If } c\ C\ B) \rangle$
 $\langle (\text{If } A'\ \text{True}\ \text{False}) = A' \rangle$
 $\langle (\text{If } A'\ \text{False}\ \text{True}) \longleftrightarrow \neg A' \rangle$
 $\langle (\text{If } A'\ \text{True}\ B') \longleftrightarrow A' \vee B' \rangle$
 $\langle (\text{If } A'\ B'\ \text{False}) \longleftrightarrow A' \wedge B' \rangle$
 $\langle (\text{If } A'\ \text{False}\ B') \longleftrightarrow \neg A' \wedge B' \rangle$
 $\langle (\text{If } A'\ B'\ \text{True}) \longleftrightarrow \neg A' \vee B' \rangle$
 $\langle x \wedge \text{True} \longleftrightarrow x \rangle$
 $\langle x \vee \text{False} \longleftrightarrow x \rangle$
for $B\ C :: 'a$ **and** $A'\ B'\ C' :: \text{bool}$
 $\langle \text{proof} \rangle$

lemma *verit-and-simplify1*:

$\langle \text{True} \wedge b \longleftrightarrow b \rangle$
 $\langle b \wedge \text{True} \longleftrightarrow b \rangle$
 $\langle \text{False} \wedge b \longleftrightarrow \text{False} \rangle$
 $\langle b \wedge \text{False} \longleftrightarrow \text{False} \rangle$
 $\langle (c \wedge \neg c) \longleftrightarrow \text{False} \rangle$
 $\langle (\neg c \wedge c) \longleftrightarrow \text{False} \rangle$
 $\langle \neg \neg a = a \rangle$
 $\langle \text{proof} \rangle$

lemmas *verit-and-simplify* = *conj-ac de-Morgan-conj disj-not1*

lemma *verit-or-simplify-1*:

$\langle \text{False} \vee b \longleftrightarrow b \rangle$
 $\langle b \vee \text{False} \longleftrightarrow b \rangle$
 $\langle b \vee \neg b \rangle$
 $\langle \neg b \vee b \rangle$
 $\langle \text{proof} \rangle$

lemmas *verit-or-simplify* = *disj-ac*

lemma *verit-not-simplify*:

$\langle \neg \neg b \longleftrightarrow b \rangle$
 $\langle \neg \text{True} \longleftrightarrow \text{False} \rangle$
 $\langle \neg \text{False} \longleftrightarrow \text{True} \rangle$
 $\langle \text{proof} \rangle$

lemma *verit-implies-simplify*:

$\langle (\neg a \longrightarrow \neg b) \longleftrightarrow (b \longrightarrow a) \rangle$
 $\langle (\text{False} \longrightarrow a) \longleftrightarrow \text{True} \rangle$
 $\langle (a \longrightarrow \text{True}) \longleftrightarrow \text{True} \rangle$
 $\langle (\text{True} \longrightarrow a) \longleftrightarrow a \rangle$
 $\langle (a \longrightarrow \text{False}) \longleftrightarrow \neg a \rangle$
 $\langle (a \longrightarrow a) \longleftrightarrow \text{True} \rangle$
 $\langle (\neg a \longrightarrow a) \longleftrightarrow a \rangle$
 $\langle (a \longrightarrow \neg a) \longleftrightarrow \neg a \rangle$
 $\langle ((a \longrightarrow b) \longrightarrow b) \longleftrightarrow a \vee b \rangle$
 $\langle \text{proof} \rangle$

lemma *verit-equiv-simplify*:

$\langle ((\neg a) = (\neg b)) \longleftrightarrow (a = b) \rangle$
 $\langle (a = a) \longleftrightarrow \text{True} \rangle$
 $\langle (a = (\neg a)) \longleftrightarrow \text{False} \rangle$
 $\langle ((\neg a) = a) \longleftrightarrow \text{False} \rangle$
 $\langle (\text{True} = a) \longleftrightarrow a \rangle$
 $\langle (a = \text{True}) \longleftrightarrow a \rangle$
 $\langle (\text{False} = a) \longleftrightarrow \neg a \rangle$
 $\langle (a = \text{False}) \longleftrightarrow \neg a \rangle$
 $\langle \neg \neg a \longleftrightarrow a \rangle$
 $\langle (\neg \text{False}) = \text{True} \rangle$
for $a\ b :: \text{bool}$
 $\langle \text{proof} \rangle$

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*:
 $\langle a :: 'a :: \text{cancel-comm-monoid-add} \rangle - a = 0 \rangle$
 $\langle a :: 'a :: \text{cancel-comm-monoid-add} \rangle - 0 = a \rangle$
 $\langle 0 - (b :: 'b :: \{\text{group-add}\}) = -b \rangle$
 $\langle - (- (b :: 'b :: \text{group-add})) = b \rangle$
 $\langle \text{proof} \rangle$

lemma *verit-sum-simplify*:
 $\langle a :: 'a :: \text{cancel-comm-monoid-add} \rangle + 0 = a \rangle$
 $\langle \text{proof} \rangle$

lemmas *verit-prod-simplify* =
mult-1
mult-1-right

lemma *verit-comp-simplify1*:
 $\langle a :: 'a :: \text{order} \rangle < a \longleftrightarrow \text{False} \rangle$
 $\langle a \leq a \rangle$
 $\langle \neg(b' \leq a') \longleftrightarrow (a' :: 'b :: \text{linorder}) < b' \rangle$
 $\langle \text{proof} \rangle$

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*:
 $\langle a :: 'a :: \text{linorder} \rangle = b \vee \neg a \leq b \vee \neg b \leq a \rangle$
 $\langle \text{proof} \rangle$

context
begin

For the reconstruction, we need to keep the order of the arguments.

named-theorems *smt-arith-multiplication* $\langle \text{Theorems to reconstruct arithmetic theorems.} \rangle$

named-theorems *smt-arith-combine* $\langle \text{Theorems to reconstruct arithmetic theo-} \rangle$

rems.⟩

named-theorems *smt-arith-simplify* ⟨Theorems to combine theorems in the LA procedure⟩

lemmas [*smt-arith-simplify*] =
div-add dvd-numeral-simp divmod-steps less-num-simps le-num-simps if-True
if-False divmod-cancel
dvd-mult dvd-mult2 less-irrefl prod.case numeral-plus-one divmod-step-def or-
der.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 nu-
meral-mod-numeral
divmod-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
divmod-steps rel-simps if-True if-False numeral-div-numeral divmod-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-uminus-conv-diff mult.left-neutral semiring-class.distrib-right
add-diff-cancel-left' add-diff-eq ring-distrib mult-minus-left minus-diff-eq

lemma [*smt-arith-simplify*]:
 ⟨ $\neg (a' :: 'a :: \text{linorder}) < b' \longleftrightarrow b' \leq a'$ ⟩
 ⟨ $\neg (a' :: 'a :: \text{linorder}) \leq b' \longleftrightarrow b' < a'$ ⟩
 ⟨ $(c::\text{int}) \bmod \text{Numeral1} = 0$ ⟩
 ⟨ $(a::\text{nat}) \bmod \text{Numeral1} = 0$ ⟩
 ⟨ $(c::\text{int}) \text{div Numeral1} = c$ ⟩
 ⟨ $a \text{div Numeral1} = a$ ⟩
 ⟨ $(c::\text{int}) \bmod 1 = 0$ ⟩
 ⟨ $a \bmod 1 = 0$ ⟩
 ⟨ $(c::\text{int}) \text{div } 1 = c$ ⟩
 ⟨ $a \text{div } 1 = a$ ⟩
 ⟨ $\neg(a' \neq b') \longleftrightarrow a' = b'$ ⟩
 ⟨*proof*⟩

lemma *div-mod-decomp*: $A = (A \text{div } n) * n + (A \bmod n)$ **for** $A :: \text{nat}$
 ⟨*proof*⟩

lemma *div-less-mono*:
fixes $A B :: \text{nat}$

assumes $A < B$ $0 < n$ **and**
 $\text{mod}: A \text{ mod } n = 0 B \text{ mod } n = 0$
shows $(A \text{ div } n) < (B \text{ div } n)$
 $\langle \text{proof} \rangle$

lemma *verit-le-mono-div*:
fixes $A B :: \text{nat}$
assumes $A < B$ $0 < n$
shows $(A \text{ div } n) + (\text{if } B \text{ mod } n = 0 \text{ then } 1 \text{ else } 0) \leq (B \text{ div } n)$
 $\langle \text{proof} \rangle$

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*: $A = (A \text{ div } n) * n + (A \text{ mod } n)$ **for** $A :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *zdiv-mono-strict*:
fixes $A B :: \text{int}$
assumes $A < B$ $0 < n$ **and**
 $\text{mod}: A \text{ mod } n = 0 B \text{ mod } n = 0$
shows $(A \text{ div } n) < (B \text{ div } n)$
 $\langle \text{proof} \rangle$

lemma *verit-le-mono-div-int*:
 $\langle A \text{ div } n + (\text{if } B \text{ mod } n = 0 \text{ then } 1 \text{ else } 0) \leq B \text{ div } n \rangle$
if $\langle A < B \rangle \langle 0 < n \rangle$
for $A B n :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *verit-less-mono-div-int2*:
fixes $A B :: \text{int}$
assumes $A \leq B$ $0 < -n$
shows $(A \text{ div } n) \geq (B \text{ div } n)$
 $\langle \text{proof} \rangle$

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*] =
 $\text{arg-cong}[\text{of } - - \langle \lambda a :: \text{nat}. a \text{ div } n * p \rangle \text{ for } n p :: \text{nat}, \text{ THEN sym}]$
 $\text{arg-cong}[\text{of } - - \langle \lambda a :: \text{int}. a \text{ div } n * p \rangle \text{ for } n p :: \text{int}, \text{ THEN sym}]$

lemma [*smt-arith-combine*]:
 $a < b \implies c < d \implies a + c + 2 \leq b + d$
 $a < b \implies c \leq d \implies a + c + 1 \leq b + d$
 $a \leq b \implies c < d \implies a + c + 1 \leq b + d$ **for** $a b c :: \text{int}$

$\langle \text{proof} \rangle$

lemma $[\text{smt-arith-combine}]$:

$a < b \implies c < d \implies a + c + 2 \leq b + d$
 $a < b \implies c \leq d \implies a + c + 1 \leq b + d$
 $a \leq b \implies c < d \implies a + c + 1 \leq b + d$ **for** $a\ b\ c :: \text{nat}$
 $\langle \text{proof} \rangle$

lemmas $[\text{smt-arith-combine}] =$

add-strict-mono
 add-less-le-mono
 add-mono
 add-le-less-mono

lemma $[\text{smt-arith-combine}]$:

$\langle m < n \implies c = d \implies m + c < n + d \rangle$
 $\langle m \leq n \implies c = d \implies m + c \leq n + d \rangle$
 $\langle c = d \implies m < n \implies m + c < n + d \rangle$
 $\langle c = d \implies m \leq n \implies m + c \leq n + d \rangle$
for $m :: \langle 'a :: \text{ordered-cancel-ab-semigroup-add} \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{verit-negate-coefficient}$:

$\langle a \leq (b :: 'a :: \{\text{ordered-ab-group-add}\}) \implies -a \geq -b \rangle$
 $\langle a < b \implies -a > -b \rangle$
 $\langle a = b \implies -a = -b \rangle$
 $\langle \text{proof} \rangle$

end

lemma verit-ite-intro :

$\langle (\text{if } a \text{ then } P \text{ (if } a \text{ then } a' \text{ else } b') \text{ else } Q) \longleftrightarrow (\text{if } a \text{ then } P \text{ } a' \text{ else } Q) \rangle$
 $\langle (\text{if } a \text{ then } P' \text{ else } Q' \text{ (if } a \text{ then } a' \text{ else } b')) \longleftrightarrow (\text{if } a \text{ then } P' \text{ else } Q' \text{ } b') \rangle$
 $\langle A = f \text{ (if } a \text{ then } R \text{ else } S) \longleftrightarrow (\text{if } a \text{ then } A = f R \text{ else } A = f S) \rangle$
 $\langle \text{proof} \rangle$

lemma verit-ite-if-cong :

fixes $x\ y :: \text{bool}$
assumes $b = c$
and $c \equiv \text{True} \implies x = u$
and $c \equiv \text{False} \implies y = v$
shows $(\text{if } b \text{ then } x \text{ else } y) \equiv (\text{if } c \text{ then } u \text{ else } v)$
 $\langle \text{proof} \rangle$

63.7 Setup

$\langle ML \rangle$

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.

declare *[[smt-solver = z3]]*

Since SMT solvers are potentially nonterminating, there is a timeout (given in seconds) to restrict their runtime.

declare *[[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.

declare *[[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.

declare *[[smt-oracle = 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.

declare *[[cvc4-options =]]*

declare *[[cvc5-options =]]*

declare *[[cvc5-proof-options = --proof-format-mode=alethe --proof-granularity=dsl-rewrite]]*

declare *[[verit-options =]]*

declare *[[z3-options =]]*

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.

declare *[[smt-infer-triggers = 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.

declare *[[cvc-extensions = 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.

declare *[[z3-extensions = false]]*

63.10 Certificates

By setting the option *smt-certificates* to the name of a file, all following applications of an SMT solver are 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.

declare `[[smt-certificates =]]`

The option *smt-read-only-certificates* controls whether only stored certificates should be used or invocation of an SMT solver is allowed. When set to *true*, no SMT solver will ever be invoked and only the existing certificates found in the configured cache are used; when set to *false* and there is no cached certificate for some proposition, then the configured SMT solver is invoked.

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 *false*.

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 *smt-trace* should be set to *true*.

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 *z3-rule* are tried prior to any implemented reconstruction procedure for all uncertain Z3 proof rules; the facts in *z3-simp* are only fed to invocations of the simplifier when reconstructing theory-specific proof steps.

lemmas `[z3-rule] =`

*refl eq-commute conj-commute disj-commute simp-thms nnf-simps
ring-distrib field-simps times-divide-eq-right times-divide-eq-left
if-True if-False not-not
NO-MATCH-def*

lemma [z3-rule]:

$$\begin{aligned}
 (P \wedge Q) &= (\neg (\neg P \vee \neg Q)) \\
 (P \wedge Q) &= (\neg (\neg Q \vee \neg P)) \\
 (\neg P \wedge Q) &= (\neg (P \vee \neg Q)) \\
 (\neg P \wedge Q) &= (\neg (\neg Q \vee P)) \\
 (P \wedge \neg Q) &= (\neg (\neg P \vee Q)) \\
 (P \wedge \neg Q) &= (\neg (Q \vee \neg P)) \\
 (\neg P \wedge \neg Q) &= (\neg (P \vee Q)) \\
 (\neg P \wedge \neg Q) &= (\neg (Q \vee P)) \\
 \langle proof \rangle
 \end{aligned}$$

lemma [z3-rule]:

$$\begin{aligned}
 (P \longrightarrow Q) &= (Q \vee \neg P) \\
 (\neg P \longrightarrow Q) &= (P \vee Q) \\
 (\neg P \longrightarrow Q) &= (Q \vee P) \\
 (True \longrightarrow P) &= P \\
 (P \longrightarrow True) &= True \\
 (False \longrightarrow P) &= True \\
 (P \longrightarrow P) &= True \\
 (\neg (A \longleftrightarrow \neg B)) &\longleftrightarrow (A \longleftrightarrow B) \\
 \langle proof \rangle
 \end{aligned}$$

lemma [z3-rule]:

$$\begin{aligned}
 ((P = Q) \longrightarrow R) &= (R \vee (Q = (\neg P))) \\
 \langle proof \rangle
 \end{aligned}$$

lemma [z3-rule]:

$$\begin{aligned}
 (\neg True) &= False \\
 (\neg False) &= True \\
 (x = x) &= True \\
 (P = True) &= P \\
 (True = P) &= P \\
 (P = False) &= (\neg P) \\
 (False = P) &= (\neg P) \\
 ((\neg P) = P) &= False \\
 (P = (\neg P)) &= False \\
 ((\neg P) = (\neg Q)) &= (P = Q) \\
 \neg (P = (\neg Q)) &= (P = Q) \\
 \neg ((\neg P) = Q) &= (P = Q) \\
 (P \neq Q) &= (Q = (\neg P)) \\
 (P = Q) &= ((\neg P \vee Q) \wedge (P \vee \neg Q)) \\
 (P \neq Q) &= ((\neg P \vee \neg Q) \wedge (P \vee Q)) \\
 \langle proof \rangle
 \end{aligned}$$

lemma [z3-rule]:

$$\begin{aligned}
 (if\ P\ then\ P\ else\ \neg P) &= True \\
 (if\ \neg P\ then\ \neg P\ else\ P) &= True \\
 (if\ P\ then\ True\ else\ False) &= P
 \end{aligned}$$

$(\text{if } P \text{ then False else True}) = (\neg P)$
 $(\text{if } P \text{ then } Q \text{ else True}) = ((\neg P) \vee Q)$
 $(\text{if } P \text{ then } Q \text{ else True}) = (Q \vee (\neg P))$
 $(\text{if } P \text{ then } Q \text{ else } \neg Q) = (P = Q)$
 $(\text{if } P \text{ then } Q \text{ else } \neg Q) = (Q = P)$
 $(\text{if } P \text{ then } \neg Q \text{ else } Q) = (P = (\neg Q))$
 $(\text{if } P \text{ then } \neg Q \text{ else } Q) = ((\neg Q) = P)$
 $(\text{if } \neg P \text{ then } x \text{ else } y) = (\text{if } P \text{ then } y \text{ else } x)$
 $(\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } x) = (\text{if } P \wedge (\neg Q) \text{ then } y \text{ else } x)$
 $(\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } x) = (\text{if } (\neg Q) \wedge P \text{ then } y \text{ else } x)$
 $(\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } y) = (\text{if } P \wedge Q \text{ then } x \text{ else } y)$
 $(\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } y) = (\text{if } Q \wedge P \text{ then } x \text{ else } y)$
 $(\text{if } P \text{ then } x \text{ else if } P \text{ then } y \text{ else } z) = (\text{if } P \text{ then } x \text{ else } z)$
 $(\text{if } P \text{ then } x \text{ else if } Q \text{ then } x \text{ else } y) = (\text{if } P \vee Q \text{ then } x \text{ else } y)$
 $(\text{if } P \text{ then } x \text{ else if } Q \text{ then } x \text{ else } y) = (\text{if } Q \vee P \text{ then } x \text{ else } y)$
 $(\text{if } P \text{ then } x = y \text{ else } x = z) = (x = (\text{if } P \text{ then } y \text{ else } z))$
 $(\text{if } P \text{ then } x = y \text{ else } y = z) = (y = (\text{if } P \text{ then } x \text{ else } z))$
 $(\text{if } P \text{ then } x = y \text{ else } z = y) = (y = (\text{if } P \text{ then } x \text{ else } z))$
 $\langle \text{proof} \rangle$

lemma [z3-rule]:

$0 + (x::\text{int}) = x$
 $x + 0 = x$
 $x + x = 2 * x$
 $0 * x = 0$
 $1 * x = x$
 $x + y = y + x$
 $\langle \text{proof} \rangle$

lemma [z3-rule]:

$P = Q \vee P \vee Q$
 $P = Q \vee \neg P \vee \neg Q$
 $(\neg P) = Q \vee \neg P \vee Q$
 $(\neg P) = Q \vee P \vee \neg Q$
 $P = (\neg Q) \vee \neg P \vee Q$
 $P = (\neg Q) \vee P \vee \neg Q$
 $P \neq Q \vee P \vee \neg Q$
 $P \neq Q \vee \neg P \vee Q$
 $P \neq (\neg Q) \vee P \vee Q$
 $(\neg P) \neq Q \vee P \vee Q$
 $P \vee Q \vee P \neq (\neg Q)$
 $P \vee Q \vee (\neg P) \neq Q$
 $P \vee \neg Q \vee P \neq Q$
 $\neg P \vee Q \vee P \neq Q$
 $P \vee y = (\text{if } P \text{ then } x \text{ else } y)$
 $P \vee (\text{if } P \text{ then } x \text{ else } y) = y$
 $\neg P \vee x = (\text{if } P \text{ then } x \text{ else } y)$
 $\neg P \vee (\text{if } P \text{ then } x \text{ else } y) = x$
 $P \vee R \vee \neg (\text{if } P \text{ then } Q \text{ else } R)$

```

  ¬ P ∨ Q ∨ ¬ (if P then Q else R)
  ¬ (if P then Q else R) ∨ ¬ P ∨ Q
  ¬ (if P then Q else R) ∨ P ∨ R
  (if P then Q else R) ∨ ¬ P ∨ ¬ Q
  (if P then Q else R) ∨ P ∨ ¬ R
  (if P then ¬ Q else R) ∨ ¬ P ∨ Q
  (if P then Q else ¬ R) ∨ P ∨ R
  ⟨proof⟩

```

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

— FIXME: *HOL.Try0-HOL* has to be imported first so that *try0-schedule* gets the value assigned value there. Otherwise, the value is the one assigned in *HOL.Try0*, which is imported transitively by both *HOL.Presburger* and *HOL.SMT*. It seems that, when merging the attributes from two theories, the value assigned in the leftmost theory has precedence.

Try0-HOL

Presburger

SMT

keywords

sledgehammer :: *diag* **and**

sledgehammer-params :: *thy-decl*

begin

⟨ML⟩

end

65 Setup for Lifting/Transfer for the set type

theory *Lifting-Set*

imports *Lifting Groups-Big*

begin

65.1 Relator and predicate properties

lemma *rel-setD1*: $\llbracket \text{rel-set } R \ A \ B; x \in A \rrbracket \implies \exists y \in B. R \ x \ y$

and *rel-setD2*: $\llbracket \text{rel-set } R \ A \ B; y \in B \rrbracket \implies \exists x \in A. R \ x \ y$

⟨proof⟩

lemma *rel-set-conversep* [*simp*]: $\text{rel-set } A^{-1-1} = (\text{rel-set } A)^{-1-1}$

$\langle \text{proof} \rangle$

lemma *rel-set-eq* [*relator-eq*]: *rel-set* (=) = (=)
 $\langle \text{proof} \rangle$

lemma *rel-set-mono* [*relator-mono*]:
assumes $A \leq B$
shows *rel-set* $A \leq$ *rel-set* B
 $\langle \text{proof} \rangle$

lemma *rel-set-OO* [*relator-distr*]: *rel-set* R *OO* *rel-set* S = *rel-set* (R *OO* S)
 $\langle \text{proof} \rangle$

lemma *Domainp-set* [*relator-domain*]:
 $\text{Domainp } (\text{rel-set } T) = (\lambda A. \text{Ball } A (\text{Domainp } T))$
 $\langle \text{proof} \rangle$

lemma *left-total-rel-set* [*transfer-rule*]:
 $\text{left-total } A \implies \text{left-total } (\text{rel-set } A)$
 $\langle \text{proof} \rangle$

lemma *left-unique-rel-set* [*transfer-rule*]:
 $\text{left-unique } A \implies \text{left-unique } (\text{rel-set } A)$
 $\langle \text{proof} \rangle$

lemma *right-total-rel-set* [*transfer-rule*]:
 $\text{right-total } A \implies \text{right-total } (\text{rel-set } A)$
 $\langle \text{proof} \rangle$

lemma *right-unique-rel-set* [*transfer-rule*]:
 $\text{right-unique } A \implies \text{right-unique } (\text{rel-set } A)$
 $\langle \text{proof} \rangle$

lemma *bi-total-rel-set* [*transfer-rule*]:
 $\text{bi-total } A \implies \text{bi-total } (\text{rel-set } A)$
 $\langle \text{proof} \rangle$

lemma *bi-unique-rel-set* [*transfer-rule*]:
 $\text{bi-unique } A \implies \text{bi-unique } (\text{rel-set } A)$
 $\langle \text{proof} \rangle$

lemma *set-relator-eq-onp* [*relator-eq-onp*]:
 $\text{rel-set } (\text{eq-onp } P) = \text{eq-onp } (\lambda A. \text{Ball } A P)$
 $\langle \text{proof} \rangle$

lemma *bi-unique-rel-set-lemma*:
assumes *bi-unique* R **and** *rel-set* R X Y
obtains f **where** $Y = \text{image } f X$ **and** *inj-on* f X **and** $\forall x \in X. R \ x \ (f \ x)$
 $\langle \text{proof} \rangle$

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*)
<proof>

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*) {} {}
<proof>

lemma *insert-transfer* [*transfer-rule*]:
 (*A* \implies *rel-set* *A* \implies *rel-set* *A*) *insert insert*
<proof>

lemma *union-transfer* [*transfer-rule*]:
 (*rel-set* *A* \implies *rel-set* *A* \implies *rel-set* *A*) *union union*
<proof>

lemma *Union-transfer* [*transfer-rule*]:
 (*rel-set* (*rel-set* *A*) \implies *rel-set* *A*) *Union Union*
<proof>

lemma *image-transfer* [*transfer-rule*]:
 ((*A* \implies *B*) \implies *rel-set* *A* \implies *rel-set* *B*) *image image*
<proof>

lemma *UNION-transfer* [*transfer-rule*]: — TODO deletion candidate
 (*rel-set* *A* \implies (*A* \implies *rel-set* *B*) \implies *rel-set* *B*) ($\lambda A f. \bigcup (f \text{ ‘ } A)$) ($\lambda A f. \bigcup (f \text{ ‘ } A)$)
<proof>

lemma *Ball-transfer* [*transfer-rule*]:
 (*rel-set* *A* \implies (*A* \implies (=)) \implies (=)) *Ball Ball*
<proof>

lemma *Bex-transfer* [*transfer-rule*]:
 (*rel-set* *A* \implies (*A* \implies (=)) \implies (=)) *Bex Bex*
<proof>

lemma *Pow-transfer* [*transfer-rule*]:
 (*rel-set* *A* \implies *rel-set* (*rel-set* *A*)) *Pow Pow*
<proof>

lemma *rel-set-transfer* [*transfer-rule*]:

$((A \text{ rel-set } B \text{ rel-set } (=)) \text{ rel-set } A \text{ rel-set } B \text{ rel-set } (=)) \text{ rel-set } rel\text{-set}$
 $\langle proof \rangle$

lemma *bind-transfer* [*transfer-rule*]:

$(rel\text{-set } A \text{ rel-set } (A \text{ rel-set } B) \text{ rel-set } B) \text{ Set.bind Set.bind}$
 $\langle proof \rangle$

lemma *INF-parametric* [*transfer-rule*]: — TODO deletion candidate

$(rel\text{-set } A \text{ rel-set } (A \text{ rel-set } HOL.eq) \text{ rel-set } HOL.eq) (\lambda A f. Inf (f ' A)) (\lambda A f. Inf (f ' A))$
 $\langle proof \rangle$

lemma *SUP-parametric* [*transfer-rule*]: — TODO deletion candidate

$(rel\text{-set } R \text{ rel-set } (R \text{ rel-set } HOL.eq) \text{ rel-set } HOL.eq) (\lambda A f. Sup (f ' A)) (\lambda A f. Sup (f ' A))$
 $\langle proof \rangle$

65.3.2 Rules requiring bi-unique, bi-total or right-total relations

lemma *member-transfer* [*transfer-rule*]:

assumes *bi-unique* *A*
shows $(A \text{ rel-set } A \text{ rel-set } (=)) (\in) (\in)$
 $\langle proof \rangle$

lemma *right-total-Collect-transfer* [*transfer-rule*]:

assumes *right-total* *A*
shows $((A \text{ rel-set } (=)) \text{ rel-set } A) (\lambda P. Collect (\lambda x. P x \wedge Domainp A x)) Collect$
 $\langle proof \rangle$

lemma *Collect-transfer* [*transfer-rule*]:

assumes *bi-total* *A*
shows $((A \text{ rel-set } (=)) \text{ rel-set } A) Collect Collect$
 $\langle proof \rangle$

lemma *inter-transfer* [*transfer-rule*]:

assumes *bi-unique* *A*
shows $(rel\text{-set } A \text{ rel-set } A \text{ rel-set } A) inter inter$
 $\langle proof \rangle$

lemma *Diff-transfer* [*transfer-rule*]:

assumes *bi-unique* *A*
shows $(rel\text{-set } A \text{ rel-set } A \text{ rel-set } A) (-) (-)$
 $\langle proof \rangle$

lemma *subset-transfer* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique* *A*

```

shows (rel-set A  $\implies$  rel-set A  $\implies$  (=)) ( $\subseteq$ ) ( $\subseteq$ )
 $\langle$ proof $\rangle$ 

context
  includes lifting-syntax
begin

lemma strict-subset-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (rel-set A  $\implies$  rel-set A  $\implies$  (=)) ( $\subset$ ) ( $\subset$ )
   $\langle$ proof $\rangle$ 

end

declare right-total-UNIV-transfer[transfer-rule]

lemma UNIV-transfer [transfer-rule]:
  assumes bi-total A
  shows (rel-set A) UNIV UNIV
   $\langle$ proof $\rangle$ 

lemma right-total-Compl-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: right-total A
  shows (rel-set A  $\implies$  rel-set A) ( $\lambda S.$  uminus S  $\cap$  Collect (Domainp A))
uminus
   $\langle$ proof $\rangle$ 

lemma Compl-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: bi-total A
  shows (rel-set A  $\implies$  rel-set A) uminus uminus
   $\langle$ proof $\rangle$ 

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)  $\implies$  rel-set A) ( $\lambda S.$   $\bigcap S \cap$  Collect (Domainp A))
Inter
   $\langle$ proof $\rangle$ 

lemma Inter-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: bi-total A
  shows (rel-set (rel-set A)  $\implies$  rel-set A) Inter Inter
   $\langle$ proof $\rangle$ 

lemma filter-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows ((A  $\implies$  (=))  $\implies$  rel-set A  $\implies$  rel-set A) Set.filter Set.filter
   $\langle$ proof $\rangle$ 

lemma finite-transfer [transfer-rule]:

```


bi-unique $A \implies (\text{rel-set } A \implies (=)) \text{ finite finite}$
 $\langle \text{proof} \rangle$

lemma *card-transfer* [*transfer-rule*]:
bi-unique $A \implies (\text{rel-set } A \implies (=)) \text{ card card}$
 $\langle \text{proof} \rangle$

context
includes *lifting-syntax*
begin

lemma *vimage-right-total-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique* B *right-total* A
shows $((A \implies B) \implies \text{rel-set } B \implies \text{rel-set } A) (\lambda f X. f -' X \cap \text{Collect } (\text{Domainp } A)) \text{ vimage}$
 $\langle \text{proof} \rangle$

end

lemma *vimage-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total* A *bi-unique* B
shows $((A \implies B) \implies \text{rel-set } B \implies \text{rel-set } A) \text{ vimage vimage}$
 $\langle \text{proof} \rangle$

lemma *Image-parametric* [*transfer-rule*]:
assumes *bi-unique* A
shows $(\text{rel-set } (\text{rel-prod } A B) \implies \text{rel-set } A \implies \text{rel-set } B) (') (')$
 $\langle \text{proof} \rangle$

lemma *inj-on-transfer* [*transfer-rule*]:
 $((A \implies B) \implies \text{rel-set } A \implies (=)) \text{ inj-on inj-on}$
if [*transfer-rule*]: *bi-unique* A *bi-unique* B
 $\langle \text{proof} \rangle$

end

lemma (**in** *comm-monoid-set*) *F-parametric* [*transfer-rule*]:
fixes $A :: 'b \Rightarrow 'c \Rightarrow \text{bool}$
assumes *bi-unique* A
shows $\text{rel-fun } (\text{rel-fun } A (=)) (\text{rel-fun } (\text{rel-set } A) (=)) F F$
 $\langle \text{proof} \rangle$

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 $(\bigcup (f -' A)) (\bigcup (g -' B))$
 $\langle \text{proof} \rangle$

```

context
  includes lifting-syntax
begin

lemma fold-graph-transfer[transfer-rule]:
  assumes bi-unique R right-total R
  shows  $((R \text{ ==== } (=) \text{ ==== } (=)) \text{ ==== } (=) \text{ ==== } \text{rel-set } R \text{ ==== } (=) \text{ ==== } (=))$  fold-graph fold-graph
  <proof>

lemma fold-transfer[transfer-rule]:
  assumes [transfer-rule]: bi-unique R right-total R
  shows  $((R \text{ ==== } (=) \text{ ==== } (=)) \text{ ==== } (=) \text{ ==== } \text{rel-set } R \text{ ==== } (=))$ 
  Finite-Set.fold Finite-Set.fold
  <proof>

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 [] = []

bundle list-syntax
begin
notation Nil (<[]>)
  and Cons (infixr <#> 65)
end

datatype-compat list

lemma [case-names Nil Cons, cases type: list]:
  — for backward compatibility – names of variables differ
   $(y = [] \implies P) \implies (\bigwedge a \text{ list. } y = a \# \text{ list} \implies P) \implies P$ 

```

⟨proof⟩

lemma [case-names Nil Cons, induct type: list]:
 — for backward compatibility – names of variables differ
 $P [] \implies (\bigwedge a \text{ list}. P \text{ list} \implies P (a \# \text{list})) \implies P \text{ list}$
 ⟨proof⟩

Compatibility:

⟨ML⟩

lemmas *inducts* = *list.induct*
lemmas *recs* = *list.rec*
lemmas *cases* = *list.case*

⟨ML⟩

lemmas *set-simps* = *list.set*

List enumeration

open-bundle *list-enumeration-syntax*
begin

syntax
 $\text{-list} :: \text{args} \Rightarrow 'a \text{ list} \quad (\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix list enumeration} \rangle \rangle [-]) \rangle)$
syntax-consts
 $\text{-list} \equiv \text{Cons}$
translations
 $[x, xs] \equiv x \# [xs]$
 $[x] \equiv x \# []$
end

66.1 Basic list processing functions

primrec (*nonexhaustive*) *last* :: 'a list \Rightarrow 'a **where**
 $\text{last } (x \# xs) = (\text{if } xs = [] \text{ then } x \text{ else } \text{last } xs)$

primrec *butlast* :: 'a list \Rightarrow 'a list **where**
 $\text{butlast } [] = []$
 $\text{butlast } (x \# xs) = (\text{if } xs = [] \text{ then } [] \text{ else } x \# \text{butlast } xs)$

lemma *set-rec*: $\text{set } xs = \text{rec-list } \{ \} \ (\lambda x \cdot \text{insert } x) \ xs$
 ⟨proof⟩

definition *coset* :: 'a list \Rightarrow 'a set **where**
 $[\text{simp}]: \text{coset } xs = \text{-- set } xs$

primrec *append* :: 'a list \Rightarrow 'a list \Rightarrow 'a list (**infixr** $\langle @ \rangle$ 65) **where**
 $\text{append-Nil}: [] @ ys = ys$

append-Cons: $(x \# xs) @ ys = x \# xs @ ys$

primrec *rev* :: 'a list \Rightarrow 'a list **where**

rev [] = [] |

rev (x # xs) = *rev* xs @ [x]

primrec *filter* :: ('a \Rightarrow bool) \Rightarrow 'a list \Rightarrow 'a list **where**

filter P [] = [] |

filter P (x # xs) = (if P x then x # *filter* P xs else *filter* P xs)

open-bundle *filter-syntax* — Special input syntax for filter
begin

syntax (*ASCII*)

-*filter* :: [pttrn, 'a list, bool] \Rightarrow 'a list ($\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix filter} \rangle \rangle [-\langle -- / - \rangle]) \rangle$)

syntax

-*filter* :: [pttrn, 'a list, bool] \Rightarrow 'a list ($\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix filter} \rangle \rangle [-\langle \leftarrow - / - \rangle]) \rangle$)

syntax-consts

-*filter* \Rightarrow *filter*

translations

[x <- xs . P] \rightarrow *CONST filter* ($\lambda x. P$) xs

end

primrec *fold* :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a list \Rightarrow 'b \Rightarrow 'b **where**

fold-Nil: *fold* f [] = id |

fold-Cons: *fold* f (x # xs) = *fold* f xs \circ f x

primrec *foldr* :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a list \Rightarrow 'b \Rightarrow 'b **where**

foldr-Nil: *foldr* f [] = id |

foldr-Cons: *foldr* f (x # xs) = f x \circ *foldr* f xs

primrec *foldl* :: ('b \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a list \Rightarrow '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 \Rightarrow 'a list **where**

concat [] = [] |

concat (x # xs) = x @ *concat* xs

primrec *drop* :: nat \Rightarrow 'a list \Rightarrow 'a list **where**

drop-Nil: *drop* n [] = [] |

drop-Cons: *drop* n (x # xs) = (case n of 0 \Rightarrow x # xs | Suc m \Rightarrow *drop* m xs)

— Warning: simpset does not contain this definition, but separate theorems for $n = 0$ and $n = \text{Suc } k$

primrec *take* :: nat \Rightarrow 'a list \Rightarrow 'a list **where**

take-Nil: $take\ n\ [] = [] \mid$
take-Cons: $take\ n\ (x \# xs) = (case\ n\ of\ 0 \Rightarrow [] \mid Suc\ m \Rightarrow x \# take\ m\ xs)$
 — Warning: simpset does not contain this definition, but separate theorems for $n = 0$ and $n = Suc\ k$

primrec (*nonexhaustive*) $nth :: 'a\ list \Rightarrow nat \Rightarrow 'a$ (**infixl** $\langle ! \rangle\ 100$) **where**
 $nth\text{-}Cons: (x \# xs) ! n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ k \Rightarrow xs ! k)$
 — Warning: simpset does not contain this definition, but separate theorems for $n = 0$ and $n = Suc\ k$

primrec *list-update* :: $'a\ list \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ list$ **where**
 $list\text{-}update\ []\ i\ v = [] \mid$
 $list\text{-}update\ (x \# xs)\ i\ v =$
 $(case\ i\ of\ 0 \Rightarrow v \# xs \mid Suc\ j \Rightarrow x \# list\text{-}update\ xs\ j\ v)$

nonterminal *lupdbinds* and *lupdbind*

open-bundle *list-update-syntax*
begin

syntax
 $-lupdbind :: ['a, 'a] \Rightarrow lupdbind \quad (\langle \langle indent=2\ notation=\langle mixfix\ update \rangle \rangle - := /$
 $- \rangle \rangle)$
 $:: lupbind \Rightarrow lupdbinds \quad (\langle \cdot \rangle)$
 $-lupdbinds :: [lupbind, lupdbinds] \Rightarrow lupdbinds \quad (\langle \cdot, / \cdot \rangle)$
 $-LUpdate :: ['a, lupdbinds] \Rightarrow 'a$
 $(\langle \langle open\text{-}block\ notation=\langle mixfix\ list\ update \rangle \rangle - / [(-)] \rangle [1000, 0] 900)$
syntax-consts
 $-LUpdate \Leftarrow list\text{-}update$
translations
 $-LUpdate\ xs\ (-lupdbinds\ b\ bs) == -LUpdate\ (-LUpdate\ xs\ b)\ bs$
 $xs[i:=x] == CONST\ list\text{-}update\ xs\ i\ x$

end

primrec *takeWhile* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $takeWhile\ P\ [] = [] \mid$
 $takeWhile\ P\ (x \# xs) = (if\ P\ x\ then\ x \# takeWhile\ P\ xs\ else\ [])$

primrec *dropWhile* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $dropWhile\ P\ [] = [] \mid$
 $dropWhile\ P\ (x \# xs) = (if\ P\ x\ then\ dropWhile\ P\ xs\ else\ x \# xs)$

primrec *zip* :: $'a\ list \Rightarrow 'b\ list \Rightarrow ('a \times 'b)\ list$ **where**
 $zip\ xs\ [] = [] \mid$
 $zip\text{-}Cons: zip\ xs\ (y \# ys) =$
 $(case\ xs\ of\ [] \Rightarrow [] \mid z \# zs \Rightarrow (z, y) \# zip\ zs\ ys)$
 — Warning: simpset does not contain this definition, but separate theorems for $xs = []$ and $xs = z \# zs$

abbreviation $\text{map2} :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow 'c \text{ list}$ **where**
 $\text{map2 } f \text{ xs ys} \equiv \text{map } (\lambda(x,y). f \ x \ y) \ (\text{zip } \text{xs } \text{ys})$

primrec $\text{product} :: 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow ('a \times 'b) \text{ list}$ **where**
 $\text{product } [] = []$ |
 $\text{product } (x\#\text{xs}) \text{ ys} = \text{map } (\text{Pair } x) \ \text{ys} \ @ \ \text{product } \text{xs } \text{ys}$

hide-const (open) product

primrec $\text{product-lists} :: 'a \text{ list list} \Rightarrow 'a \text{ list list}$ **where**
 $\text{product-lists } [] = [[]]$ |
 $\text{product-lists } (\text{xs} \# \text{xss}) = \text{concat } (\text{map } (\lambda x. \text{map } (\text{Cons } x) (\text{product-lists } \text{xss})) \ \text{xs})$

primrec $\text{upt} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list}$ ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix list inter-} \text{val} \rangle \rangle [-..</-] \rangle$) **where**
 $\text{upt-0}: [i..<0] = []$ |
 $\text{upt-Suc}: [i..<(\text{Suc } j)] = (\text{if } i \leq j \text{ then } [i..<j] \ @ \ [j] \text{ else } [])$

definition $\text{insert} :: 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
 $\text{insert } x \text{ xs} = (\text{if } x \in \text{set } \text{xs} \text{ then } \text{xs} \text{ else } x \# \text{xs})$

definition $\text{union} :: 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
 $\text{union} = \text{fold } \text{insert}$

hide-const (open) insert union
hide-fact (open) $\text{insert-def union-def}$

primrec $\text{find} :: ('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ option}$ **where**
 $\text{find } - \ [] = \text{None}$ |
 $\text{find } P \ (x\#\text{xs}) = (\text{if } P \ x \text{ then } \text{Some } x \text{ else } \text{find } P \ \text{xs})$

In the context of multisets, *count-list* is equivalent to $\text{count} \circ \text{mset}$ and it is advisable to use the latter.

primrec $\text{count-list} :: 'a \text{ list} \Rightarrow 'a \Rightarrow \text{nat}$ **where**
 $\text{count-list } [] \ y = 0$ |
 $\text{count-list } (x\#\text{xs}) \ y = (\text{if } x=y \text{ then } \text{count-list } \text{xs } y + 1 \text{ else } \text{count-list } \text{xs } y)$

definition
 $\text{extract} :: ('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow ('a \text{ list} * 'a * 'a \text{ list}) \text{ option}$
where $\text{extract } P \ \text{xs} =$
 $(\text{case } \text{dropWhile } (\text{Not} \circ P) \ \text{xs} \text{ of}$
 $[] \Rightarrow \text{None} \mid$
 $y\#\text{ys} \Rightarrow \text{Some}(\text{takeWhile } (\text{Not} \circ P) \ \text{xs}, y, \text{ys}))$

hide-const (open) extract

primrec $\text{those} :: 'a \text{ option list} \Rightarrow 'a \text{ list option}$
where

$those \ [] = Some \ [] \mid$
 $those \ (x \# xs) = (case \ x \ of$
 $\quad None \Rightarrow None$
 $\mid Some \ y \Rightarrow map-option \ (Cons \ y) \ (those \ xs))$

primrec $remove1 :: 'a \Rightarrow 'a \ list \Rightarrow 'a \ list$ **where**
 $remove1 \ x \ [] = [] \mid$
 $remove1 \ x \ (y \# xs) = (if \ x = y \ then \ xs \ else \ y \# remove1 \ x \ xs)$

primrec $removeAll :: 'a \Rightarrow 'a \ list \Rightarrow 'a \ list$ **where**
 $removeAll \ x \ [] = [] \mid$
 $removeAll \ x \ (y \# xs) = (if \ x = y \ then \ removeAll \ x \ xs \ else \ y \# removeAll \ x \ xs)$

definition $minus-list-mset :: 'a \ list \Rightarrow 'a \ list \Rightarrow 'a \ list$ **where**
 $minus-list-mset \ xs \ ys = foldr \ remove1 \ ys \ xs$

definition $minus-list-set :: 'a \ list \Rightarrow 'a \ list \Rightarrow 'a \ list$ **where**
 $minus-list-set \ xs \ ys = foldr \ removeAll \ ys \ xs$

definition $inter-list-set :: 'a \ list \Rightarrow 'a \ list \Rightarrow 'a \ list$ **where**
 $inter-list-set \ xs \ ys = filter \ (\lambda x. \ x \in set \ ys) \ xs$

primrec $distinct :: 'a \ list \Rightarrow bool$ **where**
 $distinct \ [] \longleftrightarrow True \mid$
 $distinct \ (x \# xs) \longleftrightarrow x \notin set \ xs \wedge distinct \ xs$

fun $successively :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \ list \Rightarrow bool$ **where**
 $successively \ P \ [] = True \mid$
 $successively \ P \ [x] = True \mid$
 $successively \ P \ (x \# y \# xs) = (P \ x \ y \wedge successively \ P \ (y \# xs))$

definition $distinct-adj$ **where**
 $distinct-adj = successively \ (\neq)$

primrec $remdups :: 'a \ list \Rightarrow 'a \ list$ **where**
 $remdups \ [] = [] \mid$
 $remdups \ (x \# xs) = (if \ x \in set \ xs \ then \ remdups \ xs \ else \ x \# remdups \ xs)$

fun $remdups-adj :: 'a \ list \Rightarrow 'a \ list$ **where**
 $remdups-adj \ [] = [] \mid$
 $remdups-adj \ [x] = [x] \mid$
 $remdups-adj \ (x \# y \# xs) = (if \ x = y \ then \ remdups-adj \ (x \# xs) \ else \ x \# remdups-adj \ (y \# xs))$

primrec $replicate :: nat \Rightarrow 'a \Rightarrow 'a \ list$ **where**
 $replicate-0: replicate \ 0 \ x = [] \mid$
 $replicate-Suc: replicate \ (Suc \ n) \ x = x \# replicate \ n \ x$

Function *size* is overloaded for all datatypes. Users may refer to the list

version as *length*.

abbreviation *length* :: 'a list \Rightarrow nat **where**
length \equiv size

definition *enumerate* :: nat \Rightarrow 'a list \Rightarrow (nat \times 'a) list **where**
enumerate-eq- zip : *enumerate* *n* *xs* = *zip* [*n*..*n* + *length* *xs*] *xs*

primrec *rotate1* :: 'a list \Rightarrow 'a list **where**
rotate1 [] = [] |
rotate1 (x # *xs*) = *xs* @ [x]

definition *rotate* :: nat \Rightarrow 'a list \Rightarrow 'a list **where**
rotate *n* = *rotate1* $\overset{\sim}{\sim}$ *n*

definition *nths* :: 'a list \Rightarrow nat set \Rightarrow 'a list **where**
nths *xs* *A* = *map fst* (*filter* ($\lambda p.$ *snd* *p* \in *A*) (*zip* *xs* [0..*size* *xs*]))

primrec *subseqs* :: 'a list \Rightarrow 'a list list **where**
subseqs [] = [[]] |
subseqs (x#*xs*) = (*let* *xss* = *subseqs* *xs* *in* *map* (*Cons* *x*) *xss* @ *xss*)

primrec *n-lists* :: nat \Rightarrow 'a list \Rightarrow 'a list list **where**
n-lists 0 *xs* = [[]] |
n-lists (*Suc* *n*) *xs* = *concat* (*map* ($\lambda ys.$ *map* ($\lambda y.$ *y* # *ys*) *xs*) (*n-lists* *n* *xs*))

hide-const (**open**) *n-lists*

function *splice* :: 'a list \Rightarrow 'a list \Rightarrow 'a list **where**
splice [] *ys* = *ys* |
splice (x#*xs*) *ys* = x # *splice* *ys* *xs*
 \langle proof \rangle

termination
 \langle proof \rangle

function *shuffles* **where**
shuffles [] *ys* = {*ys*}
| *shuffles* *xs* [] = {*xs*}
| *shuffles* (x # *xs*) (*y* # *ys*) = (#) *x* ‘ *shuffles* *xs* (*y* # *ys*) \cup (#) *y* ‘ *shuffles* (x # *xs*) *ys*
 \langle proof \rangle
termination \langle proof \rangle

Use only if you cannot use *Min* instead:

fun *min-list* :: 'a::ord list \Rightarrow 'a **where**
min-list (x # *xs*) = (*case* *xs* of [] \Rightarrow x | - \Rightarrow *min* x (*min-list* *xs*))

Returns first minimum:

fun *arg-min-list* :: ('a \Rightarrow ('b::linorder)) \Rightarrow 'a list \Rightarrow 'a **where**

$arg-min-list\ f\ [x] = x \mid$
 $arg-min-list\ f\ (x\#y\#zs) = (let\ m = arg-min-list\ f\ (y\#zs)\ in\ if\ f\ x \leq f\ m\ then\ x$
 $else\ m)$

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 \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a list \Rightarrow bool **where**
sorted-wrt P [] = True |
sorted-wrt P (x # ys) = (($\forall y \in set\ ys.\ P\ x\ y$) \wedge *sorted-wrt* P ys)

A class-based sorted predicate:

context *linorder*
begin

abbreviation *sorted* :: 'a list \Rightarrow bool **where**
sorted \equiv *sorted-wrt* (\leq)

lemma *sorted-simps*: *sorted* [] = True *sorted* (x # ys) = (($\forall y \in set\ ys.\ x \leq y$) \wedge
sorted ys)
 <proof>

lemma *strict-sorted-simps*: *sorted-wrt* (<) [] = True *sorted-wrt* (<) (x # ys) =
 (($\forall y \in set\ ys.\ x < y$) \wedge *sorted-wrt* (<) ys)
 <proof>

primrec *insort-key* :: ('b \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'b list \Rightarrow 'b list **where**
insort-key f x [] = [x] |
insort-key f x (y#ys) =
 (if f x \leq f y then (x#y#ys) else y#(*insort-key* f x ys))

definition *sort-key* :: ('b \Rightarrow 'a) \Rightarrow 'b list \Rightarrow 'b list **where**
sort-key f xs = foldr (*insort-key* f) xs []

definition *insort-insert-key* :: ('b \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'b list \Rightarrow 'b list **where**
insort-insert-key f x xs =
 (if f x \in f `set xs then xs else *insort-key* f x xs)

abbreviation *sort* \equiv *sort-key* ($\lambda x.\ x$)

abbreviation *insort* \equiv *insort-key* ($\lambda x.\ x$)

abbreviation *insort-insert* \equiv *insort-insert-key* ($\lambda x.\ x$)

definition *stable-sort-key* :: (('b \Rightarrow 'a) \Rightarrow 'b list \Rightarrow 'b list) \Rightarrow bool **where**
stable-sort-key sk =
 ($\forall f\ xs\ k.\ filter\ (\lambda y.\ f\ y = k)\ (sk\ f\ xs) = filter\ (\lambda y.\ f\ y = k)\ xs$)

```

[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 (λn::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 (≠) [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, e]] = [[a, c, d], [a, c, e], [b, c, d], [b, c, e]]
splice [a, b, c] [x, y, z] = [a, x, b, y, c, z]
splice [a, b, c, d] [x, y] = [a, x, b, y, c, d]
shuffles [a, b] [c, d] = {[a, b, c, d], [a, c, b, d], [a, c, d, b], [c, a, b, d], [c, a, d, 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 (λn. n < 3) [1, 2, 3, 0] = [1, 2]
dropWhile (λn. 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], 1, [0, 2])
remove1 2 [2, 0, 2, 1, 2] = [0, 2, 1, 2]
removeAll 2 [2, 0, 2, 1, 2] = [0, 1]
[a, b, c, d] ! 2 = c
[a, b, c, d][2 := x] = [a, b, x, d]
nth [a, b, c, d, e] {0, 2, 3} = [a, c, d]
subseqs [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]]
rotate1 [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 (λi. i * i) [3, - 1, 1, - 2] = - 1

```

lemma *strict-sorted-iff*: *sorted-wrt* ($<$) $l \iff \text{sorted } l \wedge \text{distinct } l$
<proof>

lemma *strict-sorted-imp-sorted*: *sorted-wrt* ($<$) $xs \implies \text{sorted } xs$
<proof>

end

66.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: $[(x,y). x \leftarrow xs, y \leftarrow ys, x \neq y]$, 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): $[e. x \leftarrow xs, \dots]$ rather than $[e \mid x \leftarrow xs, \dots]$.

The qualifiers after the dot are

generators $p \leftarrow xs$, 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 $[e. x \leftarrow xs]$ is optimized to $\text{map } (\lambda x. e) \ xs$.

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.

nonterminal *lc-qual* and *lc-quals*

open-bundle *list-comprehension-syntax*

begin

syntax

-listcompr :: $'a \Rightarrow \text{lc-qual} \Rightarrow \text{lc-quals} \Rightarrow 'a \text{ list}$ ($\langle [- \ . \ \rightarrow] \rangle$)

-lc-gen :: $'a \Rightarrow 'a \text{ list} \Rightarrow \text{lc-qual}$ ($\langle \leftarrow \leftarrow \rightarrow \rangle$)

-lc-test :: $\text{bool} \Rightarrow \text{lc-qual}$ ($\langle \rightarrow \rangle$)

-lc-end :: lc-quals ($\langle \rangle \rangle$)

-lc-quals :: $\text{lc-qual} \Rightarrow \text{lc-quals} \Rightarrow \text{lc-quals}$ ($\langle , \rightarrow \rangle$)

syntax (*ASCII*)

-lc-gen :: $'a \Rightarrow 'a \text{ list} \Rightarrow \text{lc-qual}$ ($\langle \leftarrow < - \rightarrow \rangle$)

end

$\langle ML \rangle$

code-datatype *set coset*
hide-const (**open**) *coset*

66.1.2 $[]$ and $(\#)$

lemma *not-Cons-self* [simp]:
 $xs \neq x \# xs$
 $\langle proof \rangle$

lemma *not-Cons-self2* [simp]: $x \# xs \neq xs$
 $\langle proof \rangle$

lemma *neq-Nil-conv*: $(xs \neq []) = (\exists y. xs = y \# ys)$
 $\langle proof \rangle$

lemma *tl-Nil*: $tl\ xs = [] \longleftrightarrow xs = [] \vee (\exists x. xs = [x])$
 $\langle proof \rangle$

lemmas *Nil-tl* = *tl-Nil*[*THEN eq-iff-swap*]

lemma *length-induct*:
 $(\bigwedge xs. \forall ys. length\ ys < length\ xs \longrightarrow P\ ys \Longrightarrow P\ xs) \Longrightarrow P\ xs$
 $\langle proof \rangle$

lemma *induct-list012*:
 $\llbracket P\ []; \bigwedge x. P\ [x]; \bigwedge x\ y\ zs. \llbracket P\ zs; P\ (y \# zs) \rrbracket \Longrightarrow P\ (x \# y \# zs) \rrbracket \Longrightarrow P\ xs$
 $\langle proof \rangle$

lemma *list-nonempty-induct* [*consumes 1, case-names single cons*]:
 $\llbracket xs \neq []; \bigwedge x. P\ [x]; \bigwedge xs. xs \neq [] \Longrightarrow P\ xs \Longrightarrow P\ (x \# xs) \rrbracket \Longrightarrow P\ xs$
 $\langle proof \rangle$

lemma *inj-split-Cons*: *inj-on* $(\lambda(xs, n). n \# xs)\ X$
 $\langle proof \rangle$

lemma *inj-on-Cons1* [simp]: *inj-on* $((\#)\ x)\ A$
 $\langle proof \rangle$

66.1.3 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

lemma *length-append* [simp]: $length\ (xs\ @\ ys) = length\ xs + length\ ys$
 $\langle proof \rangle$

lemma *length-map* [simp]: $length\ (map\ f\ xs) = length\ xs$
 $\langle proof \rangle$

lemma *length-rev* [*simp*]: $\text{length } (\text{rev } xs) = \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *length-tl* [*simp*]: $\text{length } (\text{tl } xs) = \text{length } xs - 1$
 $\langle \text{proof} \rangle$

lemma *length-0-conv* [*iff*]: $(\text{length } xs = 0) = (xs = [])$
 $\langle \text{proof} \rangle$

lemma *length-greater-0-conv* [*iff*]: $(0 < \text{length } xs) = (xs \neq [])$
 $\langle \text{proof} \rangle$

lemma *length-pos-if-in-set*: $x \in \text{set } xs \implies \text{length } xs > 0$
 $\langle \text{proof} \rangle$

lemma *length-Suc-conv*: $(\text{length } xs = \text{Suc } n) = (\exists y \ ys. \ xs = y \# \ ys \wedge \text{length } ys = n)$
 $\langle \text{proof} \rangle$

lemmas *Suc-length-conv* = *length-Suc-conv*[*THEN eq-iff-swap*]

lemma *Suc-le-length-iff*:
 $(\text{Suc } n \leq \text{length } xs) = (\exists x \ ys. \ xs = x \# \ ys \wedge n \leq \text{length } ys)$
 $\langle \text{proof} \rangle$

lemma *impossible-Cons*: $\text{length } xs \leq \text{length } ys \implies xs = x \# \ ys = \text{False}$
 $\langle \text{proof} \rangle$

lemma *list-induct2* [*consumes 1, case-names Nil Cons*]:
 $\text{length } xs = \text{length } ys \implies P [] [] \implies$
 $(\bigwedge x \ xs \ y \ ys. \ \text{length } xs = \text{length } ys \implies P \ xs \ ys \implies P \ (x \# \ xs) \ (y \# \ ys))$
 $\implies P \ xs \ ys$
 $\langle \text{proof} \rangle$

lemma *list-induct3* [*consumes 2, case-names Nil Cons*]:
 $\text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies P [] [] [] \implies$
 $(\bigwedge x \ xs \ y \ ys \ z \ zs. \ \text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies P \ xs \ ys \ zs$
 $\implies P \ (x \# \ xs) \ (y \# \ ys) \ (z \# \ zs))$
 $\implies P \ xs \ ys \ zs$
 $\langle \text{proof} \rangle$

lemma *list-induct4* [*consumes 3, case-names Nil Cons*]:
 $\text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies \text{length } zs = \text{length } ws \implies$
 $P [] [] [] [] \implies (\bigwedge x \ xs \ y \ ys \ z \ zs \ w \ ws. \ \text{length } xs = \text{length } ys \implies$
 $\text{length } ys = \text{length } zs \implies \text{length } zs = \text{length } ws \implies P \ xs \ ys \ zs \ ws \implies$
 $P \ (x \# \ xs) \ (y \# \ ys) \ (z \# \ zs) \ (w \# \ ws)) \implies P \ xs \ ys \ zs \ ws$
 $\langle \text{proof} \rangle$

lemma *list-induct2'*:

$$\begin{aligned} & \llbracket P \rrbracket \llbracket \rrbracket; \\ & \bigwedge x \, xs. P \, (x \# xs) \llbracket \rrbracket; \\ & \bigwedge y \, ys. P \llbracket \rrbracket (y \# ys); \\ & \bigwedge x \, xs \, y \, ys. P \, xs \, ys \implies P \, (x \# xs) \, (y \# ys) \llbracket \rrbracket \\ & \implies P \, xs \, ys \\ & \langle proof \rangle \end{aligned}$$

lemma *list-all2-iff*:

$$list\text{-}all2 \, P \, xs \, ys \longleftrightarrow length \, xs = length \, ys \wedge (\forall (x, y) \in set \, (zip \, xs \, ys). P \, x \, y)$$
 $\langle proof \rangle$

lemma *neq-if-length-neq*: $length \, xs \neq length \, ys \implies (xs = ys) == False$
 $\langle proof \rangle$

66.1.4 @ – append

global-interpretation *append*: *monoid append Nil*
 $\langle proof \rangle$

lemma *append-assoc* [*simp*]: $(xs @ ys) @ zs = xs @ (ys @ zs)$
 $\langle proof \rangle$

lemma *append-Nil2*: $xs @ [] = xs$
 $\langle proof \rangle$

lemma *append-is-Nil-conv* [*iff*]: $(xs @ ys = []) = (xs = [] \wedge ys = [])$
 $\langle proof \rangle$

lemmas *Nil-is-append-conv* [*iff*] = *append-is-Nil-conv*[*THEN eq-iff-swap*]

lemma *append-self-conv* [*iff*]: $(xs @ ys = xs) = (ys = [])$
 $\langle proof \rangle$

lemmas *self-append-conv* [*iff*] = *append-self-conv*[*THEN eq-iff-swap*]

lemma *append-eq-append-conv* [*simp*]:

$$length \, xs = length \, ys \vee length \, us = length \, vs$$

$$\implies (xs @ us = ys @ vs) = (xs = ys \wedge us = vs)$$
 $\langle proof \rangle$

lemma *append-eq-append-conv2*: $(xs @ ys = zs @ ts) =$
 $(\exists us. xs = zs @ us \wedge us @ ys = ts \vee xs @ us = zs \wedge ys = us @ ts)$
 $\langle proof \rangle$

lemma *same-append-eq* [*iff, induct-simp*]: $(xs @ ys = xs @ zs) = (ys = zs)$
 $\langle proof \rangle$

lemma *append1-eq-conv* [*iff*]: $(xs @ [x] = ys @ [y]) = (xs = ys \wedge x = y)$
 $\langle proof \rangle$

lemma *append-same-eq* [iff, induct-simp]: $(ys @ xs = zs @ xs) = (ys = zs)$
 ⟨proof⟩

lemma *append-self-conv2* [iff]: $(xs @ ys = ys) = (xs = [])$
 ⟨proof⟩

lemmas *self-append-conv2* [iff] = *append-self-conv2*[THEN eq-iff-swap]

lemma *hd-Cons-tl*: $xs \neq [] \implies hd\ xs \# tl\ xs = xs$
 ⟨proof⟩

lemma *hd-append*: $hd\ (xs @ ys) = (if\ xs = []\ then\ hd\ ys\ else\ hd\ xs)$
 ⟨proof⟩

lemma *hd-append2* [simp]: $xs \neq [] \implies hd\ (xs @ ys) = hd\ xs$
 ⟨proof⟩

lemma *tl-append*: $tl\ (xs @ ys) = (case\ xs\ of\ [] \Rightarrow tl\ ys \mid z\#zs \Rightarrow zs @ ys)$
 ⟨proof⟩

lemma *tl-append2* [simp]: $xs \neq [] \implies tl\ (xs @ ys) = tl\ xs @ ys$
 ⟨proof⟩

lemma *tl-append-if*: $tl\ (xs @ ys) = (if\ xs = []\ then\ tl\ ys\ else\ tl\ xs @ ys)$
 ⟨proof⟩

lemma *Cons-eq-append-conv*: $x\#xs = ys@zs =$
 $(ys = [] \wedge x\#xs = zs \vee (\exists\ ys'.\ x\#ys' = ys \wedge xs = ys'@zs))$
 ⟨proof⟩

lemma *append-eq-Cons-conv*: $(ys@zs = x\#xs) =$
 $(ys = [] \wedge zs = x\#xs \vee (\exists\ ys'.\ ys = x\#ys' \wedge ys'@zs = xs))$
 ⟨proof⟩

lemma *longest-common-prefix*:
 $\exists\ ps\ xs'\ ys'.\ xs = ps @ xs' \wedge ys = ps @ ys'$
 $\wedge (xs' = [] \vee ys' = [] \vee hd\ xs' \neq hd\ ys')$
 ⟨proof⟩

Trivial rules for solving @-equations automatically.

lemma *eq-Nil-appendI*: $xs = ys \implies xs = [] @ ys$
 ⟨proof⟩

lemma *Cons-eq-appendI*: $[x\#xs1 = ys; xs = xs1 @ zs] \implies x\#xs = ys @ zs$
 ⟨proof⟩

lemma *append-eq-appendI*: $[xs @ xs1 = zs; ys = xs1 @ us] \implies xs @ ys = zs @ us$

$\langle proof \rangle$

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.

$\langle ML \rangle$

66.1.5 map

lemma *hd-map*: $xs \neq [] \implies hd (map f xs) = f (hd xs)$

$\langle proof \rangle$

lemma *map-tl*: $map f (tl xs) = tl (map f xs)$

$\langle proof \rangle$

lemma *map-ext*: $(\bigwedge x. x \in set xs \implies f x = g x) \implies map f xs = map g xs$

$\langle proof \rangle$

lemma *map-ident [simp]*: $map (\lambda x. x) = (\lambda xs. xs)$

$\langle proof \rangle$

lemma *map-append [simp]*: $map f (xs @ ys) = map f xs @ map f ys$

$\langle proof \rangle$

lemma *map-map [simp]*: $map f (map g xs) = map (f \circ g) xs$

$\langle proof \rangle$

lemma *map-comp-map [simp]*: $((map f) \circ (map g)) = map(f \circ g)$

$\langle proof \rangle$

lemma *rev-map*: $rev (map f xs) = map f (rev xs)$

$\langle proof \rangle$

lemma *map-eq-conv [simp]*: $(map f xs = map g xs) = (\forall x \in set xs. f x = g x)$

$\langle proof \rangle$

lemma *map-cong [fundef-cong]*:

$xs = ys \implies (\bigwedge x. x \in set ys \implies f x = g x) \implies map f xs = map g ys$

$\langle proof \rangle$

lemma *map-is-Nil-conv [iff]*: $(map f xs = []) = (xs = [])$

$\langle proof \rangle$

lemmas *Nil-is-map-conv [iff]* = *map-is-Nil-conv [THEN eq-iff-swap]*

lemma *map-eq-Cons-conv*:

$(map f xs = y \# ys) = (\exists z zs. xs = z \# zs \wedge f z = y \wedge map f zs = ys)$

$\langle proof \rangle$

lemma *Cons-eq-map-conv*:

$(x\#xs = \text{map } f \text{ } ys) = (\exists z \text{ } zs. \text{ } ys = z\#zs \wedge x = f \text{ } z \wedge xs = \text{map } f \text{ } zs)$
 $\langle \text{proof} \rangle$

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 *ex-map-conv*:

$(\exists xs. \text{ } ys = \text{map } f \text{ } xs) = (\forall y \in \text{set } ys. \exists x. \text{ } y = f \text{ } x)$
 $\langle \text{proof} \rangle$

lemma *map-eq-imp-length-eq*:

assumes $\text{map } f \text{ } xs = \text{map } g \text{ } ys$

shows $\text{length } xs = \text{length } ys$

$\langle \text{proof} \rangle$

lemma *map-inj-on*:

assumes *map*: $\text{map } f \text{ } xs = \text{map } f \text{ } ys$ **and** *inj*: *inj-on* *f* (*set* *xs* *Un* *set* *ys*)

shows $xs = ys$

$\langle \text{proof} \rangle$

lemma *inj-on-map-eq-map*:

inj-on *f* (*set* *xs* *Un* *set* *ys*) $\implies (\text{map } f \text{ } xs = \text{map } f \text{ } ys) = (xs = ys)$
 $\langle \text{proof} \rangle$

lemma *map-injective*:

$\text{map } f \text{ } xs = \text{map } f \text{ } ys \implies \text{inj } f \implies xs = ys$
 $\langle \text{proof} \rangle$

lemma *inj-map-eq-map[simp]*: $\text{inj } f \implies (\text{map } f \text{ } xs = \text{map } f \text{ } ys) = (xs = ys)$

$\langle \text{proof} \rangle$

lemma *inj-mapI*: $\text{inj } f \implies \text{inj } (\text{map } f)$

$\langle \text{proof} \rangle$

lemma *inj-mapD*: $\text{inj } (\text{map } f) \implies \text{inj } f$

$\langle \text{proof} \rangle$

lemma *inj-map[iff]*: $\text{inj } (\text{map } f) = \text{inj } f$

$\langle \text{proof} \rangle$

lemma *inj-on-mapI*: $\text{inj-on } f \text{ } (\bigcup (\text{set } 'A)) \implies \text{inj-on } (\text{map } f) \text{ } A$

$\langle \text{proof} \rangle$

lemma *map-idI*: $(\bigwedge x. x \in \text{set } xs \implies f \text{ } x = x) \implies \text{map } f \text{ } xs = xs$

$\langle \text{proof} \rangle$

lemma *map-fun-upd [simp]*: $y \notin \text{set } xs \implies \text{map } (f(y:=v)) \text{ } xs = \text{map } f \text{ } xs$

$\langle proof \rangle$

lemma *map-fst-zip[simp]*:

$$length\ xs = length\ ys \implies map\ fst\ (zip\ xs\ ys) = xs$$

$\langle proof \rangle$

lemma *map-snd-zip[simp]*:

$$length\ xs = length\ ys \implies map\ snd\ (zip\ xs\ ys) = ys$$

$\langle proof \rangle$

lemma *map-fst-zip-take*:

$$map\ fst\ (zip\ xs\ ys) = take\ (min\ (length\ xs)\ (length\ ys))\ xs$$

$\langle proof \rangle$

lemma *map-snd-zip-take*:

$$map\ snd\ (zip\ xs\ ys) = take\ (min\ (length\ xs)\ (length\ ys))\ ys$$

$\langle proof \rangle$

lemma *map2-map-map*: $map2\ h\ (map\ f\ xs)\ (map\ g\ xs) = map\ (\lambda x. h\ (f\ x)\ (g\ x))\ xs$

$\langle proof \rangle$

functor *map*: *map*

$\langle proof \rangle$

declare *map.id* [simp]

66.1.6 rev

lemma *rev-append* [simp]: $rev\ (xs\ @\ ys) = rev\ ys\ @\ rev\ xs$

$\langle proof \rangle$

lemma *rev-rev-ident* [simp]: $rev\ (rev\ xs) = xs$

$\langle proof \rangle$

lemma *rev-involution*[simp]: $rev \circ rev = id$

$\langle proof \rangle$

lemma *rev-swap*: $(rev\ xs = ys) = (xs = rev\ ys)$

$\langle proof \rangle$

lemma *rev-is-Nil-conv* [iff]: $(rev\ xs = []) = (xs = [])$

$\langle proof \rangle$

lemmas *Nil-is-rev-conv* [iff] = *rev-is-Nil-conv*[THEN *eq-iff-swap*]

lemma *rev-singleton-conv* [simp]: $(rev\ xs = [x]) = (xs = [x])$

$\langle proof \rangle$

lemma *singleton-rev-conv* [*simp*]: $([x] = \text{rev } xs) = ([x] = xs)$
 $\langle \text{proof} \rangle$

lemma *rev-is-rev-conv* [*iff*]: $(\text{rev } xs = \text{rev } ys) = (xs = ys)$
 $\langle \text{proof} \rangle$

lemma *rev-eq-append-conv*: $\text{rev } xs = ys @ zs \longleftrightarrow xs = \text{rev } zs @ \text{rev } ys$
 $\langle \text{proof} \rangle$

lemma *append-eq-rev-conv*: $ys @ zs = \text{rev } xs \longleftrightarrow \text{rev } zs @ \text{rev } ys = xs$
 $\langle \text{proof} \rangle$

lemma *rev-eq-Cons-iff* [*iff*]: $(\text{rev } xs = y \# ys) = (xs = \text{rev } ys @ [y])$
 $\langle \text{proof} \rangle$

lemmas *Cons-eq-rev-iff* = *rev-eq-Cons-iff* [*THEN eq-iff-swap*]

lemma *inj-on-rev* [*iff*]: *inj-on* *rev A*
 $\langle \text{proof} \rangle$

lemma *rev-induct* [*case-names Nil snoc*]:
 assumes $P []$ and $\bigwedge x xs. P xs \implies P (xs @ [x])$
 shows $P xs$
 $\langle \text{proof} \rangle$

lemma *rev-exhaust* [*case-names Nil snoc*]:
 $(xs = [] \implies P) \implies (\bigwedge ys y. xs = ys @ [y] \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemmas *rev-cases* = *rev-exhaust*

lemma *rev-nonempty-induct* [*consumes 1, case-names single snoc*]:
 assumes $xs \neq []$
 and *single*: $\bigwedge x. P [x]$
 and *snoc'*: $\bigwedge x xs. xs \neq [] \implies P xs \implies P (xs @ [x])$
 shows $P xs$
 $\langle \text{proof} \rangle$

lemma *rev-induct2*:
 $[P []]$;
 $\bigwedge x xs. P (xs @ [x]) []$;
 $\bigwedge y ys. P [] (ys @ [y])$;
 $\bigwedge x xs y ys. P xs ys \implies P (xs @ [x]) (ys @ [y]) []$
 $\implies P xs ys$
 $\langle \text{proof} \rangle$

lemma *length-Suc-conv-rev*: $(\text{length } xs = \text{Suc } n) = (\exists y ys. xs = ys @ [y] \wedge \text{length } ys = n)$
 $\langle \text{proof} \rangle$

66.1.7 *set*

declare *list.set*[*code-post*] — pretty output

lemma *finite-set* [*iff*]: *finite* (*set xs*)
 ⟨*proof*⟩

lemma *set-append* [*simp*]: *set* (*xs @ ys*) = (*set xs* \cup *set ys*)
 ⟨*proof*⟩

lemma *hd-in-set*[*simp*]: *xs* \neq [] \implies *hd xs* \in *set xs*
 ⟨*proof*⟩

lemma *set-subset-Cons*: *set xs* \subseteq *set* (*x # xs*)
 ⟨*proof*⟩

lemma *set-ConsD*: *y* \in *set* (*x # xs*) \implies *y* = *x* \vee *y* \in *set xs*
 ⟨*proof*⟩

lemma *set-empty* [*iff*]: (*set xs* = {}) = (*xs* = [])
 ⟨*proof*⟩

lemmas *set-empty2*[*iff*] = *set-empty*[*THEN eq-iff-swap*]

lemma *append-eq-append-conv-if-disj*:
 (*set xs* \cup *set xs'*) \cap (*set ys* \cup *set ys'*) = {}
 \implies *xs @ ys* = *xs' @ ys'* \longleftrightarrow *xs* = *xs'* \wedge *ys* = *ys'*
 ⟨*proof*⟩

lemma *set-rev* [*simp*]: *set* (*rev xs*) = *set xs*
 ⟨*proof*⟩

lemma *set-map* [*simp*]: *set* (*map f xs*) = *f*‘(*set xs*)
 ⟨*proof*⟩

lemma *set-filter* [*simp*]: *set* (*filter P xs*) = {*x*. *x* \in *set xs* \wedge *P x*}
 ⟨*proof*⟩

lemma *set-upt* [*simp*]: *set*[*i*..*j*] = {*i*..*j*}
 ⟨*proof*⟩

lemma *atMost-upto*:
 ⟨{..*n*} = *set* [0..*Suc n*]⟩
 ⟨*proof*⟩

lemma *atLeast-upt*:
 ⟨{..*n*} = *set* [0..*n*]⟩
 ⟨*proof*⟩

lemma *greaterThanLessThan-upt*:

$\langle \{n <..< m\} = \text{set } [Suc\ n..< m] \rangle$
 $\langle \text{proof} \rangle$

lemma *atLeastLessThan-upt*:
 $\langle \{i..< j\} = \text{set } [i..< j] \rangle$
 $\langle \text{proof} \rangle$

lemma *greaterThanAtMost-upt*:
 $\langle \{n <..m\} = \text{set } [Suc\ n..< Suc\ m] \rangle$
 $\langle \text{proof} \rangle$

lemma *atLeastAtMost-upt*:
 $\langle \{n..m\} = \text{set } [n..< Suc\ m] \rangle$
 $\langle \text{proof} \rangle$

lemma *split-list*: $x \in \text{set } xs \implies \exists ys\ zs. xs = ys @ x \# zs$
 $\langle \text{proof} \rangle$

lemma *in-set-conv-decomp*: $x \in \text{set } xs \longleftrightarrow (\exists ys\ zs. xs = ys @ x \# zs)$
 $\langle \text{proof} \rangle$

lemma *split-list-first*: $x \in \text{set } xs \implies \exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } ys$
 $\langle \text{proof} \rangle$

lemma *in-set-conv-decomp-first*:
 $\langle (x \in \text{set } xs) = (\exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } ys) \rangle$
 $\langle \text{proof} \rangle$

lemma *split-list-last*: $x \in \text{set } xs \implies \exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } zs$
 $\langle \text{proof} \rangle$

lemma *in-set-conv-decomp-last*:
 $\langle (x \in \text{set } xs) = (\exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } zs) \rangle$
 $\langle \text{proof} \rangle$

lemma *split-list-prop*: $\exists x \in \text{set } xs. P\ x \implies \exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x$
 $\langle \text{proof} \rangle$

lemma *split-list-propE*:
assumes $\exists x \in \text{set } xs. P\ x$
obtains $ys\ x\ zs$ **where** $xs = ys @ x \# zs$ **and** $P\ x$
 $\langle \text{proof} \rangle$

lemma *split-list-first-prop*:
 $\exists x \in \text{set } xs. P\ x \implies$
 $\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall y \in \text{set } ys. \neg P\ y)$
 $\langle \text{proof} \rangle$

lemma *split-list-first-propE*:

assumes $\exists x \in \text{set } xs. P\ x$
obtains $ys\ x\ zs$ **where** $xs = ys @ x \# zs$ **and** $P\ x$ **and** $\forall y \in \text{set } ys. \neg P\ y$
 $\langle \text{proof} \rangle$

lemma *split-list-first-prop-iff*:
 $(\exists x \in \text{set } xs. P\ x) \longleftrightarrow$
 $(\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall y \in \text{set } ys. \neg P\ y))$
 $\langle \text{proof} \rangle$

lemma *split-list-last-prop*:
 $\exists x \in \text{set } xs. P\ x \implies$
 $\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall z \in \text{set } zs. \neg P\ z)$
 $\langle \text{proof} \rangle$

lemma *split-list-last-propE*:
assumes $\exists x \in \text{set } xs. P\ x$
obtains $ys\ x\ zs$ **where** $xs = ys @ x \# zs$ **and** $P\ x$ **and** $\forall z \in \text{set } zs. \neg P\ z$
 $\langle \text{proof} \rangle$

lemma *split-list-last-prop-iff*:
 $(\exists x \in \text{set } xs. P\ x) \longleftrightarrow$
 $(\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall z \in \text{set } zs. \neg P\ z))$
 $\langle \text{proof} \rangle$

lemma *finite-list*: $\text{finite } A \implies \exists xs. \text{set } xs = A$
 $\langle \text{proof} \rangle$

lemma *card-length*: $\text{card } (\text{set } xs) \leq \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *set-minus-filter-out*:
 $\text{set } xs - \{y\} = \text{set } (\text{filter } (\lambda x. \neg (x = y))\ xs)$
 $\langle \text{proof} \rangle$

lemma *append-Cons-eq-iff*:
 $\llbracket x \notin \text{set } xs; x \notin \text{set } ys \rrbracket \implies$
 $xs @ x \# ys = xs' @ x \# ys' \longleftrightarrow (xs = xs' \wedge ys = ys')$
 $\langle \text{proof} \rangle$

66.1.8 concat

lemma *concat-append [simp]*: $\text{concat } (xs @ ys) = \text{concat } xs @ \text{concat } ys$
 $\langle \text{proof} \rangle$

lemma *concat-eq-Nil-conv [simp]*: $(\text{concat } xss = []) = (\forall xs \in \text{set } xss. xs = [])$
 $\langle \text{proof} \rangle$

lemmas *Nil-eq-concat-conv [simp]* = *concat-eq-Nil-conv [THEN eq-iff-swap]*

lemma *set-concat* [simp]: $\text{set } (\text{concat } xs) = (\bigcup_{x \in \text{set } xs} \text{set } x)$
 ⟨proof⟩

lemma *concat-map-singleton*[simp]: $\text{concat}(\text{map } (\%x. [f\ x])\ xs) = \text{map } f\ xs$
 ⟨proof⟩

lemma *map-concat*: $\text{map } f\ (\text{concat } xs) = \text{concat } (\text{map } (\text{map } f)\ xs)$
 ⟨proof⟩

lemma *rev-concat*: $\text{rev } (\text{concat } xs) = \text{concat } (\text{map } \text{rev } (\text{rev } xs))$
 ⟨proof⟩

lemma *length-concat-rev*[simp]: $\text{length } (\text{concat } (\text{rev } xs)) = \text{length } (\text{concat } xs)$
 ⟨proof⟩

lemma *concat-eq-concat-iff*: $\forall (x, y) \in \text{set } (\text{zip } xs\ ys). \text{length } x = \text{length } y \implies \text{length } xs = \text{length } ys \implies (\text{concat } xs = \text{concat } ys) = (xs = ys)$
 ⟨proof⟩

lemma *concat-injective*: $\text{concat } xs = \text{concat } ys \implies \text{length } xs = \text{length } ys \implies \forall (x, y) \in \text{set } (\text{zip } xs\ ys). \text{length } x = \text{length } y \implies xs = ys$
 ⟨proof⟩

lemma *concat-eq-appendD*:
 assumes $\text{concat } xss = ys @ zs$ and $xss \neq []$
 shows $\exists xss1\ xs\ xs'\ xss2. xss = xss1 @ (xs @ xs') \# xss2 \wedge ys = \text{concat } xss1 @ xs$
 $xs \wedge zs = xs' @ \text{concat } xss2$
 ⟨proof⟩

lemma *concat-eq-append-conv*:
 $\text{concat } xss = ys @ zs \longleftrightarrow$
 (if $xss = []$ then $ys = [] \wedge zs = []$
 else $\exists xss1\ xs\ xs'\ xss2. xss = xss1 @ (xs @ xs') \# xss2 \wedge ys = \text{concat } xss1 @ xs$
 $\wedge zs = xs' @ \text{concat } xss2$)
 ⟨proof⟩

lemma *hd-concat*: $xs \neq []; \text{hd } xs \neq [] \implies \text{hd } (\text{concat } xs) = \text{hd } (\text{hd } xs)$
 ⟨proof⟩

⟨ML⟩

66.1.9 filter

lemma *filter-append* [simp]: $\text{filter } P\ (xs @ ys) = \text{filter } P\ xs @ \text{filter } P\ ys$
 ⟨proof⟩

lemma *rev-filter*: $\text{rev } (\text{filter } P\ xs) = \text{filter } P\ (\text{rev } xs)$

$\langle \text{proof} \rangle$

lemma *filter-filter* [simp]: $\text{filter } P (\text{filter } Q \text{ } xs) = \text{filter } (\lambda x. Q \text{ } x \wedge P \text{ } x) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *filter-concat*: $\text{filter } p (\text{concat } xs) = \text{concat } (\text{map } (\text{filter } p) \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *length-filter-le* [simp]: $\text{length } (\text{filter } P \text{ } xs) \leq \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *sum-length-filter-compl*:
 $\text{length}(\text{filter } P \text{ } xs) + \text{length}(\text{filter } (\lambda x. \neg P \text{ } x) \text{ } xs) = \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *filter-True* [simp]: $\forall x \in \text{set } xs. P \text{ } x \implies \text{filter } P \text{ } xs = xs$
 $\langle \text{proof} \rangle$

lemma *filter-False* [simp]: $\forall x \in \text{set } xs. \neg P \text{ } x \implies \text{filter } P \text{ } xs = []$
 $\langle \text{proof} \rangle$

lemma *filter-empty-conv*: $(\text{filter } P \text{ } xs = []) = (\forall x \in \text{set } xs. \neg P \text{ } x)$
 $\langle \text{proof} \rangle$

lemmas *empty-filter-conv* = *filter-empty-conv*[*THEN* *eq-iff-swap*]

lemma *filter-id-conv*: $(\text{filter } P \text{ } xs = xs) = (\forall x \in \text{set } xs. P \text{ } x)$
 $\langle \text{proof} \rangle$

lemma *filter-map*: $\text{filter } P (\text{map } f \text{ } xs) = \text{map } f (\text{filter } (P \circ f) \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *length-filter-map*[simp]:
 $\text{length } (\text{filter } P (\text{map } f \text{ } xs)) = \text{length}(\text{filter } (P \circ f) \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *filter-is-subset* [simp]: $\text{set } (\text{filter } P \text{ } xs) \leq \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *length-filter-less*:
 $\llbracket x \in \text{set } xs; \neg P \text{ } x \rrbracket \implies \text{length}(\text{filter } P \text{ } xs) < \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *length-filter-conv-card*:
 $\text{length}(\text{filter } p \text{ } xs) = \text{card}\{i. i < \text{length } xs \wedge p(xs[i])\}$
 $\langle \text{proof} \rangle$

lemma *Cons-eq-filterD*:
 $x \# xs = \text{filter } P \text{ } ys \implies$

$\exists us\ vs.\ ys = us\ @\ x\ \# \ vs \wedge (\forall u \in set\ us.\ \neg P\ u) \wedge P\ x \wedge xs = filter\ P\ vs$
 $(is\ - \implies \exists us\ vs.\ ?P\ ys\ us\ vs)$
 $\langle proof \rangle$

lemma *filter-eq-ConsD*:

$filter\ P\ ys = x\ \# \ xs \implies$
 $\exists us\ vs.\ ys = us\ @\ x\ \# \ vs \wedge (\forall u \in set\ us.\ \neg P\ u) \wedge P\ x \wedge xs = filter\ P\ vs$
 $\langle proof \rangle$

lemma *filter-eq-Cons-iff*:

$(filter\ P\ ys = x\ \# \ xs) =$
 $(\exists us\ vs.\ ys = us\ @\ x\ \# \ vs \wedge (\forall u \in set\ us.\ \neg P\ u) \wedge P\ x \wedge xs = filter\ P\ vs)$
 $\langle proof \rangle$

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 $filter\ (\lambda x.\ f\ y = f\ x)\ xs = filter\ (HOL.eq\ y)\ xs$
 $\langle proof \rangle$

lemma *filter-cong[fundef-cong]*:

$xs = ys \implies (\bigwedge x.\ x \in set\ ys \implies P\ x = Q\ x) \implies filter\ P\ xs = filter\ Q\ ys$
 $\langle proof \rangle$

66.1.10 List partitioning

primrec *partition* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \times 'a\ list$ **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*: $fst\ (partition\ P\ xs) = filter\ P\ xs$

$\langle proof \rangle$

lemma *partition-filter2*: $snd\ (partition\ P\ xs) = filter\ (Not\ o\ P)\ xs$

$\langle proof \rangle$

lemma *partition-P*:

assumes $partition\ P\ xs = (yes,\ no)$
shows $(\forall p \in set\ yes.\ P\ p) \wedge (\forall p \in set\ no.\ \neg P\ p)$
 $\langle proof \rangle$

lemma *partition-set*:

assumes $partition\ P\ xs = (yes,\ no)$
shows $set\ yes \cup set\ no = set\ xs$
 $\langle proof \rangle$

lemma *partition-filter-conv*[simp]:
 $\text{partition } f \text{ } xs = (\text{filter } f \text{ } xs, \text{filter } (\text{Not} \circ f) \text{ } xs)$
 $\langle \text{proof} \rangle$

declare *partition.simps*[simp del]

66.1.11 (!)

lemma *nth-Cons-0* [simp, code]: $(x \# xs)!0 = x$
 $\langle \text{proof} \rangle$

lemma *nth-Cons-Suc* [simp, code]: $(x \# xs)!(\text{Suc } n) = xs!n$
 $\langle \text{proof} \rangle$

declare *nth.simps* [simp del]

lemma *nth-Cons-pos*[simp]: $0 < n \implies (x \# xs)!n = xs!(n - 1)$
 $\langle \text{proof} \rangle$

lemma *nth-append*:
 $(xs @ ys)!n = (\text{if } n < \text{length } xs \text{ then } xs!n \text{ else } ys!(n - \text{length } xs))$
 $\langle \text{proof} \rangle$

lemma *nth-append-left*: $i < \text{length } xs \implies (xs @ ys)!i = xs!i$
 $\langle \text{proof} \rangle$

lemma *nth-append-right*: $i \geq \text{length } xs \implies (xs @ ys)!i = ys!(i - \text{length } xs)$
 $\langle \text{proof} \rangle$

lemma *nth-append-length* [simp]: $(xs @ x \# ys)! \text{length } xs = x$
 $\langle \text{proof} \rangle$

lemma *nth-append-length-plus*[simp]: $(xs @ ys)!(\text{length } xs + n) = ys!n$
 $\langle \text{proof} \rangle$

lemma *nth-map* [simp]: $n < \text{length } xs \implies (\text{map } f \text{ } xs)!n = f(xs!n)$
 $\langle \text{proof} \rangle$

lemma *nth-tl*: $n < \text{length } (tl \text{ } xs) \implies tl \text{ } xs!n = xs! \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *hd-conv-nth*: $xs \neq [] \implies \text{hd } xs = xs!0$
 $\langle \text{proof} \rangle$

lemma *list-eq-iff-nth-eq*:
 $(xs = ys) = (\text{length } xs = \text{length } ys \wedge (\forall i < \text{length } xs. xs!i = ys!i))$
 $\langle \text{proof} \rangle$

lemma *map-equality-iff*:

$map\ f\ xs = map\ g\ ys \longleftrightarrow length\ xs = length\ ys \wedge (\forall i < length\ ys. f\ (xs!i) = g\ (ys!i))$
 $\langle proof \rangle$

lemma *set-conv-nth*: $set\ xs = \{xs!i \mid i. i < length\ xs\}$
 $\langle proof \rangle$

lemma *in-set-conv-nth*: $(x \in set\ xs) = (\exists i < length\ xs. xs!i = x)$
 $\langle proof \rangle$

lemma *nth-equal-first-eq*:
 assumes $x \notin set\ xs$
 assumes $n \leq length\ xs$
 shows $(x \# xs) ! n = x \longleftrightarrow n = 0$ (**is** $?lhs \longleftrightarrow ?rhs$)
 $\langle proof \rangle$

lemma *nth-non-equal-first-eq*:
 assumes $x \neq y$
 shows $(x \# xs) ! n = y \longleftrightarrow xs ! (n - 1) = y \wedge n > 0$ (**is** $?lhs \longleftrightarrow ?rhs$)
 $\langle proof \rangle$

lemma *list-ball-nth*: $\llbracket n < length\ xs; \forall x \in set\ xs. P\ x \rrbracket \Longrightarrow P(xs!n)$
 $\langle proof \rangle$

lemma *nth-mem [simp]*: $n < length\ xs \Longrightarrow xs!n \in set\ xs$
 $\langle proof \rangle$

lemma *all-nth-imp-all-set*:
 $\llbracket \forall i < length\ xs. P(xs!i); x \in set\ xs \rrbracket \Longrightarrow P\ x$
 $\langle proof \rangle$

lemma *all-set-conv-all-nth*:
 $(\forall x \in set\ xs. P\ x) = (\forall i. i < length\ xs \longrightarrow P\ (xs ! i))$
 $\langle proof \rangle$

lemma *rev-nth*:
 $n < size\ xs \Longrightarrow rev\ xs ! n = xs ! (length\ xs - Suc\ n)$
 $\langle proof \rangle$

lemma *Skolem-list-nth*:
 $(\forall i < k. \exists x. P\ i\ x) = (\exists xs. size\ xs = k \wedge (\forall i < k. P\ i\ (xs!i)))$
 (**is** $- = (\exists xs. ?P\ k\ xs)$)
 $\langle proof \rangle$

66.1.12 list-update

lemma *length-list-update [simp]*: $length(xs[i:=x]) = length\ xs$
 $\langle proof \rangle$

lemma *nth-list-update*:

$$i < \text{length } xs \implies (xs[i:=x])!j = (\text{if } i = j \text{ then } x \text{ else } xs!j)$$

<proof>

lemma *nth-list-update-eq [simp]*: $i < \text{length } xs \implies (xs[i:=x])!i = x$

<proof>

lemma *nth-list-update-neq [simp]*: $i \neq j \implies xs[i:=x]!j = xs!j$

<proof>

lemma *list-update-id [simp]*: $xs[i := xs!i] = xs$

<proof>

lemma *list-update-beyond*: $\text{length } xs \leq i \implies xs[i:=x] = xs$

<proof>

lemma *list-update-nonempty [simp]*: $xs[k:=x] = [] \longleftrightarrow xs=[]$

<proof>

lemma *list-update-same-conv*:

$$i < \text{length } xs \implies (xs[i := x] = xs) = (xs!i = x)$$

<proof>

lemma *list-update-append1*:

$$i < \text{size } xs \implies (xs @ ys)[i:=x] = xs[i:=x] @ ys$$

<proof>

lemma *list-update-append*:

$$(xs @ ys) [n:=x] =$$

$$(\text{if } n < \text{length } xs \text{ then } xs[n:=x] @ ys \text{ else } xs @ (ys [n-\text{length } xs:=x]))$$

<proof>

lemma *list-update-length [simp]*:

$$(xs @ x \# ys)[\text{length } xs := y] = (xs @ y \# ys)$$

<proof>

lemma *map-update*: $\text{map } f (xs[k:=y]) = (\text{map } f xs)[k := f y]$

<proof>

lemma *rev-update*:

$$k < \text{length } xs \implies \text{rev } (xs[k:=y]) = (\text{rev } xs)[\text{length } xs - k - 1 := y]$$

<proof>

lemma *update-zip*:

$$(\text{zip } xs \ ys)[i:=xy] = \text{zip } (xs[i:=fst \ xy]) \ (ys[i:=snd \ xy])$$

<proof>

lemma *set-update-subset-insert*: $\text{set}(xs[i:=x]) \leq \text{insert } x \ (\text{set } xs)$

<proof>

lemma *set-update-subsetI*: $\llbracket \text{set } xs \subseteq A; x \in A \rrbracket \implies \text{set}(xs[i := x]) \subseteq A$
 $\langle \text{proof} \rangle$

lemma *set-update-memI*: $n < \text{length } xs \implies x \in \text{set } (xs[n := x])$
 $\langle \text{proof} \rangle$

lemma *list-update-overwrite[simp]*:
 $xs[i := x, i := y] = xs[i := y]$
 $\langle \text{proof} \rangle$

lemma *list-update-swap*:
 $i \neq i' \implies xs[i := x, i' := x'] = xs[i' := x', i := x]$
 $\langle \text{proof} \rangle$

lemma *list-update-code [code]*:
 $\llbracket i := y \rrbracket = []$
 $(x \# xs)[0 := y] = y \# xs$
 $(x \# xs)[\text{Suc } i := y] = x \# xs[i := y]$
 $\langle \text{proof} \rangle$

66.1.13 *last and butlast*

lemma *hd-Nil-eq-last*: $\text{hd } \text{Nil} = \text{last } \text{Nil}$
 $\langle \text{proof} \rangle$

lemma *last-snoc [simp]*: $\text{last } (xs @ [x]) = x$
 $\langle \text{proof} \rangle$

lemma *butlast-snoc [simp]*: $\text{butlast } (xs @ [x]) = xs$
 $\langle \text{proof} \rangle$

lemma *last-ConsL*: $xs = [] \implies \text{last}(x \# xs) = x$
 $\langle \text{proof} \rangle$

lemma *last-ConsR*: $xs \neq [] \implies \text{last}(x \# xs) = \text{last } xs$
 $\langle \text{proof} \rangle$

lemma *last-append*: $\text{last}(xs @ ys) = (\text{if } ys = [] \text{ then } \text{last } xs \text{ else } \text{last } ys)$
 $\langle \text{proof} \rangle$

lemma *last-appendL[simp]*: $ys = [] \implies \text{last}(xs @ ys) = \text{last } xs$
 $\langle \text{proof} \rangle$

lemma *last-appendR[simp]*: $ys \neq [] \implies \text{last}(xs @ ys) = \text{last } ys$
 $\langle \text{proof} \rangle$

lemma *last-tl*: $xs = [] \vee \text{tl } xs \neq [] \implies \text{last } (\text{tl } xs) = \text{last } xs$
 $\langle \text{proof} \rangle$

lemma *butlast-tl*: $\text{butlast } (\text{tl } xs) = \text{tl } (\text{butlast } xs)$
 $\langle \text{proof} \rangle$

lemma *hd-rev*: $\text{hd}(\text{rev } xs) = \text{last } xs$
 $\langle \text{proof} \rangle$

lemma *last-rev*: $\text{last}(\text{rev } xs) = \text{hd } xs$
 $\langle \text{proof} \rangle$

lemma *last-in-set[simp]*: $as \neq [] \implies \text{last } as \in \text{set } as$
 $\langle \text{proof} \rangle$

lemma *length-butlast [simp]*: $\text{length } (\text{butlast } xs) = \text{length } xs - 1$
 $\langle \text{proof} \rangle$

lemma *butlast-append*:
 $\text{butlast } (xs @ ys) = (\text{if } ys = [] \text{ then } \text{butlast } xs \text{ else } xs @ \text{butlast } ys)$
 $\langle \text{proof} \rangle$

lemma *append-butlast-last-id [simp]*:
 $xs \neq [] \implies \text{butlast } xs @ [\text{last } xs] = xs$
 $\langle \text{proof} \rangle$

lemma *in-set-butlastD*: $x \in \text{set } (\text{butlast } xs) \implies x \in \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *in-set-butlast-appendI*:
 $x \in \text{set } (\text{butlast } xs) \vee x \in \text{set } (\text{butlast } ys) \implies x \in \text{set } (\text{butlast } (xs @ ys))$
 $\langle \text{proof} \rangle$

lemma *last-drop[simp]*: $n < \text{length } xs \implies \text{last } (\text{drop } n \text{ } xs) = \text{last } xs$
 $\langle \text{proof} \rangle$

lemma *nth-butlast*:
assumes $n < \text{length } (\text{butlast } xs)$ **shows** $\text{butlast } xs ! n = xs ! n$
 $\langle \text{proof} \rangle$

lemma *last-conv-nth*: $xs \neq [] \implies \text{last } xs = xs!(\text{length } xs - 1)$
 $\langle \text{proof} \rangle$

lemma *butlast-conv-take*: $\text{butlast } xs = \text{take } (\text{length } xs - 1) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *last-list-update*:
 $xs \neq [] \implies \text{last}(xs[k:=x]) = (\text{if } k = \text{size } xs - 1 \text{ then } x \text{ else } \text{last } xs)$
 $\langle \text{proof} \rangle$

lemma *butlast-list-update*:

$butlast(xs[k:=x]) =$
 $(if\ k = size\ xs - 1\ then\ butlast\ xs\ else\ (butlast\ xs)[k:=x])$
 $\langle proof \rangle$

lemma *last-map*: $xs \neq [] \implies last\ (map\ f\ xs) = f\ (last\ xs)$
 $\langle proof \rangle$

lemma *map-butlast*: $map\ f\ (butlast\ xs) = butlast\ (map\ f\ xs)$
 $\langle proof \rangle$

lemma *snoc-eq-iff-butlast*:
 $xs @ [x] = ys \iff (ys \neq [] \wedge butlast\ ys = xs \wedge last\ ys = x)$
 $\langle proof \rangle$

corollary *longest-common-suffix*:
 $\exists ss\ xs'\ ys'.\ xs = xs' @ ss \wedge ys = ys' @ ss$
 $\wedge (xs' = [] \vee ys' = [] \vee last\ xs' \neq last\ ys')$
 $\langle proof \rangle$

lemma *butlast-rev* [simp]: $butlast\ (rev\ xs) = rev\ (tl\ xs)$
 $\langle proof \rangle$

66.1.14 take and drop

lemma *take-0*: $take\ 0\ xs = []$
 $\langle proof \rangle$

lemma *drop-0*: $drop\ 0\ xs = xs$
 $\langle proof \rangle$

lemma *take0* [simp]: $take\ 0 = (\lambda xs. [])$
 $\langle proof \rangle$

lemma *drop0* [simp]: $drop\ 0 = (\lambda x. x)$
 $\langle proof \rangle$

lemma *take-Suc-Cons* [simp]: $take\ (Suc\ n)\ (x \# xs) = x \# take\ n\ xs$
 $\langle proof \rangle$

lemma *drop-Suc-Cons* [simp]: $drop\ (Suc\ n)\ (x \# xs) = drop\ n\ xs$
 $\langle proof \rangle$

declare *take-Cons* [simp del] **and** *drop-Cons* [simp del]

lemma *take-Suc*: $xs \neq [] \implies take\ (Suc\ n)\ xs = hd\ xs \# take\ n\ (tl\ xs)$
 $\langle proof \rangle$

lemma *drop-Suc*: $drop\ (Suc\ n)\ xs = drop\ n\ (tl\ xs)$
 $\langle proof \rangle$

lemma *hd-take[simp]*: $j > 0 \implies \text{hd } (\text{take } j \text{ } xs) = \text{hd } xs$
 $\langle \text{proof} \rangle$

lemma *take-tl*: $\text{take } n \text{ } (\text{tl } xs) = \text{tl } (\text{take } (\text{Suc } n) \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *drop-tl*: $\text{drop } n \text{ } (\text{tl } xs) = \text{tl } (\text{drop } n \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *tl-take*: $\text{tl } (\text{take } n \text{ } xs) = \text{take } (n - 1) \text{ } (\text{tl } xs)$
 $\langle \text{proof} \rangle$

lemma *tl-drop*: $\text{tl } (\text{drop } n \text{ } xs) = \text{drop } n \text{ } (\text{tl } xs)$
 $\langle \text{proof} \rangle$

lemma *nth-via-drop*: $\text{drop } n \text{ } xs = y \# ys \implies xs!n = y$
 $\langle \text{proof} \rangle$

lemma *take-Suc-conv-app-nth*:
 $i < \text{length } xs \implies \text{take } (\text{Suc } i) \text{ } xs = \text{take } i \text{ } xs @ [xs!i]$
 $\langle \text{proof} \rangle$

lemma *Cons-nth-drop-Suc*:
 $i < \text{length } xs \implies (xs!i) \# (\text{drop } (\text{Suc } i) \text{ } xs) = \text{drop } i \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *length-take [simp]*: $\text{length } (\text{take } n \text{ } xs) = \min (\text{length } xs) \text{ } n$
 $\langle \text{proof} \rangle$

lemma *length-drop [simp]*: $\text{length } (\text{drop } n \text{ } xs) = (\text{length } xs - n)$
 $\langle \text{proof} \rangle$

lemma *take-all [simp]*: $\text{length } xs \leq n \implies \text{take } n \text{ } xs = xs$
 $\langle \text{proof} \rangle$

lemma *drop-all [simp]*: $\text{length } xs \leq n \implies \text{drop } n \text{ } xs = []$
 $\langle \text{proof} \rangle$

lemma *take-all-iff [simp]*: $\text{take } n \text{ } xs = xs \longleftrightarrow \text{length } xs \leq n$
 $\langle \text{proof} \rangle$

lemma *take-eq-Nil[simp]*: $(\text{take } n \text{ } xs = []) = (n = 0 \vee xs = [])$
 $\langle \text{proof} \rangle$

lemmas *take-eq-Nil2[simp]* = *take-eq-Nil[THEN eq-iff-swap]*

lemma *drop-eq-Nil* [simp]: $\text{drop } n \text{ } xs = [] \longleftrightarrow \text{length } xs \leq n$
 ⟨proof⟩

lemmas *drop-eq-Nil2* [simp] = *drop-eq-Nil*[THEN *eq-iff-swap*]

lemma *take-append* [simp]:
 $\text{take } n \text{ } (xs @ ys) = (\text{take } n \text{ } xs @ \text{take } (n - \text{length } xs) \text{ } ys)$
 ⟨proof⟩

lemma *drop-append* [simp]:
 $\text{drop } n \text{ } (xs @ ys) = \text{drop } n \text{ } xs @ \text{drop } (n - \text{length } xs) \text{ } ys$
 ⟨proof⟩

lemma *take-take* [simp]: $\text{take } n \text{ } (\text{take } m \text{ } xs) = \text{take } (\min n \text{ } m) \text{ } xs$
 ⟨proof⟩

lemma *drop-drop* [simp]: $\text{drop } n \text{ } (\text{drop } m \text{ } xs) = \text{drop } (n + m) \text{ } xs$
 ⟨proof⟩

lemma *take-drop*: $\text{take } n \text{ } (\text{drop } m \text{ } xs) = \text{drop } m \text{ } (\text{take } (n + m) \text{ } xs)$
 ⟨proof⟩

lemma *drop-take*: $\text{drop } n \text{ } (\text{take } m \text{ } xs) = \text{take } (m - n) \text{ } (\text{drop } n \text{ } xs)$
 ⟨proof⟩

lemma *append-take-drop-id* [simp]: $\text{take } n \text{ } xs @ \text{drop } n \text{ } xs = xs$
 ⟨proof⟩

lemma *take-map*: $\text{take } n \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{take } n \text{ } xs)$
 ⟨proof⟩

lemma *drop-map*: $\text{drop } n \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{drop } n \text{ } xs)$
 ⟨proof⟩

lemma *rev-take*: $\text{rev } (\text{take } i \text{ } xs) = \text{drop } (\text{length } xs - i) \text{ } (\text{rev } xs)$
 ⟨proof⟩

lemma *rev-drop*: $\text{rev } (\text{drop } i \text{ } xs) = \text{take } (\text{length } xs - i) \text{ } (\text{rev } xs)$
 ⟨proof⟩

lemma *drop-rev*: $\text{drop } n \text{ } (\text{rev } xs) = \text{rev } (\text{take } (\text{length } xs - n) \text{ } xs)$
 ⟨proof⟩

lemma *take-rev*: $\text{take } n \text{ } (\text{rev } xs) = \text{rev } (\text{drop } (\text{length } xs - n) \text{ } xs)$
 ⟨proof⟩

lemma *nth-take* [simp]: $i < n \implies (\text{take } n \text{ } xs)!i = xs!i$
 ⟨proof⟩

lemma *nth-drop [simp]*:

$$n \leq \text{length } xs \implies (\text{drop } n \text{ } xs)!i = xs!(n + i)$$

<proof>

lemma *butlast-take*:

$$n \leq \text{length } xs \implies \text{butlast } (\text{take } n \text{ } xs) = \text{take } (n - 1) \text{ } xs$$

<proof>

lemma *butlast-drop*: $\text{butlast } (\text{drop } n \text{ } xs) = \text{drop } n \text{ } (\text{butlast } xs)$

<proof>

lemma *take-butlast*: $n < \text{length } xs \implies \text{take } n \text{ } (\text{butlast } xs) = \text{take } n \text{ } xs$

<proof>

lemma *drop-butlast*: $\text{drop } n \text{ } (\text{butlast } xs) = \text{butlast } (\text{drop } n \text{ } xs)$

<proof>

lemma *butlast-power*: $(\text{butlast } \sim n) \text{ } xs = \text{take } (\text{length } xs - n) \text{ } xs$

<proof>

lemma *hd-drop-conv-nth*: $n < \text{length } xs \implies \text{hd}(\text{drop } n \text{ } xs) = xs!n$

<proof>

lemma *set-take-subset-set-take*:

$$m \leq n \implies \text{set}(\text{take } m \text{ } xs) \subseteq \text{set}(\text{take } n \text{ } xs)$$

<proof>

lemma *set-take-subset*: $\text{set}(\text{take } n \text{ } xs) \subseteq \text{set } xs$

<proof>

lemma *set-drop-subset*: $\text{set}(\text{drop } n \text{ } xs) \subseteq \text{set } xs$

<proof>

lemma *set-drop-subset-set-drop*:

$$m \geq n \implies \text{set}(\text{drop } m \text{ } xs) \subseteq \text{set}(\text{drop } n \text{ } xs)$$

<proof>

lemma *in-set-takeD*: $x \in \text{set}(\text{take } n \text{ } xs) \implies x \in \text{set } xs$

<proof>

lemma *in-set-dropD*: $x \in \text{set}(\text{drop } n \text{ } xs) \implies x \in \text{set } xs$

<proof>

lemma *append-eq-conv-conj*:

$$(xs @ ys = zs) = (xs = \text{take } (\text{length } xs) \text{ } zs \wedge ys = \text{drop } (\text{length } xs) \text{ } zs)$$

<proof>

lemma *map-eq-append-conv*:

$$\text{map } f \text{ } xs = ys @ zs \longleftrightarrow (\exists us \text{ } vs. \text{ } xs = us @ vs \wedge ys = \text{map } f \text{ } us \wedge zs = \text{map } f \text{ } vs)$$

xs)
 $\langle proof \rangle$

lemmas *append-eq-map-conv* = *map-eq-append-conv* [THEN *eq-iff-swap*]

lemma *take-add*: $take\ (i+j)\ xs = take\ i\ xs\ @\ take\ j\ (drop\ i\ xs)$
 $\langle proof \rangle$

lemma *append-eq-append-conv-if*:
 $(xs_1\ @\ xs_2 = ys_1\ @\ ys_2) =$
 $(if\ size\ xs_1 \leq size\ ys_1$
 $\ then\ xs_1 = take\ (size\ xs_1)\ ys_1 \wedge xs_2 = drop\ (size\ xs_1)\ ys_1\ @\ ys_2$
 $\ else\ take\ (size\ ys_1)\ xs_1 = ys_1 \wedge drop\ (size\ ys_1)\ xs_1\ @\ xs_2 = ys_2)$
 $\langle proof \rangle$

lemma *take-hd-drop*:
 $n < length\ xs \implies take\ n\ xs\ @\ [hd\ (drop\ n\ xs)] = take\ (Suc\ n)\ xs$
 $\langle proof \rangle$

lemma *id-take-nth-drop*:
 $i < length\ xs \implies xs = take\ i\ xs\ @\ xs!i\ \# \ drop\ (Suc\ i)\ xs$
 $\langle proof \rangle$

lemma *take-update-cancel[simp]*: $n \leq m \implies take\ n\ (xs[m := y]) = take\ n\ xs$
 $\langle proof \rangle$

lemma *drop-update-cancel[simp]*: $n < m \implies drop\ m\ (xs[n := x]) = drop\ m\ xs$
 $\langle proof \rangle$

lemma *upd-conv-take-nth-drop*:
 $i < length\ xs \implies xs[i:=a] = take\ i\ xs\ @\ a\ \# \ drop\ (Suc\ i)\ xs$
 $\langle proof \rangle$

lemma *take-update-swap*: $take\ m\ (xs[n := x]) = (take\ m\ xs)[n := x]$
 $\langle proof \rangle$

lemma *drop-update-swap*:
assumes $m \leq n$ **shows** $drop\ m\ (xs[n := x]) = (drop\ m\ xs)[n-m := x]$
 $\langle proof \rangle$

lemma *nth-image*: $l \leq size\ xs \implies nth\ xs\ ' \{0..<l\} = set(take\ l\ xs)$
 $\langle proof \rangle$

lemma *set-list-update*:
 $set\ (xs\ [i := k]) = insert\ k\ (set\ (take\ i\ xs) \cup set\ (drop\ (Suc\ i)\ xs))$
if $\langle i < length\ xs \rangle$
 $\langle proof \rangle$

66.1.15 *takeWhile* and *dropWhile*

lemma *length-takeWhile-le*: $\text{length } (\text{takeWhile } P \text{ } xs) \leq \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *takeWhile-dropWhile-id* [simp]: $\text{takeWhile } P \text{ } xs @ \text{dropWhile } P \text{ } xs = xs$
 $\langle \text{proof} \rangle$

lemma *takeWhile-append1* [simp]:
 $\llbracket x \in \text{set } xs; \neg P(x) \rrbracket \implies \text{takeWhile } P \text{ } (xs @ ys) = \text{takeWhile } P \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *takeWhile-append2* [simp]:
 $(\bigwedge x. x \in \text{set } xs \implies P \text{ } x) \implies \text{takeWhile } P \text{ } (xs @ ys) = xs @ \text{takeWhile } P \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *takeWhile-append*:
 $\text{takeWhile } P \text{ } (xs @ ys) = (\text{if } \forall x \in \text{set } xs. P \text{ } x \text{ then } xs @ \text{takeWhile } P \text{ } ys \text{ else } \text{takeWhile } P \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *takeWhile-tail*: $\neg P \text{ } x \implies \text{takeWhile } P \text{ } (xs @ (x \# l)) = \text{takeWhile } P \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *takeWhile-eq-Nil-iff*: $\text{takeWhile } P \text{ } xs = [] \iff xs = [] \vee \neg P \text{ } (\text{hd } xs)$
 $\langle \text{proof} \rangle$

lemma *takeWhile-nth*: $j < \text{length } (\text{takeWhile } P \text{ } xs) \implies \text{takeWhile } P \text{ } xs ! j = xs ! j$
 $\langle \text{proof} \rangle$

lemma *takeWhile-takeWhile*: $\text{takeWhile } Q \text{ } (\text{takeWhile } P \text{ } xs) = \text{takeWhile } (\lambda x. P \text{ } x \wedge Q \text{ } x) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *dropWhile-nth*: $j < \text{length } (\text{dropWhile } P \text{ } xs) \implies \text{dropWhile } P \text{ } xs ! j = xs ! (j + \text{length } (\text{takeWhile } P \text{ } xs))$
 $\langle \text{proof} \rangle$

lemma *length-dropWhile-le*: $\text{length } (\text{dropWhile } P \text{ } xs) \leq \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *dropWhile-append1* [simp]:
 $\llbracket x \in \text{set } xs; \neg P(x) \rrbracket \implies \text{dropWhile } P \text{ } (xs @ ys) = (\text{dropWhile } P \text{ } xs) @ ys$
 $\langle \text{proof} \rangle$

lemma *dropWhile-append2* [simp]:
 $(\bigwedge x. x \in \text{set } xs \implies P(x)) \implies \text{dropWhile } P \text{ } (xs @ ys) = \text{dropWhile } P \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *dropWhile-id[simp]*:

$(\bigwedge x. x \in \text{set } xs \implies \neg P x) \implies \text{dropWhile } P \text{ } xs = xs$
 $\langle \text{proof} \rangle$

lemma *dropWhile-append3*:

$\neg P y \implies \text{dropWhile } P (xs @ y \# ys) = \text{dropWhile } P \text{ } xs @ y \# ys$
 $\langle \text{proof} \rangle$

lemma *dropWhile-append*:

$\text{dropWhile } P (xs @ ys) = (\text{if } \forall x \in \text{set } xs. P x \text{ then } \text{dropWhile } P \text{ } ys \text{ else } \text{dropWhile } P \text{ } xs @ ys)$
 $\langle \text{proof} \rangle$

lemma *dropWhile-last*:

$x \in \text{set } xs \implies \neg P x \implies \text{last } (\text{dropWhile } P \text{ } xs) = \text{last } xs$
 $\langle \text{proof} \rangle$

lemma *set-dropWhileD*: $x \in \text{set } (\text{dropWhile } P \text{ } xs) \implies x \in \text{set } xs$

$\langle \text{proof} \rangle$

lemma *set-takeWhileD*: $x \in \text{set } (\text{takeWhile } P \text{ } xs) \implies x \in \text{set } xs \wedge P x$

$\langle \text{proof} \rangle$

lemma *takeWhile-eq-all-conv[simp]*:

$(\text{takeWhile } P \text{ } xs = xs) = (\forall x \in \text{set } xs. P x)$
 $\langle \text{proof} \rangle$

lemma *dropWhile-eq-Nil-conv[simp]*:

$(\text{dropWhile } P \text{ } xs = []) = (\forall x \in \text{set } xs. \neg P x)$
 $\langle \text{proof} \rangle$

lemma *dropWhile-eq-Cons-conv*:

$(\text{dropWhile } P \text{ } xs = y \# ys) = (xs = \text{takeWhile } P \text{ } xs @ y \# ys \wedge \neg P y)$
 $\langle \text{proof} \rangle$

lemma *dropWhile-eq-self-iff*: $\text{dropWhile } P \text{ } xs = xs \longleftrightarrow xs = [] \vee \neg P (\text{hd } xs)$

$\langle \text{proof} \rangle$

lemma *dropWhile-dropWhile1*: $(\bigwedge x. Q x \implies P x) \implies \text{dropWhile } Q (\text{dropWhile } P \text{ } xs) = \text{dropWhile } P \text{ } xs$

$\langle \text{proof} \rangle$

lemma *dropWhile-dropWhile2*: $(\bigwedge x. P x \implies Q x) \implies \text{takeWhile } P (\text{takeWhile } Q \text{ } xs) = \text{takeWhile } P \text{ } xs$

$\langle \text{proof} \rangle$

lemma *dropWhile-takeWhile*:

$(\bigwedge x. P x \implies Q x) \implies \text{dropWhile } P (\text{takeWhile } Q \text{ } xs) = \text{takeWhile } Q (\text{dropWhile } P \text{ } xs)$

$\langle \text{proof} \rangle$

lemma *distinct-takeWhile[simp]*: $\text{distinct } xs \implies \text{distinct } (\text{takeWhile } P \ xs)$
 $\langle \text{proof} \rangle$

lemma *distinct-dropWhile[simp]*: $\text{distinct } xs \implies \text{distinct } (\text{dropWhile } P \ xs)$
 $\langle \text{proof} \rangle$

lemma *takeWhile-map*: $\text{takeWhile } P \ (\text{map } f \ xs) = \text{map } f \ (\text{takeWhile } (P \circ f) \ xs)$
 $\langle \text{proof} \rangle$

lemma *dropWhile-map*: $\text{dropWhile } P \ (\text{map } f \ xs) = \text{map } f \ (\text{dropWhile } (P \circ f) \ xs)$
 $\langle \text{proof} \rangle$

lemma *takeWhile-eq-take*: $\text{takeWhile } P \ xs = \text{take } (\text{length } (\text{takeWhile } P \ xs)) \ xs$
 $\langle \text{proof} \rangle$

lemma *dropWhile-eq-drop*: $\text{dropWhile } P \ xs = \text{drop } (\text{length } (\text{takeWhile } P \ xs)) \ xs$
 $\langle \text{proof} \rangle$

lemma *hd-dropWhile*: $\text{dropWhile } P \ xs \neq [] \implies \neg P \ (\text{hd } (\text{dropWhile } P \ xs))$
 $\langle \text{proof} \rangle$

lemma *takeWhile-eq-filter*:
assumes $\bigwedge x. x \in \text{set } (\text{dropWhile } P \ xs) \implies \neg P \ x$
shows $\text{takeWhile } P \ xs = \text{filter } P \ xs$
 $\langle \text{proof} \rangle$

lemma *takeWhile-eq-take-P-nth*:
 $\llbracket \bigwedge i. \llbracket i < n ; i < \text{length } xs \rrbracket \implies P \ (xs \ ! \ i) ; n < \text{length } xs \implies \neg P \ (xs \ ! \ n) \rrbracket$
 \implies
 $\text{takeWhile } P \ xs = \text{take } n \ xs$
 $\langle \text{proof} \rangle$

lemma *nth-length-takeWhile*:
 $\text{length } (\text{takeWhile } P \ xs) < \text{length } xs \implies \neg P \ (xs \ ! \ \text{length } (\text{takeWhile } P \ xs))$
 $\langle \text{proof} \rangle$

lemma *length-takeWhile-less-P-nth*:
assumes $\text{all: } \bigwedge i. i < j \implies P \ (xs \ ! \ i)$ **and** $j \leq \text{length } xs$
shows $j \leq \text{length } (\text{takeWhile } P \ xs)$
 $\langle \text{proof} \rangle$

lemma *takeWhile-neq-rev*: $\llbracket \text{distinct } xs ; x \in \text{set } xs \rrbracket \implies$
 $\text{takeWhile } (\lambda y. y \neq x) \ (\text{rev } xs) = \text{rev } (\text{tl } (\text{dropWhile } (\lambda y. y \neq x) \ xs))$
 $\langle \text{proof} \rangle$

lemma *dropWhile-neq-rev*: $\llbracket \text{distinct } xs ; x \in \text{set } xs \rrbracket \implies$
 $\text{dropWhile } (\lambda y. y \neq x) \ (\text{rev } xs) = x \ \# \ \text{rev } (\text{takeWhile } (\lambda y. y \neq x) \ xs)$

$\langle proof \rangle$

lemma *takeWhile-not-last*:

$distinct\ xs \implies takeWhile\ (\lambda y. y \neq last\ xs)\ xs = butlast\ xs$

$\langle proof \rangle$

lemma *takeWhile-cong [fundef-cong]*:

$\llbracket l = k; \bigwedge x. x \in set\ l \implies P\ x = Q\ x \rrbracket$

$\implies takeWhile\ P\ l = takeWhile\ Q\ k$

$\langle proof \rangle$

lemma *dropWhile-cong [fundef-cong]*:

$\llbracket l = k; \bigwedge x. x \in set\ l \implies P\ x = Q\ x \rrbracket$

$\implies dropWhile\ P\ l = dropWhile\ Q\ k$

$\langle proof \rangle$

lemma *takeWhile-idem [simp]*:

$takeWhile\ P\ (takeWhile\ P\ xs) = takeWhile\ P\ xs$

$\langle proof \rangle$

lemma *dropWhile-idem [simp]*:

$dropWhile\ P\ (dropWhile\ P\ xs) = dropWhile\ P\ xs$

$\langle proof \rangle$

66.1.16 *zip*

lemma *zip-Nil [simp]*: $zip\ []\ ys = []$

$\langle proof \rangle$

lemma *zip-Cons-Cons [simp]*: $zip\ (x \# xs)\ (y \# ys) = (x, y) \# zip\ xs\ ys$

$\langle proof \rangle$

declare *zip-Cons [simp del]*

lemma *[code]*:

$zip\ []\ ys = []$

$zip\ xs\ [] = []$

$zip\ (x \# xs)\ (y \# ys) = (x, y) \# zip\ xs\ ys$

$\langle proof \rangle$

lemma *zip-Cons1*:

$zip\ (x \# xs)\ ys = (case\ ys\ of\ [] \Rightarrow [] \mid y \# ys \Rightarrow (x, y) \# zip\ xs\ ys)$

$\langle proof \rangle$

lemma *length-zip [simp]*:

$length\ (zip\ xs\ ys) = min\ (length\ xs)\ (length\ ys)$

$\langle proof \rangle$

lemma *zip-obtain-same-length*:

assumes $\bigwedge zs\ ws\ n.\ length\ zs = length\ ws \implies n = \min\ (length\ xs)\ (length\ ys)$
 $\implies zs = take\ n\ xs \implies ws = take\ n\ ys \implies P\ (zip\ zs\ ws)$
shows $P\ (zip\ xs\ ys)$
 $\langle proof \rangle$

lemma *zip-append1*:
 $zip\ (xs\ @\ ys)\ zs =$
 $zip\ xs\ (take\ (length\ xs)\ zs)\ @\ zip\ ys\ (drop\ (length\ xs)\ zs)$
 $\langle proof \rangle$

lemma *zip-append2*:
 $zip\ xs\ (ys\ @\ zs) =$
 $zip\ (take\ (length\ ys)\ xs)\ ys\ @\ zip\ (drop\ (length\ ys)\ xs)\ zs$
 $\langle proof \rangle$

lemma *zip-append [simp]*:
 $\llbracket length\ xs = length\ us \rrbracket \implies$
 $zip\ (xs@ys)\ (us@vs) = zip\ xs\ us\ @\ zip\ ys\ vs$
 $\langle proof \rangle$

lemma *zip-rev*:
 $length\ xs = length\ ys \implies zip\ (rev\ xs)\ (rev\ ys) = rev\ (zip\ xs\ ys)$
 $\langle proof \rangle$

lemma *zip-map-map*:
 $zip\ (map\ f\ xs)\ (map\ g\ ys) = map\ (\lambda\ (x,\ y).\ (f\ x,\ g\ y))\ (zip\ xs\ ys)$
 $\langle proof \rangle$

lemma *zip-map1*:
 $zip\ (map\ f\ xs)\ ys = map\ (\lambda(x,\ y).\ (f\ x,\ y))\ (zip\ xs\ ys)$
 $\langle proof \rangle$

lemma *zip-map2*:
 $zip\ xs\ (map\ f\ ys) = map\ (\lambda(x,\ y).\ (x,\ f\ y))\ (zip\ xs\ ys)$
 $\langle proof \rangle$

lemma *map-zip-map*:
 $map\ f\ (zip\ (map\ g\ xs)\ ys) = map\ (\% (x,y).\ f(g\ x,\ y))\ (zip\ xs\ ys)$
 $\langle proof \rangle$

lemma *map-zip-map2*:
 $map\ f\ (zip\ xs\ (map\ g\ ys)) = map\ (\% (x,y).\ f(x,\ g\ y))\ (zip\ xs\ ys)$
 $\langle proof \rangle$

Courtesy of Andreas Lochbihler:

lemma *zip-same-conv-map*: $zip\ xs\ xs = map\ (\lambda x.\ (x,\ x))\ xs$
 $\langle proof \rangle$

lemma *nth-zip [simp]*:

$\llbracket i < \text{length } xs; i < \text{length } ys \rrbracket \implies (\text{zip } xs \text{ } ys)!i = (xs!i, ys!i)$
 $\langle \text{proof} \rangle$

lemma *set-zip*:

$\text{set } (\text{zip } xs \text{ } ys) = \{(xs!i, ys!i) \mid i. i < \min (\text{length } xs) (\text{length } ys)\}$
 $\langle \text{proof} \rangle$

lemma *zip-same*: $((a,b) \in \text{set } (\text{zip } xs \text{ } xs)) = (a \in \text{set } xs \wedge a = b)$
 $\langle \text{proof} \rangle$

lemma *zip-update*: $\text{zip } (xs[i:=x]) (ys[i:=y]) = (\text{zip } xs \text{ } ys)[i:=(x,y)]$
 $\langle \text{proof} \rangle$

lemma *zip-replicate* [simp]:

$\text{zip } (\text{replicate } i \text{ } x) (\text{replicate } j \text{ } y) = \text{replicate } (\min i \text{ } j) (x,y)$
 $\langle \text{proof} \rangle$

lemma *zip-replicate1*: $\text{zip } (\text{replicate } n \text{ } x) \text{ } ys = \text{map } (\text{Pair } x) (\text{take } n \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *take-zip*: $\text{take } n (\text{zip } xs \text{ } ys) = \text{zip } (\text{take } n \text{ } xs) (\text{take } n \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *drop-zip*: $\text{drop } n (\text{zip } xs \text{ } ys) = \text{zip } (\text{drop } n \text{ } xs) (\text{drop } n \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *zip-takeWhile-fst*: $\text{zip } (\text{takeWhile } P \text{ } xs) \text{ } ys = \text{takeWhile } (P \circ \text{fst}) (\text{zip } xs \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *zip-takeWhile-snd*: $\text{zip } xs (\text{takeWhile } P \text{ } ys) = \text{takeWhile } (P \circ \text{snd}) (\text{zip } xs \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *set-zip-leftD*: $(x,y) \in \text{set } (\text{zip } xs \text{ } ys) \implies x \in \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *set-zip-rightD*: $(x,y) \in \text{set } (\text{zip } xs \text{ } ys) \implies y \in \text{set } ys$
 $\langle \text{proof} \rangle$

lemma *in-set-zipE*:

$(x,y) \in \text{set}(\text{zip } xs \text{ } ys) \implies (\llbracket x \in \text{set } xs; y \in \text{set } ys \rrbracket \implies R) \implies R$
 $\langle \text{proof} \rangle$

lemma *zip-map-fst-snd*: $\text{zip } (\text{map } \text{fst } zs) (\text{map } \text{snd } zs) = zs$
 $\langle \text{proof} \rangle$

lemma *zip-eq-conv*:

$\text{length } xs = \text{length } ys \implies \text{zip } xs \text{ } ys = zs \longleftrightarrow \text{map } \text{fst } zs = xs \wedge \text{map } \text{snd } zs = ys$

⟨proof⟩

lemma *in-set-zip*:

$p \in \text{set } (\text{zip } xs \ ys) \longleftrightarrow (\exists n. \ xs \ ! \ n = \text{fst } p \wedge \text{ys} \ ! \ n = \text{snd } p$
 $\wedge \ n < \text{length } xs \wedge \ n < \text{length } ys)$
 ⟨proof⟩

lemma *in-set-impl-in-set-zip1*:

assumes $\text{length } xs = \text{length } ys$
assumes $x \in \text{set } xs$
obtains y **where** $(x, y) \in \text{set } (\text{zip } xs \ ys)$
 ⟨proof⟩

lemma *in-set-impl-in-set-zip2*:

assumes $\text{length } xs = \text{length } ys$
assumes $y \in \text{set } ys$
obtains x **where** $(x, y) \in \text{set } (\text{zip } xs \ ys)$
 ⟨proof⟩

lemma *zip-eq-Nil-iff[simp]*:

$\text{zip } xs \ ys = [] \longleftrightarrow xs = [] \vee ys = []$
 ⟨proof⟩

lemmas *Nil-eq-zip-iff[simp]* = *zip-eq-Nil-iff[THEN eq-iff-swap]*

lemma *zip-eq-ConsE*:

assumes $\text{zip } xs \ ys = xy \ \# \ xys$
obtains $x \ xs' \ y \ ys'$ **where** $xs = x \ \# \ xs'$
and $ys = y \ \# \ ys'$ **and** $xy = (x, y)$
and $xys = \text{zip } xs' \ ys'$
 ⟨proof⟩

lemma *semilattice-map2*:

$\text{semilattice } (\text{map2 } (*))$ **if** $\text{semilattice } (*)$
for f (**infixl** $\langle * \rangle$ 70)
 ⟨proof⟩

lemma *pair-list-eqI*:

assumes $\text{map } \text{fst } xs = \text{map } \text{fst } ys$ **and** $\text{map } \text{snd } xs = \text{map } \text{snd } ys$
shows $xs = ys$
 ⟨proof⟩

lemma *hd-zip*:

$\langle \text{hd } (\text{zip } xs \ ys) \rangle = (\text{hd } xs, \text{hd } ys)$ **if** $\langle xs \neq [] \rangle$ **and** $\langle ys \neq [] \rangle$
 ⟨proof⟩

lemma *last-zip*:

$\langle \text{last } (\text{zip } xs \ ys) \rangle = (\text{last } xs, \text{last } ys)$ **if** $\langle xs \neq [] \rangle$ **and** $\langle ys \neq [] \rangle$
and $\langle \text{length } xs = \text{length } ys \rangle$

$\langle \text{proof} \rangle$

66.1.17 *list-all2*

lemma *list-all2-lengthD* [intro?]:

$\text{list-all2 } P \text{ } xs \text{ } ys \implies \text{length } xs = \text{length } ys$

$\langle \text{proof} \rangle$

lemma *list-all2-Nil* [iff, code]: $\text{list-all2 } P \text{ } [] \text{ } ys = (ys = [])$

$\langle \text{proof} \rangle$

lemma *list-all2-Nil2* [iff, code]: $\text{list-all2 } P \text{ } xs \text{ } [] = (xs = [])$

$\langle \text{proof} \rangle$

lemma *list-all2-Cons* [iff, code]:

$\text{list-all2 } P \text{ } (x \# xs) \text{ } (y \# ys) = (P \text{ } x \text{ } y \wedge \text{list-all2 } P \text{ } xs \text{ } ys)$

$\langle \text{proof} \rangle$

lemma *list-all2-Cons1*:

$\text{list-all2 } P \text{ } (x \# xs) \text{ } ys = (\exists z \text{ } zs. \text{ } ys = z \# zs \wedge P \text{ } x \text{ } z \wedge \text{list-all2 } P \text{ } xs \text{ } zs)$

$\langle \text{proof} \rangle$

lemma *list-all2-Cons2*:

$\text{list-all2 } P \text{ } xs \text{ } (y \# ys) = (\exists z \text{ } zs. \text{ } xs = z \# zs \wedge P \text{ } z \text{ } y \wedge \text{list-all2 } P \text{ } zs \text{ } ys)$

$\langle \text{proof} \rangle$

lemma *list-all2-induct*

[consumes 1, case-names Nil Cons, induct set: *list-all2*]:

assumes *P*: $\text{list-all2 } P \text{ } xs \text{ } ys$

assumes *Nil*: $R \text{ } [] \text{ } []$

assumes *Cons*: $\bigwedge x \text{ } xs \text{ } y \text{ } ys. \text{ } \llbracket P \text{ } x \text{ } y; \text{list-all2 } P \text{ } xs \text{ } ys; R \text{ } xs \text{ } ys \rrbracket \implies R \text{ } (x \# xs) \text{ } (y \# ys)$

shows $R \text{ } xs \text{ } ys$

$\langle \text{proof} \rangle$

lemma *list-all2-rev* [iff]:

$\text{list-all2 } P \text{ } (\text{rev } xs) \text{ } (\text{rev } ys) = \text{list-all2 } P \text{ } xs \text{ } ys$

$\langle \text{proof} \rangle$

lemma *list-all2-rev1*:

$\text{list-all2 } P \text{ } (\text{rev } xs) \text{ } ys = \text{list-all2 } P \text{ } xs \text{ } (\text{rev } ys)$

$\langle \text{proof} \rangle$

lemma *list-all2-append1*:

$\text{list-all2 } P \text{ } (xs @ ys) \text{ } zs =$

$(\exists us \text{ } vs. \text{ } zs = us @ vs \wedge \text{length } us = \text{length } xs \wedge \text{length } vs = \text{length } ys \wedge$

$\text{list-all2 } P \text{ } xs \text{ } us \wedge \text{list-all2 } P \text{ } ys \text{ } vs) \text{ (is ?lhs = ?rhs)}$

$\langle \text{proof} \rangle$

lemma *list-all2-append2*:

$list\text{-}all2\ P\ xs\ (ys\ @\ zs) =$
 $(\exists\ us\ vs.\ xs = us\ @\ vs \wedge length\ us = length\ ys \wedge length\ vs = length\ zs \wedge$
 $list\text{-}all2\ P\ us\ ys \wedge list\text{-}all2\ P\ vs\ zs)\ (\text{is } ?lhs = ?rhs)$
 $\langle proof \rangle$

lemma *list-all2-append*:

$length\ xs = length\ ys \implies$
 $list\text{-}all2\ P\ (xs@us)\ (ys@vs) = (list\text{-}all2\ P\ xs\ ys \wedge list\text{-}all2\ P\ us\ vs)$
 $\langle proof \rangle$

lemma *list-all2-appendI* [*intro?*, *trans*]:

$\llbracket list\text{-}all2\ P\ a\ b;\ list\text{-}all2\ P\ c\ d \rrbracket \implies list\text{-}all2\ P\ (a@c)\ (b@d)$
 $\langle proof \rangle$

lemma *list-all2-conv-all-nth*:

$list\text{-}all2\ P\ xs\ ys =$
 $(length\ xs = length\ ys \wedge (\forall\ i < length\ xs.\ P\ (xs!i)\ (ys!i)))$
 $\langle proof \rangle$

lemma *list-all2-trans*:

assumes *tr*: $!!a\ b\ c.\ P1\ a\ b \implies P2\ b\ c \implies P3\ a\ c$
shows $!!bs\ cs.\ list\text{-}all2\ P1\ as\ bs \implies list\text{-}all2\ P2\ bs\ cs \implies list\text{-}all2\ P3\ as\ cs$
 $(\text{is } !!bs\ cs.\ PROP\ ?Q\ as\ bs\ cs)$
 $\langle proof \rangle$

lemma *list-all2-all-nthI* [*intro?*]:

$length\ a = length\ b \implies (\bigwedge n.\ n < length\ a \implies P\ (a!n)\ (b!n)) \implies list\text{-}all2\ P\ a\ b$
 $\langle proof \rangle$

lemma *list-all2I*:

$\forall x \in set\ (zip\ a\ b).\ case\text{-}prod\ P\ x \implies length\ a = length\ b \implies list\text{-}all2\ P\ a\ b$
 $\langle proof \rangle$

lemma *list-all2-nthD*:

$\llbracket list\text{-}all2\ P\ xs\ ys;\ p < size\ xs \rrbracket \implies P\ (xs!p)\ (ys!p)$
 $\langle proof \rangle$

lemma *list-all2-nthD2*:

$\llbracket list\text{-}all2\ P\ xs\ ys;\ p < size\ ys \rrbracket \implies P\ (xs!p)\ (ys!p)$
 $\langle proof \rangle$

lemma *list-all2-map1*:

$list\text{-}all2\ P\ (map\ f\ as)\ bs = list\text{-}all2\ (\lambda x\ y.\ P\ (f\ x)\ y)\ as\ bs$
 $\langle proof \rangle$

lemma *list-all2-map2*:

$list\text{-}all2\ P\ as\ (map\ f\ bs) = list\text{-}all2\ (\lambda x\ y.\ P\ x\ (f\ y))\ as\ bs$
 $\langle proof \rangle$

lemma *list-all2-refl* [intro?]:

$$(\bigwedge x. P\ x\ x) \implies \text{list-all2}\ P\ xs\ xs$$

⟨proof⟩

lemma *list-all2-update-cong*:

$$\llbracket \text{list-all2}\ P\ xs\ ys;\ P\ x\ y \rrbracket \implies \text{list-all2}\ P\ (xs[i:=x])\ (ys[i:=y])$$

⟨proof⟩

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⟩

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⟩

lemma *list-all2-mono* [intro?]:

$$\text{list-all2}\ P\ xs\ ys \implies (\bigwedge xs\ ys. P\ xs\ ys \implies Q\ xs\ ys) \implies \text{list-all2}\ Q\ xs\ ys$$

⟨proof⟩

lemma *list-all2-eq*:

$$xs = ys \longleftrightarrow \text{list-all2}\ (=)\ xs\ ys$$

⟨proof⟩

lemma *list-eq-iff-zip-eq*:

$$xs = ys \longleftrightarrow \text{length}\ xs = \text{length}\ ys \wedge (\forall (x,y) \in \text{set}\ (\text{zip}\ xs\ ys). x = y)$$

⟨proof⟩

lemma *list-all2-same*: $\text{list-all2}\ P\ xs\ xs \longleftrightarrow (\forall x \in \text{set}\ xs. P\ x\ x)$

⟨proof⟩

lemma *zip-assoc*:

$$\text{zip}\ xs\ (\text{zip}\ ys\ zs) = \text{map}\ (\lambda((x,y),z). (x,y,z))\ (\text{zip}\ (\text{zip}\ xs\ ys)\ zs)$$

⟨proof⟩

lemma *zip-commute*: $\text{zip}\ xs\ ys = \text{map}\ (\lambda(x,y). (y,x))\ (\text{zip}\ ys\ xs)$

⟨proof⟩

lemma *zip-left-commute*:

$$\text{zip}\ xs\ (\text{zip}\ ys\ zs) = \text{map}\ (\lambda(y,(x,z)). (x,y,z))\ (\text{zip}\ ys\ (\text{zip}\ xs\ zs))$$

⟨proof⟩

lemma *zip-replicate2*: $\text{zip}\ xs\ (\text{replicate}\ n\ y) = \text{map}\ (\lambda x. (x,y))\ (\text{take}\ n\ xs)$

⟨proof⟩

66.1.18 *List.product* and *product-lists*

lemma *product-concat-map*:

List.product xs ys = concat (map (λx. map (λy. (x,y)) ys) xs)
 ⟨proof⟩

lemma *set-product[simp]*: *set (List.product xs ys) = set xs × set ys*
 ⟨proof⟩

lemma *length-product [simp]*:
*length (List.product xs ys) = length xs * length ys*
 ⟨proof⟩

lemma *product-nth*:
assumes *n < length xs * length ys*
shows *List.product xs ys ! n = (xs ! (n div length ys), ys ! (n mod length ys))*
 ⟨proof⟩

lemma *in-set-product-lists-length*:
xs ∈ set (product-lists xss) ⇒ length xs = length xss
 ⟨proof⟩

lemma *product-lists-set*:
set (product-lists xss) = {xs. list-all2 (λx ys. x ∈ set ys) xs xss} (is ?L = Collect ?R)
 ⟨proof⟩

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)
 ⟨proof⟩

lemma *fold-remove1-split*:
 $\llbracket \bigwedge x y. x \in \text{set } xs \implies y \in \text{set } xs \implies f x \circ f y = f y \circ f x; \\ x \in \text{set } xs \rrbracket$
 $\implies \text{fold } f \text{ xs} = \text{fold } f (\text{remove1 } x \text{ xs}) \circ f x$
 ⟨proof⟩

lemma *fold-cong [fundef-cong]*:
a = b ⇒ xs = ys ⇒ (λx. x ∈ set xs ⇒ f x = g x)
 $\implies \text{fold } f \text{ xs } a = \text{fold } g \text{ ys } b$
 ⟨proof⟩

lemma *fold-id*: $(\bigwedge x. x \in \text{set } xs \implies f x = \text{id}) \implies \text{fold } f \text{ xs} = \text{id}$
 ⟨proof⟩

lemma *fold-commute*:
 $(\bigwedge x. x \in \text{set } xs \implies h \circ g x = f x \circ h) \implies h \circ \text{fold } g \text{ xs} = \text{fold } f \text{ xs} \circ h$
 ⟨proof⟩

lemma *fold-commute-apply*:

assumes $\bigwedge x. x \in \text{set } xs \implies h \circ g \ x = f \ x \circ h$

shows $h \ (\text{fold } g \ xs \ s) = \text{fold } f \ xs \ (h \ s)$

$\langle \text{proof} \rangle$

lemma *fold-invariant*:

$\llbracket \bigwedge x. x \in \text{set } xs \implies Q \ x; \ P \ s; \ \bigwedge x \ s. Q \ x \implies P \ s \implies P \ (f \ x \ s) \rrbracket$

$\implies P \ (\text{fold } f \ xs \ s)$

$\langle \text{proof} \rangle$

lemma *fold-append [simp]*: $\text{fold } f \ (xs \ @ \ ys) = \text{fold } f \ ys \circ \text{fold } f \ xs$

$\langle \text{proof} \rangle$

lemma *fold-map [code-unfold]*: $\text{fold } g \ (\text{map } f \ xs) = \text{fold } (g \circ f) \ xs$

$\langle \text{proof} \rangle$

lemma *fold-filter*:

$\text{fold } f \ (\text{filter } P \ xs) = \text{fold } (\lambda x. \text{if } P \ x \text{ then } f \ x \text{ else id}) \ xs$

$\langle \text{proof} \rangle$

lemma *fold-rev*:

$(\bigwedge x \ y. x \in \text{set } xs \implies y \in \text{set } xs \implies f \ y \circ f \ x = f \ x \circ f \ y)$

$\implies \text{fold } f \ (\text{rev } xs) = \text{fold } f \ xs$

$\langle \text{proof} \rangle$

lemma *fold-Cons-rev*: $\text{fold } \text{Cons} \ xs = \text{append} \ (\text{rev } xs)$

$\langle \text{proof} \rangle$

lemma *rev-conv-fold [code]*: $\text{rev } xs = \text{fold } \text{Cons} \ xs \ []$

$\langle \text{proof} \rangle$

lemma *fold-append-concat-rev*: $\text{fold } \text{append} \ xss = \text{append} \ (\text{concat} \ (\text{rev } xss))$

$\langle \text{proof} \rangle$

lemma *fold-inject*:

assumes

$\bigwedge w \ x \ y \ z. f \ w \ x = f \ y \ z \longleftrightarrow w = y \wedge x = z$ **and**

$\bigwedge x \ y. f \ x \ y \neq a$ **and**

$\bigwedge x \ y. f \ x \ y \neq b$

shows $\text{fold } f \ xs \ a = \text{fold } f \ ys \ b \longleftrightarrow xs = ys \wedge a = b$

$\langle \text{proof} \rangle$

Finite-Set.fold and *fold*

lemma (in *comp-fun-commute-on*) *fold-set-fold-remdups*:

assumes $\text{set } xs \subseteq S$

shows $\text{Finite-Set.fold } f \ y \ (\text{set } xs) = \text{fold } f \ (\text{remdups } xs) \ y$

$\langle \text{proof} \rangle$

lemma (in *comp-fun-idem-on*) *fold-set-fold*:
 assumes $\text{set } xs \subseteq S$
 shows $\text{Finite-Set.fold } f \ y \ (\text{set } xs) = \text{fold } f \ xs \ y$
 $\langle \text{proof} \rangle$

lemma *union-set-fold* [code]: $\text{set } xs \cup A = \text{fold } \text{Set.insert } xs \ A$
 $\langle \text{proof} \rangle$

lemma *union-coset-filter* [code]:
 $\text{List.coset } xs \cup A = \text{List.coset } (\text{List.filter } (\lambda x. x \notin A) \ xs)$
 $\langle \text{proof} \rangle$

lemma *minus-set-fold* [code]: $A - \text{set } xs = \text{fold } \text{Set.remove } xs \ A$
 $\langle \text{proof} \rangle$

lemma *minus-coset-filter* [code]:
 $A - \text{List.coset } xs = \text{set } (\text{List.filter } (\lambda x. x \in A) \ xs)$
 $\langle \text{proof} \rangle$

lemma *inter-set-filter* [code]:
 $A \cap \text{set } xs = \text{set } (\text{List.filter } (\lambda x. x \in A) \ xs)$
 $\langle \text{proof} \rangle$

lemma *inter-coset-fold* [code]:
 $A \cap \text{List.coset } xs = \text{fold } \text{Set.remove } xs \ A$
 $\langle \text{proof} \rangle$

definition *abort-empty-set* :: $\langle 'a \ \text{set} \Rightarrow 'a \rangle \Rightarrow 'a$
 where [simp]: $\langle \text{abort-empty-set } F = F \ \{\} \rangle$

declare [[code abort: abort-empty-set]]

lemma (in *semilattice-set*) *set-empty-abort* [code]:
 $\langle F \ (\text{set } []) = \text{abort-empty-set } F \rangle$
 $\langle \text{proof} \rangle$

lemma (in *semilattice-set*) *set-eq-fold* [code]:
 $\langle F \ (\text{set } (x \# xs)) = \text{fold } f \ xs \ x \rangle$
 $\langle \text{proof} \rangle$

lemma (in *complete-lattice*) *Inf-set-fold*:
 $\text{Inf } (\text{set } xs) = \text{fold } \text{inf } xs \ \text{top}$
 $\langle \text{proof} \rangle$

declare *Inf-set-fold* [where 'a = 'a set, code]

lemma (in *complete-lattice*) *Sup-set-fold*:
 $\text{Sup } (\text{set } xs) = \text{fold } \text{sup } xs \ \text{bot}$
 $\langle \text{proof} \rangle$

declare *Sup-set-fold* [**where** $'a = 'a \text{ set}, \text{code}$]

lemma (**in** *complete-lattice*) *INF-set-fold*:
 $\sqcap (f \text{ ' set } xs) = \text{fold } (\text{inf} \circ f) \text{ xs top}$
 $\langle \text{proof} \rangle$

lemma (**in** *complete-lattice*) *SUP-set-fold*:
 $\sqcup (f \text{ ' set } xs) = \text{fold } (\text{sup} \circ f) \text{ xs bot}$
 $\langle \text{proof} \rangle$

66.1.20 Fold variants: *foldr* and *foldl*

Correspondence

lemma *foldr-conv-fold* [*code-abbrev*]: $\text{foldr } f \text{ xs} = \text{fold } f \text{ (rev xs)}$
 $\langle \text{proof} \rangle$

lemma *foldl-conv-fold*: $\text{foldl } f \text{ s xs} = \text{fold } (\lambda x \text{ s. } f \text{ s } x) \text{ xs s}$
 $\langle \text{proof} \rangle$

lemma *foldr-conv-foldl*: — The “Third Duality Theorem” in Bird & Wadler:
 $\text{foldr } f \text{ xs } a = \text{foldl } (\lambda x \text{ y. } f \text{ y } x) \text{ a (rev xs)}$
 $\langle \text{proof} \rangle$

lemma *foldl-conv-foldr*:
 $\text{foldl } f \text{ a xs} = \text{foldr } (\lambda x \text{ y. } f \text{ y } x) \text{ (rev xs) a}$
 $\langle \text{proof} \rangle$

lemma *foldr-fold*:
 $(\bigwedge x \text{ y. } x \in \text{set xs} \implies y \in \text{set xs} \implies f \text{ y } \circ f \text{ x} = f \text{ x } \circ f \text{ y})$
 $\implies \text{foldr } f \text{ xs} = \text{fold } f \text{ xs}$
 $\langle \text{proof} \rangle$

lemma *foldr-cong* [*fundef-cong*]:
 $a = b \implies l = k \implies (\bigwedge a \text{ x. } x \in \text{set l} \implies f \text{ x } a = g \text{ x } a) \implies \text{foldr } f \text{ l } a = \text{foldr } g \text{ k } b$
 $\langle \text{proof} \rangle$

lemma *foldl-cong* [*fundef-cong*]:
 $a = b \implies l = k \implies (\bigwedge a \text{ x. } x \in \text{set l} \implies f \text{ a } x = g \text{ a } x) \implies \text{foldl } f \text{ a } l = \text{foldl } g \text{ b } k$
 $\langle \text{proof} \rangle$

lemma *foldr-append* [*simp*]: $\text{foldr } f \text{ (xs @ ys) } a = \text{foldr } f \text{ xs } (\text{foldr } f \text{ ys } a)$
 $\langle \text{proof} \rangle$

lemma *foldl-append* [*simp*]: $\text{foldl } f \text{ a (xs @ ys)} = \text{foldl } f \text{ (foldl } f \text{ a xs) ys}$
 $\langle \text{proof} \rangle$

lemma *foldr-map* [code-unfold]: *foldr* *g* (*map* *f* *xs*) *a* = *foldr* (*g* ∘ *f*) *xs* *a*
 ⟨proof⟩

lemma *foldr-filter*:
foldr *f* (*filter* *P* *xs*) = *foldr* ($\lambda x. \text{if } P\ x \text{ then } f\ x \text{ else id}$) *xs*
 ⟨proof⟩

lemma *foldl-map* [code-unfold]:
foldl *g* *a* (*map* *f* *xs*) = *foldl* ($\lambda a\ x. g\ a\ (f\ x)$) *a* *xs*
 ⟨proof⟩

lemma *concat-conv-foldr* [code]:
concat *xss* = *foldr* *append* *xss* []
 ⟨proof⟩

lemma *foldl-inject*:
 assumes
 $\bigwedge w\ x\ y\ z. f\ w\ x = f\ y\ z \longleftrightarrow w = y \wedge x = z$ and
 $\bigwedge x\ y. f\ x\ y \neq a$ and
 $\bigwedge x\ y. f\ x\ y \neq b$
 shows *foldl* *f* *a* *xs* = *foldl* *f* *b* *ys* $\longleftrightarrow a = b \wedge xs = ys$
 ⟨proof⟩

lemma *foldr-inject*:
 assumes
 $\bigwedge w\ x\ y\ z. f\ w\ x = f\ y\ z \longleftrightarrow w = y \wedge x = z$ and
 $\bigwedge x\ y. f\ x\ y \neq a$ and
 $\bigwedge x\ y. f\ x\ y \neq b$
 shows *foldr* *f* *xs* *a* = *foldr* *f* *ys* *b* $\longleftrightarrow xs = ys \wedge a = b$
 ⟨proof⟩

66.1.21 *upt*

lemma *upt-rec*[code]: $[i..<j] = (\text{if } i < j \text{ then } i\#\text{[Suc } i..<j] \text{ else } [])$
 — *simp* does not terminate!
 ⟨proof⟩

lemmas *upt-rec-numeral*[*simp*] = *upt-rec*[of numeral *m* numeral *n*] **for** *m* *n*

lemma *upt-conv-Nil* [*simp*]: $j \leq i \implies [i..<j] = []$
 ⟨proof⟩

lemma *upt-eq-Nil-conv*[*simp*]: $([i..<j] = []) = (j = 0 \vee j \leq i)$
 ⟨proof⟩

lemma *upt-eq-Cons-conv*:
 $([i..<j] = x\#\text{xs}) = (i < j \wedge i = x \wedge [i+1..<j] = \text{xs})$
 ⟨proof⟩

lemma *upt-Suc-append*: $i \leq j \implies [i..<(Suc\ j)] = [i..<j]@[j]$
 — Only needed if *upt-Suc* is deleted from the simpset.
 $\langle proof \rangle$

lemma *upt-conv-Cons*: $i < j \implies [i..<j] = i \# [Suc\ i..<j]$
 $\langle proof \rangle$

lemma *upt-conv-Cons-Cons*: — no precondition
 $m \# n \# ns = [m..<q] \longleftrightarrow n \# ns = [Suc\ m..<q]$
 $\langle proof \rangle$

lemma *upt-add-eq-append*: $i \leq j \implies [i..<j+k] = [i..<j]@[j..<j+k]$
 — LOOPS as a simprule, since $j \leq j$.
 $\langle proof \rangle$

lemma *length-upt* [simp]: $length\ [i..<j] = j - i$
 $\langle proof \rangle$

lemma *nth-upt* [simp]: $i + k < j \implies [i..<j] ! k = i + k$
 $\langle proof \rangle$

lemma *hd-upt*[simp]: $i < j \implies hd[i..<j] = i$
 $\langle proof \rangle$

lemma *tl-upt* [simp]: $tl\ [m..<n] = [Suc\ m..<n]$
 $\langle proof \rangle$

lemma *last-upt*[simp]: $i < j \implies last[i..<j] = j - 1$
 $\langle proof \rangle$

lemma *take-upt* [simp]: $i+m \leq n \implies take\ m\ [i..<n] = [i..<i+m]$
 $\langle proof \rangle$

lemma *drop-upt*[simp]: $drop\ m\ [i..<j] = [i+m..<j]$
 $\langle proof \rangle$

lemma *map-Suc-upt*: $map\ Suc\ [m..<n] = [Suc\ m..<Suc\ n]$
 $\langle proof \rangle$

lemma *map-add-upt*: $map\ (\lambda i. i + n)\ [0..<m] = [n..<m + n]$
 $\langle proof \rangle$

lemma *nth-map-upt*: $i < n-m \implies (map\ f\ [m..<n]) ! i = f(m+i)$
 $\langle proof \rangle$

lemma *map-decr-upt*: $map\ (\lambda n. n - Suc\ 0)\ [Suc\ m..<Suc\ n] = [m..<n]$
 $\langle proof \rangle$

lemma *map-upt-Suc*: $map\ f\ [0..<Suc\ n] = f\ 0 \# map\ (\lambda i. f\ (Suc\ i))\ [0..<n]$

$\langle \text{proof} \rangle$

lemma *nth-take-lemma*:

$k \leq \text{length } xs \implies k \leq \text{length } ys \implies$
 $(\bigwedge i. i < k \longrightarrow xs!i = ys!i) \implies \text{take } k \text{ } xs = \text{take } k \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *nth-equalityI*:

$\llbracket \text{length } xs = \text{length } ys; \bigwedge i. i < \text{length } xs \implies xs!i = ys!i \rrbracket \implies xs = ys$
 $\langle \text{proof} \rangle$

lemma *map-nth*:

$\text{map } (\lambda i. xs ! i) [0..<\text{length } xs] = xs$
 $\langle \text{proof} \rangle$

lemma *list-all2-antisym*:

$\llbracket (\bigwedge x y. \llbracket P \text{ } x \text{ } y; Q \text{ } y \text{ } x \rrbracket \implies x = y); \text{list-all2 } P \text{ } xs \text{ } ys; \text{list-all2 } Q \text{ } ys \text{ } xs \rrbracket$
 $\implies xs = ys$
 $\langle \text{proof} \rangle$

lemma *take-equalityI*: $(\forall i. \text{take } i \text{ } xs = \text{take } i \text{ } ys) \implies xs = ys$

— The famous take-lemma.

$\langle \text{proof} \rangle$

lemma *take-Cons'*:

$\text{take } n \text{ } (x \# xs) = (\text{if } n = 0 \text{ then } [] \text{ else } x \# \text{take } (n - 1) \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *drop-Cons'*:

$\text{drop } n \text{ } (x \# xs) = (\text{if } n = 0 \text{ then } x \# xs \text{ else } \text{drop } (n - 1) \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *nth-Cons'*: $(x \# xs)!n = (\text{if } n = 0 \text{ then } x \text{ else } xs!(n - 1))$

$\langle \text{proof} \rangle$

lemma *take-Cons-numeral* [simp]:

$\text{take } (\text{numeral } v) \text{ } (x \# xs) = x \# \text{take } (\text{numeral } v - 1) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *drop-Cons-numeral* [simp]:

$\text{drop } (\text{numeral } v) \text{ } (x \# xs) = \text{drop } (\text{numeral } v - 1) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *nth-Cons-numeral* [simp]:

$(x \# xs) ! \text{numeral } v = xs ! (\text{numeral } v - 1)$
 $\langle \text{proof} \rangle$

lemma *map-upt-eqI*:

$\langle \text{map } f [m..<n] = xs \rangle \text{ if } \langle \text{length } xs = n - m \rangle$

$\langle \bigwedge i. i < \text{length } xs \implies xs ! i = f (m + i) \rangle$
 $\langle \text{proof} \rangle$

66.1.22 upto: interval-list on int

function upto :: int \Rightarrow int \Rightarrow int list ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix list interval} \rangle \rangle [-..-] \rangle$) **where**

upto i j = (if i \leq j then i # [i+1..j] else [])
 $\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

declare upto.simps[simp del]

lemmas upto-rec-numeral [simp] =
 upto.simps[of numeral m numeral n]
 upto.simps[of numeral m - numeral n]
 upto.simps[of - numeral m numeral n]
 upto.simps[of - numeral m - numeral n] **for** m n

lemma upto-empty[simp]: j < i \implies [i..j] = []
 $\langle \text{proof} \rangle$

lemma upto-single[simp]: [i..i] = [i]
 $\langle \text{proof} \rangle$

lemma upto-Nil[simp]: [i..j] = [] \longleftrightarrow j < i
 $\langle \text{proof} \rangle$

lemmas upto-Nil2[simp] = upto-Nil[THEN eq-iff-swap]

lemma upto-rec1: i \leq j \implies [i..j] = i # [i+1..j]
 $\langle \text{proof} \rangle$

lemma upto-rec2: i \leq j \implies [i..j] = [i..j - 1] @ [j]
 $\langle \text{proof} \rangle$

lemma length-upto[simp]: length [i..j] = nat(j - i + 1)
 $\langle \text{proof} \rangle$

lemma set-upto[simp]: set [i..j] = {i..j}
 $\langle \text{proof} \rangle$

lemma nth-upto[simp]: i + int k \leq j \implies [i..j] ! k = i + int k
 $\langle \text{proof} \rangle$

lemma upto-split1:
 i \leq j \implies j \leq k \implies [i..k] = [i..j-1] @ [j..k]
 $\langle \text{proof} \rangle$

lemma *upto-split2*:

$i \leq j \implies j \leq k \implies [i..k] = [i..j] @ [j+1..k]$
 $\langle \text{proof} \rangle$

lemma *upto-split3*: $\llbracket i \leq j; j \leq k \rrbracket \implies [i..k] = [i..j-1] @ j \# [j+1..k]$
 $\langle \text{proof} \rangle$

Tail recursive version for code generation:

definition *upto-aux* :: $\text{int} \Rightarrow \text{int} \Rightarrow \text{int list} \Rightarrow \text{int list}$ **where**
upto-aux i j $js = [i..j] @ js$

lemma *upto-aux-rec* [*code*]:

upto-aux i j $js = (\text{if } j < i \text{ then } js \text{ else } \text{upto-aux } i \ (j - 1) \ (j \# js))$
 $\langle \text{proof} \rangle$

lemma *upto-code*[*code*]: $[i..j] = \text{upto-aux } i \ j \ []$
 $\langle \text{proof} \rangle$

66.1.23 successively

lemma *successively-Cons*:

successively $P \ (x \# xs) \longleftrightarrow xs = [] \vee P \ x \ (\text{hd } xs) \wedge \text{successively } P \ xs$
 $\langle \text{proof} \rangle$

lemma *successively-cong* [*cong*]:

assumes $\bigwedge x \ y. x \in \text{set } xs \implies y \in \text{set } xs \implies P \ x \ y \longleftrightarrow Q \ x \ y \ xs = ys$
shows $\text{successively } P \ xs \longleftrightarrow \text{successively } Q \ ys$
 $\langle \text{proof} \rangle$

lemma *successively-append-iff*:

successively $P \ (xs @ ys) \longleftrightarrow$
 $\text{successively } P \ xs \wedge \text{successively } P \ ys \wedge$
 $(xs = [] \vee ys = [] \vee P \ (\text{last } xs) \ (\text{hd } ys))$
 $\langle \text{proof} \rangle$

lemma *successively-if-sorted-wrt*: $\text{sorted-wrt } P \ xs \implies \text{successively } P \ xs$
 $\langle \text{proof} \rangle$

lemma *successively-iff-sorted-wrt-strong*:

assumes $\bigwedge x \ y \ z. x \in \text{set } xs \implies y \in \text{set } xs \implies z \in \text{set } xs \implies$
 $P \ x \ y \implies P \ y \ z \implies P \ x \ z$
shows $\text{successively } P \ xs \longleftrightarrow \text{sorted-wrt } P \ xs$
 $\langle \text{proof} \rangle$

lemma *successively-conv-sorted-wrt*:

assumes *transp* P

shows $\text{successively } P \text{ } xs \longleftrightarrow \text{sorted-wrt } P \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *successively-rev* [simp]: $\text{successively } P \text{ } (\text{rev } xs) \longleftrightarrow \text{successively } (\lambda x y. P \text{ } y \text{ } x) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *successively-map*: $\text{successively } P \text{ } (\text{map } f \text{ } xs) \longleftrightarrow \text{successively } (\lambda x y. P \text{ } (f \text{ } x) \text{ } (f \text{ } y)) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *successively-mono*:
assumes $\text{successively } P \text{ } xs$
assumes $\bigwedge x y. x \in \text{set } xs \implies y \in \text{set } xs \implies P \text{ } x \text{ } y \implies Q \text{ } x \text{ } y$
shows $\text{successively } Q \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *successively-altdef*:
 $\text{successively} = (\lambda P. \text{rec-list True } (\lambda x \text{ } xs \text{ } b. \text{case } xs \text{ of } [] \Rightarrow \text{True} \mid y \# - \Rightarrow P \text{ } x \text{ } y \wedge b))$
 $\langle \text{proof} \rangle$

66.1.24 *distinct and remdups and remdups-adj*

lemma *distinct-tl*: $\text{distinct } xs \implies \text{distinct } (\text{tl } xs)$
 $\langle \text{proof} \rangle$

lemma *distinct-append* [simp]:
 $\text{distinct } (xs @ ys) = (\text{distinct } xs \wedge \text{distinct } ys \wedge \text{set } xs \cap \text{set } ys = \{\})$
 $\langle \text{proof} \rangle$

lemma *distinct-rev*[simp]: $\text{distinct}(\text{rev } xs) = \text{distinct } xs$
 $\langle \text{proof} \rangle$

lemma *set-remdups* [simp]: $\text{set } (\text{remdups } xs) = \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *distinct-remdups* [iff]: $\text{distinct } (\text{remdups } xs)$
 $\langle \text{proof} \rangle$

lemma *distinct-remdups-id*: $\text{distinct } xs \implies \text{remdups } xs = xs$
 $\langle \text{proof} \rangle$

lemma *remdups-id-iff-distinct* [simp]: $\text{remdups } xs = xs \longleftrightarrow \text{distinct } xs$
 $\langle \text{proof} \rangle$

lemma *finite-distinct-list*: $\text{finite } A \implies \exists xs. \text{set } xs = A \wedge \text{distinct } xs$
 $\langle \text{proof} \rangle$

lemma *remdups-eq-nil-iff* [simp]: $(\text{remdups } x = []) = (x = [])$
 ⟨proof⟩

lemmas *remdups-eq-nil-right-iff* [simp] = *remdups-eq-nil-iff* [THEN *eq-iff-swap*]

lemma *length-remdups-leq* [iff]: $\text{length}(\text{remdups } xs) \leq \text{length } xs$
 ⟨proof⟩

lemma *length-remdups-eq* [iff]:
 $(\text{length } (\text{remdups } xs) = \text{length } xs) = (\text{remdups } xs = xs)$
 ⟨proof⟩

lemma *remdups-filter*: $\text{remdups}(\text{filter } P \text{ } xs) = \text{filter } P (\text{remdups } xs)$
 ⟨proof⟩

lemma *distinct-map*:
 $\text{distinct}(\text{map } f \text{ } xs) = (\text{distinct } xs \wedge \text{inj-on } f (\text{set } xs))$
 ⟨proof⟩

lemma *distinct-map-filter*:
 $\text{distinct } (\text{map } f \text{ } xs) \implies \text{distinct } (\text{map } f (\text{filter } P \text{ } xs))$
 ⟨proof⟩

lemma *distinct-filter* [simp]: $\text{distinct } xs \implies \text{distinct } (\text{filter } P \text{ } xs)$
 ⟨proof⟩

lemma *distinct-upt* [simp]: $\text{distinct}[i..<j]$
 ⟨proof⟩

lemma *distinct-upto* [simp]: $\text{distinct}[i..j]$
 ⟨proof⟩

lemma *distinct-take* [simp]: $\text{distinct } xs \implies \text{distinct } (\text{take } i \text{ } xs)$
 ⟨proof⟩

lemma *distinct-drop* [simp]: $\text{distinct } xs \implies \text{distinct } (\text{drop } i \text{ } xs)$
 ⟨proof⟩

lemma *distinct-list-update*:
 assumes d : $\text{distinct } xs$ and a : $a \notin \text{set } xs - \{xs[i]\}$
 shows $\text{distinct } (xs[i:=a])$
 ⟨proof⟩

lemma *distinct-concat-rev* [simp]: $\text{distinct } (\text{concat } (\text{rev } xs)) = \text{distinct } (\text{concat } xs)$
 ⟨proof⟩

lemma *distinct-concat*:
 $\llbracket \text{distinct } xs;$
 $\bigwedge ys. ys \in \text{set } xs \implies \text{distinct } ys;$

$$\bigwedge ys\ zs. \llbracket ys \in \text{set } xs ; zs \in \text{set } xs ; ys \neq zs \rrbracket \implies \text{set } ys \cap \text{set } zs = \{\}$$

$$\rrbracket \implies \text{distinct } (\text{concat } xs)$$

$$\langle \text{proof} \rangle$$

An iff-version of $\llbracket \text{distinct } ?xs ; \bigwedge ys. ys \in \text{set } ?xs \implies \text{distinct } ys ; \bigwedge ys\ zs. \llbracket ys \in \text{set } ?xs ; zs \in \text{set } ?xs ; ys \neq zs \rrbracket \implies \text{set } ys \cap \text{set } zs = \{\} \rrbracket \implies \text{distinct } (\text{concat } ?xs)$ is available further down as *distinct-concat-iff*.

It is best to avoid the following indexed version of *distinct*, but sometimes it is useful.

lemma *distinct-conv-nth*: $\text{distinct } xs = (\forall i < \text{size } xs. \forall j < \text{size } xs. i \neq j \longrightarrow xs!i \neq xs!j)$
 $\langle \text{proof} \rangle$

lemma *nth-eq-iff-index-eq*:

$$\llbracket \text{distinct } xs ; i < \text{length } xs ; j < \text{length } xs \rrbracket \implies (xs!i = xs!j) = (i = j)$$
 $\langle \text{proof} \rangle$

lemma *distinct-Ex1*:

$$\text{distinct } xs \implies x \in \text{set } xs \implies (\exists !i. i < \text{length } xs \wedge xs ! i = x)$$
 $\langle \text{proof} \rangle$

lemma *inj-on-nth*: $\text{distinct } xs \implies \forall i \in I. i < \text{length } xs \implies \text{inj-on } (nth\ xs)\ I$
 $\langle \text{proof} \rangle$

lemma *bij-betw-nth*:
assumes $\text{distinct } xs\ A = \{..
shows $\text{bij-betw } (!)\ xs\ A\ B$
 $\langle \text{proof} \rangle$$

lemma *set-update-distinct*: $\llbracket \text{distinct } xs ; n < \text{length } xs \rrbracket \implies$

$$\text{set}(xs[n := x]) = \text{insert } x\ (\text{set } xs - \{xs!n\})$$
 $\langle \text{proof} \rangle$

lemma *distinct-swap[simp]*: $\llbracket i < \text{size } xs ; j < \text{size } xs \rrbracket \implies$

$$\text{distinct}(xs[i := xs!j, j := xs!i]) = \text{distinct } xs$$
 $\langle \text{proof} \rangle$

lemma *set-swap[simp]*:

$$\llbracket i < \text{size } xs ; j < \text{size } xs \rrbracket \implies \text{set}(xs[i := xs!j, j := xs!i]) = \text{set } xs$$
 $\langle \text{proof} \rangle$

lemma *distinct-card*: $\text{distinct } xs \implies \text{card } (\text{set } xs) = \text{size } xs$
 $\langle \text{proof} \rangle$

lemma *card-distinct*: $\text{card } (\text{set } xs) = \text{size } xs \implies \text{distinct } xs$
 $\langle \text{proof} \rangle$

lemma *distinct-length-filter*: $\text{distinct } xs \implies \text{length } (\text{filter } P\ xs) = \text{card } (\{x. P\ x\})$

Int set xs)
<proof>

lemma *not-distinct-decomp*: $\neg \text{distinct } ws \implies \exists xs\ ys\ zs\ y. ws = xs @ [y] @ ys @ [y] @ zs$
<proof>

lemma *not-distinct-conv-prefix*:
defines $\text{dec } as\ xs\ y\ ys \equiv y \in \text{set } xs \wedge \text{distinct } xs \wedge as = xs @ y \# ys$
shows $\neg \text{distinct } as \longleftrightarrow (\exists xs\ y\ ys. \text{dec } as\ xs\ y\ ys) \text{ (is ?L = ?R)}$
<proof>

lemma *distinct-product*:
 $\text{distinct } xs \implies \text{distinct } ys \implies \text{distinct } (\text{List.product } xs\ ys)$
<proof>

lemma *distinct-product-lists*:
assumes $\forall xs \in \text{set } xss. \text{distinct } xs$
shows $\text{distinct } (\text{product-lists } xss)$
<proof>

lemma *length-remdups-concat*:
 $\text{length } (\text{remdups } (\text{concat } xss)) = \text{card } (\bigcup xs \in \text{set } xss. \text{set } xs)$
<proof>

lemma *remdups-append2*:
 $\text{remdups } (xs @ \text{remdups } ys) = \text{remdups } (xs @ ys)$
<proof>

lemma *length-remdups-card-conv*: $\text{length}(\text{remdups } xs) = \text{card}(\text{set } xs)$
<proof>

lemma *remdups-remdups*: $\text{remdups } (\text{remdups } xs) = \text{remdups } xs$
<proof>

lemma *distinct-butlast*:
assumes $\text{distinct } xs$
shows $\text{distinct } (\text{butlast } xs)$
<proof>

lemma *remdups-map-remdups*:
 $\text{remdups } (\text{map } f\ (\text{remdups } xs)) = \text{remdups } (\text{map } f\ xs)$
<proof>

lemma *distinct-zipI1*:
assumes $\text{distinct } xs$
shows $\text{distinct } (\text{zip } xs\ ys)$
<proof>

lemma *distinct-zipI2*:

assumes *distinct ys*
shows *distinct (zip xs ys)*
 ⟨proof⟩

lemma *set-take-disj-set-drop-if-distinct*:
 $\text{distinct } vs \implies i \leq j \implies \text{set } (\text{take } i \text{ } vs) \cap \text{set } (\text{drop } j \text{ } vs) = \{\}$
 ⟨proof⟩

lemma *distinct-singleton*: $\text{distinct } [x]$ ⟨proof⟩

lemma *distinct-length-2-or-more*:
 $\text{distinct } (a \# b \# xs) \longleftrightarrow (a \neq b \wedge \text{distinct } (a \# xs) \wedge \text{distinct } (b \# xs))$
 ⟨proof⟩

lemma *remdups-adj-altdef*: $(\text{remdups-adj } xs = ys) \longleftrightarrow$
 $(\exists f :: \text{nat} \Rightarrow \text{nat}. \text{mono } f \wedge f' \{0 \dots \text{size } xs\} = \{0 \dots \text{size } ys\}$
 $\wedge (\forall i < \text{size } xs. xs!i = ys!(f i))$
 $\wedge (\forall i. i + 1 < \text{size } xs \longrightarrow (xs!i = xs!(i+1) \longleftrightarrow f i = f(i+1)))) \text{ (is ?L } \longleftrightarrow$
 $(\exists f. ?p f xs ys))$
 ⟨proof⟩

lemma *hd-remdups-adj[simp]*: $\text{hd } (\text{remdups-adj } xs) = \text{hd } xs$
 ⟨proof⟩

lemma *remdups-adj-Cons*: $\text{remdups-adj } (x \# xs) =$
 $(\text{case } \text{remdups-adj } xs \text{ of } [] \Rightarrow [x] \mid y \# xs \Rightarrow \text{if } x = y \text{ then } y \# xs \text{ else } x \# y \#$
 $xs)$
 ⟨proof⟩

lemma *remdups-adj-append-two*:
 $\text{remdups-adj } (xs @ [x, y]) = \text{remdups-adj } (xs @ [x]) @ (\text{if } x = y \text{ then } [] \text{ else } [y])$
 ⟨proof⟩

lemma *remdups-adj-adjacent*:
 $\text{Suc } i < \text{length } (\text{remdups-adj } xs) \implies \text{remdups-adj } xs ! i \neq \text{remdups-adj } xs ! \text{Suc } i$
 ⟨proof⟩

lemma *remdups-adj-rev[simp]*: $\text{remdups-adj } (\text{rev } xs) = \text{rev } (\text{remdups-adj } xs)$
 ⟨proof⟩

lemma *remdups-adj-length[simp]*: $\text{length } (\text{remdups-adj } xs) \leq \text{length } xs$
 ⟨proof⟩

lemma *remdups-adj-length-ge1[simp]*: $xs \neq [] \implies \text{length } (\text{remdups-adj } xs) \geq \text{Suc } 0$
 ⟨proof⟩

lemma *remdups-adj-Nil-iff[simp]*: $\text{remdups-adj } xs = [] \longleftrightarrow xs = []$
 ⟨proof⟩

lemma *remdups-adj-set[simp]*: $\text{set } (\text{remdups-adj } xs) = \text{set } xs$
 ⟨proof⟩

lemma *last-remdups-adj [simp]*: $\text{last } (\text{remdups-adj } xs) = \text{last } xs$
 ⟨proof⟩

lemma *remdups-adj-Cons-alt[simp]*: $x \# \text{tl } (\text{remdups-adj } (x \# xs)) = \text{remdups-adj } (x \# xs)$
 ⟨proof⟩

lemma *remdups-adj-distinct*: $\text{distinct } xs \implies \text{remdups-adj } xs = xs$
 ⟨proof⟩

lemma *remdups-adj-append*:
 $\text{remdups-adj } (xs_1 @ x \# xs_2) = \text{remdups-adj } (xs_1 @ [x]) @ \text{tl } (\text{remdups-adj } (x \# xs_2))$
 ⟨proof⟩

lemma *remdups-adj-singleton*:
 $\text{remdups-adj } xs = [x] \implies xs = \text{replicate } (\text{length } xs) \ x$
 ⟨proof⟩

lemma *remdups-adj-map-injective*:
 assumes *inj f*
 shows $\text{remdups-adj } (\text{map } f \ xs) = \text{map } f \ (\text{remdups-adj } xs)$
 ⟨proof⟩

lemma *remdups-adj-replicate*:
 $\text{remdups-adj } (\text{replicate } n \ x) = (\text{if } n = 0 \text{ then } [] \text{ else } [x])$
 ⟨proof⟩

lemma *remdups-upt [simp]*: $\text{remdups } [m..<n] = [\text{remdups-adj } m..<n]$
 ⟨proof⟩

lemma *successively-remdups-adjI*:
 $\text{successively } P \ xs \implies \text{successively } P \ (\text{remdups-adj } xs)$
 ⟨proof⟩

lemma *successively-remdups-adj-iff*:
 $(\bigwedge x. x \in \text{set } xs \implies P \ x \ x) \implies$
 $\text{successively } P \ (\text{remdups-adj } xs) \longleftrightarrow \text{successively } P \ xs$
 ⟨proof⟩

lemma *successively-conv-nth*:
 $\text{successively } P \ xs \longleftrightarrow (\forall i. \text{Suc } i < \text{length } xs \longrightarrow P \ (xs \ ! \ i) \ (xs \ ! \ \text{Suc } i))$
 ⟨proof⟩

lemma *successively-nth*: $\text{successively } P \text{ } xs \implies \text{Suc } i < \text{length } xs \implies P \text{ } (xs ! i) \text{ } (xs ! \text{Suc } i)$
 <proof>

lemma *distinct-adj-conv-nth*:
 $\text{distinct-adj } xs \longleftrightarrow (\forall i. \text{Suc } i < \text{length } xs \longrightarrow xs ! i \neq xs ! \text{Suc } i)$
 <proof>

lemma *distinct-adj-nth*: $\text{distinct-adj } xs \implies \text{Suc } i < \text{length } xs \implies xs ! i \neq xs ! \text{Suc } i$
 <proof>

lemma *remdups-adj-Cons'*:
 $\text{remdups-adj } (x \# xs) = x \# \text{remdups-adj } (\text{dropWhile } (\lambda y. y = x) \text{ } xs)$
 <proof>

lemma *remdups-adj-singleton-iff*:
 $\text{length } (\text{remdups-adj } xs) = \text{Suc } 0 \longleftrightarrow xs \neq [] \wedge xs = \text{replicate } (\text{length } xs) \text{ } (\text{hd } xs)$
 <proof>

lemma *tl-remdups-adj*:
 $ys \neq [] \implies \text{tl } (\text{remdups-adj } ys) = \text{remdups-adj } (\text{dropWhile } (\lambda x. x = \text{hd } ys) \text{ } (\text{tl } ys))$
 <proof>

lemma *remdups-adj-append-dropWhile*:
 $\text{remdups-adj } (xs @ y \# ys) = \text{remdups-adj } (xs @ [y]) @ \text{remdups-adj } (\text{dropWhile } (\lambda x. x = y) \text{ } ys)$
 <proof>

lemma *remdups-adj-append'*:
assumes $xs = [] \vee ys = [] \vee \text{last } xs \neq \text{hd } ys$
shows $\text{remdups-adj } (xs @ ys) = \text{remdups-adj } xs @ \text{remdups-adj } ys$
 <proof>

lemma *remdups-adj-append''*: $xs \neq [] \implies \text{remdups-adj } (xs @ ys) = \text{remdups-adj } xs @ \text{remdups-adj } (\text{dropWhile } (\lambda y. y = \text{last } xs) \text{ } ys)$
 <proof>

lemma *remdups-filter-last*:
 $\text{last } [x \leftarrow \text{remdups } xs. P \text{ } x] = \text{last } [x \leftarrow xs. P \text{ } x]$
 <proof>

lemma *remdups-append*:
 $\text{set } xs \subseteq \text{set } ys \implies \text{remdups } (xs @ ys) = \text{remdups } ys$
 <proof>

lemma *remdups-concat*:

remdups (concat (remdups xs)) = remdups (concat xs)

<proof>

66.2 *distinct-adj*

lemma *distinct-adj-Nil* [simp]: *distinct-adj []*

and *distinct-adj-singleton* [simp]: *distinct-adj [x]*

and *distinct-adj-Cons-Cons* [simp]: *distinct-adj (x # y # xs) \longleftrightarrow x \neq y \wedge distinct-adj (y # xs)*

<proof>

lemma *distinct-adj-Cons*: *distinct-adj (x # xs) \longleftrightarrow xs = [] \vee x \neq hd xs \wedge distinct-adj xs*

<proof>

lemma *distinct-adj-ConsD*: *distinct-adj (x # xs) \implies distinct-adj xs*

<proof>

lemma *distinct-adj-remdups-adj*[simp]: *distinct-adj (remdups-adj xs)*

<proof>

lemma *distinct-adj-altdef*: *distinct-adj xs \longleftrightarrow remdups-adj xs = xs*

<proof>

lemma *distinct-adj-rev* [simp]: *distinct-adj (rev xs) \longleftrightarrow distinct-adj xs*

<proof>

lemma *distinct-adj-append-iff*:

distinct-adj (xs @ ys) \longleftrightarrow

distinct-adj xs \wedge distinct-adj ys \wedge (xs = [] \vee ys = [] \vee last xs \neq hd ys)

<proof>

lemma *distinct-adj-appendD1* [dest]: *distinct-adj (xs @ ys) \implies distinct-adj xs*

and *distinct-adj-appendD2* [dest]: *distinct-adj (xs @ ys) \implies distinct-adj ys*

<proof>

lemma *distinct-adj-mapI*: *distinct-adj xs \implies inj-on f (set xs) \implies distinct-adj (map f xs)*

<proof>

lemma *distinct-adj-mapD*: *distinct-adj (map f xs) \implies distinct-adj xs*

<proof>

lemma *distinct-adj-map-iff*: *inj-on f (set xs) \implies distinct-adj (map f xs) \longleftrightarrow distinct-adj xs*

<proof>

lemma *distinct-adj-conv-length-remdups-adj*:

$distinct\text{-}adj\ xs \longleftrightarrow length\ (remdups\text{-}adj\ xs) = length\ xs$
 $\langle proof \rangle$

66.2.1 *insert*

lemma *in-set-insert* [simp]:

$x \in set\ xs \implies List.insert\ x\ xs = xs$
 $\langle proof \rangle$

lemma *not-in-set-insert* [simp]:

$x \notin set\ xs \implies List.insert\ x\ xs = x \# xs$
 $\langle proof \rangle$

lemma *insert-Nil* [simp]: $List.insert\ x\ [] = [x]$

$\langle proof \rangle$

lemma *set-insert* [simp]: $set\ (List.insert\ x\ xs) = insert\ x\ (set\ xs)$

$\langle proof \rangle$

lemma *distinct-insert* [simp]: $distinct\ (List.insert\ x\ xs) = distinct\ xs$

$\langle proof \rangle$

lemma *insert-remdups*:

$List.insert\ x\ (remdups\ xs) = remdups\ (List.insert\ x\ xs)$
 $\langle proof \rangle$

66.2.2 *List.union*

This is all one should need to know about union:

lemma *set-union*[simp]: $set\ (List.union\ xs\ ys) = set\ xs \cup set\ ys$

$\langle proof \rangle$

lemma *distinct-union*[simp]: $distinct\ (List.union\ xs\ ys) = distinct\ ys$

$\langle proof \rangle$

66.2.3 *find*

lemma *find-None-iff*: $List.find\ P\ xs = None \longleftrightarrow \neg (\exists x. x \in set\ xs \wedge P\ x)$

$\langle proof \rangle$

lemmas *find-None-iff2* = *find-None-iff*[*THEN* *eq-iff-swap*]

lemma *find-Some-iff*:

$List.find\ P\ xs = Some\ x \longleftrightarrow$
 $(\exists i < length\ xs. P\ (xs[i]) \wedge x = xs[i] \wedge (\forall j < i. \neg P\ (xs[j])))$
 $\langle proof \rangle$

lemmas *find-Some-iff2* = *find-Some-iff*[*THEN* *eq-iff-swap*]

lemma *find-cong[fundef-cong]*:
 assumes $xs = ys$ and $\bigwedge x. x \in \text{set } ys \implies P\ x = Q\ x$
 shows $\text{List.find } P\ xs = \text{List.find } Q\ ys$
 $\langle \text{proof} \rangle$

lemma *find-dropWhile*:
 $\text{List.find } P\ xs = (\text{case dropWhile } (\text{Not} \circ P)\ xs$
 of $[] \Rightarrow \text{None}$
 $| x \# - \Rightarrow \text{Some } x)$
 $\langle \text{proof} \rangle$

66.2.4 count-list

This library is intentionally minimal. See the remark about multisets at the point above where *count-list* is defined.

lemma *count-list-append[simp]*: $\text{count-list } (xs @ ys)\ x = \text{count-list } xs\ x + \text{count-list } ys\ x$
 $\langle \text{proof} \rangle$

lemma *count-list-0-iff*: $\text{count-list } xs\ x = 0 \iff x \notin \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *count-notin[simp]*: $x \notin \text{set } xs \implies \text{count-list } xs\ x = 0$
 $\langle \text{proof} \rangle$

lemma *count-le-length*: $\text{count-list } xs\ x \leq \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *count-list-map-ge*: $\text{count-list } xs\ x \leq \text{count-list } (\text{map } f\ xs)\ (f\ x)$
 $\langle \text{proof} \rangle$

lemma *count-list-inj-map*:
 $\llbracket \text{inj-on } f\ (\text{set } xs); x \in \text{set } xs \rrbracket \implies \text{count-list } (\text{map } f\ xs)\ (f\ x) = \text{count-list } xs\ x$
 $\langle \text{proof} \rangle$

lemma *count-list-map-conv*:
 assumes $\text{inj } f$ shows $\text{count-list } (\text{map } f\ xs)\ (f\ x) = \text{count-list } xs\ x$
 $\langle \text{proof} \rangle$

lemma *count-list-rev[simp]*: $\text{count-list } (\text{rev } xs)\ x = \text{count-list } xs\ x$
 $\langle \text{proof} \rangle$

lemma *sum-count-set*:
 $\text{set } xs \subseteq X \implies \text{finite } X \implies \text{sum } (\text{count-list } xs)\ X = \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *count-list-Suc-split-first*:
 assumes $\text{count-list } xs\ x = \text{Suc } n$
 shows $\exists \text{ pref rest. } xs = \text{pref } @\ x \# \text{ rest} \wedge x \notin \text{set } \text{pref} \wedge \text{count-list } \text{rest}\ x = n$

$\langle \text{proof} \rangle$

lemma *count-list-eq-length-filter*: $\text{count-list } xs \ y = \text{length}(\text{filter } ((=) \ y) \ xs)$
 $\langle \text{proof} \rangle$

lemma *split-list-cycles*:

$\exists \text{pref } xss. xs = \text{pref} @ \text{concat } xss \wedge x \notin \text{set pref} \wedge (\forall ys \in \text{set } xss. \exists zs. ys = x \# zs)$
 $\langle \text{proof} \rangle$

66.2.5 *List.extract*

lemma *extract-None-iff*: $\text{List.extract } P \ xs = \text{None} \longleftrightarrow \neg (\exists x \in \text{set } xs. P \ x)$
 $\langle \text{proof} \rangle$

lemma *extract-SomeE*:

$\text{List.extract } P \ xs = \text{Some } (ys, y, zs) \implies$
 $xs = ys @ y \# zs \wedge P \ y \wedge \neg (\exists y \in \text{set } ys. P \ y)$
 $\langle \text{proof} \rangle$

lemma *extract-Some-iff*:

$\text{List.extract } P \ xs = \text{Some } (ys, y, zs) \longleftrightarrow$
 $xs = ys @ y \# zs \wedge P \ y \wedge \neg (\exists y \in \text{set } ys. P \ y)$
 $\langle \text{proof} \rangle$

lemma *extract-Nil-code* [code]: $\text{List.extract } P \ [] = \text{None}$
 $\langle \text{proof} \rangle$

lemma *extract-Cons-code* [code]:

$\text{List.extract } P \ (x \# xs) = (\text{if } P \ x \text{ then } \text{Some } ([], x, xs) \text{ else}$
 $(\text{case } \text{List.extract } P \ xs \text{ of}$
 $\text{None} \Rightarrow \text{None} \mid$
 $\text{Some } (ys, y, zs) \Rightarrow \text{Some } (x \# ys, y, zs)))$
 $\langle \text{proof} \rangle$

66.2.6 *remove1*

lemma *count-list-remove1* [simp]:

$\text{count-list } (\text{remove1 } a \ xs) \ b = \text{count-list } xs \ b - (\text{if } a=b \text{ then } 1 \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *remove1-append*:

$\text{remove1 } x \ (xs @ ys) =$
 $(\text{if } x \in \text{set } xs \text{ then } \text{remove1 } x \ xs @ ys \text{ else } xs @ \text{remove1 } x \ ys)$
 $\langle \text{proof} \rangle$

lemma *remove1-commute*: $\text{remove1 } x \ (\text{remove1 } y \ zs) = \text{remove1 } y \ (\text{remove1 } x \ zs)$
 $\langle \text{proof} \rangle$

lemma *in-set-remove1* [simp]:

$a \neq b \implies a \in \text{set}(\text{remove1 } b \ xs) = (a \in \text{set } xs)$
 ⟨proof⟩

lemma *set-remove1-subset*: $\text{set}(\text{remove1 } x \ xs) \subseteq \text{set } xs$
 ⟨proof⟩

lemma *set-remove1-eq* [simp]: $\text{distinct } xs \implies \text{set}(\text{remove1 } x \ xs) = \text{set } xs - \{x\}$
 ⟨proof⟩

lemma *length-remove1*:
 $\text{length}(\text{remove1 } x \ xs) = (\text{if } x \in \text{set } xs \text{ then } \text{length } xs - 1 \text{ else } \text{length } xs)$
 ⟨proof⟩

lemma *remove1-filter-not*[simp]:
 $\neg P \ x \implies \text{remove1 } x \ (\text{filter } P \ xs) = \text{filter } P \ xs$
 ⟨proof⟩

lemma *filter-remove1*:
 $\text{filter } Q \ (\text{remove1 } x \ xs) = \text{remove1 } x \ (\text{filter } Q \ xs)$
 ⟨proof⟩

lemma *notin-set-remove1*[simp]: $x \notin \text{set } xs \implies x \notin \text{set}(\text{remove1 } y \ xs)$
 ⟨proof⟩

lemma *distinct-remove1*[simp]: $\text{distinct } xs \implies \text{distinct}(\text{remove1 } x \ xs)$
 ⟨proof⟩

lemma *remove1-remdups*:
 $\text{distinct } xs \implies \text{remove1 } x \ (\text{remdups } xs) = \text{remdups } (\text{remove1 } x \ xs)$
 ⟨proof⟩

lemma *remove1-idem*: $x \notin \text{set } xs \implies \text{remove1 } x \ xs = xs$
 ⟨proof⟩

lemma *remove1-split*:
 $a \in \text{set } xs \implies \text{remove1 } a \ xs = ys \longleftrightarrow (\exists \ ls \ rs. \ xs = ls @ a \# rs \wedge a \notin \text{set } ls \wedge$
 $ys = ls @ rs)$
 ⟨proof⟩

lemma *foldr-fold-remove1*[code-unfold]: $\text{foldr } \text{remove1} = \text{fold } \text{remove1}$
 ⟨proof⟩

66.2.7 removeAll

lemma *removeAll-filter-not-eq*:
 $\text{removeAll } x = \text{filter } (\lambda y. \ x \neq y)$
 ⟨proof⟩

lemma *removeAll-append*[simp]:

$\text{removeAll } x \ (xs \ @ \ ys) = \text{removeAll } x \ xs \ @ \ \text{removeAll } x \ ys$
 $\langle \text{proof} \rangle$

lemma *removeAll-commute*: $\text{removeAll } x \ (\text{removeAll } y \ zs) = \text{removeAll } y \ (\text{removeAll } x \ zs)$
 $\langle \text{proof} \rangle$

lemma *set-removeAll[simp]*: $\text{set}(\text{removeAll } x \ xs) = \text{set } xs - \{x\}$
 $\langle \text{proof} \rangle$

lemma *removeAll-id[simp]*: $x \notin \text{set } xs \implies \text{removeAll } x \ xs = xs$
 $\langle \text{proof} \rangle$

lemma *length-removeAll*:
 $\text{length}(\text{removeAll } x \ xs) = \text{length } xs - \text{count-list } xs \ x$
 $\langle \text{proof} \rangle$

lemma *removeAll-filter-not[simp]*:
 $\neg P \ x \implies \text{removeAll } x \ (\text{filter } P \ xs) = \text{filter } P \ xs$
 $\langle \text{proof} \rangle$

lemma *distinct-removeAll*:
 $\text{distinct } xs \implies \text{distinct } (\text{removeAll } x \ xs)$
 $\langle \text{proof} \rangle$

lemma *distinct-remove1-removeAll*:
 $\text{distinct } xs \implies \text{remove1 } x \ xs = \text{removeAll } x \ xs$
 $\langle \text{proof} \rangle$

lemma *map-removeAll-inj-on*: $\text{inj-on } f \ (\text{insert } x \ (\text{set } xs)) \implies$
 $\text{map } f \ (\text{removeAll } x \ xs) = \text{removeAll } (f \ x) \ (\text{map } f \ xs)$
 $\langle \text{proof} \rangle$

lemma *map-removeAll-inj*: $\text{inj } f \implies$
 $\text{map } f \ (\text{removeAll } x \ xs) = \text{removeAll } (f \ x) \ (\text{map } f \ xs)$
 $\langle \text{proof} \rangle$

lemma *length-removeAll-less-eq [simp]*:
 $\text{length } (\text{removeAll } x \ xs) \leq \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *length-removeAll-less [termination-simp]*:
 $x \in \text{set } xs \implies \text{length } (\text{removeAll } x \ xs) < \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *distinct-concat-iff*: $\text{distinct } (\text{concat } xs) \longleftrightarrow$
 $\text{distinct } (\text{removeAll } [] \ xs) \wedge$
 $(\forall \ ys. \ ys \in \text{set } xs \longrightarrow \text{distinct } ys) \wedge$
 $(\forall \ ys \ zs. \ ys \in \text{set } xs \wedge \ zs \in \text{set } xs \wedge \ ys \neq zs \longrightarrow \text{set } ys \cap \text{set } zs = \{\})$

<proof>

lemma *foldr-fold-removeAll*[code-unfold]: *foldr removeAll = fold removeAll*
<proof>

66.2.8 *minus-list-mset*

The difference of two lists viewed as multisets. Conceptually, the result of *minus-list-mset* is only determined up to permutation, i.e. up to the multiset of elements. Thus this function comes into its own in connection with multisets where $mset(minus-list-mset\ xs\ ys) = mset\ xs - mset\ ys$ is proved. Lemma *count-list-minus-list-mset* is the equivalent on the list level.

lemma *minus-list-mset-Nil2* [simp]: *minus-list-mset xs [] = xs*
<proof>

lemma *minus-list-mset-Cons2* [simp]: *minus-list-mset xs (y#ys) = remove1 y (minus-list-mset xs ys)*
<proof>

lemma *count-list-minus-list-mset*[simp]:
count-list (minus-list-mset xs ys) a = count-list xs a - count-list ys a
<proof>

lemma *minus-list-set-subset-minus-list-mset*: *set xs - set ys \subseteq set(minus-list-mset xs ys)*
<proof>

lemma *minus-list-mset-remove1-commute*:
minus-list-mset (remove1 x xs) ys = remove1 x (minus-list-mset xs ys)
<proof>

lemma *minus-list-mset-append* [simp]:
minus-list-mset xs (ys@zs) = minus-list-mset (minus-list-mset xs ys) zs
<proof>

lemma *minus-list-mset-Nil1* [simp]: *minus-list-mset [] xs = []*
<proof>

lemma *minus-list-mset-Cons1*: *minus-list-mset (x#xs) ys = (if $x \in set\ ys$ then *minus-list-mset xs (remove1 x ys)* else $x \# (minus-list-mset xs ys)$)*
<proof>

lemma *length-minus-list-mset*: *length(minus-list-mset xs ys) \leq length xs*
<proof>

lemma *minus-list-mset-subset*:
set (minus-list-mset xs ys) \subseteq set xs

$\langle \text{proof} \rangle$

lemma *distinct-minus-list-mset*:

assumes *distinct xs*

shows *distinct (minus-list-mset xs ys)*

$\langle \text{proof} \rangle$

lemma *set-minus-list-mset-distinct*:

assumes *distinct xs* **shows** *set (minus-list-mset xs ys) = set xs - set ys*

$\langle \text{proof} \rangle$

66.2.9 minus-list-set

The difference of two lists viewed as sets. Conceptually, the result of *minus-list-set* is only determined up to the set of elements:

lemma *set-minus-list-set[simp]*: *set(minus-list-set xs ys) = set xs - set ys*

$\langle \text{proof} \rangle$

lemma *minus-list-set-Nil2[simp]*: *minus-list-set xs [] = xs*

$\langle \text{proof} \rangle$

lemma *minus-list-set-Cons2[simp]*: *minus-list-set xs (y#ys) = removeAll y (minus-list-set xs ys)*

$\langle \text{proof} \rangle$

lemma *minus-list-set-eq-filter*: *minus-list-set xs ys = filter ($\lambda x. x \notin \text{set } ys$) xs*

$\langle \text{proof} \rangle$

lemma *minus-list-set-removeAll-commute*:

minus-list-set (removeAll x xs) ys = removeAll x (minus-list-set xs ys)

$\langle \text{proof} \rangle$

lemma *minus-list-set-Nil1 [simp]*: *minus-list-set [] xs = []*

$\langle \text{proof} \rangle$

lemma *minus-list-set-Cons1*: *minus-list-set (x#xs) ys =*

(if $x \in \text{set } ys$ then minus-list-set xs ys else $x \# (\text{minus-list-set xs ys})$)

$\langle \text{proof} \rangle$

lemma *minus-list-set-append2[simp]*:

minus-list-set xs (ys @ zs) = minus-list-set (minus-list-set xs ys) zs

$\langle \text{proof} \rangle$

lemma *length-minus-list-set*: *length(minus-list-set xs ys) \leq length xs*

$\langle \text{proof} \rangle$

lemma *distinct-minus-list-set*: *distinct xs \implies distinct (minus-list-set xs ys)*

$\langle \text{proof} \rangle$

66.2.10 *inter-list-set*

The intersection of two lists viewed as sets. Conceptually, the result of *inter-list-set* is only determined up to the set of elements:

lemma *set-inter-list-set[simp]*: $\text{set}(\text{inter-list-set } xs \ ys) = \text{set } xs \cap \text{set } ys$
 $\langle \text{proof} \rangle$

lemma *inter-list-set-Nil[simp]*: $\text{inter-list-set } [] \ xs = []$
 $\langle \text{proof} \rangle$

lemma *inter-list-set-Cons[simp]*: $\text{inter-list-set } (x \# xs) \ ys =$
 $(\text{if } x \in \text{set } ys \text{ then } x \# \text{inter-list-set } xs \ ys \text{ else } \text{inter-list-set } xs \ ys)$
 $\langle \text{proof} \rangle$

lemma *inter-list-set-Nil2[simp]*: $\text{inter-list-set } xs \ [] = []$
 $\langle \text{proof} \rangle$

lemma *distinct-inter-list-set[simp]*: $\text{distinct } xs \implies \text{distinct } (\text{inter-list-set } xs \ ys)$
 $\langle \text{proof} \rangle$

lemma *inter-list-set-append[simp]*:
 $\text{inter-list-set } (xs \ @ \ ys) \ zs = \text{inter-list-set } xs \ zs \ @ \ \text{inter-list-set } ys \ zs$
 $\langle \text{proof} \rangle$

lemma *length-inter-list-set*: $\text{length}(\text{inter-list-set } xs \ ys) \leq \text{length } xs$
 $\langle \text{proof} \rangle$

66.2.11 *replicate*

lemma *length-replicate [simp]*: $\text{length } (\text{replicate } n \ x) = n$
 $\langle \text{proof} \rangle$

lemma *replicate-eqI*:
assumes $\text{length } xs = n$ **and** $\bigwedge y. y \in \text{set } xs \implies y = x$
shows $xs = \text{replicate } n \ x$
 $\langle \text{proof} \rangle$

lemma *Ex-list-of-length*: $\exists xs. \text{length } xs = n$
 $\langle \text{proof} \rangle$

lemma *map-replicate [simp]*: $\text{map } f \ (\text{replicate } n \ x) = \text{replicate } n \ (f \ x)$
 $\langle \text{proof} \rangle$

lemma *map-replicate-const*:
 $\text{map } (\lambda x. k) \ lst = \text{replicate } (\text{length } lst) \ k$
 $\langle \text{proof} \rangle$

lemma *replicate-app-Cons-same*:
 $(\text{replicate } n \ x) \ @ \ (x \# xs) = x \# \text{replicate } n \ x \ @ \ xs$

$\langle \text{proof} \rangle$

lemma *rev-replicate* [simp]: $\text{rev} (\text{replicate } n \ x) = \text{replicate } n \ x$
 $\langle \text{proof} \rangle$

lemma *replicate-add*: $\text{replicate } (n + m) \ x = \text{replicate } n \ x @ \text{replicate } m \ x$
 $\langle \text{proof} \rangle$

Courtesy of Matthias Daum:

lemma *append-replicate-commute*:
 $\text{replicate } n \ x @ \text{replicate } k \ x = \text{replicate } k \ x @ \text{replicate } n \ x$
 $\langle \text{proof} \rangle$

Courtesy of Andreas Lochbihler:

lemma *filter-replicate*:
 $\text{filter } P (\text{replicate } n \ x) = (\text{if } P \ x \text{ then } \text{replicate } n \ x \text{ else } [])$
 $\langle \text{proof} \rangle$

lemma *hd-replicate* [simp]: $n \neq 0 \implies \text{hd} (\text{replicate } n \ x) = x$
 $\langle \text{proof} \rangle$

lemma *tl-replicate* [simp]: $\text{tl} (\text{replicate } n \ x) = \text{replicate } (n - 1) \ x$
 $\langle \text{proof} \rangle$

lemma *last-replicate* [simp]: $n \neq 0 \implies \text{last} (\text{replicate } n \ x) = x$
 $\langle \text{proof} \rangle$

lemma *nth-replicate* [simp]: $i < n \implies (\text{replicate } n \ x) ! i = x$
 $\langle \text{proof} \rangle$

Courtesy of Matthias Daum (2 lemmas):

lemma *take-replicate* [simp]: $\text{take } i (\text{replicate } k \ x) = \text{replicate } (\min i \ k) \ x$
 $\langle \text{proof} \rangle$

lemma *drop-replicate* [simp]: $\text{drop } i (\text{replicate } k \ x) = \text{replicate } (k - i) \ x$
 $\langle \text{proof} \rangle$

lemma *set-replicate-Suc*: $\text{set} (\text{replicate } (\text{Suc } n) \ x) = \{x\}$
 $\langle \text{proof} \rangle$

lemma *set-replicate* [simp]: $n \neq 0 \implies \text{set} (\text{replicate } n \ x) = \{x\}$
 $\langle \text{proof} \rangle$

lemma *set-replicate-conv-if*: $\text{set} (\text{replicate } n \ x) = (\text{if } n = 0 \text{ then } \{\} \text{ else } \{x\})$
 $\langle \text{proof} \rangle$

lemma *in-set-replicate* [simp]: $(x \in \text{set} (\text{replicate } n \ y)) = (x = y \wedge n \neq 0)$
 $\langle \text{proof} \rangle$

lemma *card-set-1-iff-replicate*:

$\text{card}(\text{set } xs) = \text{Suc } 0 \longleftrightarrow xs \neq [] \wedge (\exists x. xs = \text{replicate } (\text{length } xs) \ x)$
 $\langle \text{proof} \rangle$

lemma *Ball-set-replicate[simp]*:

$(\forall x \in \text{set}(\text{replicate } n \ a). P \ x) = (P \ a \vee n=0)$
 $\langle \text{proof} \rangle$

lemma *Bex-set-replicate[simp]*:

$(\exists x \in \text{set}(\text{replicate } n \ a). P \ x) = (P \ a \wedge n \neq 0)$
 $\langle \text{proof} \rangle$

lemma *replicate-append-same*:

$\text{replicate } i \ x \ @ \ [x] = x \ # \ \text{replicate } i \ x$
 $\langle \text{proof} \rangle$

lemma *map-replicate-trivial*:

$\text{map } (\lambda i. \ x) \ [0..<i] = \text{replicate } i \ x$
 $\langle \text{proof} \rangle$

lemma *concat-replicate-trivial[simp]*:

$\text{concat } (\text{replicate } i \ []) = []$
 $\langle \text{proof} \rangle$

lemma *concat-replicate-single[simp]*: $\text{concat } (\text{replicate } m \ [a]) = \text{replicate } m \ a$
 $\langle \text{proof} \rangle$

lemma *replicate-empty[simp]*: $(\text{replicate } n \ x = []) \longleftrightarrow n=0$
 $\langle \text{proof} \rangle$

lemmas *empty-replicate[simp] = replicate-empty[THEN eq-iff-swap]*

lemma *replicate-eq-replicate[simp]*:

$(\text{replicate } m \ x = \text{replicate } n \ y) \longleftrightarrow (m=n \wedge (m \neq 0 \longrightarrow x=y))$
 $\langle \text{proof} \rangle$

lemma *takeWhile-replicate[simp]*:

$\text{takeWhile } P \ (\text{replicate } n \ x) = (\text{if } P \ x \ \text{then } \text{replicate } n \ x \ \text{else } [])$
 $\langle \text{proof} \rangle$

lemma *dropWhile-replicate[simp]*:

$\text{dropWhile } P \ (\text{replicate } n \ x) = (\text{if } P \ x \ \text{then } [] \ \text{else } \text{replicate } n \ x)$
 $\langle \text{proof} \rangle$

lemma *replicate-length-filter*:

$\text{replicate } (\text{length } (\text{filter } (\lambda y. \ x = y) \ xs)) \ x = \text{filter } (\lambda y. \ x = y) \ xs$
 $\langle \text{proof} \rangle$

lemma *comm-append-are-replicate*:

$xs @ ys = ys @ xs \implies \exists m n zs. \text{concat} (\text{replicate } m \text{ } zs) = xs \wedge \text{concat} (\text{replicate } n \text{ } zs) = ys$
 $\langle \text{proof} \rangle$

lemma *comm-append-is-replicate*:

fixes $xs \ ys :: 'a \text{ list}$
assumes $xs \neq [] \ ys \neq []$
assumes $xs @ ys = ys @ xs$
shows $\exists n \ zs. n > 1 \wedge \text{concat} (\text{replicate } n \text{ } zs) = xs @ ys$
 $\langle \text{proof} \rangle$

lemma *Cons-replicate-eq*:

$x \# xs = \text{replicate } n \text{ } y \longleftrightarrow x = y \wedge n > 0 \wedge xs = \text{replicate } (n - 1) \text{ } x$
 $\langle \text{proof} \rangle$

lemma *replicate-length-same*:

$(\forall y \in \text{set } xs. y = x) \implies \text{replicate} (\text{length } xs) \ x = xs$
 $\langle \text{proof} \rangle$

lemma *foldr-replicate* [simp]:

$\text{foldr } f (\text{replicate } n \text{ } x) = f \ x \ \overset{\sim}{\sim} n$
 $\langle \text{proof} \rangle$

lemma *fold-replicate* [simp]:

$\text{fold } f (\text{replicate } n \text{ } x) = f \ x \ \overset{\sim}{\sim} n$
 $\langle \text{proof} \rangle$

66.2.12 enumerate

lemma *enumerate-simps* [simp, code]:

$\text{enumerate } n \ [] = []$
 $\text{enumerate } n \ (x \# xs) = (n, x) \# \text{enumerate } (\text{Suc } n) \ xs$
 $\langle \text{proof} \rangle$

lemma *length-enumerate* [simp]:

$\text{length} (\text{enumerate } n \ xs) = \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *map-fst-enumerate* [simp]:

$\text{map } \text{fst} (\text{enumerate } n \ xs) = [n..<n + \text{length } xs]$
 $\langle \text{proof} \rangle$

lemma *map-snd-enumerate* [simp]:

$\text{map } \text{snd} (\text{enumerate } n \ xs) = xs$
 $\langle \text{proof} \rangle$

lemma *in-set-enumerate-eq*:

$p \in \text{set} (\text{enumerate } n \ xs) \longleftrightarrow n \leq \text{fst } p \wedge \text{fst } p < \text{length } xs + n \wedge \text{nth } xs (\text{fst } p - n) = \text{snd } p$

<proof>

lemma *nth-enumerate-eq*: $m < \text{length } xs \implies \text{enumerate } n \text{ } xs ! m = (n + m, xs ! m)$

<proof>

lemma *enumerate-replicate-eq*:

$\text{enumerate } n (\text{replicate } m \ a) = \text{map } (\lambda q. (q, a)) [n..<n + m]$

<proof>

lemma *enumerate-Suc-eq*:

$\text{enumerate } (\text{Suc } n) \ xs = \text{map } (\text{apfst } \text{Suc}) (\text{enumerate } n \ xs)$

<proof>

lemma *distinct-enumerate [simp]*:

$\text{distinct } (\text{enumerate } n \ xs)$

<proof>

lemma *enumerate-append-eq*:

$\text{enumerate } n \ (xs @ ys) = \text{enumerate } n \ xs @ \text{enumerate } (n + \text{length } xs) \ ys$

<proof>

lemma *enumerate-map-upt*:

$\text{enumerate } n \ (\text{map } f \ [n..<m]) = \text{map } (\lambda k. (k, f \ k)) [n..<m]$

<proof>

66.2.13 rotate1 and rotate

lemma *rotate0[simp]*: $\text{rotate } 0 = \text{id}$

<proof>

lemma *rotate-Suc[simp]*: $\text{rotate } (\text{Suc } n) \ xs = \text{rotate1}(\text{rotate } n \ xs)$

<proof>

lemma *rotate-add*:

$\text{rotate } (m+n) = \text{rotate } m \circ \text{rotate } n$

<proof>

lemma *rotate-rotate*: $\text{rotate } m \ (\text{rotate } n \ xs) = \text{rotate } (m+n) \ xs$

<proof>

lemma *rotate1-map*: $\text{rotate1 } (\text{map } f \ xs) = \text{map } f \ (\text{rotate1 } xs)$

<proof>

lemma *rotate1-rotate-swap*: $\text{rotate1 } (\text{rotate } n \ xs) = \text{rotate } n \ (\text{rotate1 } xs)$

<proof>

lemma *rotate1-length01[simp]*: $\text{length } xs \leq 1 \implies \text{rotate1 } xs = xs$

<proof>

lemma *rotate-length01*[simp]: $\text{length } xs \leq 1 \implies \text{rotate } n \text{ } xs = xs$
 ⟨proof⟩

lemma *rotate1-hd-tl*: $xs \neq [] \implies \text{rotate1 } xs = \text{tl } xs @ [\text{hd } xs]$
 ⟨proof⟩

lemma *rotate-drop-take*:
 $\text{rotate } n \text{ } xs = \text{drop } (n \bmod \text{length } xs) \text{ } xs @ \text{take } (n \bmod \text{length } xs) \text{ } xs$
 ⟨proof⟩

lemma *rotate-conv-mod*: $\text{rotate } n \text{ } xs = \text{rotate } (n \bmod \text{length } xs) \text{ } xs$
 ⟨proof⟩

lemma *rotate-id*[simp]: $n \bmod \text{length } xs = 0 \implies \text{rotate } n \text{ } xs = xs$
 ⟨proof⟩

lemma *length-rotate1*[simp]: $\text{length}(\text{rotate1 } xs) = \text{length } xs$
 ⟨proof⟩

lemma *length-rotate*[simp]: $\text{length}(\text{rotate } n \text{ } xs) = \text{length } xs$
 ⟨proof⟩

lemma *distinct1-rotate*[simp]: $\text{distinct}(\text{rotate1 } xs) = \text{distinct } xs$
 ⟨proof⟩

lemma *distinct-rotate*[simp]: $\text{distinct}(\text{rotate } n \text{ } xs) = \text{distinct } xs$
 ⟨proof⟩

lemma *rotate-map*: $\text{rotate } n \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{rotate } n \text{ } xs)$
 ⟨proof⟩

lemma *set-rotate1*[simp]: $\text{set}(\text{rotate1 } xs) = \text{set } xs$
 ⟨proof⟩

lemma *set-rotate*[simp]: $\text{set}(\text{rotate } n \text{ } xs) = \text{set } xs$
 ⟨proof⟩

lemma *rotate1-replicate*[simp]: $\text{rotate1 } (\text{replicate } n \text{ } a) = \text{replicate } n \text{ } a$
 ⟨proof⟩

lemma *rotate1-is-Nil-conv*[simp]: $(\text{rotate1 } xs = []) = (xs = [])$
 ⟨proof⟩

lemma *rotate-is-Nil-conv*[simp]: $(\text{rotate } n \text{ } xs = []) = (xs = [])$
 ⟨proof⟩

lemma *rotate-rev*:
 $\text{rotate } n \text{ } (\text{rev } xs) = \text{rev}(\text{rotate } (\text{length } xs - (n \bmod \text{length } xs)) \text{ } xs)$

⟨proof⟩

lemma *hd-rotate-conv-nth*:

assumes $xs \neq []$ shows $hd(\text{rotate } n \text{ } xs) = xs!(n \bmod \text{length } xs)$
 ⟨proof⟩

lemma *rotate-append*: $\text{rotate } (\text{length } l) \ (l @ q) = q @ l$

⟨proof⟩

lemma *nth-rotate*:

$\langle \text{rotate } m \text{ } xs ! n = xs ! ((m + n) \bmod \text{length } xs) \rangle$ if $\langle n < \text{length } xs \rangle$
 ⟨proof⟩

lemma *nth-rotate1*:

$\langle \text{rotate1 } xs ! n = xs ! (\text{Suc } n \bmod \text{length } xs) \rangle$ if $\langle n < \text{length } xs \rangle$
 ⟨proof⟩

lemma *inj-rotate1*: $\text{inj } \text{rotate1}$

⟨proof⟩

lemma *surj-rotate1*: $\text{surj } \text{rotate1}$

⟨proof⟩

lemma *bij-rotate1*: $\text{bij } (\text{rotate1} :: 'a \text{ list} \Rightarrow 'a \text{ list})$

⟨proof⟩

lemma *rotate1-fixpoint-card*: $\text{rotate1 } xs = xs \implies xs = [] \vee \text{card}(\text{set } xs) = 1$

⟨proof⟩

66.2.14 *nths* — a generalization of (!) to sets

lemma *nths-empty* [simp]: $\text{nths } xs \ \{\} = []$

⟨proof⟩

lemma *nths-nil* [simp]: $\text{nths } [] \ A = []$

⟨proof⟩

lemma *nths-all*: $\forall i < \text{length } xs. i \in I \implies \text{nths } xs \ I = xs$

⟨proof⟩

lemma *length-nths*:

$\text{length } (\text{nths } xs \ I) = \text{card}\{i. i < \text{length } xs \wedge i \in I\}$
 ⟨proof⟩

lemma *nths-shift-lemma-Suc*:

$\text{map fst } (\text{filter } (\lambda p. P(\text{Suc}(\text{snd } p))) \ (\text{zip } xs \ is)) =$
 $\text{map fst } (\text{filter } (\lambda p. P(\text{snd } p)) \ (\text{zip } xs \ (\text{map } \text{Suc } is)))$
 ⟨proof⟩

lemma *nths-shift-lemma*:

$$\begin{aligned} \text{map fst (filter } (\lambda p. \text{snd } p \in A) \text{ (zip xs [i..<i + length xs]))} = \\ \text{map fst (filter } (\lambda p. \text{snd } p + i \in A) \text{ (zip xs [0..<length xs]))} \end{aligned}$$

<proof>

lemma *nths-append*:

$$\text{nths } (l @ l') A = \text{nths } l A @ \text{nths } l' \{j. j + \text{length } l \in A\}$$

<proof>

lemma *nths-Cons*:

$$\text{nths } (x \# l) A = (\text{if } 0 \in A \text{ then } [x] \text{ else } []) @ \text{nths } l \{j. \text{Suc } j \in A\}$$

<proof>

lemma *nths-map*: $\text{nths } (\text{map } f \text{ xs}) I = \text{map } f \text{ (nths xs } I)$

<proof>

lemma *set-nths*: $\text{set}(\text{nths xs } I) = \{xs!i \mid i. i < \text{size xs} \wedge i \in I\}$

<proof>

lemma *set-nths-subset*: $\text{set}(\text{nths xs } I) \subseteq \text{set xs}$

<proof>

lemma *notin-set-nthsI[simp]*: $x \notin \text{set xs} \implies x \notin \text{set}(\text{nths xs } I)$

<proof>

lemma *in-set-nthsD*: $x \in \text{set}(\text{nths xs } I) \implies x \in \text{set xs}$

<proof>

lemma *nths-singleton [simp]*: $\text{nths } [x] A = (\text{if } 0 \in A \text{ then } [x] \text{ else } [])$

<proof>

lemma *distinct-nthsI[simp]*: $\text{distinct xs} \implies \text{distinct } (\text{nths xs } I)$

<proof>

lemma *nths-upt-eq-take [simp]*: $\text{nths } l \{..<n\} = \text{take } n \text{ } l$

<proof>

lemma *nths-nths*: $\text{nths } (\text{nths xs } A) B = \text{nths xs } \{i \in A. \exists j \in B. \text{card } \{i' \in A. i' < i\} = j\}$

<proof>

lemma *drop-eq-nths*: $\text{drop } n \text{ xs} = \text{nths xs } \{i. i \geq n\}$

<proof>

lemma *nths-drop*: $\text{nths } (\text{drop } n \text{ xs}) I = \text{nths xs } ((+) n \text{ ` } I)$

<proof>

lemma *filter-eq-nths*: $\text{filter } P \text{ xs} = \text{nths xs } \{i. i < \text{length xs} \wedge P(xs!i)\}$

<proof>

lemma *filter-in-nths*:

$distinct\ xs \implies filter\ (\%x. x \in set\ (nth\ xs\ s))\ xs = nth\ xs\ s$
 $\langle proof \rangle$

66.2.15 *subseqs and List.n-lists*

lemma *length-subseqs*: $length\ (subseqs\ xs) = 2 \wedge length\ xs$
 $\langle proof \rangle$

lemma *subseqs-powset*: $set\ 'set\ (subseqs\ xs) = Pow\ (set\ xs)$
 $\langle proof \rangle$

lemma *distinct-set-subseqs*:

assumes *distinct xs*

shows *distinct (map set (subseqs xs))*

$\langle proof \rangle$

lemma *n-lists-Nil [simp]*: $List.n\ lists\ n\ [] = (if\ n = 0\ then\ [[]]\ else\ [])$
 $\langle proof \rangle$

lemma *length-n-lists-elem*: $ys \in set\ (List.n\ lists\ n\ xs) \implies length\ ys = n$
 $\langle proof \rangle$

lemma *set-n-lists*: $set\ (List.n\ lists\ n\ xs) = \{ys. length\ ys = n \wedge set\ ys \subseteq set\ xs\}$
 $\langle proof \rangle$

lemma *subseqs-refl*: $xs \in set\ (subseqs\ xs)$
 $\langle proof \rangle$

lemma *subset-subseqs*: $X \subseteq set\ xs \implies X \in set\ 'set\ (subseqs\ xs)$
 $\langle proof \rangle$

lemma *Cons-in-subseqsD*: $y \# ys \in set\ (subseqs\ xs) \implies ys \in set\ (subseqs\ xs)$
 $\langle proof \rangle$

lemma *subseqs-distinctD*: $[ys \in set\ (subseqs\ xs); distinct\ xs] \implies distinct\ ys$
 $\langle proof \rangle$

66.2.16 *splice*

lemma *splice-Nil2 [simp]*: $splice\ xs\ [] = xs$
 $\langle proof \rangle$

lemma *length-splice [simp]*: $length\ (splice\ xs\ ys) = length\ xs + length\ ys$
 $\langle proof \rangle$

lemma *split-Nil-iff [simp]*: $splice\ xs\ ys = [] \longleftrightarrow xs = [] \wedge ys = []$
 $\langle proof \rangle$

lemma *splice-replicate*[simp]: $\text{splice } (\text{replicate } m \ x) \ (\text{replicate } n \ x) = \text{replicate } (m+n) \ x$
 ⟨proof⟩

66.2.17 shuffles

lemma *shuffles-commutes*: $\text{shuffles } xs \ ys = \text{shuffles } ys \ xs$
 ⟨proof⟩

lemma *Nil-in-shuffles*[simp]: $[] \in \text{shuffles } xs \ ys \longleftrightarrow xs = [] \wedge ys = []$
 ⟨proof⟩

lemma *shufflesE*:

$zs \in \text{shuffles } xs \ ys \implies$
 $(zs = xs \implies ys = [] \implies P) \implies$
 $(zs = ys \implies xs = [] \implies P) \implies$
 $(\bigwedge x \ xs' \ z \ zs'. \ xs = x \ \# \ xs' \implies zs = z \ \# \ zs' \implies x = z \implies zs' \in \text{shuffles } xs' \ ys \implies P) \implies$
 $(\bigwedge y \ ys' \ z \ zs'. \ ys = y \ \# \ ys' \implies zs = z \ \# \ zs' \implies y = z \implies zs' \in \text{shuffles } xs \ ys' \implies P) \implies P$
 ⟨proof⟩

lemma *Cons-in-shuffles-iff*:

$z \ \# \ zs \in \text{shuffles } xs \ ys \longleftrightarrow$
 $(xs \neq [] \wedge \text{hd } xs = z \wedge zs \in \text{shuffles } (\text{tl } xs) \ ys \vee$
 $ys \neq [] \wedge \text{hd } ys = z \wedge zs \in \text{shuffles } xs \ (\text{tl } ys))$
 ⟨proof⟩

lemma *splice-in-shuffles* [simp, intro]: $\text{splice } xs \ ys \in \text{shuffles } xs \ ys$
 ⟨proof⟩

lemma *Nil-in-shufflesI*: $xs = [] \implies ys = [] \implies [] \in \text{shuffles } xs \ ys$
 ⟨proof⟩

lemma *Cons-in-shuffles-leftI*: $zs \in \text{shuffles } xs \ ys \implies z \ \# \ zs \in \text{shuffles } (z \ \# \ xs) \ ys$
 ⟨proof⟩

lemma *Cons-in-shuffles-rightI*: $zs \in \text{shuffles } xs \ ys \implies z \ \# \ zs \in \text{shuffles } xs \ (z \ \# \ ys)$
 ⟨proof⟩

lemma *finite-shuffles* [simp, intro]: $\text{finite } (\text{shuffles } xs \ ys)$
 ⟨proof⟩

lemma *length-shuffles*: $zs \in \text{shuffles } xs \ ys \implies \text{length } zs = \text{length } xs + \text{length } ys$
 ⟨proof⟩

lemma *set-shuffles*: $zs \in \text{shuffles } xs \ ys \implies \text{set } zs = \text{set } xs \cup \text{set } ys$

$\langle \text{proof} \rangle$

lemma *distinct-disjoint-shuffles*:

assumes *distinct xs distinct ys set xs \cap set ys = {} zs \in shuffles xs ys*

shows *distinct zs*

$\langle \text{proof} \rangle$

lemma *Cons-shuffles-subset1*: $(\#) \ x \text{ ‘ shuffles xs ys } \subseteq \text{ shuffles } (x \# \text{ xs}) \text{ ys}$

$\langle \text{proof} \rangle$

lemma *Cons-shuffles-subset2*: $(\#) \ y \text{ ‘ shuffles xs ys } \subseteq \text{ shuffles xs } (y \# \text{ ys})$

$\langle \text{proof} \rangle$

lemma *filter-shuffles*:

filter P ‘ shuffles xs ys = shuffles (filter P xs) (filter P ys)

$\langle \text{proof} \rangle$

lemma *filter-shuffles-disjoint1*:

assumes *set xs \cap set ys = {} zs \in shuffles xs ys*

shows *filter $(\lambda x. x \in \text{set xs})$ zs = xs (is filter ?P - = -)*

and *filter $(\lambda x. x \notin \text{set xs})$ zs = ys (is filter ?Q - = -)*

$\langle \text{proof} \rangle$

lemma *filter-shuffles-disjoint2*:

assumes *set xs \cap set ys = {} zs \in shuffles xs ys*

shows *filter $(\lambda x. x \in \text{set ys})$ zs = ys filter $(\lambda x. x \notin \text{set ys})$ zs = xs*

$\langle \text{proof} \rangle$

lemma *partition-in-shuffles*:

xs \in shuffles (filter P xs) (filter $(\lambda x. \neg P \ x)$ xs)

$\langle \text{proof} \rangle$

lemma *inv-image-partition*:

assumes $\bigwedge x. x \in \text{set xs} \implies P \ x \bigwedge y. y \in \text{set ys} \implies \neg P \ y$

shows *partition P - ‘ {(xs, ys)} = shuffles xs ys*

$\langle \text{proof} \rangle$

66.2.18 Transpose

function *transpose* **where**

transpose [] = [] |

transpose ([_ # xss] = transpose xss |

transpose ((x # xs) # xss) =

(x # [h. (h # t) \leftarrow xss]) # transpose (xs # [t. (h # t) \leftarrow xss])

$\langle \text{proof} \rangle$

lemma *transpose-aux-filter-head*:

concat (map (case-list [] $(\lambda h \ t. [h])$) xss) =

map $(\lambda xs. \text{hd xs})$ (filter $(\lambda ys. ys \neq [])$ xss)

$\langle \text{proof} \rangle$

lemma *transpose-aux-filter-tail*:

$\text{concat } (\text{map } (\text{case-list } [] (\lambda h t. [t])) xss) =$
 $\text{map } (\lambda xs. \text{tl } xs) (\text{filter } (\lambda ys. ys \neq []) xss)$
 $\langle \text{proof} \rangle$

lemma *transpose-aux-max*:

$\text{max } (\text{Suc } (\text{length } xs)) (\text{foldr } (\lambda xs. \text{max } (\text{length } xs)) xss 0) =$
 $\text{Suc } (\text{max } (\text{length } xs) (\text{foldr } (\lambda x. \text{max } (\text{length } x - \text{Suc } 0)) (\text{filter } (\lambda ys. ys \neq [])$
 $xss) 0))$
 $(\text{is } \text{max} - ?\text{foldB} = \text{Suc } (\text{max} - ?\text{foldA}))$
 $\langle \text{proof} \rangle$

termination *transpose*

$\langle \text{proof} \rangle$

lemma *transpose-empty*: $(\text{transpose } xs = []) \longleftrightarrow (\forall x \in \text{set } xs. x = [])$

$\langle \text{proof} \rangle$

lemma *length-transpose*:

fixes $xs :: 'a \text{ list list}$
shows $\text{length } (\text{transpose } xs) = \text{foldr } (\lambda xs. \text{max } (\text{length } xs)) xs 0$
 $\langle \text{proof} \rangle$

lemma *nth-transpose*:

fixes $xs :: 'a \text{ list list}$
assumes $i < \text{length } (\text{transpose } xs)$
shows $\text{transpose } xs ! i = \text{map } (\lambda xs. xs ! i) (\text{filter } (\lambda ys. i < \text{length } ys) xs)$
 $\langle \text{proof} \rangle$

lemma *transpose-map-map*:

$\text{transpose } (\text{map } (\text{map } f) xs) = \text{map } (\text{map } f) (\text{transpose } xs)$
 $\langle \text{proof} \rangle$

66.2.19 *min* and *arg-min*

lemma *min-list-Min*: $xs \neq [] \implies \text{min-list } xs = \text{Min } (\text{set } xs)$

$\langle \text{proof} \rangle$

lemma *f-arg-min-list-f*: $xs \neq [] \implies f (\text{arg-min-list } f xs) = \text{Min } (f ` (\text{set } xs))$

$\langle \text{proof} \rangle$

lemma *arg-min-list-in*: $xs \neq [] \implies \text{arg-min-list } f xs \in \text{set } xs$

$\langle \text{proof} \rangle$

66.2.20 (In)finiteness

lemma *finite-list-length*: $\text{finite } \{xs :: ('a :: \text{finite}) \text{ list}. \text{length } xs = n\}$

$\langle \text{proof} \rangle$

lemma *finite-maxlen:*

finite ($M::'a$ list set) $\implies \exists n. \forall s \in M. \text{size } s < n$
 <proof>

lemma *lists-length-Suc-eq:*

$\{xs. \text{set } xs \subseteq A \wedge \text{length } xs = \text{Suc } n\} =$
 $(\lambda(xs, n). n \# xs) \cdot (\{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\} \times A)$
 <proof>

lemma

assumes *finite* A

shows *finite-lists-length-eq:* *finite* $\{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\}$

and *card-lists-length-eq:* $\text{card } \{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\} = (\text{card } A)^n$

<proof>

lemma *finite-lists-length-le:*

assumes *finite* A **shows** *finite* $\{xs. \text{set } xs \subseteq A \wedge \text{length } xs \leq n\}$

(**is** *finite* ? S)

<proof>

lemma *card-lists-length-le:*

assumes *finite* A **shows** $\text{card } \{xs. \text{set } xs \subseteq A \wedge \text{length } xs \leq n\} = (\sum i \leq n. \text{card } A^i)$

<proof>

lemma *finite-subset-distinct:*

assumes *finite* A

shows *finite* $\{xs. \text{set } xs \subseteq A \wedge \text{distinct } xs\}$ (**is** *finite* ? S)

<proof>

lemma *card-lists-distinct-length-eq:*

assumes *finite* A $k \leq \text{card } A$

shows $\text{card } \{xs. \text{length } xs = k \wedge \text{distinct } xs \wedge \text{set } xs \subseteq A\} = \prod \{\text{card } A - k + 1 \dots \text{card } A\}$

<proof>

lemma *card-lists-distinct-length-eq':*

assumes $k < \text{card } A$

shows $\text{card } \{xs. \text{length } xs = k \wedge \text{distinct } xs \wedge \text{set } xs \subseteq A\} = \prod \{\text{card } A - k + 1 \dots \text{card } A\}$

<proof>

lemma *infinite-UNIV-listI:* $\neg \text{finite}(\text{UNIV}::'a \text{ list set})$

<proof>

lemma *same-length-different:*

assumes $xs \neq ys$ **and** $\text{length } xs = \text{length } ys$

shows $\exists \text{pre } x \text{ } xs' \text{ } y \text{ } ys'. x \neq y \wedge xs = \text{pre} @ [x] @ xs' \wedge ys = \text{pre} @ [y] @ ys'$

$\langle proof \rangle$

66.3 Sorting

66.3.1 *sorted-wrt*

Sometimes the second equation in the definition of *sorted-wrt* is too aggressive because it relates each list element to *all* its successors. Then this equation should be removed and *sorted-wrt2-simps* should be added instead.

lemma *sorted-wrt1*: *sorted-wrt* P $[x] = True$

$\langle proof \rangle$

lemma *sorted-wrt2*: *transp* $P \implies \text{sorted-wrt } P (x \# y \# zs) = (P x y \wedge \text{sorted-wrt } P (y \# zs))$

$\langle proof \rangle$

lemmas *sorted-wrt2-simps* = *sorted-wrt1 sorted-wrt2*

lemma *sorted-wrt-true* [*simp*]:

sorted-wrt $(\lambda - . True)$ xs

$\langle proof \rangle$

lemma *sorted-wrt-append*:

sorted-wrt $P (xs @ ys) \longleftrightarrow$

sorted-wrt $P xs \wedge \text{sorted-wrt } P ys \wedge (\forall x \in \text{set } xs. \forall y \in \text{set } ys. P x y)$

$\langle proof \rangle$

lemma *sorted-wrt-map*:

sorted-wrt $R (\text{map } f xs) = \text{sorted-wrt } (\lambda x y. R (f x) (f y)) xs$

$\langle proof \rangle$

lemma

assumes *sorted-wrt* $f xs$

shows *sorted-wrt-take*[*simp*]: *sorted-wrt* $f (\text{take } n xs)$

and *sorted-wrt-drop*[*simp*]: *sorted-wrt* $f (\text{drop } n xs)$

$\langle proof \rangle$

lemma *sorted-wrt-dropWhile*[*simp*]: *sorted-wrt* $R xs \implies \text{sorted-wrt } R (\text{dropWhile } P xs)$

$\langle proof \rangle$

lemma *sorted-wrt-takeWhile*[*simp*]: *sorted-wrt* $R xs \implies \text{sorted-wrt } R (\text{takeWhile } P xs)$

$\langle proof \rangle$

lemma *sorted-wrt-filter*:

sorted-wrt $f xs \implies \text{sorted-wrt } f (\text{filter } P xs)$

$\langle proof \rangle$

lemma *sorted-wrt-rev*:

$\text{sorted-wrt } P \text{ (rev } xs) = \text{sorted-wrt } (\lambda x y. P y x) xs$
 $\langle \text{proof} \rangle$

lemma *sorted-wrt-mono-rel*:

$(\bigwedge x y. \llbracket x \in \text{set } xs; y \in \text{set } xs; P x y \rrbracket \implies Q x y) \implies \text{sorted-wrt } P xs \implies \text{sorted-wrt } Q xs$
 $\langle \text{proof} \rangle$

lemma *sorted-wrt01*: $\text{length } xs \leq 1 \implies \text{sorted-wrt } P xs$

$\langle \text{proof} \rangle$

lemma *sorted-wrt-iff-nth-less*:

$\text{sorted-wrt } P xs = (\forall i j. i < j \longrightarrow j < \text{length } xs \longrightarrow P (xs ! i) (xs ! j))$
 $\langle \text{proof} \rangle$

lemma *sorted-wrt-nth-less*:

$\llbracket \text{sorted-wrt } P xs; i < j; j < \text{length } xs \rrbracket \implies P (xs ! i) (xs ! j)$
 $\langle \text{proof} \rangle$

lemma *sorted-wrt-iff-nth-Suc-transp*: **assumes** $\text{transp } P$

shows $\text{sorted-wrt } P xs \longleftrightarrow (\forall i. \text{Suc } i < \text{length } xs \longrightarrow P (xs ! i) (xs ! (\text{Suc } i)))$ (**is**
 $?L = ?R$)
 $\langle \text{proof} \rangle$

lemma *sorted-wrt-upt[simp]*: $\text{sorted-wrt } (<) [m..<n]$

$\langle \text{proof} \rangle$

lemma *sorted-wrt-upto[simp]*: $\text{sorted-wrt } (<) [i..j]$

$\langle \text{proof} \rangle$

Each element is greater or equal to its index:

lemma *sorted-wrt-less-idx*:

$\text{sorted-wrt } (<) ns \implies i < \text{length } ns \implies i \leq ns ! i$
 $\langle \text{proof} \rangle$

66.3.2 *sorted*

context *linorder*

begin

Sometimes the second equation in the definition of *sorted* is too aggressive because it relates each list element to *all* its successors. Then this equation should be removed and *sorted2-simps* should be added instead. Executable code is one such use case.

lemma *sorted0*: $\text{sorted } [] = \text{True}$

$\langle \text{proof} \rangle$

lemma *sorted1*: $\text{sorted } [x] = \text{True}$

$\langle \text{proof} \rangle$

lemma *sorted2*: $\text{sorted } (x \# y \# zs) = (x \leq y \wedge \text{sorted } (y \# zs))$
 $\langle \text{proof} \rangle$

lemmas *sorted2-simps* = *sorted1 sorted2*

lemma *sorted-append*:
 $\text{sorted } (xs @ ys) = (\text{sorted } xs \wedge \text{sorted } ys \wedge (\forall x \in \text{set } xs. \forall y \in \text{set } ys. x \leq y))$
 $\langle \text{proof} \rangle$

lemma *sorted-map*:
 $\text{sorted } (\text{map } f \text{ } xs) = \text{sorted-wrt } (\lambda x \ y. f \ x \leq f \ y) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *sorted01*: $\text{length } xs \leq 1 \implies \text{sorted } xs$
 $\langle \text{proof} \rangle$

lemma *sorted-tl*:
 $\text{sorted } xs \implies \text{sorted } (\text{tl } xs)$
 $\langle \text{proof} \rangle$

lemma *sorted-iff-nth-mono-less*:
 $\text{sorted } xs = (\forall i \ j. i < j \longrightarrow j < \text{length } xs \longrightarrow xs ! i \leq xs ! j)$
 $\langle \text{proof} \rangle$

lemma *sorted-iff-nth-mono*:
 $\text{sorted } xs = (\forall i \ j. i \leq j \longrightarrow j < \text{length } xs \longrightarrow xs ! i \leq xs ! j)$
 $\langle \text{proof} \rangle$

lemma *sorted-nth-mono*:
 $\text{sorted } xs \implies i \leq j \implies j < \text{length } xs \implies xs ! i \leq xs ! j$
 $\langle \text{proof} \rangle$

lemma *sorted-iff-nth-Suc*:
 $\text{sorted } xs \longleftrightarrow (\forall i. \text{Suc } i < \text{length } xs \longrightarrow xs ! i \leq xs ! (\text{Suc } i))$
 $\langle \text{proof} \rangle$

lemma *sorted-rev-nth-mono*:
 $\text{sorted } (\text{rev } xs) \implies i \leq j \implies j < \text{length } xs \implies xs ! j \leq xs ! i$
 $\langle \text{proof} \rangle$

lemma *sorted-rev-iff-nth-mono*:
 $\text{sorted } (\text{rev } xs) \longleftrightarrow (\forall i \ j. i \leq j \longrightarrow j < \text{length } xs \longrightarrow xs ! j \leq xs ! i) \text{ (is ?L = ?R)}$
 $\langle \text{proof} \rangle$

lemma *sorted-rev-iff-nth-Suc*:
 $\text{sorted } (\text{rev } xs) \longleftrightarrow (\forall i. \text{Suc } i < \text{length } xs \longrightarrow xs ! (\text{Suc } i) \leq xs ! i)$
 $\langle \text{proof} \rangle$

lemma *sorted-map-remove1*:

sorted (map *f xs*) \implies *sorted* (map *f* (remove1 *x xs*))
 ⟨proof⟩

lemma *sorted-remove1*: *sorted xs* \implies *sorted* (remove1 *a xs*)

⟨proof⟩

lemma *sorted-butlast*:

assumes *sorted xs*
shows *sorted* (butlast *xs*)
 ⟨proof⟩

lemma *sorted-replicate* [simp]: *sorted*(replicate *n x*)

⟨proof⟩

lemma *sorted-remdups*[simp]:

sorted xs \implies *sorted* (remdups *xs*)
 ⟨proof⟩

lemma *sorted-remdups-adj*[simp]:

sorted xs \implies *sorted* (remdups-adj *xs*)
 ⟨proof⟩

lemma *sorted-nths*: *sorted xs* \implies *sorted* (nths *xs I*)

⟨proof⟩

lemma *sorted-distinct-set-unique*:

assumes *sorted xs distinct xs sorted ys distinct ys set xs = set ys*
shows *xs = ys*
 ⟨proof⟩

lemma *map-sorted-distinct-set-unique*:

assumes *inj-on f (set xs \cup 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*
 ⟨proof⟩

lemma *sorted-dropWhile*: *sorted xs* \implies *sorted* (dropWhile *P xs*)

⟨proof⟩

lemma *sorted-takeWhile*: *sorted xs* \implies *sorted* (takeWhile *P xs*)

⟨proof⟩

lemma *sorted-filter*:

sorted (map *f xs*) \implies *sorted* (map *f* (filter *P xs*))
 ⟨proof⟩

lemma *foldr-max-sorted*:
 assumes *sorted* (*rev xs*)
 shows *foldr max xs y* = (if *xs* = [] then *y* else *max (xs ! 0) y*)
 ⟨*proof*⟩

lemma *filter-equals-takeWhile-sorted-rev*:
 assumes *sorted*: *sorted (rev (map f xs))*
 shows *filter* ($\lambda x. t < f x$) *xs* = *takeWhile* ($\lambda x. t < f x$) *xs*
 (is *filter ?P xs* = *?tW*)
 ⟨*proof*⟩

lemma *sorted-map-same*:
sorted (map f (filter ($\lambda x. f x = g xs$) xs))
 ⟨*proof*⟩

lemma *sorted-same*:
sorted (filter ($\lambda x. x = g xs$) xs)
 ⟨*proof*⟩

end

lemma *sorted-upt[simp]*: *sorted [m..*n*]*
 ⟨*proof*⟩

lemma *sorted-upto[simp]*: *sorted [m..*n*]*
 ⟨*proof*⟩

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.

context *linorder*
begin

lemma *set-insort-key*:
set (insort-key f x xs) = *insert x (set xs)*
 ⟨*proof*⟩

lemma *length-insort [simp]*:
length (insort-key f x xs) = *Suc (length xs)*
 ⟨*proof*⟩

lemma *insort-key-left-comm*:
 assumes *f x* ≠ *f y*

shows $\text{insort-key } f \ y \ (\text{insort-key } f \ x \ xs) = \text{insort-key } f \ x \ (\text{insort-key } f \ y \ xs)$
 $\langle \text{proof} \rangle$

lemma *insort-left-comm*:
 $\text{insort } x \ (\text{insort } y \ xs) = \text{insort } y \ (\text{insort } x \ xs)$
 $\langle \text{proof} \rangle$

lemma *comp-fun-commute-insort*: *comp-fun-commute insort*
 $\langle \text{proof} \rangle$

lemma *sort-key-simps* [simp]:
 $\text{sort-key } f \ [] = []$
 $\text{sort-key } f \ (x \# xs) = \text{insort-key } f \ x \ (\text{sort-key } f \ xs)$
 $\langle \text{proof} \rangle$

lemma *sort-key-conv-fold*:
assumes *inj-on* $f \ (\text{set } xs)$
shows $\text{sort-key } f \ xs = \text{fold } (\text{insort-key } f) \ xs \ []$
 $\langle \text{proof} \rangle$

lemma *sort-conv-fold*:
 $\text{sort } xs = \text{fold insort } xs \ []$
 $\langle \text{proof} \rangle$

lemma *length-sort*[simp]: $\text{length } (\text{sort-key } f \ xs) = \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *set-sort*[simp]: $\text{set}(\text{sort-key } f \ xs) = \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *distinct-insort*: $\text{distinct } (\text{insort-key } f \ x \ xs) = (x \notin \text{set } xs \wedge \text{distinct } xs)$
 $\langle \text{proof} \rangle$

lemma *distinct-insort-key*:
 $\text{distinct } (\text{map } f \ (\text{insort-key } f \ x \ xs)) = (f \ x \notin f \ ` \ \text{set } xs \wedge (\text{distinct } (\text{map } f \ xs)))$
 $\langle \text{proof} \rangle$

lemma *distinct-sort*[simp]: $\text{distinct } (\text{sort-key } f \ xs) = \text{distinct } xs$
 $\langle \text{proof} \rangle$

lemma *sorted-insort-key*: $\text{sorted } (\text{map } f \ (\text{insort-key } f \ x \ xs)) = \text{sorted } (\text{map } f \ xs)$
 $\langle \text{proof} \rangle$

lemma *sorted-insort*: $\text{sorted } (\text{insort } x \ xs) = \text{sorted } xs$
 $\langle \text{proof} \rangle$

theorem *sorted-sort-key* [simp]: $\text{sorted } (\text{map } f \ (\text{sort-key } f \ xs))$
 $\langle \text{proof} \rangle$

theorem *sorted-sort* [simp]: *sorted* (*sort xs*)
 ⟨*proof*⟩

lemma *insort-not-Nil* [simp]:
insort-key f a xs ≠ []
 ⟨*proof*⟩

lemma *insort-is-Cons*: $\forall x \in \text{set } xs. f a \leq f x \implies \text{insort-key } f a xs = a \# xs$
 ⟨*proof*⟩

lemma *sort-key-id-if-sorted*: *sorted* (*map f xs*) $\implies \text{sort-key } f xs = xs$
 ⟨*proof*⟩

Subsumed by *sorted* (*map ?f ?xs*) $\implies \text{sort-key } ?f ?xs = ?xs$ but easier to find:

lemma *sorted-sort-id*: *sorted xs* $\implies \text{sort } xs = xs$
 ⟨*proof*⟩

lemma *sort-replicate* [simp]: *sort* (*replicate n x*) = *replicate n x*
 ⟨*proof*⟩

lemma *insort-key-remove1*:
 assumes *a* ∈ *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*
 ⟨*proof*⟩

lemma *insort-remove1*:
 assumes *a* ∈ *set xs* and *sorted xs*
 shows *insort a* (*remove1 a xs*) = *xs*
 ⟨*proof*⟩

lemma *finite-sorted-distinct-unique*:
 assumes *finite A* shows $\exists ! xs. \text{set } xs = A \wedge \text{sorted } xs \wedge \text{distinct } xs$
 ⟨*proof*⟩

lemma *insort-insert-key-triv*:
 $f x \in f \text{ ` set } xs \implies \text{insort-insert-key } f x xs = xs$
 ⟨*proof*⟩

lemma *insort-insert-triv*:
 $x \in \text{set } xs \implies \text{insort-insert } x xs = xs$
 ⟨*proof*⟩

lemma *insort-insert-insort-key*:
 $f x \notin f \text{ ` set } xs \implies \text{insort-insert-key } f x xs = \text{insort-key } f x xs$
 ⟨*proof*⟩

lemma *insort-insert-insort*:
 $x \notin \text{set } xs \implies \text{insort-insert } x xs = \text{insort } x xs$

$\langle \text{proof} \rangle$

lemma *set-insort-insert*:

$\text{set } (\text{insort-insert } x \text{ } xs) = \text{insert } x \text{ } (\text{set } xs)$

$\langle \text{proof} \rangle$

lemma *distinct-insort-insert*:

assumes *distinct xs*

shows *distinct (insort-insert-key f x xs)*

$\langle \text{proof} \rangle$

lemma *sorted-insort-insert-key*:

assumes *sorted (map f xs)*

shows *sorted (map f (insort-insert-key f x xs))*

$\langle \text{proof} \rangle$

lemma *sorted-insort-insert*:

assumes *sorted xs*

shows *sorted (insort-insert x xs)*

$\langle \text{proof} \rangle$

lemma *filter-insort-triv*:

$\neg P \ x \implies \text{filter } P \ (\text{insort-key } f \ x \ xs) = \text{filter } P \ xs$

$\langle \text{proof} \rangle$

lemma *filter-insort*:

$\text{sorted } (\text{map } f \ xs) \implies P \ x \implies \text{filter } P \ (\text{insort-key } f \ x \ xs) = \text{insort-key } f \ x \ (\text{filter } P \ xs)$

$\langle \text{proof} \rangle$

lemma *filter-sort*:

$\text{filter } P \ (\text{sort-key } f \ xs) = \text{sort-key } f \ (\text{filter } P \ xs)$

$\langle \text{proof} \rangle$

lemma *remove1-insort-key [simp]*:

$\text{remove1 } x \ (\text{insort-key } f \ x \ xs) = xs$

$\langle \text{proof} \rangle$

end

lemma *sort-upt [simp]*: $\text{sort } [m..<n] = [m..<n]$

$\langle \text{proof} \rangle$

lemma *sort-upto [simp]*: $\text{sort } [i..j] = [i..j]$

$\langle \text{proof} \rangle$

lemma *sorted-find-Min*:

$\text{sorted } xs \implies \exists x \in \text{set } xs. P \ x \implies \text{List.find } P \ xs = \text{Some } (\text{Min } \{x \in \text{set } xs. P \ x\})$

$\langle \text{proof} \rangle$

lemma *sorted-enumerate* [simp]: *sorted* (map *fst* (*enumerate* *n* *xs*))
 ⟨*proof*⟩

lemma *sorted-insort-is-snoc*: *sorted* *xs* $\implies \forall x \in \text{set } xs. a \geq x \implies \text{insort } a \text{ } xs = xs @ [a]$
 ⟨*proof*⟩

Stability of *sort-key*:

lemma *sort-key-stable*: *filter* ($\lambda y. f \ y = k$) (*sort-key* *f* *xs*) = *filter* ($\lambda y. f \ y = k$) *xs*
 ⟨*proof*⟩

corollary *stable-sort-key-sort-key*: *stable-sort-key* *sort-key*
 ⟨*proof*⟩

lemma *sort-key-const*: *sort-key* ($\lambda x. c$) *xs* = *xs*
 ⟨*proof*⟩

66.3.4 transpose on sorted lists

lemma *sorted-transpose* [simp]: *sorted* (rev (map *length* (*transpose* *xs*)))
 ⟨*proof*⟩

lemma *transpose-max-length*:
 foldr ($\lambda xs. \max (\text{length } xs)$) (*transpose* *xs*) 0 = *length* (*filter* ($\lambda x. x \neq []$) *xs*)
 (is ?L = ?R)
 ⟨*proof*⟩

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*⟩

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* ($\lambda ys. i < \text{length } ys$) *xs*)
 shows *transpose* *xs* ! *i* ! *j* = *xs* ! *j* ! *i*
 ⟨*proof*⟩

lemma *transpose-column-length*:
 fixes *xs* :: 'a list list
 assumes *sorted*: *sorted* (rev (map *length* *xs*)) and *i* < *length* *xs*
 shows *length* (*filter* ($\lambda ys. i < \text{length } ys$) (*transpose* *xs*)) = *length* (*xs* ! *i*)
 ⟨*proof*⟩

lemma *transpose-column*:

```

fixes  $xs :: 'a \text{ list list}$ 
assumes  $sorted: sorted \ (rev \ (map \ length \ xs)) \ \text{and} \ i < length \ xs$ 
shows  $map \ (\lambda ys. \ ys \ ! \ i) \ (filter \ (\lambda ys. \ i < length \ ys) \ (transpose \ xs))$ 
 $\quad = xs \ ! \ i \ (\text{is} \ ?R = -)$ 
 $\langle proof \rangle$ 

```

```

lemma transpose-transpose:
fixes  $xs :: 'a \text{ list list}$ 
assumes  $sorted: sorted \ (rev \ (map \ length \ xs))$ 
shows  $transpose \ (transpose \ xs) = takeWhile \ (\lambda x. \ x \neq []) \ xs \ (\text{is} \ ?L = ?R)$ 
 $\langle proof \rangle$ 

```

```

theorem transpose-rectangle:
assumes  $xs = [] \implies n = 0$ 
assumes  $rect: \bigwedge i. \ i < length \ xs \implies length \ (xs \ ! \ i) = n$ 
shows  $transpose \ xs = map \ (\lambda i. \ map \ (\lambda j. \ xs \ ! \ j \ ! \ i) \ [0..<length \ xs]) \ [0..<n]$ 
 $\quad (\text{is} \ ?trans = ?map)$ 
 $\langle proof \rangle$ 

```

66.3.5 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 *set*).

Note: this is a generalisation of the older *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 *sorted-list-of-set* as seen further below.

```

definition (in linorder) sorted-key-list-of-set ::  $('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ list}$ 
where  $sorted\_key\_list\_of\_set \ f \equiv folding\_on.F \ (insert\_key \ f) \ []$ 

```

```

locale folding-insert-key = lo?: linorder less-eq ::  $'a \Rightarrow 'a \Rightarrow bool$  less
for less-eq (infix  $\prec\preceq$  50) and less (infix  $\prec\prec$  50) +
fixes S
fixes  $f :: 'b \Rightarrow 'a$ 
assumes inj-on: inj-on f S
begin

```

```

lemma insert-key-commute:
 $x \in S \implies y \in S \implies insert\_key \ f \ y \ o \ insert\_key \ f \ x = insert\_key \ f \ x \ o \ insert\_key \ f \ y$ 
 $\langle proof \rangle$ 

```

```

sublocale fold-insert-key: folding-on S insert-key f []
rewrites  $folding\_on.F \ (insert\_key \ f) \ [] = sorted\_key\_list\_of\_set \ f$ 
 $\langle proof \rangle$ 

```

lemma *idem-if-sorted-distinct*:

assumes $\text{set } xs \subseteq S$ **and** $\text{sorted } (\text{map } f \text{ } xs)$ $\text{distinct } xs$
shows $\text{sorted-key-list-of-set } f \text{ } (\text{set } xs) = xs$

$\langle \text{proof} \rangle$

lemma *sorted-key-list-of-set-empty*:

$\text{sorted-key-list-of-set } f \text{ } \{\} = []$

$\langle \text{proof} \rangle$

lemma *sorted-key-list-of-set-insert*:

assumes $\text{insert } x \text{ } A \subseteq S$ **and** $\text{finite } A$ $x \notin A$
shows $\text{sorted-key-list-of-set } f \text{ } (\text{insert } x \text{ } A)$
 $= \text{insort-key } f \text{ } x \text{ } (\text{sorted-key-list-of-set } f \text{ } A)$

$\langle \text{proof} \rangle$

lemma *sorted-key-list-of-set-insert-remove [simp]*:

assumes $\text{insert } x \text{ } A \subseteq S$ **and** $\text{finite } A$
shows $\text{sorted-key-list-of-set } f \text{ } (\text{insert } x \text{ } A)$
 $= \text{insort-key } f \text{ } x \text{ } (\text{sorted-key-list-of-set } f \text{ } (A - \{x\}))$

$\langle \text{proof} \rangle$

lemma *sorted-key-list-of-set-eq-Nil-iff [simp]*:

assumes $A \subseteq S$ **and** $\text{finite } A$
shows $\text{sorted-key-list-of-set } f \text{ } A = [] \longleftrightarrow A = \{\}$

$\langle \text{proof} \rangle$

lemma *set-sorted-key-list-of-set [simp]*:

assumes $A \subseteq S$ **and** $\text{finite } A$
shows $\text{set } (\text{sorted-key-list-of-set } f \text{ } A) = A$

$\langle \text{proof} \rangle$

lemma *sorted-sorted-key-list-of-set [simp]*:

assumes $A \subseteq S$
shows $\text{sorted } (\text{map } f \text{ } (\text{sorted-key-list-of-set } f \text{ } A))$

$\langle \text{proof} \rangle$

lemma *distinct-if-distinct-map*: $\text{distinct } (\text{map } f \text{ } xs) \implies \text{distinct } xs$

$\langle \text{proof} \rangle$

lemma *distinct-sorted-key-list-of-set [simp]*:

assumes $A \subseteq S$
shows $\text{distinct } (\text{map } f \text{ } (\text{sorted-key-list-of-set } f \text{ } A))$

$\langle \text{proof} \rangle$

lemma *length-sorted-key-list-of-set [simp]*:

assumes $A \subseteq S$
shows $\text{length } (\text{sorted-key-list-of-set } f \text{ } A) = \text{card } A$

$\langle \text{proof} \rangle$

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* $x \ A \subseteq S$ **and** *finite* A
shows *sorted-key-list-of-set* $f \ (A - \{x\}) = \text{remove1 } x \ (\text{sorted-key-list-of-set } f \ A)$
 $\langle \text{proof} \rangle$

lemma *strict-sorted-key-list-of-set* [simp]:
 $A \subseteq S \implies \text{sorted-wrt } (<) \ (\text{map } f \ (\text{sorted-key-list-of-set } f \ A))$
 $\langle \text{proof} \rangle$

lemma *finite-set-strict-sorted*:
assumes $A \subseteq S$ **and** *finite* A
obtains l **where** *sorted-wrt* $(<) \ (\text{map } f \ l)$ *set* $l = A$ *length* $l = \text{card } A$
 $\langle \text{proof} \rangle$

lemma (**in** *linorder*) *strict-sorted-equal*:
assumes *sorted-wrt* $(<) \ xs$
and *sorted-wrt* $(<) \ ys$
and *set* $ys = \text{set } xs$
shows $ys = xs$
 $\langle \text{proof} \rangle$

lemma (**in** *linorder*) *strict-sorted-equal-Uniq*: $\exists \leq_1 xs. \text{sorted-wrt } (<) \ xs \wedge \text{set } xs = A$
 $\langle \text{proof} \rangle$

lemma *sorted-key-list-of-set-inject*:
assumes $A \subseteq S \ B \subseteq S$
assumes *sorted-key-list-of-set* $f \ A = \text{sorted-key-list-of-set } f \ B$ *finite* A *finite* B
shows $A = B$
 $\langle \text{proof} \rangle$

lemma *sorted-key-list-of-set-unique*:
assumes $A \subseteq S$ **and** *finite* A
shows *sorted-wrt* $(<) \ (\text{map } f \ l) \wedge \text{set } l = A \wedge \text{length } l = \text{card } A$
 $\longleftrightarrow \text{sorted-key-list-of-set } f \ A = l$
 $\langle \text{proof} \rangle$

end

context *linorder*
begin

definition *sorted-list-of-set* $\equiv \text{sorted-key-list-of-set } (\lambda x::'a. x)$

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* $(\lambda x. x) = \text{sorted-list-of-set}$
and $\bigwedge xs. \text{map } (\lambda x. x) xs \equiv xs$
and $\bigwedge X. (X \subseteq UNIV) \equiv \text{True}$
and $\bigwedge x. x \in UNIV \equiv \text{True}$
and $\bigwedge P. (\text{True} \implies P) \equiv \text{Trueprop } P$
and $\bigwedge P Q. (\text{True} \implies \text{PROP } P \implies \text{PROP } Q) \equiv (\text{PROP } P \implies \text{True} \implies \text{PROP } Q)$
 $\langle \text{proof} \rangle$

lemma *ex1-sorted-list-for-set-if-finite*:
 $\text{finite } X \implies \exists ! xs. \text{sorted-wrt } (<) xs \wedge \text{set } xs = X$
 $\langle \text{proof} \rangle$

Alias theorems for backwards compatibility and ease of use.

lemmas *sorted-list-of-set = sorted-list-of-set.sorted-key-list-of-set* **and**
 $\text{sorted-list-of-set-empty} = \text{sorted-list-of-set.sorted-key-list-of-set-empty}$ **and**
 $\text{sorted-list-of-set-insert} = \text{sorted-list-of-set.sorted-key-list-of-set-insert}$ **and**
 $\text{sorted-list-of-set-insert-remove} = \text{sorted-list-of-set.sorted-key-list-of-set-insert-remove}$
and
 $\text{sorted-list-of-set-eq-Nil-iff} = \text{sorted-list-of-set.sorted-key-list-of-set-eq-Nil-iff}$
and
 $\text{set-sorted-list-of-set} = \text{sorted-list-of-set.set-sorted-key-list-of-set}$ **and**
 $\text{sorted-sorted-list-of-set} = \text{sorted-list-of-set.sorted-sorted-key-list-of-set}$ **and**
 $\text{distinct-sorted-list-of-set} = \text{sorted-list-of-set.distinct-sorted-key-list-of-set}$ **and**
 $\text{length-sorted-list-of-set} = \text{sorted-list-of-set.length-sorted-key-list-of-set}$ **and**
 $\text{sorted-list-of-set-remove} = \text{sorted-list-of-set.sorted-key-list-of-set-remove}$ **and**
 $\text{strict-sorted-list-of-set} = \text{sorted-list-of-set.strict-sorted-key-list-of-set}$ **and**
 $\text{sorted-list-of-set-inject} = \text{sorted-list-of-set.sorted-key-list-of-set-inject}$ **and**
 $\text{sorted-list-of-set-unique} = \text{sorted-list-of-set.sorted-key-list-of-set-unique}$ **and**
 $\text{finite-set-strict-sorted} = \text{sorted-list-of-set.finite-set-strict-sorted}$

lemma *sorted-list-of-set-sort-remdups* [code]:
 $\text{sorted-list-of-set } (\text{set } xs) = \text{sort } (\text{remdups } xs)$
 $\langle \text{proof} \rangle$

end

lemma *sorted-list-of-set-range* [simp]:
 $\text{sorted-list-of-set } \{m..<n\} = [m..<n]$
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-set-lessThan-Suc* [simp]:
 $\text{sorted-list-of-set } \{..<\text{Suc } k\} = \text{sorted-list-of-set } \{..<k\} @ [k]$
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-set-atMost-Suc* [simp]:

sorted-list-of-set $\{..Suc\ k\} = \text{sorted-list-of-set } \{..k\} @ [Suc\ k]$
 $\langle \text{proof} \rangle$

lemma *sorted-lift-of-set-eq-upto* [simp]:
 $\langle \text{sorted-list-of-set } \{k..l\} = [k..l] \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-set-nonempty*:
assumes *finite A A* $\neq \{\}$
shows *sorted-list-of-set A = Min A # sorted-list-of-set (A - {Min A})*
 $\langle \text{proof} \rangle$

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*}*
 $\langle \text{proof} \rangle$

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*}*
 $\langle \text{proof} \rangle$

lemma *nth-sorted-list-of-set-greaterThanLessThan*:
 $n < j - Suc\ i \implies \text{sorted-list-of-set } \{i<..*j*\} ! n = Suc\ (i+n)$
 $\langle \text{proof} \rangle$

lemma *nth-sorted-list-of-set-greaterThanAtMost*:
 $n < j - i \implies \text{sorted-list-of-set } \{i<..*j*\} ! n = Suc\ (i+n)$
 $\langle \text{proof} \rangle$

lemma *sorted-wrt-induct* [consumes 1, case-names Nil Cons]:
assumes *sorted-wrt R xs*
assumes $P []$
 $\bigwedge x\ xs. (\bigwedge y. y \in \text{set } xs \implies R\ x\ y) \implies P\ xs \implies P\ (x \# xs)$
shows $P\ xs$
 $\langle \text{proof} \rangle$

lemma *sorted-wrt-trans-induct* [consumes 2, case-names Nil single Cons]:
assumes *sorted-wrt R xs transp R*
assumes $P [] \bigwedge x. P\ [x]$
 $\bigwedge x\ y\ xs. R\ x\ y \implies P\ (y \# xs) \implies P\ (x \# y \# xs)$
shows $P\ xs$
 $\langle \text{proof} \rangle$

lemmas *sorted-induct* [consumes 1, case-names Nil single Cons] =
sorted-wrt-trans-induct[OF - preorder-class.transp-on-le]

lemma *sorted-wrt-map-mono*:

assumes *sorted-wrt* R xs
assumes $\bigwedge x y. x \in \text{set } xs \implies y \in \text{set } xs \implies R x y \implies R' (f x) (f y)$
shows *sorted-wrt* R' $(\text{map } f xs)$
 $\langle \text{proof} \rangle$

lemma *sorted-map-mono*:
assumes *sorted* xs **and** *mono-on* $(\text{set } xs)$ f
shows *sorted* $(\text{map } f xs)$
 $\langle \text{proof} \rangle$

66.3.6 *lists*: the list-forming operator over sets

inductive-set
 $lists :: 'a \text{ set} \Rightarrow 'a \text{ list set}$
for $A :: 'a \text{ set}$
where
 $Nil [intro!, simp]: [] \in lists A$
 $| Cons [intro!, simp]: [a \in A; l \in lists A] \implies a \# l \in lists A$

inductive-cases *listsE* $[elim!]: x \# l \in lists A$
inductive-cases *listspE* $[elim!]: listsp A (x \# l)$

inductive-simps *listsp-simps* $[code]$:
 $listsp A []$
 $listsp A (x \# xs)$

lemma *listsp-mono* $[mono]: A \leq B \implies listsp A \leq listsp B$
 $\langle \text{proof} \rangle$

lemmas *lists-mono* = *listsp-mono* $[to-set]$

lemma *listsp-infI*:
assumes $l: listsp A l$ **shows** $listsp B l \implies listsp (inf A B) l$ $\langle \text{proof} \rangle$

lemmas *lists-IntI* = *listsp-infI* $[to-set]$

lemma *listsp-inf-eq* $[simp]: listsp (inf A B) = inf (listsp A) (listsp B)$
 $\langle \text{proof} \rangle$

lemmas *listsp-conj-eq* $[simp] = listsp-inf-eq$ $[simplified \text{ inf-fun-def inf-bool-def}]$

lemmas *lists-Int-eq* $[simp] = listsp-inf-eq$ $[to-set]$

lemma *Cons-in-lists-iff* $[simp]: x \# xs \in lists A \longleftrightarrow x \in A \wedge xs \in lists A$
 $\langle \text{proof} \rangle$

lemma *append-in-listsp-conv* $[iff]: (listsp A (xs @ ys)) = (listsp A xs \wedge listsp A ys)$
 $\langle \text{proof} \rangle$

lemmas *append-in-lists-conv* [iff] = *append-in-listsp-conv* [to-set]

lemma *in-listsp-conv-set*: (*listsp* *A xs*) = ($\forall x \in \text{set } xs. A\ x$)
 — eliminate *listsp* in favour of *set*
 <proof>

lemmas *in-lists-conv-set* [code-unfold] = *in-listsp-conv-set* [to-set]

lemma *in-listspD* [dest!]: *listsp* *A xs* $\implies \forall x \in \text{set } xs. A\ x$
 <proof>

lemmas *in-listsD* [dest!] = *in-listspD* [to-set]

lemma *in-listspI* [intro!]: $\forall x \in \text{set } xs. A\ x \implies \text{listsp } A\ xs$
 <proof>

lemmas *in-listsI* [intro!] = *in-listspI* [to-set]

lemma *mono-lists*: *mono lists*
 <proof>

lemma *lists-eq-set*: *lists* *A* = {*xs*. *set xs* $\leq A$ }
 <proof>

lemma *lists-empty* [simp]: *lists* {} = {}
 <proof>

lemma *lists-UNIV* [simp]: *lists* *UNIV* = *UNIV*
 <proof>

lemma *lists-image*: *lists* (*f*‘*A*) = *map f* ‘ *lists A*
 <proof>

lemma *inj-on-map-lists*: **assumes** *inj-on f A*
shows *inj-on* (*map f*) (*lists A*)
 <proof>

lemma *bij-lists*: *bij-betw f X Y* $\implies \text{bij-betw } (\text{map } f) (\text{lists } X) (\text{lists } Y)$
 <proof>

lemma *replicate-in-lists*: *a* $\in A \implies \text{replicate } k\ a \in \text{lists } A$
 <proof>

66.3.7 Inductive definition for membership

inductive *ListMem* :: ‘*a* \Rightarrow ‘*a list* \Rightarrow *bool*

where

elem: *ListMem* *x* (*x* # *xs*)

| *insert*: $ListMem\ x\ xs \implies ListMem\ x\ (y \# xs)$

lemma *ListMem-iff*: $(ListMem\ x\ xs) = (x \in set\ xs)$
 $\langle proof \rangle$

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* :: $'a\ set \Rightarrow 'a\ list\ set \Rightarrow 'a\ list\ set$ **where**
 $set-Cons\ A\ XS = \{z. \exists x\ xs. z = x \# xs \wedge x \in A \wedge xs \in XS\}$

lemma *set-Cons-sing-Nil* [*simp*]: $set-Cons\ A\ \{\}\ = (\%x. [x])\ 'A$
 $\langle proof \rangle$

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* :: $'a\ set\ list \Rightarrow 'a\ list\ set$ **where**
 $listset\ [] = \{\}\ |$
 $listset\ (A \# As) = set-Cons\ A\ (listset\ As)$

66.3.9 Transitive Closure on Lists

Use $^+$ on binary relations if possible. Transitive closure on lists is useful for executable definitions on the list level. Is not efficient, naive closure computation.

definition *trans-list-step* $ps = [(a,c). (a,b) \leftarrow ps, (b',c) \leftarrow ps, b=b']$

lemma *set-trans-list-step-subset-trancl*: $set\ (trans-list-step\ ps) \subseteq (set\ ps)^{+}$
 $\langle proof \rangle$

function *trancl-list* :: $('a * 'a)\ list \Rightarrow ('a * 'a)\ list$ **where**
 $trancl-list\ ps =$
 $(let\ ps' = trans-list-step\ ps$
 $in\ if\ set\ ps' \subseteq set\ ps\ then\ ps\ else\ trancl-list\ (List.union\ ps'\ ps))$
 $\langle proof \rangle$

termination
 $\langle proof \rangle$

declare *trancl-list.simps*[*code*, *simp del*]

lemma *set-trancl-list*: $set(trancl-list\ ps) = (set\ ps)^{+}$
 $\langle proof \rangle$

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
 $lexn :: ('a \times 'a) \text{ set} \Rightarrow \text{nat} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ set}$ **where**
 $lexn \ r \ 0 = \{\}$ |
 $lexn \ r \ (Suc \ n) =$
 $(\text{map-prod } (\%(x, xs). \ x \# xs) \ (\%(x, xs). \ x \# xs)) \ ' (r \ < *lex* > \ lexn \ r \ n)) \ Int$
 $\{(xs, ys). \ length \ xs = Suc \ n \wedge \ length \ ys = Suc \ n\}$

definition $lex :: ('a \times 'a) \text{ set} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ set}$ **where**
 $lex \ r = (\bigcup n. \ lexn \ r \ n)$ — Holds only between lists of the same length

definition $lenlex :: ('a \times 'a) \text{ set} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ set}$ **where**
 $lenlex \ r = \text{inv-image } (\text{less-than } < *lex* > \ lex \ r) \ (\lambda xs. \ (length \ xs, \ xs))$
 — Compares lists by their length and then lexicographically

lemma $wf\text{-}lexn$: **assumes** $wf \ r$ **shows** $wf \ (lexn \ r \ n)$
 $\langle proof \rangle$

lemma $lexn\text{-}length$:
 $(xs, ys) \in lexn \ r \ n \implies length \ xs = n \wedge length \ ys = n$
 $\langle proof \rangle$

lemma $wf\text{-}lex \ [intro!]$:
assumes $wf \ r$ **shows** $wf \ (lex \ r)$
 $\langle proof \rangle$

lemma $lexn\text{-}conv$:
 $lexn \ r \ n =$
 $\{(xs, ys). \ length \ xs = n \wedge \ length \ ys = n \wedge$
 $(\exists xys \ x \ y \ xs' \ ys'. \ xs = xys \ @ \ x \# xs' \wedge \ ys = xys \ @ \ y \ \# \ ys' \wedge \ (x, y) \in r)\}$
 $(\text{is } ?L \ n = ?R \ n \text{ is } - = \{(xs, ys). \ ?len \ n \ xs \wedge \ ?len \ n \ ys \wedge \ (\exists xys. \ ?P \ xs \ ys \ xys)\})$
 $\langle proof \rangle$

By Mathias Fleury:

proposition $lexn\text{-}transI$:
assumes $trans \ r$ **shows** $trans \ (lexn \ r \ n)$
 $\langle proof \rangle$

corollary $lex\text{-}transI$:
assumes $trans \ r$ **shows** $trans \ (lex \ r)$
 $\langle proof \rangle$

lemma $lex\text{-}conv$:
 $lex \ r =$

$\{(xs,ys). \text{length } xs = \text{length } ys \wedge$
 $(\exists xys\ x\ y\ xs'\ ys'.\ xs = xys @ x \# xs' \wedge ys = xys @ y \# ys' \wedge (x, y) \in r)\}$
 $\langle \text{proof} \rangle$

lemma *wf-lenlex* [intro!]: $wf\ r \implies wf\ (\text{lenlex } r)$
 $\langle \text{proof} \rangle$

lemma *lenlex-conv*:
 $\text{lenlex } r = \{(xs,ys). \text{length } xs < \text{length } ys \vee$
 $\text{length } xs = \text{length } ys \wedge (xs, ys) \in \text{lex } r\}$
 $\langle \text{proof} \rangle$

lemma *total-lenlex*:
assumes *total* r
shows *total* $(\text{lenlex } r)$
 $\langle \text{proof} \rangle$

lemma *lenlex-transI* [intro]: $\text{trans } r \implies \text{trans } (\text{lenlex } r)$
 $\langle \text{proof} \rangle$

lemma *Nil-notin-lex* [iff]: $([], ys) \notin \text{lex } r$
 $\langle \text{proof} \rangle$

lemma *Nil2-notin-lex* [iff]: $(xs, []) \notin \text{lex } r$
 $\langle \text{proof} \rangle$

lemma *Cons-in-lex* [simp]:
 $(x \# xs, y \# ys) \in \text{lex } r \longleftrightarrow (x, y) \in r \wedge \text{length } xs = \text{length } ys \vee x = y \wedge (xs,$
 $ys) \in \text{lex } r$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *Nil-lenlex-iff1* [simp]: $([], ns) \in \text{lenlex } r \longleftrightarrow ns \neq []$
and *Nil-lenlex-iff2* [simp]: $(ns, []) \notin \text{lenlex } r$
 $\langle \text{proof} \rangle$

lemma *Cons-lenlex-iff*:
 $((m \# ms, n \# ns) \in \text{lenlex } r) \longleftrightarrow$
 $\text{length } ms < \text{length } ns$
 $\vee \text{length } ms = \text{length } ns \wedge (m,n) \in r$
 $\vee (m = n \wedge (ms,ns) \in \text{lenlex } r)$
 $\langle \text{proof} \rangle$

lemma *lenlex-irreflexive*: $(\bigwedge x. (x,x) \notin r) \implies (xs,xs) \notin \text{lenlex } r$
 $\langle \text{proof} \rangle$

lemma *lenlex-trans*:
 $\llbracket (x,y) \in \text{lenlex } r; (y,z) \in \text{lenlex } r; \text{trans } r \rrbracket \implies (x,z) \in \text{lenlex } r$
 $\langle \text{proof} \rangle$

lemma *lenlex-length*: $(ms, ns) \in \text{lenlex } r \implies \text{length } ms \leq \text{length } ns$
 $\langle \text{proof} \rangle$

lemma *lex-append-rightI*:
 $(xs, ys) \in \text{lex } r \implies \text{length } vs = \text{length } us \implies (xs @ us, ys @ vs) \in \text{lex } r$
 $\langle \text{proof} \rangle$

lemma *lex-append-leftI*:
 $(ys, zs) \in \text{lex } r \implies (xs @ ys, xs @ zs) \in \text{lex } r$
 $\langle \text{proof} \rangle$

lemma *lex-append-leftD*:
 $\forall x. (x, x) \notin r \implies (xs @ ys, xs @ zs) \in \text{lex } r \implies (ys, zs) \in \text{lex } r$
 $\langle \text{proof} \rangle$

lemma *lex-append-left-iff*:
 $\forall x. (x, x) \notin r \implies (xs @ ys, xs @ zs) \in \text{lex } r \longleftrightarrow (ys, zs) \in \text{lex } r$
 $\langle \text{proof} \rangle$

lemma *lex-take-index*:
assumes $(xs, ys) \in \text{lex } r$
obtains i **where** $i < \text{length } xs$ **and** $i < \text{length } ys$ **and** $\text{take } i \text{ } xs = \text{take } i \text{ } ys$
and $(xs ! i, ys ! i) \in r$
 $\langle \text{proof} \rangle$

lemma *irrefl-lex*: $\text{irrefl } r \implies \text{irrefl } (\text{lex } r)$
 $\langle \text{proof} \rangle$

lemma *lexl-not-refl [simp]*: $\text{irrefl } r \implies (x, x) \notin \text{lex } r$
 $\langle \text{proof} \rangle$

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* :: $('a \times 'a) \text{ set} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ set}$ **where**
 $\text{lexord } r = \{(x, y). \exists a \ v. y = x @ a \ \# \ v \vee$
 $(\exists u \ a \ b \ v \ w. (a, b) \in r \wedge x = u @ (a \ \# \ v) \wedge y = u @ (b \ \# \ w))\}$

lemma *lexord-Nil-left[simp]*: $([], y) \in \text{lexord } r = (\exists a \ x. y = a \ \# \ x)$
 $\langle \text{proof} \rangle$

lemma *lexord-Nil-right[simp]*: $(x, []) \notin \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-cons-cons[simp]*:
 $(a \ \# \ x, b \ \# \ y) \in \text{lexord } r \longleftrightarrow (a, b) \in r \vee (a = b \wedge (x, y) \in \text{lexord } r)$ (**is ?lhs = ?rhs**)

$\langle \text{proof} \rangle$

lemmas *lexord-simps* = *lexord-Nil-left* *lexord-Nil-right* *lexord-cons-cons*

lemma *lexord-same-pref-iff*:

$(xs @ ys, xs @ zs) \in \text{lexord } r \longleftrightarrow (\exists x \in \text{set } xs. (x, x) \in r) \vee (ys, zs) \in \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-same-pref-if-irrefl[simp]*:

$\text{irrefl } r \implies (xs @ ys, xs @ zs) \in \text{lexord } r \longleftrightarrow (ys, zs) \in \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-append-rightI*: $\exists b z. y = b \# z \implies (x, x @ y) \in \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-append-left-rightI*:

$(a, b) \in r \implies (u @ a \# x, u @ b \# y) \in \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-append-leftI*: $(u, v) \in \text{lexord } r \implies (x @ u, x @ v) \in \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-append-leftD*:

$\llbracket (x @ u, x @ v) \in \text{lexord } r; (\forall a. (a, a) \notin r) \rrbracket \implies (u, v) \in \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-take-index-conv*:

$((x, y) \in \text{lexord } r) =$
 $((\text{length } x < \text{length } y \wedge \text{take } (\text{length } x) y = x) \vee$
 $(\exists i. i < \min(\text{length } x)(\text{length } y) \wedge \text{take } i x = \text{take } i y \wedge (x!i, y!i) \in r))$
 $\langle \text{proof} \rangle$

lemma *lexord-lex*: $(x, y) \in \text{lex } r = ((x, y) \in \text{lexord } r \wedge \text{length } x = \text{length } y)$
 $\langle \text{proof} \rangle$

lemma *lexord-sufI*:

assumes $(u, w) \in \text{lexord } r$ $\text{length } w \leq \text{length } u$

shows $(u @ v, w @ z) \in \text{lexord } r$

$\langle \text{proof} \rangle$

lemma *lexord-sufE*:

assumes $(xs @ zs, ys @ qs) \in \text{lexord } r$ $xs \neq ys$ $\text{length } xs = \text{length } ys$ $\text{length } zs = \text{length } qs$

shows $(xs, ys) \in \text{lexord } r$

$\langle \text{proof} \rangle$

lemma *lexord-irreflexive*: $\forall x. (x, x) \notin r \implies (xs, xs) \notin \text{lexord } r$
 $\langle \text{proof} \rangle$

By René Thiemann:

lemma *lexord-partial-trans*:

$(\bigwedge x y z. x \in \text{set } xs \implies (x,y) \in r \implies (y,z) \in r \implies (x,z) \in r)$
 $\implies (xs,ys) \in \text{lexord } r \implies (ys,zs) \in \text{lexord } r \implies (xs,zs) \in \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-trans*:

$\llbracket (x, y) \in \text{lexord } r; (y, z) \in \text{lexord } r; \text{trans } r \rrbracket \implies (x, z) \in \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-transI*: $\text{trans } r \implies \text{trans } (\text{lexord } r)$

$\langle \text{proof} \rangle$

lemma *total-lexord*: $\text{total } r \implies \text{total } (\text{lexord } r)$

$\langle \text{proof} \rangle$

corollary *lexord-linear*: $(\forall a b. (a,b) \in r \vee a = b \vee (b,a) \in r) \implies (x,y) \in \text{lexord } r \vee x = y \vee (y,x) \in \text{lexord } r$

$\langle \text{proof} \rangle$

lemma *lexord-irrefl*:

$\text{irrefl } R \implies \text{irrefl } (\text{lexord } R)$

$\langle \text{proof} \rangle$

lemma *lexord-asy*:

assumes *asy* R

shows *asy* $(\text{lexord } R)$

$\langle \text{proof} \rangle$

lemma *lexord-asymmetric*:

assumes *asy* R

assumes *hyp*: $(a, b) \in \text{lexord } R$

shows $(b, a) \notin \text{lexord } R$

$\langle \text{proof} \rangle$

lemma *asy-lex*: $\text{asy } R \implies \text{asy } (\text{lex } R)$

$\langle \text{proof} \rangle$

lemma *asy-lenlex*: $\text{asy } R \implies \text{asy } (\text{lenlex } R)$

$\langle \text{proof} \rangle$

lemma *lenlex-append1*:

assumes *len*: $(us,xs) \in \text{lenlex } R$ **and** *eq*: $\text{length } vs = \text{length } ys$

shows $(us @ vs, xs @ ys) \in \text{lenlex } R$

$\langle \text{proof} \rangle$

lemma *lenlex-append2* [*simp*]:

assumes *irrefl* R

shows $(us @ xs, us @ ys) \in \text{lenlex } R \longleftrightarrow (xs, ys) \in \text{lenlex } R$

$\langle \text{proof} \rangle$

Predicate version of lexicographic order integrated with Isabelle’s order type classes. Author: Andreas Lochbihler

context *ord*
begin

context
 notes *[[inductive-internals]]*
begin

inductive *lexordp* :: ‘*a* list \Rightarrow ‘*a* list \Rightarrow bool

where

Nil: *lexordp* [] (*y* # *ys*)
 | *Cons*: $x < y \implies \text{lexordp } (x \# xs) (y \# ys)$
 | *Cons-eq*:
 $\llbracket \neg x < y; \neg y < x; \text{lexordp } xs \ ys \rrbracket \implies \text{lexordp } (x \# xs) (y \# ys)$

end

lemma *lexordp-simps* [*simp*, *code*]:

lexordp [] *ys* $\longleftrightarrow ys \neq []$
 lexordp *xs* [] $\longleftrightarrow \text{False}$
 lexordp (*x* # *xs*) (*y* # *ys*) $\longleftrightarrow x < y \vee \neg y < x \wedge \text{lexordp } xs \ ys$
 <proof>

inductive *lexordp-eq* :: ‘*a* list \Rightarrow ‘*a* list \Rightarrow bool **where**

Nil: *lexordp-eq* [] *ys*
 | *Cons*: $x < y \implies \text{lexordp-eq } (x \# xs) (y \# ys)$
 | *Cons-eq*: $\llbracket \neg x < y; \neg y < x; \text{lexordp-eq } xs \ ys \rrbracket \implies \text{lexordp-eq } (x \# xs) (y \# ys)$

lemma *lexordp-eq-simps* [*simp*, *code*]:

lexordp-eq [] *ys* $\longleftrightarrow \text{True}$
 lexordp-eq *xs* [] $\longleftrightarrow xs = []$
 lexordp-eq (*x* # *xs*) (*y* # *ys*) $\longleftrightarrow x < y \vee \neg y < x \wedge \text{lexordp-eq } xs \ ys$
 <proof>

lemma *lexordp-append-rightI*: $ys \neq \text{Nil} \implies \text{lexordp } xs (xs @ ys)$

 <proof>

lemma *lexordp-append-left-rightI*: $x < y \implies \text{lexordp } (us @ x \# xs) (us @ y \# ys)$

 <proof>

lemma *lexordp-eq-refl*: *lexordp-eq* *xs* *xs*

 <proof>

lemma *lexordp-append-leftI*: $\text{lexordp } us \ vs \implies \text{lexordp } (xs @ us) (xs @ vs)$

 <proof>

lemma *lexordp-append-leftD*: $\llbracket \text{lexordp } (xs @ us) (xs @ vs); \forall a. \neg a < a \rrbracket \implies \text{lexordp } us \ vs$

$\langle \text{proof} \rangle$

lemma *lexordp-irreflexive*:
assumes *irrefl*: $\forall x. \neg x < x$
shows $\neg \text{lexordp } xs \ xs$
 $\langle \text{proof} \rangle$

lemma *lexordp-into-lexordp-eq*:
 $\text{lexordp } xs \ ys \implies \text{lexordp-eq } xs \ ys$
 $\langle \text{proof} \rangle$

lemma *lexordp-eq-pref*: $\text{lexordp-eq } u \ (u @ v)$
 $\langle \text{proof} \rangle$

end

declare *ord.lexordp-simps* [*simp*, *code*]
declare *ord.lexordp-eq-simps* [*simp*, *code*]

context *order*
begin

lemma *lexordp-antisym*:
assumes $\text{lexordp } xs \ ys \ \text{lexordp } ys \ xs$
shows *False*
 $\langle \text{proof} \rangle$

lemma *lexordp-irreflexive'*: $\neg \text{lexordp } xs \ xs$
 $\langle \text{proof} \rangle$

end

context *linorder* **begin**

lemma *lexordp-cases* [*consumes 1*, *case-names Nil Cons Cons-eq*, *cases pred: lexordp*]:
assumes $\text{lexordp } xs \ ys$
obtains $(Nil) \ y \ ys' \text{ where } xs = [] \ ys = y \# ys'$
 $| (Cons) \ x \ xs' \ y \ ys' \text{ where } xs = x \# xs' \ ys = y \# ys' \ x < y$
 $| (Cons-eq) \ x \ xs' \ ys' \text{ where } xs = x \# xs' \ ys = x \# ys' \ \text{lexordp } xs' \ ys'$
 $\langle \text{proof} \rangle$

lemma *lexordp-induct* [*consumes 1*, *case-names Nil Cons Cons-eq*, *induct pred: lexordp*]:
assumes *major*: $\text{lexordp } xs \ ys$
and *Nil*: $\bigwedge y \ ys. P [] (y \# ys)$
and *Cons*: $\bigwedge x \ xs \ y \ ys. x < y \implies P (x \# xs) (y \# ys)$
and *Cons-eq*: $\bigwedge x \ xs \ ys. [\text{lexordp } xs \ ys; P \ xs \ ys] \implies P (x \# xs) (x \# ys)$
shows $P \ xs \ ys$

<proof>

lemma *lexordp-iff*:

$lexordp\ xs\ ys \longleftrightarrow (\exists\ x\ vs.\ ys = xs\ @\ x\ \# \ vs) \vee (\exists\ us\ a\ b\ vs\ ws.\ a < b \wedge xs = us\ @\ a\ \# \ vs \wedge ys = us\ @\ b\ \# \ ws)$
(is ?lhs = ?rhs)
<proof>

lemma *lexordp-conv-lexord*:

$lexordp\ xs\ ys \longleftrightarrow (xs, ys) \in lexord\ \{(x, y). x < y\}$
<proof>

lemma *lexordp-eq-antisym*:

assumes *lexordp-eq xs ys lexordp-eq ys xs*
shows $xs = ys$
<proof>

lemma *lexordp-eq-trans*:

assumes *lexordp-eq xs ys and lexordp-eq ys zs*
shows *lexordp-eq xs zs*
<proof>

lemma *lexordp-trans*:

assumes *lexordp xs ys lexordp ys zs*
shows *lexordp xs zs*
<proof>

lemma *lexordp-linear*: $lexordp\ xs\ ys \vee xs = ys \vee lexordp\ ys\ xs$

<proof>

lemma *lexordp-conv-lexordp-eq*: $lexordp\ xs\ ys \longleftrightarrow lexordp-eq\ xs\ ys \wedge \neg lexordp-eq\ ys\ xs$

(is ?lhs \longleftrightarrow ?rhs)
<proof>

lemma *lexordp-eq-conv-lexord*: $lexordp-eq\ xs\ ys \longleftrightarrow xs = ys \vee lexordp\ xs\ ys$

<proof>

lemma *lexordp-eq-linear*: $lexordp-eq\ xs\ ys \vee lexordp-eq\ ys\ xs$

<proof>

lemma *lexordp-linorder*: *class.linorder lexordp-eq lexordp*

<proof>

end

66.4.3 Lexicographic combination of measure functions

These are useful for termination proofs

definition $\text{measures } fs = \text{inv-image } (\text{lex less-than}) (\%a. \text{map } (\%f. f \ a) \ fs)$

lemma $\text{wf-measures[simp]}: \text{wf } (\text{measures } fs)$
 $\langle \text{proof} \rangle$

lemma $\text{in-measures[simp]}:$
 $(x, y) \in \text{measures } [] = \text{False}$
 $(x, y) \in \text{measures } (f \# fs)$
 $= (f \ x < f \ y \vee (f \ x = f \ y \wedge (x, y) \in \text{measures } fs))$
 $\langle \text{proof} \rangle$

lemma $\text{measures-less}: f \ x < f \ y \implies (x, y) \in \text{measures } (f \# fs)$
 $\langle \text{proof} \rangle$

lemma $\text{measures-lesseq}: f \ x \leq f \ y \implies (x, y) \in \text{measures } fs \implies (x, y) \in \text{measures } (f \# fs)$
 $\langle \text{proof} \rangle$

66.4.4 Lifting Relations to Lists: one element

definition $\text{listrel1} :: ('a \times 'a) \text{ set} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ set}$ **where**
 $\text{listrel1 } r = \{(xs, ys) \mid$

$\exists us \ z \ z' \ vs. \ xs = us @ z \# vs \wedge (z, z') \in r \wedge ys = us @ z' \# vs\}$

lemma $\text{listrel1I}:$
 $\llbracket (x, y) \in r; \ xs = us @ x \# vs; \ ys = us @ y \# vs \rrbracket \implies$
 $(xs, ys) \in \text{listrel1 } r$
 $\langle \text{proof} \rangle$

lemma $\text{listrel1E}:$
 $\llbracket (xs, ys) \in \text{listrel1 } r;$
 $\quad !!x \ y \ us \ vs. \llbracket (x, y) \in r; \ xs = us @ x \# vs; \ ys = us @ y \# vs \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma $\text{not-Nil-listrel1} \ [\text{iff}]: ([], xs) \notin \text{listrel1 } r$
 $\langle \text{proof} \rangle$

lemma $\text{not-listrel1-Nil} \ [\text{iff}]: (xs, []) \notin \text{listrel1 } r$
 $\langle \text{proof} \rangle$

lemma $\text{Cons-listrel1-Cons} \ [\text{iff}]:$
 $(x \# xs, y \# ys) \in \text{listrel1 } r \longleftrightarrow$
 $(x, y) \in r \wedge xs = ys \vee x = y \wedge (xs, ys) \in \text{listrel1 } r$
 $\langle \text{proof} \rangle$

lemma $\text{listrel1I1}: (x, y) \in r \implies (x \# xs, y \# xs) \in \text{listrel1 } r$
 $\langle \text{proof} \rangle$

lemma *listrel1I2*: $(xs, ys) \in \text{listrel1 } r \implies (x \# xs, x \# ys) \in \text{listrel1 } r$
 $\langle \text{proof} \rangle$

lemma *append-listrel1I*:
 $(xs, ys) \in \text{listrel1 } r \wedge us = vs \vee xs = ys \wedge (us, vs) \in \text{listrel1 } r$
 $\implies (xs @ us, ys @ vs) \in \text{listrel1 } r$
 $\langle \text{proof} \rangle$

lemma *Cons-listrel1E1*[*elim!*]:
assumes $(x \# xs, ys) \in \text{listrel1 } r$
and $\bigwedge y. ys = y \# xs \implies (x, y) \in r \implies R$
and $\bigwedge zs. ys = x \# zs \implies (xs, zs) \in \text{listrel1 } r \implies R$
shows R
 $\langle \text{proof} \rangle$

lemma *Cons-listrel1E2*[*elim!*]:
assumes $(xs, y \# ys) \in \text{listrel1 } r$
and $\bigwedge x. xs = x \# ys \implies (x, y) \in r \implies R$
and $\bigwedge zs. xs = y \# zs \implies (zs, ys) \in \text{listrel1 } r \implies R$
shows R
 $\langle \text{proof} \rangle$

lemma *snoc-listrel1-snoc-iff*:
 $(xs @ [x], ys @ [y]) \in \text{listrel1 } r$
 $\longleftrightarrow (xs, ys) \in \text{listrel1 } r \wedge x = y \vee xs = ys \wedge (x, y) \in r$ (**is** $?L \longleftrightarrow ?R$)
 $\langle \text{proof} \rangle$

lemma *listrel1-eq-len*: $(xs, ys) \in \text{listrel1 } r \implies \text{length } xs = \text{length } ys$
 $\langle \text{proof} \rangle$

lemma *listrel1-mono*:
 $r \subseteq s \implies \text{listrel1 } r \subseteq \text{listrel1 } s$
 $\langle \text{proof} \rangle$

lemma *listrel1-converse*: $\text{listrel1 } (r^{-1}) = (\text{listrel1 } r)^{-1}$
 $\langle \text{proof} \rangle$

lemma *in-listrel1-converse*:
 $(x, y) \in \text{listrel1 } (r^{-1}) \longleftrightarrow (x, y) \in (\text{listrel1 } r)^{-1}$
 $\langle \text{proof} \rangle$

lemma *listrel1-iff-update*:
 $(xs, ys) \in (\text{listrel1 } r)$
 $\longleftrightarrow (\exists y n. (xs ! n, y) \in r \wedge n < \text{length } xs \wedge ys = xs[n:=y])$ (**is** $?L \longleftrightarrow ?R$)
 $\langle \text{proof} \rangle$

Accessible part and wellfoundedness:

lemma *Cons-acc-listrel1I* [*intro!*]:

$x \in \text{Wellfounded.acc } r \implies xs \in \text{Wellfounded.acc } (\text{listrel1 } r) \implies (x \# xs) \in \text{Wellfounded.acc } (\text{listrel1 } r)$
 $\langle \text{proof} \rangle$

lemma *lists-accD*: $xs \in \text{lists } (\text{Wellfounded.acc } r) \implies xs \in \text{Wellfounded.acc } (\text{listrel1 } r)$
 $\langle \text{proof} \rangle$

lemma *lists-accI*: $xs \in \text{Wellfounded.acc } (\text{listrel1 } r) \implies xs \in \text{lists } (\text{Wellfounded.acc } r)$
 $\langle \text{proof} \rangle$

lemma *wf-listrel1-iff[simp]*: $\text{wf}(\text{listrel1 } r) = \text{wf } r$
 $\langle \text{proof} \rangle$

66.4.5 Lifting Relations to Lists: all elements

inductive-set

listrel :: ('a × 'b) set \Rightarrow ('a list × 'b list) set
 for *r* :: ('a × 'b) set

where

Nil: $([], []) \in \text{listrel } r$
 $| \text{Cons}$: $\llbracket (x, y) \in r; (xs, ys) \in \text{listrel } r \rrbracket \implies (x \# xs, y \# ys) \in \text{listrel } r$

inductive-cases *listrel-Nil1* [*elim!*]: $([], xs) \in \text{listrel } r$

inductive-cases *listrel-Nil2* [*elim!*]: $(xs, []) \in \text{listrel } r$

inductive-cases *listrel-Cons1* [*elim!*]: $(y \# ys, xs) \in \text{listrel } r$

inductive-cases *listrel-Cons2* [*elim!*]: $(xs, y \# ys) \in \text{listrel } r$

lemma *listrel-eq-len*: $(xs, ys) \in \text{listrel } r \implies \text{length } xs = \text{length } ys$
 $\langle \text{proof} \rangle$

lemma *listrel-iff-zip* [*code-unfold*]: $(xs, ys) \in \text{listrel } r \longleftrightarrow \text{length } xs = \text{length } ys \wedge (\forall (x, y) \in \text{set}(\text{zip } xs \text{ } ys). (x, y) \in r) \text{ (is } ?L \longleftrightarrow ?R)$
 $\langle \text{proof} \rangle$

lemma *listrel-iff-nth*: $(xs, ys) \in \text{listrel } r \longleftrightarrow \text{length } xs = \text{length } ys \wedge (\forall n < \text{length } xs. (xs!n, ys!n) \in r) \text{ (is } ?L \longleftrightarrow ?R)$
 $\langle \text{proof} \rangle$

lemma *listrel-mono*: $r \subseteq s \implies \text{listrel } r \subseteq \text{listrel } s$
 $\langle \text{proof} \rangle$

lemma *listrel-subset*:

assumes $r \subseteq A \times A$ **shows** $\text{listrel } r \subseteq \text{lists } A \times \text{lists } A$
 $\langle \text{proof} \rangle$

lemma *listrel-refl-on*:

assumes *refl-on A r* **shows** *refl-on (lists A) (listrel r)*
 $\langle \text{proof} \rangle$

lemma *listrel-sym*: $\text{sym } r \implies \text{sym } (\text{listrel } r)$
 $\langle \text{proof} \rangle$

lemma *listrel-trans*:
assumes *trans r* **shows** *trans (listrel r)*
 $\langle \text{proof} \rangle$

theorem *equiv-listrel*: $\text{equiv } A \ r \implies \text{equiv } (\text{lists } A) \ (\text{listrel } r)$
 $\langle \text{proof} \rangle$

lemma *listrel-rtrancl-refl*[*iff*]: $(x, x) \in \text{listrel}(r^*)$
 $\langle \text{proof} \rangle$

lemma *listrel-rtrancl-trans*:
 $\llbracket (x, y) \in \text{listrel}(r^*); (y, z) \in \text{listrel}(r^*) \rrbracket \implies (x, z) \in \text{listrel}(r^*)$
 $\langle \text{proof} \rangle$

lemma *listrel-Nil* [*simp*]: $\text{listrel } r \text{ “ } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *listrel-Cons*:
 $\text{listrel } r \text{ “ } \{x \# xs\} = \text{set-Cons } (r \text{ “ } \{x\}) \ (\text{listrel } r \text{ “ } \{xs\})$
 $\langle \text{proof} \rangle$

Relating *listrel1*, *listrel* and closures:

lemma *listrel1-rtrancl-subset-rtrancl-listrel1*: $\text{listrel1 } (r^*) \subseteq (\text{listrel1 } r)^*$
 $\langle \text{proof} \rangle$

lemma *rtrancl-listrel1-eq-len*: $(x, y) \in (\text{listrel1 } r)^* \implies \text{length } x = \text{length } y$
 $\langle \text{proof} \rangle$

lemma *rtrancl-listrel1-ConsI1*:
 $(x, y) \in (\text{listrel1 } r)^* \implies (x \# xs, x \# ys) \in (\text{listrel1 } r)^*$
 $\langle \text{proof} \rangle$

lemma *rtrancl-listrel1-ConsI2*:
 $(x, y) \in r^* \implies (x, y) \in (\text{listrel1 } r)^* \implies (x \# xs, y \# ys) \in (\text{listrel1 } r)^*$
 $\langle \text{proof} \rangle$

lemma *listrel1-subset-listrel*:
 $r \subseteq r' \implies \text{refl } r' \implies \text{listrel1 } r \subseteq \text{listrel}(r')$
 $\langle \text{proof} \rangle$

lemma *listrel-reflcl-if-listrel1*:
 $(x, y) \in \text{listrel1 } r \implies (x, y) \in \text{listrel}(r^*)$
 $\langle \text{proof} \rangle$

lemma *listrel-rtrancl-eq-rtrancl-listrel1*: $\text{listrel } (r^*) = (\text{listrel1 } r)^*$
 ⟨proof⟩

lemma *rtrancl-listrel1-if-listrel*:
 $(xs, ys) \in \text{listrel } r \implies (xs, ys) \in (\text{listrel1 } r)^*$
 ⟨proof⟩

lemma *listrel-subset-rtrancl-listrel1*: $\text{listrel } r \subseteq (\text{listrel1 } r)^*$
 ⟨proof⟩

66.5 Size function

lemma *[measure-function]*: $\text{is-measure } f \implies \text{is-measure } (\text{size-list } f)$
 ⟨proof⟩

lemma *[measure-function]*: $\text{is-measure } f \implies \text{is-measure } (\text{size-option } f)$
 ⟨proof⟩

lemma *size-list-estimation[termination-simp]*:
 $x \in \text{set } xs \implies y < f x \implies y < \text{size-list } f xs$
 ⟨proof⟩

lemma *size-list-estimation'[termination-simp]*:
 $x \in \text{set } xs \implies y \leq f x \implies y \leq \text{size-list } f xs$
 ⟨proof⟩

lemma *size-list-map[simp]*: $\text{size-list } f (\text{map } g xs) = \text{size-list } (f \circ g) xs$
 ⟨proof⟩

lemma *size-list-append[simp]*: $\text{size-list } f (xs @ ys) = \text{size-list } f xs + \text{size-list } f ys$
 ⟨proof⟩

lemma *size-list-pointwise[termination-simp]*:
 $(\bigwedge x. x \in \text{set } xs \implies f x \leq g x) \implies \text{size-list } f xs \leq \text{size-list } g xs$
 ⟨proof⟩

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]*:
 $\text{List.bind } [] f = []$
 $\text{List.bind } (x \# xs) f = f x @ \text{List.bind } xs f$
 ⟨proof⟩

lemma *list-bind-cong [fundef-cong]*:

assumes $xs = ys \ (\bigwedge x. x \in \text{set } xs \implies f\ x = g\ x)$
shows $\text{List.bind } xs\ f = \text{List.bind } ys\ g$
 $\langle \text{proof} \rangle$

lemma *set-list-bind*: $\text{set } (\text{List.bind } xs\ f) = (\bigcup_{x \in \text{set } xs} \text{set } (f\ x))$
 $\langle \text{proof} \rangle$

66.7 Code generation

66.7.1 Counterparts for set-related operations

context
begin

qualified definition *member* :: $\langle 'a\ \text{list} \Rightarrow 'a \Rightarrow \text{bool} \rangle$ — only for code generation
where *member-iff* [*code-abbrev*, *simp*]: $\langle \text{member } xs\ x \longleftrightarrow x \in \text{set } xs \rangle$

Use *member* only for generating executable code. Otherwise use $x \in \text{set } xs$ instead — it is much easier to reason about.

qualified lemma *member-code* [*code*, *no-atp*]:

$\langle \text{member } []\ y \longleftrightarrow \text{False} \rangle$
 $\langle \text{member } (x \# xs)\ y \longleftrightarrow x = y \vee \text{member } xs\ y \rangle$
 $\langle \text{proof} \rangle$ **lemma** *Collect-member* [*code-unfold*, *no-atp*]: — make preprocessor setup
confluent
 $\langle \{x. \text{List.member } xs\ x \wedge P\ x\} = \text{Set.filter } P\ (\text{set } xs) \rangle$
 $\langle \text{proof} \rangle$ **lemma** *Collect-pair-member* [*code-unfold*, *no-atp*]: — make preprocessor
setup confluent
 $\langle \{(x, y). \text{List.member } xs\ (x, y) \wedge P\ x\ y\} = \text{Set.filter } (\lambda(x, y). P\ x\ y)\ (\text{set } xs) \rangle$
 $\langle \text{proof} \rangle$ **lemma** *Collect-triple-member* [*code-unfold*, *no-atp*]: — make preprocessor
setup confluent
 $\langle \{(x, y, z). \text{List.member } xs\ (x, y, z) \wedge P\ x\ y\ z\} = \text{Set.filter } (\lambda(x, y, z). P\ x\ y\ z)\ (\text{set } xs) \rangle$
 $\langle \text{proof} \rangle$

end

lemma *list-all-iff* [*code-abbrev*]:
 $\langle \text{list-all } P\ xs \longleftrightarrow \text{Ball } (\text{set } xs)\ P \rangle$
 $\langle \text{proof} \rangle$

definition *list-ex* :: $\langle ('a \Rightarrow \text{bool}) \Rightarrow 'a\ \text{list} \Rightarrow \text{bool} \rangle$
where *list-ex-iff* [*code-abbrev*]: $\langle \text{list-ex } P\ xs \longleftrightarrow \text{Bex } (\text{set } xs)\ P \rangle$

definition *list-ex1* :: $\langle ('a \Rightarrow \text{bool}) \Rightarrow 'a\ \text{list} \Rightarrow \text{bool} \rangle$
where *list-ex1-iff* [*code-abbrev*]: $\langle \text{list-ex1 } P\ xs \longleftrightarrow \text{Set.can-select } P\ (\text{set } xs) \rangle$

Usually you should prefer $\forall x \in \text{set } xs$, $\exists x \in \text{set } xs$ and $\exists! x. x \in \text{set } xs \wedge$ - over *list-all*, *list-ex* and *list-ex1* in specifications.

lemma *list-all-Nil-iff* [*code*, *no-atp*]:

$\langle \text{list-all } P [] \longleftrightarrow \text{True} \rangle$
 $\langle \text{proof} \rangle$

lemma *list-all-Cons-iff* [*code, no-atp*]:
 $\langle \text{list-all } P (x \# xs) \longleftrightarrow P x \wedge \text{list-all } P xs \rangle$
 $\langle \text{proof} \rangle$

lemma *list-ex-Nil-iff* [*simp, code, no-atp*]:
 $\langle \text{list-ex } P [] \longleftrightarrow \text{False} \rangle$
 $\langle \text{proof} \rangle$

lemma *list-ex-Cons-iff* [*simp, code, no-atp*]:
 $\langle \text{list-ex } P (x \# xs) \longleftrightarrow P x \vee \text{list-ex } P xs \rangle$
 $\langle \text{proof} \rangle$

lemma *list-ex1-Nil-iff* [*simp, code, no-atp*]:
 $\langle \text{list-ex1 } P [] \longleftrightarrow \text{False} \rangle$
 $\langle \text{proof} \rangle$

lemma *list-ex1-Cons-iff* [*simp, code, no-atp*]:
 $\langle \text{list-ex1 } P (x \# xs) \longleftrightarrow (\text{if } P x \text{ then list-all } (\lambda y. \neg P y \vee x = y) xs \text{ else list-ex1 } P xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *list-all-append* [*simp*]:
 $\langle \text{list-all } P (xs @ ys) \longleftrightarrow \text{list-all } P xs \wedge \text{list-all } P ys \rangle$
 $\langle \text{proof} \rangle$

lemma *list-ex-append* [*simp*]:
 $\langle \text{list-ex } P (xs @ ys) \longleftrightarrow \text{list-ex } P xs \vee \text{list-ex } P ys \rangle$
 $\langle \text{proof} \rangle$

lemma *list-all-rev* [*simp*]:
 $\langle \text{list-all } P (\text{rev } xs) \longleftrightarrow \text{list-all } P xs \rangle$
 $\langle \text{proof} \rangle$

lemma *list-ex-rev* [*simp*]:
 $\langle \text{list-ex } P (\text{rev } xs) \longleftrightarrow \text{list-ex } P xs \rangle$
 $\langle \text{proof} \rangle$

lemma *list-all-length*:
 $\langle \text{list-all } P xs \longleftrightarrow (\forall n < \text{length } xs. P (xs ! n)) \rangle$
 $\langle \text{proof} \rangle$

lemma *list-ex-length*:
 $\langle \text{list-ex } P xs \longleftrightarrow (\exists n < \text{length } xs. P (xs ! n)) \rangle$
 $\langle \text{proof} \rangle$

lemma *list-all-cong* [*fundef-cong*]:

$\langle \text{list-all } f \text{ } xs = \text{list-all } g \text{ } ys \rangle$
if $\langle xs = ys \rangle \langle (\bigwedge x. x \in \text{set } ys \implies f \text{ } x = g \text{ } x) \rangle$
 $\langle \text{proof} \rangle$

lemma *list-ex-cong* [*fundef-cong*]:
 $\langle \text{list-ex } f \text{ } xs = \text{list-ex } g \text{ } ys \rangle$
if $\langle xs = ys \rangle \langle (\bigwedge x. x \in \text{set } ys \implies f \text{ } x = g \text{ } x) \rangle$
 $\langle \text{proof} \rangle$

context
begin

qualified definition *superset* :: $\langle 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \rangle$
where *superset-iff* [*code-abbrev*, *simp*]: $\langle \text{superset } ys \text{ } xs \longleftrightarrow \text{set } xs \subseteq \text{set } ys \rangle$

lemma [*code*, *no-atp*]:
 $\langle \text{superset } xs = \text{list-all } (\lambda x. x \in \text{set } xs) \rangle$
 $\langle \text{proof} \rangle$

end

Executable checks for relations on sets

definition *listrel1p* :: $\langle ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \rangle$ — only for code generation
where $\langle \text{listrel1p } r \text{ } xs \text{ } ys \longleftrightarrow (xs, ys) \in \text{listrel1 } \{(x, y). r \text{ } x \text{ } y\} \rangle$

lemma [*code-unfold*]:
 $\langle (xs, ys) \in \text{listrel1 } r \longleftrightarrow \text{listrel1p } (\lambda x \text{ } y. (x, y) \in r) \text{ } xs \text{ } ys \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle \text{listrel1p } r \text{ } [] \longleftrightarrow \text{False} \rangle$
 $\langle \text{listrel1p } r \text{ } xs \text{ } [] \longleftrightarrow \text{False} \rangle$
 $\langle \text{listrel1p } r \text{ } (x \# xs) \text{ } (y \# ys) \longleftrightarrow$
 $\quad r \text{ } x \text{ } y \wedge xs = ys \vee x = y \wedge \text{listrel1p } r \text{ } xs \text{ } ys \rangle$
 $\langle \text{proof} \rangle$

definition *lexordp* :: $\langle ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \rangle$ — only for code generation
where $\langle \text{lexordp } r \text{ } xs \text{ } ys \longleftrightarrow (xs, ys) \in \text{lexord } \{(x, y). r \text{ } x \text{ } y\} \rangle$

lemma [*code-unfold*]:
 $\langle (xs, ys) \in \text{lexord } r = \text{lexordp } (\lambda x \text{ } y. (x, y) \in r) \text{ } xs \text{ } ys \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle \text{lexordp } r \text{ } xs \text{ } [] \longleftrightarrow \text{False} \rangle$
 $\langle \text{lexordp } r \text{ } [] \text{ } (y \# ys) \longleftrightarrow \text{True} \rangle$
 $\langle \text{lexordp } r \text{ } (x \# xs) \text{ } (y \# ys) \longleftrightarrow$

$r\ x\ y \vee (x = y \wedge \text{lexordp}\ r\ xs\ ys)$
 $\langle \text{proof} \rangle$

Executable intervals

context *preorder*
begin

lemma *forall-less-eq-iff* [*code-unfold*]:
 $\langle (\forall n \leq b. P\ n) \longleftrightarrow (\forall n \in \{..b\}. P\ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *exists-less-eq-iff* [*code-unfold*]:
 $\langle (\exists n \leq b. P\ n) \longleftrightarrow (\exists n \in \{..b\}. P\ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *forall-less-iff* [*code-unfold*]:
 $\langle (\forall n < b. P\ n) \longleftrightarrow (\forall n \in \{..**b\}. P\ n) \rangle**$
 $\langle \text{proof} \rangle$

lemma *exists-less-iff* [*code-unfold*]:
 $\langle (\exists n < b. P\ n) \longleftrightarrow (\exists n \in \{..**b\}. P\ n) \rangle**$
 $\langle \text{proof} \rangle$

lemma *forall-greater-eq-iff* [*code-unfold*]:
 $\langle (\forall n \geq a. P\ n) \longleftrightarrow (\forall n \in \{a.. \}. P\ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *exists-greater-eq-iff* [*code-unfold*]:
 $\langle (\exists n \geq a. P\ n) \longleftrightarrow (\exists n \in \{a.. \}. P\ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *forall-greater-iff* [*code-unfold*]:
 $\langle (\forall n > a. P\ n) \longleftrightarrow (\forall n \in \{a<.. \}. P\ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *exists-greater-iff* [*code-unfold*]:
 $\langle (\exists n > a. P\ n) \longleftrightarrow (\exists n \in \{a<.. \}. P\ n) \rangle$
 $\langle \text{proof} \rangle$

end

class *interval* = *linorder* + *comm-semiring-1-cancel* +
assumes *finite-atLeastAtMost*: $\langle \text{finite } \{a..b\} \rangle$
assumes *dec-less-imp-less-eq*: $\langle a - 1 < b \implies a \leq b \rangle$
assumes *less-inc-imp-less-eq*: $\langle a < b + 1 \implies a \leq b \rangle$
assumes *dec-greater-eq-self-imp-bot*: $\langle a \leq a - 1 \implies a \leq c \rangle$
assumes *inc-less-eq-self-imp-top*: $\langle b + 1 \leq b \implies d \leq b \rangle$
begin

context
begin

qualified lemma *less-imp-less-eq-dec*:

$\langle c < b \implies a < b \implies a \leq b - 1 \rangle$

$\langle \text{proof} \rangle$ **lemma** *less-imp-in-less-eq*:

$\langle a < c \implies a < b \implies a + 1 \leq b \rangle$

$\langle \text{proof} \rangle$ **lemma** *less-eq-dec-imp-less*:

$\langle c < b \implies a \leq b - 1 \implies a < b \rangle$

$\langle \text{proof} \rangle$ **lemma** *inc-less-eq-imp-less*:

$\langle a < c \implies a + 1 \leq b \implies a < b \rangle$

$\langle \text{proof} \rangle$ **definition** *interval* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \text{ list} \rangle$ — only for code generation

where *interval-eq*: $\langle \text{interval } a \ b = \text{sorted-list-of-set } \{a..b\} \rangle$

qualified lemma *set-interval-eq* [simp]:

$\langle \text{set } (\text{interval } a \ b) = \{a..b\} \rangle$

$\langle \text{proof} \rangle$ **lemma** *distinct-interval* [simp]:

$\langle \text{distinct } (\text{interval } a \ b) \rangle$

$\langle \text{proof} \rangle$ **lemma** *interval-code* [code]:

$\langle \text{interval } a \ b = (\text{if } a < b \text{ then } a \ \# \ \text{interval } (a + 1) \ b \text{ else if } a = b \text{ then } [a] \text{ else } []) \rangle$

$\langle \text{proof} \rangle$ **lemma** *atLeastAtMost-eq-interval* [code]:

$\langle \{a..b\} = \text{set } (\text{interval } a \ b) \rangle$

$\langle \text{proof} \rangle$ **lemma** *atLeastLessThan-eq-interval* [code]:

$\langle \{a..<b\} = (\text{let } d = b - 1 \text{ in if } d < b \text{ then set } (\text{interval } a \ d) \text{ else } \{\}) \rangle$

$\langle \text{proof} \rangle$ **lemma** *greaterThanAtMost-eq-interval* [code]:

$\langle \{a<..b\} = (\text{let } c = a + 1 \text{ in if } a < c \text{ then set } (\text{interval } c \ b) \text{ else } \{\}) \rangle$

$\langle \text{proof} \rangle$ **lemma** *greaterThanLessThan-eq-interval* [code]:

$\langle \{a<..**b\} = (\text{let } c = a + 1; d = b - 1 \text{ in if } a < c \ \wedge \ d < b \text{ then set } (\text{interval } c \ d) \text{ else } \{\}) \rangle**$

$\langle \text{proof} \rangle$ **definition** *all-interval* :: $\langle ('a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool} \rangle$ — only for code generation

where *all-interval-iff* [code-post, simp]: $\langle \text{all-interval } P \ a \ b \longleftrightarrow (\forall n \in \{a..b\}. P \ n) \rangle$

qualified lemma *all-interval-code* [code]:

$\langle \text{all-interval } P \ a \ b \longleftrightarrow ((a < b \longrightarrow P \ a \ \wedge \ \text{all-interval } P \ (a + 1) \ b) \ \wedge \ (a = b \longrightarrow P \ a)) \rangle$

$\langle \text{proof} \rangle$ **lemma** *forall-atLeastAtMost-iff* [code-unfold]:

$\langle (\forall n \in \{a..b\}. P \ n) \longleftrightarrow \text{all-interval } P \ a \ b \rangle$

$\langle \text{proof} \rangle$ **lemma** *exists-atLeastAtMost-iff* [code-unfold]:

$\langle (\exists n \in \{a..b\}. P \ n) \longleftrightarrow \neg \text{all-interval } (\text{Not } \circ P) \ a \ b \rangle$

$\langle \text{proof} \rangle$ **lemma** *forall-atLeastLessThan-iff* [code-unfold]:

$\langle (\forall n \in \{a..**b\}. P \ n) \longleftrightarrow (\text{let } d = b - 1 \text{ in } d < b \longrightarrow \text{all-interval } P \ a \ d) \rangle**$

$\langle \text{proof} \rangle$ **lemma** *exists-atLeastLessThan-iff* [code-unfold]:

$\langle (\exists n \in \{a..**b\}. P \ n) \longleftrightarrow (\text{let } d = b - 1 \text{ in } d < b \ \wedge \ \neg \text{all-interval } (\text{Not } \circ P) \ a \ d) \rangle**$

$\langle \text{proof} \rangle$ **lemma** *forall-greaterThanAtMost-iff* [code-unfold]:

$\langle (\forall n \in \{a<..**b\}. P \ n) \longleftrightarrow (\text{let } c = a + 1 \text{ in } a < c \longrightarrow \text{all-interval } P \ c \ b) \rangle**$

$\langle \text{proof} \rangle$ **lemma** *exists-greaterThanAtMost-iff* [code-unfold]:
 $\langle (\exists n \in \{a < .. b\}. P\ n) \longleftrightarrow (\text{let } c = a + 1 \text{ in } a < c \wedge \neg \text{all-interval } (\text{Not} \circ P)\ c\ b) \rangle$
 $\langle \text{proof} \rangle$ **lemma** *forall-greaterThanLessThan-iff* [code-unfold]:
 $\langle (\forall n \in \{a < .. < b\}. P\ n) \longleftrightarrow (\text{let } c = a + 1; d = b - 1 \text{ in } a < c \longrightarrow d < b \longrightarrow \text{all-interval } P\ c\ d) \rangle$
 $\langle \text{proof} \rangle$ **lemma** *exists-greaterThanLessThan-iff* [code-unfold]:
 $\langle (\exists n \in \{a < .. < b\}. P\ n) \longleftrightarrow (\text{let } c = a + 1; d = b - 1 \text{ in } a < c \wedge d < b \wedge \neg \text{all-interval } (\text{Not} \circ P)\ c\ d) \rangle$
 $\langle \text{proof} \rangle$

end

end

class *interval-top* = *interval* + *order-top*
begin

lemma *atLeast-eq-atLeastAtMost-top* [code, code-unfold]:
 $\langle \{a ..\} = \{a .. \text{top}\} \rangle$
 $\langle \text{proof} \rangle$

lemma *greaterThan-eq-greaterThanAtMost-top* [code, code-unfold]:
 $\langle \{a < ..\} = \{a < .. \text{top}\} \rangle$
 $\langle \text{proof} \rangle$

end

class *interval-bot* = *interval* + *order-bot*
begin

lemma *atMost-eq-atLeastAtMost-bot* [code, code-unfold]:
 $\langle \{..b\} = \{\text{bot} .. b\} \rangle$
 $\langle \text{proof} \rangle$

lemma *lessThan-eq-atLeastLessThan-bot* [code, code-unfold]:
 $\langle \{..<b\} = \{\text{bot} .. <b\} \rangle$
 $\langle \text{proof} \rangle$

end

instance *nat* :: *interval-bot*
 $\langle \text{proof} \rangle$

instance *int* :: *interval*
 $\langle \text{proof} \rangle$

context
begin

qualified lemma *interval-eq-upt* [simp]:
 $\langle \text{List.interval } m \ n = [m..<\text{Suc } n] \rangle$
 $\langle \text{proof} \rangle$ **lemma** *interval-eq-upto* [simp]:
 $\langle \text{List.interval } i \ k = [i..k] \rangle$
 $\langle \text{proof} \rangle$

end

66.7.2 Special implementations

context
begin

qualified definition *map-tailrec-rev* :: $\langle ('a \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow 'b \text{ list} \rangle$ —
only for code generation
where *map-tailrec-rev* [simp]: $\langle \text{map-tailrec-rev } f \ as \ bs = \text{rev } (\text{map } f \ as) \ @ \ bs \rangle$

Optional tail recursive version of *map*. Can avoid stack overflow in some target languages. Do not use for proving.

qualified lemma *map-tailrec-rev-code* [code, no-atp]:
 $\langle \text{map-tailrec-rev } f \ [] \ bs = bs \rangle$
 $\langle \text{map-tailrec-rev } f \ (a \ # \ as) \ bs = \text{map-tailrec-rev } f \ as \ (f \ a \ # \ bs) \rangle$
 $\langle \text{proof} \rangle$ **definition** *map-tailrec* :: $\langle ('a \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \rangle$ — only for code generation
where *map-tailrec-eq* [simp]: $\langle \text{map-tailrec} = \text{map} \rangle$

qualified lemma *map-tailrec-code* [code, no-atp]:
 $\langle \text{map-tailrec } f \ as = \text{rev } (\text{map-tailrec-rev } f \ as \ []) \rangle$
 $\langle \text{proof} \rangle$

Potential code equation:

qualified lemma *map-eq-map-tailrec*:
 $\langle \text{map} = \text{map-tailrec} \rangle$
 $\langle \text{proof} \rangle$

end

definition *map-filter* :: $\langle ('a \Rightarrow 'b \text{ option}) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \rangle$
where [code-post]: *map-filter* $f \ xs = \text{map } (\text{the} \circ f) \ (\text{filter } (\lambda x. f \ x \neq \text{None}) \ xs)$

Operation *map-filter* avoids intermediate lists on execution – do not use for proving.

lemma *map-filter-simps* [simp, code, no-atp]:
 $\langle \text{map-filter } f \ [] = [] \rangle$
 $\langle \text{map-filter } f \ (x \ # \ xs) = (\text{case } f \ x \ \text{of } \text{None} \Rightarrow \text{map-filter } f \ xs \mid \text{Some } y \Rightarrow y \ # \text{map-filter } f \ xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *map-filter-map-filter* [code-unfold]:
 $\langle \text{map } f \text{ (filter } P \text{ } xs) = \text{map-filter } (\lambda x. \text{ if } P \text{ } x \text{ then Some } (f \text{ } x) \text{ else None) } xs \rangle$
 $\langle \text{proof} \rangle$

hide-const (open) *map-filter*

66.7.3 Operations for optimization and efficiency

context
begin

qualified definition *null* :: $\langle 'a \text{ list} \Rightarrow \text{bool} \rangle$ — only for code generation
where *null-iff* [code-abbrev, simp]: $\langle \text{null } xs \longleftrightarrow xs = [] \rangle$

qualified lemma *null-code* [code, no-atp]:
 $\langle \text{null } [] \longleftrightarrow \text{True} \rangle$
 $\langle \text{null } (x \# xs) \longleftrightarrow \text{False} \rangle$
 $\langle \text{proof} \rangle$ **lemma** *equal-Nil-null* [code-unfold, no-atp]:
 $\langle \text{HOL.equal } xs \ [] \longleftrightarrow \text{null } xs \rangle$
 $\langle \text{HOL.equal } [] = \text{null} \rangle$
 $\langle \text{proof} \rangle$ **definition** *length-tailrec* :: $\langle 'a \text{ list} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$ — only for code generation
where *length-tailrec-eq* [simp]: $\langle \text{length-tailrec } xs = (+) (\text{length } xs) \rangle$

optimized code (tail-recursive) for *length*

qualified lemma *length-tailrec-code* [code, no-atp]:
 $\langle \text{length-tailrec } [] \text{ } n = n \rangle$
 $\langle \text{length-tailrec } (x \# xs) \text{ } n = \text{length-tailrec } xs \text{ } (\text{Suc } n) \rangle$
 $\langle \text{proof} \rangle$ **lemma** *length-code* [code, no-atp]:
 $\langle \text{length } xs = \text{length-tailrec } xs \text{ } 0 \rangle$
 $\langle \text{proof} \rangle$ **definition** *maps* :: $\langle 'a \Rightarrow 'b \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \rangle$ — only for code generation
where *maps-eq* [code-abbrev, simp]: $\langle \text{maps } f \text{ } xs = \text{concat } (\text{map } f \text{ } xs) \rangle$

Operation *maps* avoids intermediate lists on execution – do not use for proving.

qualified lemma *maps-code* [code, no-atp]:
 $\langle \text{maps } f \text{ } [] = [] \rangle$
 $\langle \text{maps } f \text{ } (x \# xs) = f \text{ } x \text{ @ maps } f \text{ } xs \rangle$
 $\langle \text{proof} \rangle$

end

66.7.4 Implementation of sets by lists

lemma *is-empty-set* [code]:
 $\text{Set.is-empty } (\text{set } xs) \longleftrightarrow \text{List.null } xs$
 $\langle \text{proof} \rangle$

lemma *empty-set* [code]:

$\{\} = \text{set } []$
 $\langle \text{proof} \rangle$

lemma *UNIV-coset* [code]:

$\text{UNIV} = \text{List.coset } []$
 $\langle \text{proof} \rangle$

lemma *compl-set* [code]:

$-\text{ set } xs = \text{List.coset } xs$
 $\langle \text{proof} \rangle$

lemma *compl-coset* [code]:

$-\text{ List.coset } xs = \text{set } xs$
 $\langle \text{proof} \rangle$

lemma [code]:

$x \in \text{set } xs \longleftrightarrow \text{List.member } xs \ x$
 $x \in \text{List.coset } xs \longleftrightarrow \neg \text{List.member } xs \ x$
 $\langle \text{proof} \rangle$

lemma *insert-code* [code]:

$\text{insert } x \ (\text{set } xs) = \text{set } (\text{List.insert } x \ xs)$
 $\text{insert } x \ (\text{List.coset } xs) = \text{List.coset } (\text{removeAll } x \ xs)$
 $\langle \text{proof} \rangle$

lemma *remove-code* [code]:

$\text{Set.remove } x \ (\text{set } xs) = \text{set } (\text{removeAll } x \ xs)$
 $\text{Set.remove } x \ (\text{List.coset } xs) = \text{List.coset } (\text{List.insert } x \ xs)$
 $\langle \text{proof} \rangle$

lemma *filter-set* [code]:

$\text{Set.filter } P \ (\text{set } xs) = \text{set } (\text{filter } P \ xs)$
 $\langle \text{proof} \rangle$

lemma *image-set* [code]:

$\text{image } f \ (\text{set } xs) = \text{set } (\text{map } f \ xs)$
 $\langle \text{proof} \rangle$

lemma *subset-code* [code]:

$\text{set } xs \subseteq B \longleftrightarrow (\forall x \in \text{set } xs. \ x \in B)$
 $A \subseteq \text{List.coset } ys \longleftrightarrow (\forall y \in \text{set } ys. \ y \notin A)$
 $\text{List.coset } [] \subseteq \text{set } [] \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *Ball-set* [code]:

$\text{Ball } (\text{set } xs) \ P \longleftrightarrow \text{list-all } P \ xs$
 $\langle \text{proof} \rangle$

lemma *Bex-set* [code]:

$Bex\ (set\ xs)\ P \longleftrightarrow list\text{-}ex\ P\ xs$

⟨proof⟩

lemma *card-set* [code]:

$card\ (set\ xs) = length\ (remdups\ xs)$

⟨proof⟩

lemma *the-elem-set* [code]:

$the\text{-}elem\ (set\ [x]) = x$

⟨proof⟩

lemma *Pow-set* [code]:

$Pow\ (set\ []) = \{\{\}\}$

$Pow\ (set\ (x\ \# \ xs)) = (let\ A = Pow\ (set\ xs)\ in\ A \cup insert\ x\ \text{'}\ A)$

⟨proof⟩

lemma *these-set-code* [code]:

$\langle Option.\text{these}\ (set\ xs) = set\ (List.\text{map-filter}\ (\lambda x.\ x)\ xs) \rangle$

⟨proof⟩

lemma *image-filter-set-eq* [code]:

$\langle Option.\text{image-filter}\ f\ (set\ xs) = set\ (List.\text{map-filter}\ f\ xs) \rangle$

⟨proof⟩

lemma *can-select-set-list-ex1* [code]:

$Set.\text{can-select}\ P\ (set\ A) = list\text{-}ex1\ P\ A$

⟨proof⟩

lemma *product-code* [code]:

$Product\text{-}Type.\text{product}\ (set\ xs)\ (set\ ys) = set\ [(x,\ y).\ x \leftarrow xs,\ y \leftarrow ys]$

⟨proof⟩

lemma *Id-on-set* [code]:

$Id\text{-}on\ (set\ xs) = set\ [(x,\ x).\ x \leftarrow xs]$

⟨proof⟩

lemma *Image-code* [code]:

$R\ \text{"}\ S = Option.\text{image-filter}\ (\lambda(x,\ y).\ \text{if}\ x \in S\ \text{then}\ Some\ y\ \text{else}\ None)\ R$

⟨proof⟩

lemma *tranc1-set-ntranc1* [code]:

$tranc1\ (set\ xs) = ntranc1\ (card\ (set\ xs) - 1)\ (set\ xs)$

⟨proof⟩

lemma *set-relcomp* [code]:

$set\ rxs\ O\ set\ yzs = set\ ([fst\ xy,\ snd\ yz).\ xy \leftarrow rxs,\ yz \leftarrow yzs,\ snd\ xy = fst\ yz])$

⟨proof⟩

lemma *wf-set*:
wf (*set xs*) = *acyclic* (*set xs*)
 ⟨*proof*⟩

lemma *wf-code-set* [*code*]:
wf-code (*set xs*) = *acyclic* (*set xs*)
 ⟨*proof*⟩

66.7.5 Pretty lists

⟨*ML*⟩

code-printing
type-constructor *list* \rightarrow
 (*SML*) - *list*
 and (*OCaml*) - *list*
 and (*Haskell*) ![(*-*)]
 and (*Scala*) *List*[(*-*)]
| **constant** *Nil* \rightarrow
 (*SML*) []
 and (*OCaml*) []
 and (*Haskell*) []
 and (*Scala*) !*Nil*
| **class-instance** *list* :: *equal* \rightarrow
 (*Haskell*) –
| **constant** *HOL.equal* :: '*a list* \Rightarrow '*a list* \Rightarrow *bool* \rightarrow
 (*Haskell*) **infix** 4 ==

⟨*ML*⟩

code-reserved
 (*SML*) *list*
 and (*OCaml*) *list*

66.7.6 Use convenient predefined operations

code-printing
constant (@) \rightarrow
 (*SML*) **infixr** 7 @
 and (*OCaml*) **infixr** 6 @
 and (*Haskell*) **infixr** 5 ++
 and (*Scala*) **infixl** 7 ++
| **constant** *map* \rightarrow
 (*Haskell*) *map*
| **constant** *filter* \rightarrow
 (*Haskell*) *filter*
| **constant** *concat* \rightarrow
 (*Haskell*) *concat*
| **constant** *List.maps* \rightarrow
 (*Haskell*) *concatMap*

```

| constant rev  $\rightarrow$ 
  (Haskell) reverse
| constant zip  $\rightarrow$ 
  (Haskell) zip
| constant List.null  $\rightarrow$ 
  (Haskell) null
| constant takeWhile  $\rightarrow$ 
  (Haskell) takeWhile
| constant dropWhile  $\rightarrow$ 
  (Haskell) dropWhile
| constant list-all  $\rightarrow$ 
  (Haskell) all
| constant list-ex  $\rightarrow$ 
  (Haskell) any

```

66.8 Setup for Lifting/Transfer

66.8.1 Transfer rules for the Transfer package

context includes *lifting-syntax*
begin

lemma *tl-transfer* [*transfer-rule*]:
 (*list-all2* *A* \implies *list-all2* *A*) *tl tl*
<proof>

lemma *butlast-transfer* [*transfer-rule*]:
 (*list-all2* *A* \implies *list-all2* *A*) *butlast butlast*
<proof>

lemma *append-transfer* [*transfer-rule*]:
 (*list-all2* *A* \implies *list-all2* *A* \implies *list-all2* *A*) *append append*
<proof>

lemma *rev-transfer* [*transfer-rule*]:
 (*list-all2* *A* \implies *list-all2* *A*) *rev rev*
<proof>

lemma *filter-transfer* [*transfer-rule*]:
 ((*A* \implies (*=*)) \implies *list-all2* *A* \implies *list-all2* *A*) *filter filter*
<proof>

lemma *fold-transfer* [*transfer-rule*]:
 ((*A* \implies *B* \implies *B*) \implies *list-all2* *A* \implies *B* \implies *B*) *fold fold*
<proof>

lemma *foldr-transfer* [*transfer-rule*]:
 ((*A* \implies *B* \implies *B*) \implies *list-all2* *A* \implies *B* \implies *B*) *foldr foldr*
<proof>

lemma *foldl-transfer* [*transfer-rule*]:

$((B \text{====>} A \text{====>} B) \text{====>} B \text{====>} \text{list-all2 } A \text{====>} B) \text{ foldl foldl}$
 $\langle \text{proof} \rangle$

lemma *concat-transfer* [*transfer-rule*]:

$(\text{list-all2 } (\text{list-all2 } A) \text{====>} \text{list-all2 } A) \text{ concat concat}$
 $\langle \text{proof} \rangle$

lemma *drop-transfer* [*transfer-rule*]:

$((=) \text{====>} \text{list-all2 } A \text{====>} \text{list-all2 } A) \text{ drop drop}$
 $\langle \text{proof} \rangle$

lemma *take-transfer* [*transfer-rule*]:

$((=) \text{====>} \text{list-all2 } A \text{====>} \text{list-all2 } A) \text{ take take}$
 $\langle \text{proof} \rangle$

lemma *list-update-transfer* [*transfer-rule*]:

$(\text{list-all2 } A \text{====>} (=) \text{====>} A \text{====>} \text{list-all2 } A) \text{ list-update list-update}$
 $\langle \text{proof} \rangle$

lemma *takeWhile-transfer* [*transfer-rule*]:

$((A \text{====>} (=)) \text{====>} \text{list-all2 } A \text{====>} \text{list-all2 } A) \text{ takeWhile takeWhile}$
 $\langle \text{proof} \rangle$

lemma *dropWhile-transfer* [*transfer-rule*]:

$((A \text{====>} (=)) \text{====>} \text{list-all2 } A \text{====>} \text{list-all2 } A) \text{ dropWhile dropWhile}$
 $\langle \text{proof} \rangle$

lemma *zip-transfer* [*transfer-rule*]:

$(\text{list-all2 } A \text{====>} \text{list-all2 } B \text{====>} \text{list-all2 } (\text{rel-prod } A \text{ } B)) \text{ zip zip}$
 $\langle \text{proof} \rangle$

lemma *product-transfer* [*transfer-rule*]:

$(\text{list-all2 } A \text{====>} \text{list-all2 } B \text{====>} \text{list-all2 } (\text{rel-prod } A \text{ } B)) \text{ List.product List.product}$
 $\langle \text{proof} \rangle$

lemma *product-lists-transfer* [*transfer-rule*]:

$(\text{list-all2 } (\text{list-all2 } A) \text{====>} \text{list-all2 } (\text{list-all2 } A)) \text{ product-lists product-lists}$
 $\langle \text{proof} \rangle$

lemma *insert-transfer* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique* *A*

shows $(A \text{====>} \text{list-all2 } A \text{====>} \text{list-all2 } A) \text{ List.insert List.insert}$
 $\langle \text{proof} \rangle$

lemma *find-transfer* [*transfer-rule*]:

$((A \text{====>} (=)) \text{====>} \text{list-all2 } A \text{====>} \text{rel-option } A) \text{ List.find List.find}$
 $\langle \text{proof} \rangle$

lemma *those-transfer* [transfer-rule]:
 (list-all2 (rel-option P) ==> rel-option (list-all2 P)) those those
 <proof>

lemma *remove1-transfer* [transfer-rule]:
 assumes [transfer-rule]: bi-unique A
 shows (A ==> list-all2 A ==> list-all2 A) remove1 remove1
 <proof>

lemma *removeAll-transfer* [transfer-rule]:
 assumes [transfer-rule]: bi-unique A
 shows (A ==> list-all2 A ==> list-all2 A) removeAll removeAll
 <proof>

lemma *successively-transfer* [transfer-rule]:
 ((A ==> A ==> (=)) ==> list-all2 A ==> (=)) successively successively
 <proof>

lemma *distinct-transfer* [transfer-rule]:
 assumes [transfer-rule]: bi-unique A
 shows (list-all2 A ==> (=)) distinct distinct
 <proof>

lemma *distinct-adj-transfer* [transfer-rule]:
 assumes bi-unique A
 shows (list-all2 A ==> (=)) distinct-adj distinct-adj
 <proof>

lemma *remdups-transfer* [transfer-rule]:
 assumes [transfer-rule]: bi-unique A
 shows (list-all2 A ==> list-all2 A) remdups remdups
 <proof>

lemma *remdups-adj-transfer* [transfer-rule]:
 assumes [transfer-rule]: bi-unique A
 shows (list-all2 A ==> list-all2 A) remdups-adj remdups-adj
 <proof>

lemma *replicate-transfer* [transfer-rule]:
 ((=) ==> A ==> list-all2 A) replicate replicate
 <proof>

lemma *length-transfer* [transfer-rule]:
 (list-all2 A ==> (=)) length length
 <proof>

lemma *rotate1-transfer* [transfer-rule]:
 (list-all2 A ==> list-all2 A) rotate1 rotate1
 <proof>

lemma *rotate-transfer* [transfer-rule]:

$((=) \implies \text{list-all2 } A \implies \text{list-all2 } A) \text{ rotate rotate}$
 $\langle \text{proof} \rangle$

lemma *nths-transfer* [transfer-rule]:

$(\text{list-all2 } A \implies \text{rel-set } (=) \implies \text{list-all2 } A) \text{ nths nths}$
 $\langle \text{proof} \rangle$

lemma *subseqs-transfer* [transfer-rule]:

$(\text{list-all2 } A \implies \text{list-all2 } (\text{list-all2 } A)) \text{ subseqs subseqs}$
 $\langle \text{proof} \rangle$

lemma *partition-transfer* [transfer-rule]:

$((A \implies (=)) \implies \text{list-all2 } A \implies \text{rel-prod } (\text{list-all2 } A) (\text{list-all2 } A))$
 $\text{partition partition}$
 $\langle \text{proof} \rangle$

lemma *lists-transfer* [transfer-rule]:

$(\text{rel-set } A \implies \text{rel-set } (\text{list-all2 } A)) \text{ lists lists}$
 $\langle \text{proof} \rangle$

lemma *set-Cons-transfer* [transfer-rule]:

$(\text{rel-set } A \implies \text{rel-set } (\text{list-all2 } A) \implies \text{rel-set } (\text{list-all2 } A))$
 set-Cons set-Cons
 $\langle \text{proof} \rangle$

lemma *listset-transfer* [transfer-rule]:

$(\text{list-all2 } (\text{rel-set } A) \implies \text{rel-set } (\text{list-all2 } A)) \text{ listset listset}$
 $\langle \text{proof} \rangle$

lemma *null-transfer* [transfer-rule]:

$(\text{list-all2 } A \implies (=)) \text{ List.null List.null}$
 $\langle \text{proof} \rangle$

lemma *list-all-transfer* [transfer-rule]:

$((A \implies (=)) \implies \text{list-all2 } A \implies (=)) \text{ list-all list-all}$
 $\langle \text{proof} \rangle$

lemma *list-ex-transfer* [transfer-rule]:

$((A \implies (=)) \implies \text{list-all2 } A \implies (=)) \text{ list-ex list-ex}$
 $\langle \text{proof} \rangle$

lemma *splice-transfer* [transfer-rule]:

$(\text{list-all2 } A \implies \text{list-all2 } A \implies \text{list-all2 } A) \text{ splice splice}$
 $\langle \text{proof} \rangle$

lemma *shuffles-transfer* [transfer-rule]:

$(\text{list-all2 } A \implies \text{list-all2 } A \implies \text{rel-set } (\text{list-all2 } A)) \text{ shuffles shuffles}$

<proof>

lemma *rtranc1-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)*) *rtranc1 rtranc1*
<proof>

lemma *monotone-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
shows ((*A ==> A ==> (=)*) ==> (*B ==> B ==> (=)*) ==> (*A ==> B ==> (=)*)) *monotone monotone*
<proof>

lemma *fun-ord-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total C*
shows ((*A ==> B ==> (=)*) ==> (*C ==> A ==> (C ==> B) ==> (=)*)) *fun-ord fun-ord*
<proof>

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*))
fun-lub fun-lub
<proof>

end

66.9 Misc

lemma *Ball-set-list-all*:
Ball (set xs) P <=> list-all P xs
<proof>

lemma *Bex-set-list-ex*:
Bex (set xs) P <=> list-ex P xs
<proof>

end

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*]:

$F\ [] = 1$

⟨*proof*⟩

lemma *Cons* [*simp*]:

$F\ (x\ \# \ xs) = x\ * \ F\ xs$

⟨*proof*⟩

lemma *append* [*simp*]:

$F\ (xs\ @ \ ys) = F\ xs\ * \ F\ ys$

⟨*proof*⟩

end

locale *comm-monoid-list* = *comm-monoid* + *monoid-list*

begin

lemma *rev* [*simp*]:

$F\ (rev\ xs) = F\ xs$

⟨*proof*⟩

end

locale *comm-monoid-list-set* = *list: comm-monoid-list* + *set: comm-monoid-set*

begin

lemma *distinct-set-conv-list*:

$distinct\ xs \implies set.F\ g\ (set\ xs) = list.F\ (map\ g\ xs)$

⟨*proof*⟩

lemma *set-conv-list* [*code*]:

$set.F\ g\ (set\ xs) = list.F\ (map\ g\ (remdups\ xs))$

⟨*proof*⟩

lemma *list-conv-set-nth*:

$list.F\ xs = set.F\ (\lambda i. \ xs\ !\ i)\ \{0..<length\ xs\}$

⟨*proof*⟩

lemma *atLeastAtMost-conv-list* [*code-unfold*]:

⟨ $set.F\ g\ \{a..b\} = list.F\ (map\ g\ (List.interval\ a\ b))$ ⟩

⟨*proof*⟩

lemma *atLeastLessThan-conv-list* [*code-unfold*]:

⟨ $set.F\ g\ \{a..<b\} = (let\ d = b - 1\ in\ if\ d < b$

$then\ list.F\ (map\ g\ (List.interval\ a\ d))$

$else\ 1)$ ⟩

⟨proof⟩

lemma *greaterThanAtMost-conv-list* [code-unfold]:

⟨set.F g {a<..b} = (let c = a + 1 in if a < c
 then list.F (map g (List.interval c b))
 else 1)⟩
 ⟨proof⟩

lemma *greaterThanLessThan-conv-list* [code-unfold]:

⟨set.F g {a<..**<**b} = (let c = a + 1; d = b - 1 in if a < c ∧ d < b
 then list.F (map g (List.interval (a + 1) (b - 1)))
 else 1)⟩
 ⟨proof⟩

end

67.1 List summation

context *monoid-add*

begin

sublocale *sum-list: monoid-list plus 0*

defines

sum-list = *sum-list.F* ⟨proof⟩

end

context *comm-monoid-add*

begin

sublocale *sum-list: comm-monoid-list plus 0*

rewrites

monoid-list.F plus 0 = *sum-list*
 ⟨proof⟩

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⟩

end

Some syntactic sugar for summing a function over a list:

open-bundle *sum-list-syntax*

begin

syntax (*ASCII*)

-sum-list :: *pttrn* => '*a list* => '*b* => '*b* (⟨⟨indent=3 notation=⟨binder

```

SUM ›› SUM -<--. -) › [0, 51, 10] 10)
syntax
  -sum-list :: pttrn => 'a list => 'b => 'b    (⟨⟨indent=3 notation=⟨binder
Σ ›› Σ -<--. -) › [0, 51, 10] 10)
syntax-consts
  -sum-list == sum-list
translations — Beware of argument permutation!
  Σ x ← xs. b == CONST sum-list (CONST map (λx. b) xs)

end

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) (+) (+)
  ⟨proof⟩

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 = plus (sum-list (rev xs))
  ⟨proof⟩

lemma sum-list-of-nat: sum-list (map of-nat xs) = of-nat (sum-list xs)
  ⟨proof⟩

lemma sum-list-of-int: sum-list (map of-int xs) = of-int (sum-list xs)
  ⟨proof⟩

lemma count-list-concat: count-list (concat xss) x = sum-list (map (λxs. count-list
xs x) xss)
  ⟨proof⟩

lemma (in comm-monoid-add) sum-list-map-remove1:
  x ∈ set xs == => sum-list (map f xs) = f x + sum-list (map f (remove1 x xs))
  ⟨proof⟩

lemma (in monoid-add) size-list-conv-sum-list:
  size-list f xs = sum-list (map f xs) + size xs
  ⟨proof⟩

```

lemma (in monoid-add) length-concat:
 $length\ (concat\ xss) = sum-list\ (map\ length\ xss)$
 ⟨proof⟩

lemma (in monoid-add) length-product-lists:
 $length\ (product-lists\ xss) = foldr\ (*)\ (map\ length\ xss)\ 1$
 ⟨proof⟩

lemma (in monoid-add) sum-list-map-filter:
 assumes $\bigwedge x. x \in set\ xs \implies \neg P\ x \implies f\ x = 0$
 shows $sum-list\ (map\ f\ (filter\ P\ xs)) = sum-list\ (map\ f\ xs)$
 ⟨proof⟩

lemma sum-list-filter-le-nat:
 fixes $f :: 'a \Rightarrow nat$
 shows $sum-list\ (map\ f\ (filter\ P\ xs)) \leq sum-list\ (map\ f\ xs)$
 ⟨proof⟩

lemma (in comm-monoid-add) distinct-sum-list-conv-Sum:
 $distinct\ xs \implies sum-list\ xs = Sum\ (set\ xs)$
 ⟨proof⟩

lemma sum-list-upt[simp]:
 $m \leq n \implies sum-list\ [m..<n] = \sum\ \{m..<n\}$
 ⟨proof⟩

context ordered-comm-monoid-add
begin

lemma sum-list-nonneg: $(\bigwedge x. x \in set\ xs \implies 0 \leq x) \implies 0 \leq sum-list\ xs$
 ⟨proof⟩

lemma sum-list-nonpos: $(\bigwedge x. x \in set\ xs \implies x \leq 0) \implies sum-list\ xs \leq 0$
 ⟨proof⟩

lemma sum-list-nonneg-eq-0-iff:
 $(\bigwedge x. x \in set\ xs \implies 0 \leq x) \implies sum-list\ xs = 0 \longleftrightarrow (\forall x \in set\ xs. x = 0)$
 ⟨proof⟩

end

context canonically-ordered-monoid-add
begin

lemma sum-list-eq-0-iff [simp]:
 $sum-list\ ns = 0 \longleftrightarrow (\forall n \in set\ ns. n = 0)$
 ⟨proof⟩

lemma member-le-sum-list:

$x \in \text{set } xs \implies x \leq \text{sum-list } xs$
 $\langle \text{proof} \rangle$

lemma *elem-le-sum-list*:

$k < \text{size } ns \implies ns ! k \leq \text{sum-list } (ns)$
 $\langle \text{proof} \rangle$

end

lemma (*in ordered-cancel-comm-monoid-diff*) *sum-list-update*:

$k < \text{size } xs \implies \text{sum-list } (xs[k := x]) = \text{sum-list } xs + x - xs ! k$
 $\langle \text{proof} \rangle$

lemma (*in monoid-add*) *sum-list-triv*:

$(\sum x \leftarrow xs. r) = \text{of-nat } (\text{length } xs) * r$
 $\langle \text{proof} \rangle$

lemma (*in monoid-add*) *sum-list-0* [*simp*]:

$(\sum x \leftarrow xs. 0) = 0$
 $\langle \text{proof} \rangle$

For non-Abelian groups xs needs to be reversed on one side:

lemma (*in ab-group-add*) *uminus-sum-list-map*:

$-\text{sum-list } (\text{map } f xs) = \text{sum-list } (\text{map } (\text{uminus} \circ f) xs)$
 $\langle \text{proof} \rangle$

lemma (*in comm-monoid-add*) *sum-list-addf*:

$(\sum x \leftarrow xs. f x + g x) = \text{sum-list } (\text{map } f xs) + \text{sum-list } (\text{map } g xs)$
 $\langle \text{proof} \rangle$

lemma (*in ab-group-add*) *sum-list-subtractf*:

$(\sum x \leftarrow xs. f x - g x) = \text{sum-list } (\text{map } f xs) - \text{sum-list } (\text{map } g xs)$
 $\langle \text{proof} \rangle$

lemma (*in semiring-0*) *sum-list-const-mult*:

$(\sum x \leftarrow xs. c * f x) = c * (\sum x \leftarrow xs. f x)$
 $\langle \text{proof} \rangle$

lemma (*in semiring-0*) *sum-list-mult-const*:

$(\sum x \leftarrow xs. f x * c) = (\sum x \leftarrow xs. f x) * c$
 $\langle \text{proof} \rangle$

lemma (*in ordered-ab-group-add-abs*) *sum-list-abs*:

$|\text{sum-list } xs| \leq \text{sum-list } (\text{map } \text{abs } xs)$
 $\langle \text{proof} \rangle$

lemma *sum-list-mono*:

fixes $f g :: 'a \Rightarrow 'b :: \{\text{monoid-add, ordered-ab-semigroup-add}\}$

shows $(\bigwedge x. x \in \text{set } xs \implies f x \leq g x) \implies (\sum x \leftarrow xs. f x) \leq (\sum x \leftarrow xs. g x)$

<proof>

lemma *sum-list-strict-mono*:

fixes $f\ g :: 'a \Rightarrow 'b :: \{\text{monoid-add, strict-ordered-ab-semigroup-add}\}$

shows $\llbracket xs \neq []; \bigwedge x. x \in \text{set } xs \implies f\ x < g\ x \rrbracket$

$\implies \text{sum-list } (\text{map } f\ xs) < \text{sum-list } (\text{map } g\ xs)$

<proof>

A much more general version of this monotonicity lemma can be formulated with multisets and the multiset order

lemma *sum-list-mono2*: **fixes** $xs :: 'a :: \text{ordered-comm-monoid-add list}$

shows $\llbracket \text{length } xs = \text{length } ys; \bigwedge i. i < \text{length } xs \longrightarrow xs!i \leq ys!i \rrbracket$

$\implies \text{sum-list } xs \leq \text{sum-list } ys$

<proof>

lemma (in *monoid-add*) *sum-list-distinct-conv-sum-set*:

$\text{distinct } xs \implies \text{sum-list } (\text{map } f\ xs) = \text{sum } f\ (\text{set } xs)$

<proof>

lemma (in *monoid-add*) *interv-sum-list-conv-sum-set-nat*:

$\text{sum-list } (\text{map } f\ [m..<n]) = \text{sum } f\ (\text{set } [m..<n])$

<proof>

lemma (in *monoid-add*) *interv-sum-list-conv-sum-set-int*:

$\text{sum-list } (\text{map } f\ [k..l]) = \text{sum } f\ (\text{set } [k..l])$

<proof>

General equivalence between *sum-list* and *sum*

lemma (in *monoid-add*) *sum-list-sum-nth*:

$\text{sum-list } xs = (\sum i = 0 ..< \text{length } xs. xs!i)$

<proof>

lemma *sum-list-map-eq-sum-count*:

$\text{sum-list } (\text{map } f\ xs) = \text{sum } (\lambda x. \text{count-list } xs\ x * f\ x)\ (\text{set } xs)$

<proof>

lemma *sum-list-map-eq-sum-count2*:

assumes $\text{set } xs \subseteq X\ \text{finite } X$

shows $\text{sum-list } (\text{map } f\ xs) = \text{sum } (\lambda x. \text{count-list } xs\ x * f\ x)\ X$

<proof>

lemma *sum-list-replicate*: $\text{sum-list } (\text{replicate } n\ c) = \text{of-nat } n * c$

<proof>

lemma *sum-list-nonneg*:

$(\bigwedge x. x \in \text{set } xs \implies (x :: 'a :: \text{ordered-comm-monoid-add}) \geq 0) \implies \text{sum-list } xs \geq 0$

<proof>

lemma *sum-list-Suc*:

$sum\text{-}list\ (map\ (\lambda x. Suc(f\ x))\ xs) = sum\text{-}list\ (map\ f\ xs) + length\ xs$
 $\langle proof \rangle$

lemma (*in monoid-add*) *sum-list-map-filter'*:

$sum\text{-}list\ (map\ f\ (filter\ P\ xs)) = sum\text{-}list\ (map\ (\lambda x. if\ P\ x\ then\ f\ x\ else\ 0)\ xs)$
 $\langle proof \rangle$

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 $f :: nat \Rightarrow ('b :: ordered\text{-}comm\text{-}monoid\text{-}add)$
assumes *mono*: $\bigwedge x\ y. x \leq y \implies f\ x \leq f\ y$
shows *sorted-wrt* $(<) ns \implies$
 $(\sum i \in \{0..<length\ ns\}. f\ i) \leq (\sum i \leftarrow ns. f\ i)$
 $\langle proof \rangle$

lemma *member-le-sum-list*:

fixes $x :: 'a :: ordered\text{-}comm\text{-}monoid\text{-}add$
assumes $x \in set\ xs \bigwedge x. x \in set\ xs \implies x \geq 0$
shows $x \leq sum\text{-}list\ xs$
 $\langle proof \rangle$

67.2 Horner sums

context *comm-semiring-0*

begin

definition *horner-sum* :: $\langle ('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'a \rangle$

where *horner-sum-foldr*: $\langle horner\text{-}sum\ f\ a\ xs = foldr\ (\lambda x\ b. f\ x + a * b)\ xs\ 0 \rangle$

lemma *horner-sum-simps* [*simp*]:

$\langle horner\text{-}sum\ f\ a\ [] = 0 \rangle$
 $\langle horner\text{-}sum\ f\ a\ (x \# xs) = f\ x + a * horner\text{-}sum\ f\ a\ xs \rangle$
 $\langle proof \rangle$

lemma *horner-sum-eq-sum-funpow*:

$\langle horner\text{-}sum\ f\ a\ xs = (\sum n = 0..<length\ xs. ((*)\ a\ \frown\ n)\ (f\ (xs\ !\ n))) \rangle$
 $\langle proof \rangle$

end

context

includes *lifting-syntax*

begin

lemma *horner-sum-transfer* [*transfer-rule*]:

```

  ⟨((B ==> A) ==> A ==> list-all2 B ==> A) horner-sum horner-sum⟩
  if [transfer-rule]: ⟨A 0 0⟩
    and [transfer-rule]: ⟨(A ==> A ==> A) (+) (+)⟩
    and [transfer-rule]: ⟨(A ==> A ==> A) (*) (*)⟩
  ⟨proof⟩

```

end

context *comm-semiring-1*
begin

lemma *horner-sum-eq-sum*:
 ⟨horner-sum f a xs = (∑ n = 0..<length xs. f (xs ! n) * a ^ n)⟩
 ⟨proof⟩

lemma *horner-sum-append*:
 ⟨horner-sum f a (xs @ ys) = horner-sum f a xs + a ^ length xs * horner-sum f a ys⟩
 ⟨proof⟩

end

context *linordered-semidom*
begin

lemma *horner-sum-nonnegative*:
 ⟨0 ≤ horner-sum of-bool 2 bs⟩
 ⟨proof⟩

end

context *discrete-linordered-semidom*
begin

lemma *horner-sum-bound*:
 ⟨horner-sum of-bool 2 bs < 2 ^ length bs⟩
 ⟨proof⟩

lemma *horner-sum-of-bool-2-less*:
 ⟨(horner-sum of-bool 2 bs) < 2 ^ length bs⟩
 ⟨proof⟩

end

lemma *nat-horner-sum [simp]*:
 ⟨nat (horner-sum of-bool 2 bs) = horner-sum of-bool 2 bs⟩
 ⟨proof⟩

context *discrete-linordered-semidom*

begin

lemma *horner-sum-less-iff-lexordp-eq*:

$\langle \text{horner-sum of-bool } 2 \text{ } bs \leq \text{horner-sum of-bool } 2 \text{ } cs \longleftrightarrow \text{lexordp-eq } (\text{rev } bs) (\text{rev } cs) \rangle$

if $\langle \text{length } bs = \text{length } cs \rangle$

$\langle \text{proof} \rangle$

lemma *horner-sum-less-iff-lexordp*:

$\langle \text{horner-sum of-bool } 2 \text{ } bs < \text{horner-sum of-bool } 2 \text{ } cs \longleftrightarrow \text{ord-class.lexordp } (\text{rev } bs) (\text{rev } cs) \rangle$

if $\langle \text{length } bs = \text{length } cs \rangle$

$\langle \text{proof} \rangle$

end

67.3 Further facts about *List.n-lists*

lemma *length-n-lists*: $\text{length } (\text{List.n-lists } n \text{ } xs) = \text{length } xs \wedge n$

$\langle \text{proof} \rangle$

lemma *distinct-n-lists*:

assumes *distinct xs*

shows *distinct (List.n-lists n xs)*

$\langle \text{proof} \rangle$

67.4 Tools setup

lemmas *sum-code = sum.set-conv-list*

lemma *sum-set-upto-conv-sum-list-int*:

$\text{sum } f \text{ (set } [i..j::\text{int}]) = \text{sum-list } (\text{map } f [i..j])$

$\langle \text{proof} \rangle$

lemma *sum-set-upt-conv-sum-list-nat*:

$\text{sum } f \text{ (set } [m..<n]) = \text{sum-list } (\text{map } f [m..<n])$

$\langle \text{proof} \rangle$

67.5 List product

context *monoid-mult*

begin

sublocale *prod-list: monoid-list times 1*

defines

$\text{prod-list} = \text{prod-list.F } \langle \text{proof} \rangle$

end

context *comm-monoid-mult*

begin

sublocale *prod-list: comm-monoid-list times 1*

rewrites

monoid-list.F times 1 = prod-list

<proof>

sublocale *prod: comm-monoid-list-set times 1*

rewrites

monoid-list.F times 1 = prod-list

and *comm-monoid-set.F times 1 = prod*

<proof>

end

Some syntactic sugar:

open-bundle *prod-list-syntax*

begin

syntax (*ASCII*)

-prod-list :: pttrn ==> 'a list ==> 'b ==> 'b (*<(<indent=3 notation=<binder
PROD>>PROD -<-.-) [0, 51, 10] 10)*

syntax

-prod-list :: pttrn ==> 'a list ==> 'b ==> 'b (*<(<indent=3 notation=<binder
prod>>prod -<-.-) [0, 51, 10] 10)*

syntax-consts

-prod-list == prod-list

translations — Beware of argument permutation!

$\prod x \leftarrow xs. b \equiv \text{CONST } \text{prod-list } (\text{CONST } \text{map } (\lambda x. b) xs)$

end

context

includes *lifting-syntax*

begin

lemma *prod-list-transfer [transfer-rule]:*

(list-all2 A ==> A) prod-list prod-list

if *[transfer-rule]: A 1 1 (A ==> A ==> A) (*) (*)*

<proof>

end

lemma *prod-list-zero-iff:*

prod-list xs = 0 <=> (0 :: 'a :: {semiring-no-zero-divisors, semiring-1}) ∈ set xs

<proof>

lemma *prod-list-nonneg: ($\bigwedge x. (x :: 'a :: \text{ordered-semiring-1}) \in \text{set } xs \implies x \geq 0$)
 $\implies \text{prod-list } xs \geq 0$*

$\langle \text{proof} \rangle$

lemma *prod-list-replicate*[simp]: *prod-list* (*replicate* *n* *a*) = *a* \wedge *n*
 $\langle \text{proof} \rangle$

lemma *prod-list-power*:
fixes *xs* :: 'a :: comm-monoid-mult *list*
shows *prod-list* *xs* \wedge *n* = $(\prod_{x \leftarrow xs} x \wedge n)$
 $\langle \text{proof} \rangle$

lemma *prod-list-dvd*:
assumes (*x* :: 'a :: comm-monoid-mult) \in *set xs*
shows *x* *dvd* *prod-list xs*
 $\langle \text{proof} \rangle$

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*]:
 $\langle (\bigwedge a. a \text{ div } 2 = a \implies P \ a)$
 $\implies (\bigwedge a \ b. P \ a \implies (\text{of-bool } b + 2 * a) \text{ div } 2 = a \implies P \ (\text{of-bool } b + 2 * a))$
 $\implies P \ a \rangle$
assumes *bits-mod-div-trivial* [simp]: $\langle a \text{ mod } b \text{ div } b = 0 \rangle$
and *half-div-exp-eq*: $\langle a \text{ div } 2 \text{ div } 2 \wedge n = a \text{ div } 2 \wedge \text{Suc } n \rangle$
and *even-double-div-exp-iff*: $\langle 2 \wedge \text{Suc } n \neq 0 \implies \text{even } (2 * a \text{ div } 2 \wedge \text{Suc } n)$
 $\longleftrightarrow \text{even } (a \text{ div } 2 \wedge n) \rangle$
fixes *bit* :: 'a \Rightarrow nat \Rightarrow bool
assumes *bit-iff-odd*: $\langle \text{bit } a \ n \longleftrightarrow \text{odd } (a \text{ div } 2 \wedge n) \rangle$
begin

Having *bit* as definitional class operation takes into account that specific instances can be implemented differently wrt. code generation.

lemma *half-1* [simp]:
 $\langle 1 \text{ div } 2 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *div-exp-eq-funpow-half*:
 $\langle a \text{ div } 2 \wedge n = ((\lambda a. a \text{ div } 2) \frown n) \ a \rangle$
 $\langle \text{proof} \rangle$

lemma *div-exp-eq*:

$\langle a \text{ div } 2 \wedge m \text{ div } 2 \wedge n = a \text{ div } 2 \wedge (m + n) \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-0*:

$\langle \text{bit } a \ 0 \longleftrightarrow \text{odd } a \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-Suc*:

$\langle \text{bit } a \ (\text{Suc } n) \longleftrightarrow \text{bit } (a \text{ div } 2) \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-rec*:

$\langle \text{bit } a \ n \longleftrightarrow (\text{if } n = 0 \text{ then odd } a \text{ else bit } (a \text{ div } 2) \ (n - 1)) \rangle$
 $\langle \text{proof} \rangle$

context

fixes *a*

assumes *stable*: $\langle a \text{ div } 2 = a \rangle$

begin

lemma *bits-stable-imp-add-self*:

$\langle a + a \text{ mod } 2 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *stable-imp-bit-iff-odd*:

$\langle \text{bit } a \ n \longleftrightarrow \text{odd } a \rangle$
 $\langle \text{proof} \rangle$

end

lemma *bit-iff-odd-imp-stable*:

$\langle a \text{ div } 2 = a \rangle$ **if** $\langle \bigwedge n. \text{bit } a \ n \longleftrightarrow \text{odd } a \rangle$
 $\langle \text{proof} \rangle$

lemma *even-succ-div-exp [simp]*:

$\langle (1 + a) \text{ div } 2 \wedge n = a \text{ div } 2 \wedge n \rangle$ **if** $\langle \text{even } a \rangle$ **and** $\langle n > 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *even-succ-mod-exp [simp]*:

$\langle (1 + a) \text{ mod } 2 \wedge n = 1 + (a \text{ mod } 2 \wedge n) \rangle$ **if** $\langle \text{even } a \rangle$ **and** $\langle n > 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *half-numeral-Bit1-eq [simp]*:

$\langle \text{numeral } (\text{num.Bit1 } m) \text{ div } 2 = \text{numeral } (\text{num.Bit0 } m) \text{ div } 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *double-half-numeral-Bit-0-eq [simp]*:

$\langle 2 * (\text{numeral } (\text{num.Bit0 } m) \text{ div } 2) = \text{numeral } (\text{num.Bit0 } m) \rangle$
 $\langle (\text{numeral } (\text{num.Bit0 } m) \text{ div } 2) * 2 = \text{numeral } (\text{num.Bit0 } m) \rangle$

$\langle \text{proof} \rangle$

named-theorems *bit-simps* $\langle \text{Simplification rules for } \mathbf{const} \langle \text{bit} \rangle \rangle$

definition *possible-bit* :: $\langle 'a \text{ itself} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$

where $\langle \text{possible-bit TYPE('a)} \ n \longleftrightarrow 2 \wedge n \neq 0 \rangle$

— This auxiliary avoids non-termination with extensionality.

lemma *possible-bit-0* [simp]:

$\langle \text{possible-bit TYPE('a)} \ 0 \rangle$

$\langle \text{proof} \rangle$

lemma *fold-possible-bit*:

$\langle 2 \wedge n = 0 \longleftrightarrow \neg \text{possible-bit TYPE('a)} \ n \rangle$

$\langle \text{proof} \rangle$

lemma *bit-imp-possible-bit*:

$\langle \text{possible-bit TYPE('a)} \ n \rangle \text{ if } \langle \text{bit } a \ n \rangle$

$\langle \text{proof} \rangle$

lemma *impossible-bit*:

$\langle \neg \text{bit } a \ n \rangle \text{ if } \langle \neg \text{possible-bit TYPE('a)} \ n \rangle$

$\langle \text{proof} \rangle$

lemma *possible-bit-less-imp*:

$\langle \text{possible-bit TYPE('a)} \ j \rangle \text{ if } \langle \text{possible-bit TYPE('a)} \ i \rangle \langle j \leq i \rangle$

$\langle \text{proof} \rangle$

lemma *possible-bit-min* [simp]:

$\langle \text{possible-bit TYPE('a)} \ (\min \ i \ j) \longleftrightarrow \text{possible-bit TYPE('a)} \ i \vee \text{possible-bit TYPE('a)} \ j \rangle$

$\langle \text{proof} \rangle$

lemma *bit-eqI*:

$\langle a = b \rangle \text{ if } \langle \bigwedge n. \text{possible-bit TYPE('a)} \ n \implies \text{bit } a \ n \longleftrightarrow \text{bit } b \ n \rangle$

$\langle \text{proof} \rangle$

lemma *bit-eq-rec*:

$\langle a = b \longleftrightarrow (\text{even } a \longleftrightarrow \text{even } b) \wedge a \text{ div } 2 = b \text{ div } 2 \rangle \text{ (is } \langle ?P = ?Q \rangle)$

$\langle \text{proof} \rangle$

lemma *bit-eq-iff*:

$\langle a = b \longleftrightarrow (\forall n. \text{possible-bit TYPE('a)} \ n \longrightarrow \text{bit } a \ n \longleftrightarrow \text{bit } b \ n) \rangle$

$\langle \text{proof} \rangle$

lemma *bit-0-eq* [simp]:

$\langle \text{bit } 0 = \perp \rangle$

$\langle \text{proof} \rangle$

lemma *bit-double-Suc-iff*:

$\langle \text{bit } (2 * a) \text{ (Suc } n) \longleftrightarrow \text{possible-bit TYPE('a) (Suc } n) \wedge \text{bit } a \text{ } n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-double-iff [bit-simps]*:

$\langle \text{bit } (2 * a) \text{ } n \longleftrightarrow \text{possible-bit TYPE('a) } n \wedge n \neq 0 \wedge \text{bit } a \text{ (} n - 1 \text{)} \rangle$
 $\langle \text{proof} \rangle$

lemma *even-bit-succ-iff*:

$\langle \text{bit } (1 + a) \text{ } n \longleftrightarrow \text{bit } a \text{ } n \vee n = 0 \rangle \text{ if } \langle \text{even } a \rangle$
 $\langle \text{proof} \rangle$

lemma *odd-bit-iff-bit-pred*:

$\langle \text{bit } a \text{ } n \longleftrightarrow \text{bit } (a - 1) \text{ } n \vee n = 0 \rangle \text{ if } \langle \text{odd } a \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-exp-iff [bit-simps]*:

$\langle \text{bit } (2 \wedge m) \text{ } n \longleftrightarrow \text{possible-bit TYPE('a) } n \wedge n = m \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-1-iff [bit-simps]*:

$\langle \text{bit } 1 \text{ } n \longleftrightarrow n = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-2-iff [bit-simps]*:

$\langle \text{bit } 2 \text{ } n \longleftrightarrow \text{possible-bit TYPE('a) } 1 \wedge n = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-of-bool-iff [bit-simps]*:

$\langle \text{bit (of-bool } b) \text{ } n \longleftrightarrow n = 0 \wedge b \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-mod-2-iff [simp]*:

$\langle \text{bit } (a \bmod 2) \text{ } n \longleftrightarrow n = 0 \wedge \text{odd } a \rangle$
 $\langle \text{proof} \rangle$

lemma *stable-index*:

obtains m **where** $\langle \text{possible-bit TYPE('a) } m \rangle$

$\langle \bigwedge n. \text{possible-bit TYPE('a) } n \implies n \geq m \implies \text{bit } a \text{ } n \longleftrightarrow \text{bit } a \text{ } m \rangle$
 $\langle \text{proof} \rangle$

end

lemma *nat-bit-induct [case-names zero even odd]*:

$\langle P \text{ } n \rangle \text{ if zero: } \langle P \text{ } 0 \rangle$

and *even*: $\langle \bigwedge n. P \text{ } n \implies n > 0 \implies P \text{ (} 2 * n \text{)} \rangle$

and *odd*: $\langle \bigwedge n. P \text{ } n \implies P \text{ (Suc (} 2 * n \text{))} \rangle$

$\langle \text{proof} \rangle$

instantiation *nat* :: *semiring-bits*
begin

definition *bit-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$
where $\langle \text{bit-nat } m \ n \longleftrightarrow \text{odd } (m \text{ div } 2 \wedge n) \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma *possible-bit-nat* [*simp*]:
 $\langle \text{possible-bit } \text{TYPE}(\text{nat}) \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-Suc-0-iff* [*bit-simps*]:
 $\langle \text{bit } (\text{Suc } 0) \ n \longleftrightarrow n = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *not-bit-Suc-0-Suc* [*simp*]:
 $\langle \neg \text{bit } (\text{Suc } 0) \ (\text{Suc } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *not-bit-Suc-0-numeral* [*simp*]:
 $\langle \neg \text{bit } (\text{Suc } 0) \ (\text{numeral } n) \rangle$
 $\langle \text{proof} \rangle$

context *semiring-bits*
begin

lemma *bit-of-nat-iff* [*bit-simps*]:
 $\langle \text{bit } (\text{of-nat } m) \ n \longleftrightarrow \text{possible-bit } \text{TYPE}('a) \ n \wedge \text{bit } m \ n \rangle$
 $\langle \text{proof} \rangle$

end

lemma *int-bit-induct* [*case-names zero minus even odd*]:
 $\langle P \ k \rangle$ **if** *zero-int*: $\langle P \ 0 \rangle$
and *minus-int*: $\langle P \ (- \ 1) \rangle$
and *even-int*: $\langle \bigwedge k. P \ k \Longrightarrow k \neq 0 \Longrightarrow P \ (k * 2) \rangle$
and *odd-int*: $\langle \bigwedge k. P \ k \Longrightarrow k \neq -1 \Longrightarrow P \ (1 + (k * 2)) \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

instantiation *int* :: *semiring-bits*
begin

definition *bit-int* :: $\langle \text{int} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$
where $\langle \text{bit-int } k \ n \longleftrightarrow \text{odd } (k \text{ div } 2 \wedge n) \rangle$

instance

$\langle \text{proof} \rangle$

end

lemma *possible-bit-int* [simp]:

$\langle \text{possible-bit TYPE(int)}\ n \rangle$

$\langle \text{proof} \rangle$

lemma *bit-nat-iff* [bit-simps]:

$\langle \text{bit (nat } k) \ n \longleftrightarrow k \geq 0 \wedge \text{bit } k \ n \rangle$

$\langle \text{proof} \rangle$

68.2 Bit operations

class *semiring-bit-operations* = *semiring-bits* +

fixes *and* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$ (**infixr** $\langle \text{AND} \rangle$ 64)

and *or* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$ (**infixr** $\langle \text{OR} \rangle$ 59)

and *xor* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$ (**infixr** $\langle \text{XOR} \rangle$ 59)

and *mask* :: $\langle \text{nat} \Rightarrow 'a \rangle$

and *set-bit* :: $\langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle$

and *unset-bit* :: $\langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle$

and *flip-bit* :: $\langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle$

and *push-bit* :: $\langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle$

and *drop-bit* :: $\langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle$

and *take-bit* :: $\langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle$

assumes *and-rec*: $\langle a \text{ AND } b = \text{of_bool } (\text{odd } a \wedge \text{odd } b) + 2 * ((a \text{ div } 2) \text{ AND } (b \text{ div } 2)) \rangle$

and *or-rec*: $\langle a \text{ OR } b = \text{of_bool } (\text{odd } a \vee \text{odd } b) + 2 * ((a \text{ div } 2) \text{ OR } (b \text{ div } 2)) \rangle$

and *xor-rec*: $\langle a \text{ XOR } b = \text{of_bool } (\text{odd } a \neq \text{odd } b) + 2 * ((a \text{ div } 2) \text{ XOR } (b \text{ div } 2)) \rangle$

and *mask-eq-exp-minus-1*: $\langle \text{mask } n = 2^{\wedge} n - 1 \rangle$

and *set-bit-eq-or*: $\langle \text{set-bit } n \ a = a \text{ OR } \text{push-bit } n \ 1 \rangle$

and *unset-bit-eq-or-xor*: $\langle \text{unset-bit } n \ a = (a \text{ OR } \text{push-bit } n \ 1) \text{ XOR } \text{push-bit } n \ 1 \rangle$

and *flip-bit-eq-xor*: $\langle \text{flip-bit } n \ a = a \text{ XOR } \text{push-bit } n \ 1 \rangle$

and *push-bit-eq-mult*: $\langle \text{push-bit } n \ a = a * 2^{\wedge} n \rangle$

and *drop-bit-eq-div*: $\langle \text{drop-bit } n \ a = a \text{ div } 2^{\wedge} n \rangle$

and *take-bit-eq-mod*: $\langle \text{take-bit } n \ a = a \text{ mod } 2^{\wedge} n \rangle$

begin

We want the bitwise operations to bind slightly weaker than + and −.

Logically, *push-bit*, *drop-bit* and *take-bit* are just aliases; having them as separate operations makes proofs easier, otherwise proof automation would fiddle with concrete expressions $(2::'a)^n$ 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 im-

plemented differently wrt. code generation.

lemma *bit-iff-odd-drop-bit*:

$$\langle \text{bit } a \ n \longleftrightarrow \text{odd } (\text{drop-bit } n \ a) \rangle$$

<proof>

lemma *even-drop-bit-iff-not-bit*:

$$\langle \text{even } (\text{drop-bit } n \ a) \longleftrightarrow \neg \text{bit } a \ n \rangle$$

<proof>

lemma *bit-and-iff [bit-simps]*:

$$\langle \text{bit } (a \ \text{AND} \ b) \ n \longleftrightarrow \text{bit } a \ n \wedge \text{bit } b \ n \rangle$$

<proof>

lemma *bit-or-iff [bit-simps]*:

$$\langle \text{bit } (a \ \text{OR} \ b) \ n \longleftrightarrow \text{bit } a \ n \vee \text{bit } b \ n \rangle$$

<proof>

lemma *bit-xor-iff [bit-simps]*:

$$\langle \text{bit } (a \ \text{XOR} \ b) \ n \longleftrightarrow \text{bit } a \ n \neq \text{bit } b \ n \rangle$$

<proof>

sublocale *and: semilattice* $\langle (\text{AND}) \rangle$

<proof>

sublocale *or: semilattice-neutr* $\langle (\text{OR}) \rangle \ 0$

<proof>

sublocale *xor: comm-monoid* $\langle (\text{XOR}) \rangle \ 0$

<proof>

lemma *even-and-iff*:

$$\langle \text{even } (a \ \text{AND} \ b) \longleftrightarrow \text{even } a \vee \text{even } b \rangle$$

<proof>

lemma *even-or-iff*:

$$\langle \text{even } (a \ \text{OR} \ b) \longleftrightarrow \text{even } a \wedge \text{even } b \rangle$$

<proof>

lemma *even-xor-iff*:

$$\langle \text{even } (a \ \text{XOR} \ b) \longleftrightarrow (\text{even } a \longleftrightarrow \text{even } b) \rangle$$

<proof>

lemma *zero-and-eq [simp]*:

$$\langle 0 \ \text{AND} \ a = 0 \rangle$$

<proof>

lemma *and-zero-eq [simp]*:

$$\langle a \ \text{AND} \ 0 = 0 \rangle$$

<proof>

lemma *one-and-eq*:

$\langle 1 \text{ AND } a = a \text{ mod } 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *and-one-eq*:

$\langle a \text{ AND } 1 = a \text{ mod } 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *one-or-eq*:

$\langle 1 \text{ OR } a = a + \text{of-bool}(\text{even } a) \rangle$
 $\langle \text{proof} \rangle$

lemma *or-one-eq*:

$\langle a \text{ OR } 1 = a + \text{of-bool}(\text{even } a) \rangle$
 $\langle \text{proof} \rangle$

lemma *one-xor-eq*:

$\langle 1 \text{ XOR } a = a + \text{of-bool}(\text{even } a) - \text{of-bool}(\text{odd } a) \rangle$
 $\langle \text{proof} \rangle$

lemma *xor-one-eq*:

$\langle a \text{ XOR } 1 = a + \text{of-bool}(\text{even } a) - \text{of-bool}(\text{odd } a) \rangle$
 $\langle \text{proof} \rangle$

lemma *xor-self-eq* [simp]:

$\langle a \text{ XOR } a = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-0* [simp]:

$\langle \text{mask } 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *inc-mask-eq-exp*:

$\langle \text{mask } n + 1 = 2 \wedge n \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-eq-iff-eq-exp*:

$\langle \text{mask } n = a \iff a + 1 = 2 \wedge n \rangle$
 $\langle \text{proof} \rangle$

lemma *eq-mask-iff-eq-exp*:

$\langle a = \text{mask } n \iff a + 1 = 2 \wedge n \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-Suc-double*:

$\langle \text{mask}(\text{Suc } n) = 1 \text{ OR } 2 * \text{mask } n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-mask-iff* [bit-simps]:

$\langle \text{bit } (\text{mask } m) \ n \longleftrightarrow \text{possible-bit TYPE('a)} \ n \wedge n < m \rangle$
 $\langle \text{proof} \rangle$

lemma *even-mask-iff*:

$\langle \text{even } (\text{mask } n) \longleftrightarrow n = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-Suc-0* [simp]:

$\langle \text{mask } (\text{Suc } 0) = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-Suc-exp*:

$\langle \text{mask } (\text{Suc } n) = 2 \wedge n \text{ OR } \text{mask } n \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-numeral*:

$\langle \text{mask } (\text{numeral } n) = 1 + 2 * \text{mask } (\text{pred-numeral } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-0-id* [simp]:

$\langle \text{push-bit } 0 = \text{id} \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-Suc* [simp]:

$\langle \text{push-bit } (\text{Suc } n) \ a = \text{push-bit } n \ (a * 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-double*:

$\langle \text{push-bit } n \ (a * 2) = \text{push-bit } n \ a * 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-push-bit-iff* [bit-simps]:

$\langle \text{bit } (\text{push-bit } m \ a) \ n \longleftrightarrow m \leq n \wedge \text{possible-bit TYPE('a)} \ n \wedge \text{bit } a \ (n - m) \rangle$
 $\langle \text{proof} \rangle$

lemma *funpow-double-eq-push-bit*:

$\langle \text{times } 2 \ \sim n = \text{push-bit } n \rangle$
 $\langle \text{proof} \rangle$

lemma *div-push-bit-of-1-eq-drop-bit*:

$\langle a \ \text{div} \ \text{push-bit } n \ 1 = \text{drop-bit } n \ a \rangle$
 $\langle \text{proof} \rangle$

lemma *bits-ident*:

$\langle \text{push-bit } n \ (\text{drop-bit } n \ a) + \text{take-bit } n \ a = a \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-push-bit* [simp]:

$\langle \text{push-bit } m \ (\text{push-bit } n \ a) = \text{push-bit } (m + n) \ a \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-of-0* [simp]:
 $\langle \text{push-bit } n \ 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-of-1* [simp]:
 $\langle \text{push-bit } n \ 1 = 2 \wedge^n \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-add*:
 $\langle \text{push-bit } n \ (a + b) = \text{push-bit } n \ a + \text{push-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-numeral* [simp]:
 $\langle \text{push-bit } (\text{numeral } l) \ (\text{numeral } k) = \text{push-bit } (\text{pred-numeral } l) \ (\text{numeral } (\text{Num.Bit0 } k)) \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-drop-bit-eq* [bit-simps]:
 $\langle \text{bit } (\text{drop-bit } n \ a) = \text{bit } a \circ (+) \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *disjunctive-xor-eq-or*:
 $\langle a \ \text{XOR } b = a \ \text{OR } b \rangle \text{ if } \langle a \ \text{AND } b = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *disjunctive-add-eq-or*:
 $\langle a + b = a \ \text{OR } b \rangle \text{ if } \langle a \ \text{AND } b = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *disjunctive-add-eq-xor*:
 $\langle a + b = a \ \text{XOR } b \rangle \text{ if } \langle a \ \text{AND } b = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-0* [simp]:
 $\text{take-bit } 0 \ a = 0$
 $\langle \text{proof} \rangle$

lemma *bit-take-bit-iff* [bit-simps]:
 $\langle \text{bit } (\text{take-bit } m \ a) \ n \longleftrightarrow n < m \wedge \text{bit } a \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-Suc*:
 $\langle \text{take-bit } (\text{Suc } n) \ a = \text{take-bit } n \ (a \text{ div } 2) * 2 + a \text{ mod } 2 \rangle \text{ (is } \langle ?lhs = ?rhs \rangle)$
 $\langle \text{proof} \rangle$

lemma *take-bit-rec*:

$\langle \text{take-bit } n \ a = (\text{if } n = 0 \text{ then } 0 \text{ else take-bit } (n - 1) \ (a \text{ div } 2) * 2 + a \text{ mod } 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-Suc-0* [simp]:
 $\langle \text{take-bit } (\text{Suc } 0) \ a = a \text{ mod } 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-of-0* [simp]:
 $\langle \text{take-bit } n \ 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-of-1* [simp]:
 $\langle \text{take-bit } n \ 1 = \text{of-bool } (n > 0) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-of-0* [simp]:
 $\langle \text{drop-bit } n \ 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-of-1* [simp]:
 $\langle \text{drop-bit } n \ 1 = \text{of-bool } (n = 0) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-0* [simp]:
 $\langle \text{drop-bit } 0 = \text{id} \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-Suc*:
 $\langle \text{drop-bit } (\text{Suc } n) \ a = \text{drop-bit } n \ (a \text{ div } 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-rec*:
 $\langle \text{drop-bit } n \ a = (\text{if } n = 0 \text{ then } a \text{ else drop-bit } (n - 1) \ (a \text{ div } 2)) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-half*:
 $\langle \text{drop-bit } n \ (a \text{ div } 2) = \text{drop-bit } n \ a \text{ div } 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-of-bool* [simp]:
 $\langle \text{drop-bit } n \ (\text{of-bool } b) = \text{of-bool } (n = 0 \wedge b) \rangle$
 $\langle \text{proof} \rangle$

lemma *even-take-bit-eq* [simp]:
 $\langle \text{even } (\text{take-bit } n \ a) \longleftrightarrow n = 0 \vee \text{even } a \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-take-bit* [simp]:
 $\langle \text{take-bit } m \ (\text{take-bit } n \ a) = \text{take-bit } (\min m \ n) \ a \rangle$

$\langle \text{proof} \rangle$

lemma *drop-bit-drop-bit* [simp]:

$\langle \text{drop-bit } m \ (\text{drop-bit } n \ a) = \text{drop-bit } (m + n) \ a \rangle$

$\langle \text{proof} \rangle$

lemma *push-bit-take-bit*:

$\langle \text{push-bit } m \ (\text{take-bit } n \ a) = \text{take-bit } (m + n) \ (\text{push-bit } m \ a) \rangle$

$\langle \text{proof} \rangle$

lemma *take-bit-push-bit*:

$\langle \text{take-bit } m \ (\text{push-bit } n \ a) = \text{push-bit } n \ (\text{take-bit } (m - n) \ a) \rangle$

$\langle \text{proof} \rangle$

lemma *take-bit-drop-bit*:

$\langle \text{take-bit } m \ (\text{drop-bit } n \ a) = \text{drop-bit } n \ (\text{take-bit } (m + n) \ a) \rangle$

$\langle \text{proof} \rangle$

lemma *drop-bit-take-bit*:

$\langle \text{drop-bit } m \ (\text{take-bit } n \ a) = \text{take-bit } (n - m) \ (\text{drop-bit } m \ a) \rangle$

$\langle \text{proof} \rangle$

lemma *even-push-bit-iff* [simp]:

$\langle \text{even } (\text{push-bit } n \ a) \longleftrightarrow n \neq 0 \ \vee \ \text{even } a \rangle$

$\langle \text{proof} \rangle$

lemma *stable-imp-drop-bit-eq*:

$\langle \text{drop-bit } n \ a = a \rangle$

if $\langle a \ \text{div } 2 = a \rangle$

$\langle \text{proof} \rangle$

lemma *stable-imp-take-bit-eq*:

$\langle \text{take-bit } n \ a = (\text{if even } a \text{ then } 0 \text{ else mask } n) \rangle$

if $\langle a \ \text{div } 2 = a \rangle$

$\langle \text{proof} \rangle$

lemma *exp-dvdE*:

assumes $\langle 2^n \ \text{dvd } a \rangle$

obtains b **where** $\langle a = \text{push-bit } n \ b \rangle$

$\langle \text{proof} \rangle$

lemma *take-bit-eq-0-iff*:

$\langle \text{take-bit } n \ a = 0 \longleftrightarrow 2^n \ \text{dvd } a \rangle \ (\text{is } \langle ?P \longleftrightarrow ?Q \rangle)$

$\langle \text{proof} \rangle$

lemma *take-bit-tightened*:

$\langle \text{take-bit } m \ a = \text{take-bit } m \ b \rangle \ \text{if } \langle \text{take-bit } n \ a = \text{take-bit } n \ b \rangle \ \text{and } \langle m \leq n \rangle$

$\langle \text{proof} \rangle$

lemma *take-bit-eq-self-iff-drop-bit-eq-0*:

$\langle \text{take-bit } n \ a = a \longleftrightarrow \text{drop-bit } n \ a = 0 \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$)
 $\langle \text{proof} \rangle$

lemma *impossible-bit-imp-take-bit-eq-self*:

$\langle \text{take-bit } n \ a = a \rangle$ if $\langle \neg \text{possible-bit TYPE('a) } n \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-exp-eq*:

$\langle \text{drop-bit } m \ (2 \wedge n) = \text{of_bool } (m \leq n \wedge \text{possible-bit TYPE('a) } n) * 2 \wedge (n - m) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-and [simp]*:

$\langle \text{take-bit } n \ (a \ \text{AND} \ b) = \text{take-bit } n \ a \ \text{AND} \ \text{take-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-or [simp]*:

$\langle \text{take-bit } n \ (a \ \text{OR} \ b) = \text{take-bit } n \ a \ \text{OR} \ \text{take-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-xor [simp]*:

$\langle \text{take-bit } n \ (a \ \text{XOR} \ b) = \text{take-bit } n \ a \ \text{XOR} \ \text{take-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-and [simp]*:

$\langle \text{push-bit } n \ (a \ \text{AND} \ b) = \text{push-bit } n \ a \ \text{AND} \ \text{push-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-or [simp]*:

$\langle \text{push-bit } n \ (a \ \text{OR} \ b) = \text{push-bit } n \ a \ \text{OR} \ \text{push-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-xor [simp]*:

$\langle \text{push-bit } n \ (a \ \text{XOR} \ b) = \text{push-bit } n \ a \ \text{XOR} \ \text{push-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-and [simp]*:

$\langle \text{drop-bit } n \ (a \ \text{AND} \ b) = \text{drop-bit } n \ a \ \text{AND} \ \text{drop-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-or [simp]*:

$\langle \text{drop-bit } n \ (a \ \text{OR} \ b) = \text{drop-bit } n \ a \ \text{OR} \ \text{drop-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-xor [simp]*:

$\langle \text{drop-bit } n \ (a \ \text{XOR} \ b) = \text{drop-bit } n \ a \ \text{XOR} \ \text{drop-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-of-mask [simp]*:

$\langle \text{take-bit } m \text{ (mask } n) = \text{mask (min } m \text{ } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-eq-mask*:

$\langle \text{take-bit } n \text{ } a = a \text{ AND mask } n \rangle$
 $\langle \text{proof} \rangle$

lemma *or-eq-0-iff*:

$\langle a \text{ OR } b = 0 \iff a = 0 \wedge b = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-iff-and-drop-bit-eq-1*:

$\langle \text{bit } a \text{ } n \iff \text{drop-bit } n \text{ } a \text{ AND } 1 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-iff-and-push-bit-not-eq-0*:

$\langle \text{bit } a \text{ } n \iff a \text{ AND push-bit } n \text{ } 1 \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-set-bit-iff [bit-simps]*:

$\langle \text{bit (set-bit } m \text{ } a) \text{ } n \iff \text{bit } a \text{ } n \vee (m = n \wedge \text{possible-bit TYPE('a) } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *even-set-bit-iff*:

$\langle \text{even (set-bit } m \text{ } a) \iff \text{even } a \wedge m \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-unset-bit-iff [bit-simps]*:

$\langle \text{bit (unset-bit } m \text{ } a) \text{ } n \iff \text{bit } a \text{ } n \wedge m \neq n \rangle$
 $\langle \text{proof} \rangle$

lemma *even-unset-bit-iff*:

$\langle \text{even (unset-bit } m \text{ } a) \iff \text{even } a \vee m = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-flip-bit-iff [bit-simps]*:

$\langle \text{bit (flip-bit } m \text{ } a) \text{ } n \iff (m = n \iff \neg \text{bit } a \text{ } n) \wedge \text{possible-bit TYPE('a) } n \rangle$
 $\langle \text{proof} \rangle$

lemma *even-flip-bit-iff*:

$\langle \text{even (flip-bit } m \text{ } a) \iff \neg (\text{even } a \iff m = 0) \rangle$
 $\langle \text{proof} \rangle$

lemma *and-exp-eq-0-iff-not-bit*:

$\langle a \text{ AND } 2 \wedge n = 0 \iff \neg \text{bit } a \text{ } n \rangle \text{ (is } \langle ?P \iff ?Q \rangle)$
 $\langle \text{proof} \rangle$

lemma *bit-sum-mult-2-cases*:

assumes a : $\langle \forall j. \neg \text{bit } a \text{ (Suc } j) \rangle$

shows $\langle \text{bit } (a + 2 * b) \ n = (\text{if } n = 0 \text{ then odd } a \text{ else bit } (2 * b) \ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *set-bit-0*:
 $\langle \text{set-bit } 0 \ a = 1 + 2 * (a \ \text{div } 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *set-bit-Suc*:
 $\langle \text{set-bit } (\text{Suc } n) \ a = a \ \text{mod } 2 + 2 * \text{set-bit } n \ (a \ \text{div } 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *unset-bit-0*:
 $\langle \text{unset-bit } 0 \ a = 2 * (a \ \text{div } 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *unset-bit-Suc*:
 $\langle \text{unset-bit } (\text{Suc } n) \ a = a \ \text{mod } 2 + 2 * \text{unset-bit } n \ (a \ \text{div } 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *flip-bit-0*:
 $\langle \text{flip-bit } 0 \ a = \text{of-bool } (\text{even } a) + 2 * (a \ \text{div } 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *flip-bit-Suc*:
 $\langle \text{flip-bit } (\text{Suc } n) \ a = a \ \text{mod } 2 + 2 * \text{flip-bit } n \ (a \ \text{div } 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *flip-bit-eq-if*:
 $\langle \text{flip-bit } n \ a = (\text{if bit } a \ n \text{ then unset-bit else set-bit}) \ n \ a \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-set-bit-eq*:
 $\langle \text{take-bit } n \ (\text{set-bit } m \ a) = (\text{if } n \leq m \text{ then take-bit } n \ a \text{ else set-bit } m \ (\text{take-bit } n \ a)) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-unset-bit-eq*:
 $\langle \text{take-bit } n \ (\text{unset-bit } m \ a) = (\text{if } n \leq m \text{ then take-bit } n \ a \text{ else unset-bit } m \ (\text{take-bit } n \ a)) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-flip-bit-eq*:
 $\langle \text{take-bit } n \ (\text{flip-bit } m \ a) = (\text{if } n \leq m \text{ then take-bit } n \ a \text{ else flip-bit } m \ (\text{take-bit } n \ a)) \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-Suc-numeral [simp]*:
 $\langle \text{push-bit } (\text{Suc } n) \ (\text{numeral } k) = \text{push-bit } n \ (\text{numeral } (\text{Num.Bit0 } k)) \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-eq-0-iff* [simp]:

$\langle \text{mask } n = 0 \longleftrightarrow n = 0 \rangle$

$\langle \text{proof} \rangle$

lemma *bit-horner-sum-bit-iff* [bit-simps]:

$\langle \text{bit } (\text{horner-sum of-bool } 2 \text{ } bs) \text{ } n \longleftrightarrow \text{possible-bit TYPE('a) } n \wedge n < \text{length } bs \wedge bs ! n \rangle$

$\langle \text{proof} \rangle$

lemma *horner-sum-bit-eq-take-bit*:

$\langle \text{horner-sum of-bool } 2 \text{ } (\text{map } (\text{bit } a) [0..<n]) = \text{take-bit } n \text{ } a \rangle$

$\langle \text{proof} \rangle$

lemma *take-bit-horner-sum-bit-eq*:

$\langle \text{take-bit } n \text{ } (\text{horner-sum of-bool } 2 \text{ } bs) = \text{horner-sum of-bool } 2 \text{ } (\text{take } n \text{ } bs) \rangle$

$\langle \text{proof} \rangle$

lemma *take-bit-sum*:

$\langle \text{take-bit } n \text{ } a = (\sum k = 0..<n. \text{push-bit } k \text{ } (\text{of-bool } (\text{bit } a \text{ } k))) \rangle$

$\langle \text{proof} \rangle$

lemma *set-bit-eq*:

$\langle \text{set-bit } n \text{ } a = a + \text{of-bool } (\neg \text{bit } a \text{ } n) * 2^n \rangle$

$\langle \text{proof} \rangle$

end

class *ring-bit-operations* = *semiring-bit-operations* + *ring-parity* +

fixes *not* :: $\langle 'a \Rightarrow 'a \rangle$ ($\langle \text{NOT} \rangle$)

assumes *not-eq-complement*: $\langle \text{NOT } a = - a - 1 \rangle$

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]:

$\langle (-1) \bmod 2 = 1 \rangle$

$\langle \text{proof} \rangle$

lemma *minus-eq-not-plus-1*:

$\langle - a = \text{NOT } a + 1 \rangle$

$\langle \text{proof} \rangle$

lemma *minus-eq-not-minus-1*:

$\langle - a = \text{NOT } (a - 1) \rangle$

$\langle \text{proof} \rangle$

lemma *not-rec*:

$\langle NOT\ a = of\text{-}bool\ (even\ a) + 2 * NOT\ (a\ div\ 2) \rangle$
 $\langle proof \rangle$

lemma *decr-eq-not-minus*:

$\langle a - 1 = NOT\ (-\ a) \rangle$
 $\langle proof \rangle$

lemma *even-not-iff* [simp]:

$\langle even\ (NOT\ a) \longleftrightarrow odd\ a \rangle$
 $\langle proof \rangle$

lemma *bit-not-iff* [bit-simps]:

$\langle bit\ (NOT\ a)\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \wedge \neg\ bit\ a\ n \rangle$
 $\langle proof \rangle$

lemma *bit-not-exp-iff* [bit-simps]:

$\langle bit\ (NOT\ (2 \wedge m))\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \wedge n \neq m \rangle$
 $\langle proof \rangle$

lemma *bit-minus-iff* [bit-simps]:

$\langle bit\ (-\ a)\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \wedge \neg\ bit\ (a - 1)\ n \rangle$
 $\langle proof \rangle$

lemma *bit-minus-1-iff* [simp]:

$\langle bit\ (-\ 1)\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \rangle$
 $\langle proof \rangle$

lemma *bit-minus-exp-iff* [bit-simps]:

$\langle bit\ (-\ (2 \wedge m))\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \wedge n \geq m \rangle$
 $\langle proof \rangle$

lemma *bit-minus-2-iff* [simp]:

$\langle bit\ (-\ 2)\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \wedge n > 0 \rangle$
 $\langle proof \rangle$

lemma *bit-decr-iff*:

$\langle bit\ (a - 1)\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \wedge \neg\ bit\ (-\ a)\ n \rangle$
 $\langle proof \rangle$

lemma *bit-not-iff-eq*:

$\langle bit\ (NOT\ a)\ n \longleftrightarrow 2 \wedge n \neq 0 \wedge \neg\ bit\ a\ n \rangle$
 $\langle proof \rangle$

lemma *not-one-eq* [simp]:

$\langle NOT\ 1 = -\ 2 \rangle$
 $\langle proof \rangle$

sublocale *and*: *semilattice-neutr* $\langle (AND) \rangle \leftarrow 1$

$\langle proof \rangle$

sublocale *bit: abstract-boolean-algebra* $\langle (AND) \rangle \langle (OR) \rangle NOT\ 0 \langle -\ 1 \rangle$
 $\langle proof \rangle$

sublocale *bit: abstract-boolean-algebra-sym-diff* $\langle (AND) \rangle \langle (OR) \rangle NOT\ 0 \langle -\ 1 \rangle$
 $\langle (XOR) \rangle$
 $\langle proof \rangle$

lemma *and-eq-not-not-or*:
 $\langle a\ AND\ b = NOT\ (NOT\ a\ OR\ NOT\ b) \rangle$
 $\langle proof \rangle$

lemma *or-eq-not-not-and*:
 $\langle a\ OR\ b = NOT\ (NOT\ a\ AND\ NOT\ b) \rangle$
 $\langle proof \rangle$

lemma *not-add-distrib*:
 $\langle NOT\ (a + b) = NOT\ a - b \rangle$
 $\langle proof \rangle$

lemma *not-diff-distrib*:
 $\langle NOT\ (a - b) = NOT\ a + b \rangle$
 $\langle proof \rangle$

lemma *and-eq-minus-1-iff*:
 $\langle a\ AND\ b = -\ 1 \iff a = -\ 1 \wedge b = -\ 1 \rangle$
 $\langle proof \rangle$

lemma *disjunctive-and-not-eq-xor*:
 $\langle a\ AND\ NOT\ b = a\ XOR\ b \rangle$ **if** $\langle NOT\ a\ AND\ b = 0 \rangle$
 $\langle proof \rangle$

lemma *disjunctive-diff-eq-and-not*:
 $\langle a - b = a\ AND\ NOT\ b \rangle$ **if** $\langle NOT\ a\ AND\ b = 0 \rangle$
 $\langle proof \rangle$

lemma *disjunctive-diff-eq-xor*:
 $\langle a\ AND\ NOT\ b = a\ XOR\ b \rangle$ **if** $\langle NOT\ a\ AND\ b = 0 \rangle$
 $\langle proof \rangle$

lemma *push-bit-minus*:
 $\langle push-bit\ n\ (-\ a) = -\ push-bit\ n\ a \rangle$
 $\langle proof \rangle$

lemma *take-bit-not-take-bit*:
 $\langle take-bit\ n\ (NOT\ (take-bit\ n\ a)) = take-bit\ n\ (NOT\ a) \rangle$
 $\langle proof \rangle$

lemma *take-bit-not-iff*:

$\langle \text{take-bit } n \text{ (NOT } a) = \text{take-bit } n \text{ (NOT } b) \longleftrightarrow \text{take-bit } n \text{ } a = \text{take-bit } n \text{ } b \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-not-eq-mask-diff*:

$\langle \text{take-bit } n \text{ (NOT } a) = \text{mask } n - \text{take-bit } n \text{ } a \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-eq-take-bit-minus-one*:

$\langle \text{mask } n = \text{take-bit } n \text{ (} - 1 \text{)} \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-minus-one-eq-mask* [simp]:

$\langle \text{take-bit } n \text{ (} - 1 \text{)} = \text{mask } n \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-exp-eq-not-mask*:

$\langle - (2 \wedge n) = \text{NOT (mask } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-minus-one-eq-not-mask* [simp]:

$\langle \text{push-bit } n \text{ (} - 1 \text{)} = \text{NOT (mask } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-not-mask-eq-0*:

$\langle \text{take-bit } m \text{ (NOT (mask } n)) = 0 \rangle \text{ if } \langle n \geq m \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-eq-minus-one-if-not-possible-bit*:

$\langle \text{mask } n = - 1 \rangle \text{ if } \langle \neg \text{possible-bit TYPE('a) } n \rangle$
 $\langle \text{proof} \rangle$

lemma *unset-bit-eq-and-not*:

$\langle \text{unset-bit } n \text{ } a = a \text{ AND NOT (push-bit } n \text{ } 1) \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-Suc-minus-numeral* [simp]:

$\langle \text{push-bit (Suc } n) \text{ (} - \text{ numeral } k) = \text{push-bit } n \text{ (} - \text{ numeral (Num.Bit0 } k)) \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-minus-numeral* [simp]:

$\langle \text{push-bit (numeral } l) \text{ (} - \text{ numeral } k) = \text{push-bit (pred-numeral } l) \text{ (} - \text{ numeral (Num.Bit0 } k)) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-Suc-minus-1-eq*:

$\langle \text{take-bit (Suc } n) \text{ (} - 1 \text{)} = 2 \wedge \text{Suc } n - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-numeral-minus-1-eq*:

$\langle \text{take-bit } (\text{numeral } k) \text{ } (-\ 1) = 2 \wedge \text{numeral } k - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-mask-eq*:

$\langle \text{push-bit } m \text{ } (\text{mask } n) = \text{mask } (n + m) \text{ AND NOT } (\text{mask } m) \rangle$
 $\langle \text{proof} \rangle$

lemma *slice-eq-mask*:

$\langle \text{push-bit } n \text{ } (\text{take-bit } m \text{ } (\text{drop-bit } n \ a)) = a \text{ AND mask } (m + n) \text{ AND NOT } (\text{mask } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-numeral-minus-1* [simp]:

$\langle \text{push-bit } (\text{numeral } n) \text{ } (-\ 1) = -\ (2 \wedge \text{numeral } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *unset-bit-eq*:

$\langle \text{unset-bit } n \ a = a - \text{of-bool } (\text{bit } a \ n) * 2 \wedge n \rangle$
 $\langle \text{proof} \rangle$

end

68.3 Common algebraic structure

class *linordered-euclidean-semiring-bit-operations* =
linordered-euclidean-semiring + *semiring-bit-operations*
begin

lemma *possible-bit* [simp]:

$\langle \text{possible-bit } \text{TYPE}(a) \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-of-exp* [simp]:

$\langle \text{take-bit } m \text{ } (2 \wedge n) = \text{of-bool } (n < m) * 2 \wedge n \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-of-2* [simp]:

$\langle \text{take-bit } n \ 2 = \text{of-bool } (2 \leq n) * 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-eq-0-iff* [simp]:

$\langle \text{push-bit } n \ a = 0 \longleftrightarrow a = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-add*:

$\langle \text{take-bit } n \text{ } (\text{take-bit } n \ a + \text{take-bit } n \ b) = \text{take-bit } n \ (a + b) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-of-1-eq-0-iff* [simp]:

$\langle \text{take-bit } n \ 1 = 0 \longleftrightarrow n = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-Suc-bit0* [simp]:
 $\langle \text{drop-bit } (\text{Suc } n) \ (\text{numeral } (\text{Num.Bit0 } k)) = \text{drop-bit } n \ (\text{numeral } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-Suc-bit1* [simp]:
 $\langle \text{drop-bit } (\text{Suc } n) \ (\text{numeral } (\text{Num.Bit1 } k)) = \text{drop-bit } n \ (\text{numeral } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-numeral-bit0* [simp]:
 $\langle \text{drop-bit } (\text{numeral } l) \ (\text{numeral } (\text{Num.Bit0 } k)) = \text{drop-bit } (\text{pred-numeral } l) \ (\text{numeral } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-numeral-bit1* [simp]:
 $\langle \text{drop-bit } (\text{numeral } l) \ (\text{numeral } (\text{Num.Bit1 } k)) = \text{drop-bit } (\text{pred-numeral } l) \ (\text{numeral } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-Suc-1* [simp]:
 $\langle \text{take-bit } (\text{Suc } n) \ 1 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-Suc-bit0*:
 $\langle \text{take-bit } (\text{Suc } n) \ (\text{numeral } (\text{Num.Bit0 } k)) = \text{take-bit } n \ (\text{numeral } k) * 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-Suc-bit1*:
 $\langle \text{take-bit } (\text{Suc } n) \ (\text{numeral } (\text{Num.Bit1 } k)) = \text{take-bit } n \ (\text{numeral } k) * 2 + 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-numeral-1* [simp]:
 $\langle \text{take-bit } (\text{numeral } l) \ 1 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-numeral-bit0*:
 $\langle \text{take-bit } (\text{numeral } l) \ (\text{numeral } (\text{Num.Bit0 } k)) = \text{take-bit } (\text{pred-numeral } l) \ (\text{numeral } k) * 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-numeral-bit1*:
 $\langle \text{take-bit } (\text{numeral } l) \ (\text{numeral } (\text{Num.Bit1 } k)) = \text{take-bit } (\text{pred-numeral } l) \ (\text{numeral } k) * 2 + 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-of-nat-iff-bit* [bit-simps]:
 $\langle \text{bit } (\text{of-nat } m) \ n \longleftrightarrow \text{bit } m \ n \rangle$

⟨proof⟩

lemma *drop-bit-mask-eq*:

⟨drop-bit m (mask n) = mask $(n - m)$ ⟩

⟨proof⟩

lemma *bit-push-bit-iff'*:

⟨bit (push-bit m a) n \longleftrightarrow $m \leq n \wedge$ bit a $(n - m)$ ⟩

⟨proof⟩

lemma *mask-half*:

⟨mask n div 2 = mask $(n - 1)$ ⟩

⟨proof⟩

lemma *take-bit-Suc-from-most*:

⟨take-bit (Suc n) a = $2 \wedge n$ * of-bool (bit a n) + take-bit n a ⟩

⟨proof⟩

lemma *take-bit-nonnegative* [simp]:

⟨ $0 \leq$ take-bit n a ⟩

⟨proof⟩

lemma *not-take-bit-negative* [simp]:

⟨ \neg take-bit n $a < 0$ ⟩

⟨proof⟩

lemma *bit-imp-take-bit-positive*:

⟨ $0 <$ take-bit m a ⟩ **if** ⟨ $n < m$ ⟩ **and** ⟨bit a n ⟩

⟨proof⟩

lemma *take-bit-mult*:

⟨take-bit n (take-bit n a * take-bit n b) = take-bit n (a * b)⟩

⟨proof⟩

lemma *drop-bit-push-bit*:

⟨drop-bit m (push-bit n a) = drop-bit $(m - n)$ (push-bit $(n - m)$ a)⟩

⟨proof⟩

end

68.4 Instance *int*

locale *fold2-bit-int* =

fixes $f :: \langle \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$

begin

context

begin


```

function  $F :: \langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$ 
  where  $\langle F\ k\ l = (\text{if } k \in \{0, -1\} \wedge l \in \{0, -1\}$ 
     $\text{then } -\text{of-bool } (f\ (\text{odd } k)\ (\text{odd } l))$ 
     $\text{else of-bool } (f\ (\text{odd } k)\ (\text{odd } l)) + 2 * (F\ (k\ \text{div } 2)\ (l\ \text{div } 2))) \rangle$ 
   $\langle \text{proof} \rangle$  termination  $\langle \text{proof} \rangle$ 

declare  $F.\text{sims}$  [ $\text{simp del}$ ]

lemma  $\text{rec}$ :
   $\langle F\ k\ l = \text{of-bool } (f\ (\text{odd } k)\ (\text{odd } l)) + 2 * (F\ (k\ \text{div } 2)\ (l\ \text{div } 2)) \rangle$ 
  for  $k\ l :: \text{int}$ 
   $\langle \text{proof} \rangle$ 

lemma  $\text{bit-iff}$ :
   $\langle \text{bit } (F\ k\ l)\ n \longleftrightarrow f\ (\text{bit } k\ n)\ (\text{bit } l\ n) \rangle$  for  $k\ l :: \text{int}$ 
   $\langle \text{proof} \rangle$ 

end

end

instantiation  $\text{int} :: \text{ring-bit-operations}$ 
begin

definition  $\text{not-int} :: \langle \text{int} \Rightarrow \text{int} \rangle$ 
  where  $\langle \text{not-int } k = -\ k - 1 \rangle$ 

global-interpretation  $\text{and-int}$ :  $\text{fold2-bit-int } \langle (\wedge) \rangle$ 
  defines  $\text{and-int} = \text{and-int}.F\ \langle \text{proof} \rangle$ 

global-interpretation  $\text{or-int}$ :  $\text{fold2-bit-int } \langle (\vee) \rangle$ 
  defines  $\text{or-int} = \text{or-int}.F\ \langle \text{proof} \rangle$ 

global-interpretation  $\text{xor-int}$ :  $\text{fold2-bit-int } \langle (\neq) \rangle$ 
  defines  $\text{xor-int} = \text{xor-int}.F\ \langle \text{proof} \rangle$ 

definition  $\text{mask-int} :: \langle \text{nat} \Rightarrow \text{int} \rangle$ 
  where  $\langle \text{mask } n = (2 :: \text{int})^{\wedge} n - 1 \rangle$ 

definition  $\text{push-bit-int} :: \langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$ 
  where  $\langle \text{push-bit-int } n\ k = k * 2^{\wedge} n \rangle$ 

definition  $\text{drop-bit-int} :: \langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$ 
  where  $\langle \text{drop-bit-int } n\ k = k\ \text{div } 2^{\wedge} n \rangle$ 

definition  $\text{take-bit-int} :: \langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$ 
  where  $\langle \text{take-bit-int } n\ k = k\ \text{mod } 2^{\wedge} n \rangle$ 

definition  $\text{set-bit-int} :: \langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$ 

```

where $\langle \text{set-bit } n \ k = k \text{ OR } \text{push-bit } n \ 1 \rangle$ **for** $k :: \text{int}$

definition $\text{unset-bit-int} :: \langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$

where $\langle \text{unset-bit } n \ k = k \text{ AND } \text{NOT } (\text{push-bit } n \ 1) \rangle$ **for** $k :: \text{int}$

definition $\text{flip-bit-int} :: \langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$

where $\langle \text{flip-bit } n \ k = k \text{ XOR } \text{push-bit } n \ 1 \rangle$ **for** $k :: \text{int}$

lemma not-int-div-2 :

$\langle \text{NOT } k \text{ div } 2 = \text{NOT } (k \text{ div } 2) \rangle$ **for** $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma bit-not-int-iff :

$\langle \text{bit } (\text{NOT } k) \ n \longleftrightarrow \neg \text{bit } k \ n \rangle$ **for** $k :: \text{int}$

$\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instance $\text{int} :: \text{linordered-euclidean-semiring-bit-operations}$ $\langle \text{proof} \rangle$

context $\text{ring-bit-operations}$

begin

lemma even-of-int-iff :

$\langle \text{even } (\text{of-int } k) \longleftrightarrow \text{even } k \rangle$

$\langle \text{proof} \rangle$

lemma bit-of-int-iff $[\text{bit-simps}]$:

$\langle \text{bit } (\text{of-int } k) \ n \longleftrightarrow \text{possible-bit } \text{TYPE}('a) \ n \wedge \text{bit } k \ n \rangle$

$\langle \text{proof} \rangle$

lemma push-bit-of-int :

$\langle \text{push-bit } n \ (\text{of-int } k) = \text{of-int } (\text{push-bit } n \ k) \rangle$

$\langle \text{proof} \rangle$

lemma of-int-push-bit :

$\langle \text{of-int } (\text{push-bit } n \ k) = \text{push-bit } n \ (\text{of-int } k) \rangle$

$\langle \text{proof} \rangle$

lemma take-bit-of-int :

$\langle \text{take-bit } n \ (\text{of-int } k) = \text{of-int } (\text{take-bit } n \ k) \rangle$

$\langle \text{proof} \rangle$

lemma of-int-take-bit :

$\langle \text{of-int } (\text{take-bit } n \ k) = \text{take-bit } n \ (\text{of-int } k) \rangle$

$\langle \text{proof} \rangle$

lemma *of-int-not-eq*:

$\langle \text{of-int } (\text{NOT } k) = \text{NOT } (\text{of-int } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *of-int-not-numeral*:

$\langle \text{of-int } (\text{NOT } (\text{numeral } k)) = \text{NOT } (\text{numeral } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *of-int-and-eq*:

$\langle \text{of-int } (k \text{ AND } l) = \text{of-int } k \text{ AND } \text{of-int } l \rangle$
 $\langle \text{proof} \rangle$

lemma *of-int-or-eq*:

$\langle \text{of-int } (k \text{ OR } l) = \text{of-int } k \text{ OR } \text{of-int } l \rangle$
 $\langle \text{proof} \rangle$

lemma *of-int-xor-eq*:

$\langle \text{of-int } (k \text{ XOR } l) = \text{of-int } k \text{ XOR } \text{of-int } l \rangle$
 $\langle \text{proof} \rangle$

lemma *of-int-mask-eq*:

$\langle \text{of-int } (\text{mask } n) = \text{mask } n \rangle$
 $\langle \text{proof} \rangle$

end

lemma *take-bit-int-less-exp* [simp]:

$\langle \text{take-bit } n \ k < 2 \wedge n \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-int-eq-self-iff*:

$\langle \text{take-bit } n \ k = k \longleftrightarrow 0 \leq k \wedge k < 2 \wedge n \rangle (\text{is } \langle ?P \longleftrightarrow ?Q \rangle)$
 $\text{for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-int-eq-self*:

$\langle \text{take-bit } n \ k = k \rangle \text{ if } \langle 0 \leq k \rangle \langle k < 2 \wedge n \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *mask-nonnegative-int* [simp]:

$\langle \text{mask } n \geq (0 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *not-mask-negative-int* [simp]:

$\langle \neg \text{mask } n < (0 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *not-nonnegative-int-iff* [simp]:

$\langle \text{NOT } k \geq 0 \longleftrightarrow k < 0 \rangle \text{ for } k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *not-negative-int-iff* [simp]:
 $\langle \text{NOT } k < 0 \longleftrightarrow k \geq 0 \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *and-nonnegative-int-iff* [simp]:
 $\langle k \text{ AND } l \geq 0 \longleftrightarrow k \geq 0 \vee l \geq 0 \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *and-negative-int-iff* [simp]:
 $\langle k \text{ AND } l < 0 \longleftrightarrow k < 0 \wedge l < 0 \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *and-less-eq*:
 $\langle k \text{ AND } l \leq k \rangle \text{ if } \langle l < 0 \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *or-nonnegative-int-iff* [simp]:
 $\langle k \text{ OR } l \geq 0 \longleftrightarrow k \geq 0 \wedge l \geq 0 \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *or-negative-int-iff* [simp]:
 $\langle k \text{ OR } l < 0 \longleftrightarrow k < 0 \vee l < 0 \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *or-greater-eq*:
 $\langle k \text{ OR } l \geq k \rangle \text{ if } \langle l \geq 0 \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *xor-nonnegative-int-iff* [simp]:
 $\langle k \text{ XOR } l \geq 0 \longleftrightarrow (k \geq 0 \longleftrightarrow l \geq 0) \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *xor-negative-int-iff* [simp]:
 $\langle k \text{ XOR } l < 0 \longleftrightarrow (k < 0) \neq (l < 0) \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *OR-upper*:
 $\langle x \text{ OR } y < 2^{\wedge n} \rangle \text{ if } \langle 0 \leq x \rangle \langle x < 2^{\wedge n} \rangle \langle y < 2^{\wedge n} \rangle \text{ for } x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *XOR-upper*:
 $\langle x \text{ XOR } y < 2^{\wedge n} \rangle \text{ if } \langle 0 \leq x \rangle \langle x < 2^{\wedge n} \rangle \langle y < 2^{\wedge n} \rangle \text{ for } x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *AND-lower* [simp]:
 $\langle 0 \leq x \text{ AND } y \rangle \text{ if } \langle 0 \leq x \rangle \text{ for } x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *OR-lower* [simp]:

$\langle 0 \leq x \text{ OR } y \rangle$ **if** $\langle 0 \leq x \rangle \langle 0 \leq y \rangle$ **for** $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *XOR-lower* [simp]:

$\langle 0 \leq x \text{ XOR } y \rangle$ **if** $\langle 0 \leq x \rangle \langle 0 \leq y \rangle$ **for** $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *AND-upper1* [simp]:

$\langle x \text{ AND } y \leq x \rangle$ **if** $\langle 0 \leq x \rangle$ **for** $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *AND-upper1'* [simp]:

$\langle y \text{ AND } x \leq z \rangle$ **if** $\langle 0 \leq y \rangle \langle y \leq z \rangle$ **for** $x \ y \ z :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *AND-upper1''* [simp]:

$\langle y \text{ AND } x < z \rangle$ **if** $\langle 0 \leq y \rangle \langle y < z \rangle$ **for** $x \ y \ z :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *AND-upper2* [simp]:

$\langle x \text{ AND } y \leq y \rangle$ **if** $\langle 0 \leq y \rangle$ **for** $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *AND-upper2'* [simp]:

$\langle x \text{ AND } y \leq z \rangle$ **if** $\langle 0 \leq y \rangle \langle y \leq z \rangle$ **for** $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *AND-upper2''* [simp]:

$\langle x \text{ AND } y < z \rangle$ **if** $\langle 0 \leq y \rangle \langle y < z \rangle$ **for** $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *plus-and-or*:

$\langle (x \text{ AND } y) + (x \text{ OR } y) = x + y \rangle$ **for** $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *push-bit-minus-one*:

$\langle \text{push-bit } n \ (-1 :: \text{int}) = -(2^n) \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-1-div-exp-eq-int*:

$\langle -1 \text{ div } (2 :: \text{int})^n = -1 \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-minus-one* [simp]:

$\langle \text{drop-bit } n \ (-1 :: \text{int}) = -1 \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-minus*:

$\langle \text{take-bit } n \ (- \ \text{take-bit } n \ k) = \text{take-bit } n \ (- \ k) \rangle$
for $k :: \text{int}$
 $\langle \text{proof} \rangle$

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 $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma (*in ring-1*) *of-nat-nat-take-bit-eq* [*simp*]:

$\langle \text{of-nat } (\text{nat } (\text{take-bit } n \ k)) = \text{of-int } (\text{take-bit } n \ k) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-minus-small-eq*:

$\langle \text{take-bit } n \ (- \ k) = 2^{\wedge n} - k \rangle$ **if** $\langle 0 < k \rangle \ \langle k \leq 2^{\wedge n} \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *push-bit-nonnegative-int-iff* [*simp*]:

$\langle \text{push-bit } n \ k \geq 0 \longleftrightarrow k \geq 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *push-bit-negative-int-iff* [*simp*]:

$\langle \text{push-bit } n \ k < 0 \longleftrightarrow k < 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *drop-bit-nonnegative-int-iff* [*simp*]:

$\langle \text{drop-bit } n \ k \geq 0 \longleftrightarrow k \geq 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *drop-bit-negative-int-iff* [*simp*]:

$\langle \text{drop-bit } n \ k < 0 \longleftrightarrow k < 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *set-bit-nonnegative-int-iff* [*simp*]:

$\langle \text{set-bit } n \ k \geq 0 \longleftrightarrow k \geq 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *set-bit-negative-int-iff* [*simp*]:

$\langle \text{set-bit } n \ k < 0 \longleftrightarrow k < 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *unset-bit-nonnegative-int-iff* [*simp*]:

$\langle \text{unset-bit } n \ k \geq 0 \longleftrightarrow k \geq 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *unset-bit-negative-int-iff* [*simp*]:

$\langle \text{unset-bit } n \ k < 0 \longleftrightarrow k < 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *flip-bit-nonnegative-int-iff* [simp]:
 $\langle \text{flip-bit } n \ k \geq 0 \longleftrightarrow k \geq 0 \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *flip-bit-negative-int-iff* [simp]:
 $\langle \text{flip-bit } n \ k < 0 \longleftrightarrow k < 0 \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *set-bit-greater-eq*:
 $\langle \text{set-bit } n \ k \geq k \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *unset-bit-less-eq*:
 $\langle \text{unset-bit } n \ k \leq k \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *and-int-unfold*:
 $\langle k \text{ AND } l = (\text{if } k = 0 \vee l = 0 \text{ then } 0 \text{ else if } k = -1 \text{ then } l \text{ else if } l = -1 \text{ then } k$
 $\text{ else } (k \bmod 2) * (l \bmod 2) + 2 * ((k \text{ div } 2) \text{ AND } (l \text{ div } 2))) \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *or-int-unfold*:
 $\langle k \text{ OR } l = (\text{if } k = -1 \vee l = -1 \text{ then } -1 \text{ else if } k = 0 \text{ then } l \text{ else if } l = 0 \text{ then } k$
 $\text{ else } \max (k \bmod 2) (l \bmod 2) + 2 * ((k \text{ div } 2) \text{ OR } (l \text{ div } 2))) \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *xor-int-unfold*:
 $\langle k \text{ XOR } l = (\text{if } k = -1 \text{ then } \text{NOT } l \text{ else if } l = -1 \text{ then } \text{NOT } k \text{ else if } k = 0$
 $\text{ then } l \text{ else if } l = 0 \text{ then } k$
 $\text{ else } |k \bmod 2 - l \bmod 2| + 2 * ((k \text{ div } 2) \text{ XOR } (l \text{ div } 2))) \rangle \text{ for } k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *bit-minus-int-iff*:
 $\langle \text{bit } (-k) \ n \longleftrightarrow \text{bit } (\text{NOT } (k - 1)) \ n \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-incr-eq*:
 $\langle \text{take-bit } n \ (k + 1) = 1 + \text{take-bit } n \ k \rangle \text{ if } \langle \text{take-bit } n \ k \neq 2^n - 1 \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-decr-eq*:
 $\langle \text{take-bit } n \ (k - 1) = \text{take-bit } n \ k - 1 \rangle \text{ if } \langle \text{take-bit } n \ k \neq 0 \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-int-greater-eq*:
 $\langle k + 2^n \leq \text{take-bit } n \ k \rangle \text{ if } \langle k < 0 \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-int-less-eq*:
 $\langle \text{take-bit } n \ k \leq k - 2 \wedge n \rangle$ **if** $\langle 2 \wedge n \leq k \rangle$ **and** $\langle n > 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-int-less-eq-self-iff*:
 $\langle \text{take-bit } n \ k \leq k \longleftrightarrow 0 \leq k \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$) **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-int-less-self-iff*:
 $\langle \text{take-bit } n \ k < k \longleftrightarrow 2 \wedge n \leq k \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-int-less-eq-mask*:
 $\langle \text{take-bit } n \ k \leq \text{mask } n \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-int-greater-self-iff*:
 $\langle k < \text{take-bit } n \ k \longleftrightarrow k < 0 \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-int-greater-eq-self-iff*:
 $\langle k \leq \text{take-bit } n \ k \longleftrightarrow k < 2 \wedge n \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-tightened-less-eq-int*:
 $\langle \text{take-bit } m \ k \leq \text{take-bit } n \ k \rangle$ **if** $\langle m \leq n \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *not-exp-less-eq-0-int* [simp]:
 $\langle \neg 2 \wedge n \leq (0 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *int-bit-bound*:
fixes $k :: \text{int}$
obtains n **where** $\langle \bigwedge m. n \leq m \implies \text{bit } k \ m \longleftrightarrow \text{bit } k \ n \rangle$
and $\langle n > 0 \implies \text{bit } k \ (n - 1) \neq \text{bit } k \ n \rangle$
 $\langle \text{proof} \rangle$

68.5 Instance *nat*

instantiation *nat* :: *semiring-bit-operations*
begin

definition *and-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle m \ \text{AND} \ n = \text{nat} \ (\text{int } m \ \text{AND} \ \text{int } n) \rangle$ **for** $m \ n :: \text{nat}$

definition *or-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle m \ \text{OR} \ n = \text{nat} \ (\text{int } m \ \text{OR} \ \text{int } n) \rangle$ **for** $m \ n :: \text{nat}$

definition *xor-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle m \text{ XOR } n = \text{nat } (\text{int } m \text{ XOR } \text{int } n) \rangle$ **for** $m \ n :: \text{nat}$

definition *mask-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{mask } n = (2 :: \text{nat}) \wedge n - 1 \rangle$

definition *push-bit-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{push-bit-nat } n \ m = m * 2 \wedge n \rangle$

definition *drop-bit-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{drop-bit-nat } n \ m = m \text{ div } 2 \wedge n \rangle$

definition *take-bit-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{take-bit-nat } n \ m = m \text{ mod } 2 \wedge n \rangle$

definition *set-bit-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{set-bit } m \ n = n \text{ OR } \text{push-bit } m \ 1 \rangle$ **for** $m \ n :: \text{nat}$

definition *unset-bit-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{unset-bit } m \ n = (n \text{ OR } \text{push-bit } m \ 1) \text{ XOR } \text{push-bit } m \ 1 \rangle$ **for** $m \ n :: \text{nat}$

definition *flip-bit-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{flip-bit } m \ n = n \text{ XOR } \text{push-bit } m \ 1 \rangle$ **for** $m \ n :: \text{nat}$

instance $\langle \text{proof} \rangle$

end

instance *nat* :: *linordered-euclidean-semiring-bit-operations* $\langle \text{proof} \rangle$

context *semiring-bit-operations*
begin

lemma *push-bit-of-nat*:
 $\langle \text{push-bit } n \ (\text{of-nat } m) = \text{of-nat } (\text{push-bit } n \ m) \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-push-bit*:
 $\langle \text{of-nat } (\text{push-bit } m \ n) = \text{push-bit } m \ (\text{of-nat } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-of-nat*:
 $\langle \text{take-bit } n \ (\text{of-nat } m) = \text{of-nat } (\text{take-bit } n \ m) \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-take-bit*:
 $\langle \text{of-nat } (\text{take-bit } n \ m) = \text{take-bit } n \ (\text{of-nat } m) \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-and-eq*:

$\langle \text{of-nat } (m \text{ AND } n) = \text{of-nat } m \text{ AND of-nat } n \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-or-eq*:

$\langle \text{of-nat } (m \text{ OR } n) = \text{of-nat } m \text{ OR of-nat } n \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-xor-eq*:

$\langle \text{of-nat } (m \text{ XOR } n) = \text{of-nat } m \text{ XOR of-nat } n \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-mask-eq*:

$\langle \text{of-nat } (\text{mask } n) = \text{mask } n \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-set-bit-eq*:

$\langle \text{of-nat } (\text{set-bit } n \ m) = \text{set-bit } n \ (\text{of-nat } m) \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-unset-bit-eq*:

$\langle \text{of-nat } (\text{unset-bit } n \ m) = \text{unset-bit } n \ (\text{of-nat } m) \rangle$
 $\langle \text{proof} \rangle$

lemma *of-nat-flip-bit-eq*:

$\langle \text{of-nat } (\text{flip-bit } n \ m) = \text{flip-bit } n \ (\text{of-nat } m) \rangle$
 $\langle \text{proof} \rangle$

end

context *linordered-euclidean-semiring-bit-operations*
begin

lemma *drop-bit-of-nat*:

$\text{drop-bit } n \ (\text{of-nat } m) = \text{of-nat } (\text{drop-bit } n \ m)$
 $\langle \text{proof} \rangle$

lemma *of-nat-drop-bit*:

$\langle \text{of-nat } (\text{drop-bit } m \ n) = \text{drop-bit } m \ (\text{of-nat } n) \rangle$
 $\langle \text{proof} \rangle$

end

lemma *take-bit-nat-less-exp* [simp]:

$\langle \text{take-bit } n \ m < 2 \wedge n \rangle \text{ for } n \ m :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *take-bit-nat-eq-self-iff*:

$\langle \text{take-bit } n \ m = m \longleftrightarrow m < 2 \wedge n \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$) **for** $n \ m :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *take-bit-nat-eq-self*:
 $\langle \text{take-bit } n \ m = m \rangle$ **if** $\langle m < 2 \wedge n \rangle$ **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *take-bit-nat-less-eq-self* [simp]:
 $\langle \text{take-bit } n \ m \leq m \rangle$ **for** $n \ m :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *take-bit-nat-less-self-iff*:
 $\langle \text{take-bit } n \ m < m \longleftrightarrow 2 \wedge n \leq m \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$) **for** $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *Suc-0-and-eq* [simp]:
 $\langle \text{Suc } 0 \text{ AND } n = n \bmod 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *and-Suc-0-eq* [simp]:
 $\langle n \text{ AND } \text{Suc } 0 = n \bmod 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *Suc-0-or-eq*:
 $\langle \text{Suc } 0 \text{ OR } n = n + \text{of-bool } (\text{even } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *or-Suc-0-eq*:
 $\langle n \text{ OR } \text{Suc } 0 = n + \text{of-bool } (\text{even } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *Suc-0-xor-eq*:
 $\langle \text{Suc } 0 \text{ XOR } n = n + \text{of-bool } (\text{even } n) - \text{of-bool } (\text{odd } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *xor-Suc-0-eq*:
 $\langle n \text{ XOR } \text{Suc } 0 = n + \text{of-bool } (\text{even } n) - \text{of-bool } (\text{odd } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *and-nat-unfold* [code]:
 $\langle m \text{ AND } n = (\text{if } m = 0 \vee n = 0 \text{ then } 0 \text{ else } (m \bmod 2) * (n \bmod 2) + 2 * ((m \div 2) \text{ AND } (n \div 2))) \rangle$
for $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *or-nat-unfold* [code]:
 $\langle m \text{ OR } n = (\text{if } m = 0 \text{ then } n \text{ else if } n = 0 \text{ then } m$
 $\text{else } \max (m \bmod 2) (n \bmod 2) + 2 * ((m \div 2) \text{ OR } (n \div 2))) \rangle$ **for** $m \ n ::$
 nat

⟨proof⟩

lemma *xor-nat-unfold* [code]:

⟨ $m \text{ XOR } n = (\text{if } m = 0 \text{ then } n \text{ else if } n = 0 \text{ then } m$
 $\text{else } (m \bmod 2 + n \bmod 2) \bmod 2 + 2 * ((m \text{ div } 2) \text{ XOR } (n \text{ div } 2)))$ ⟩ **for** $m \ n$
 $:: \text{ nat}$
 ⟨proof⟩

lemma [code]:

⟨ $\text{unset-bit } 0 \ m = 2 * (m \text{ div } 2)$ ⟩
 ⟨ $\text{unset-bit } (\text{Suc } n) \ m = m \bmod 2 + 2 * \text{unset-bit } n \ (m \text{ div } 2)$ ⟩ **for** $m \ n :: \text{ nat}$
 ⟨proof⟩

lemma *push-bit-of-Suc-0* [simp]:

⟨ $\text{push-bit } n \ (\text{Suc } 0) = 2 \wedge n$ ⟩
 ⟨proof⟩

lemma *take-bit-of-Suc-0* [simp]:

⟨ $\text{take-bit } n \ (\text{Suc } 0) = \text{of-bool } (0 < n)$ ⟩
 ⟨proof⟩

lemma *drop-bit-of-Suc-0* [simp]:

⟨ $\text{drop-bit } n \ (\text{Suc } 0) = \text{of-bool } (n = 0)$ ⟩
 ⟨proof⟩

lemma *Suc-mask-eq-exp*:

⟨ $\text{Suc } (\text{mask } n) = 2 \wedge n$ ⟩
 ⟨proof⟩

lemma *less-eq-mask*:

⟨ $n \leq \text{mask } n$ ⟩
 ⟨proof⟩

lemma *less-mask*:

⟨ $n < \text{mask } n$ ⟩ **if** ⟨ $\text{Suc } 0 < n$ ⟩
 ⟨proof⟩

lemma *mask-nat-less-exp* [simp]:

⟨ $(\text{mask } n :: \text{ nat}) < 2 \wedge n$ ⟩
 ⟨proof⟩

lemma *mask-nat-positive-iff* [simp]:

⟨ $(0 :: \text{ nat}) < \text{mask } n \longleftrightarrow 0 < n$ ⟩
 ⟨proof⟩

lemma *take-bit-nat-less-eq-mask*:

⟨ $\text{take-bit } n \ m \leq \text{mask } n$ ⟩ **for** $m :: \text{ nat}$
 ⟨proof⟩

lemma *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** $q :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *push-bit-nat-eq*:

$\langle \text{push-bit } n \ (\text{nat } k) = \text{nat } (\text{push-bit } n \ k) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-nat-eq*:

$\langle \text{drop-bit } n \ (\text{nat } k) = \text{nat } (\text{drop-bit } n \ k) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-nat-eq*:

$\langle \text{take-bit } n \ (\text{nat } k) = \text{nat } (\text{take-bit } n \ k) \rangle$ **if** $\langle k \geq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-take-bit-eq*:

$\langle \text{nat } (\text{take-bit } n \ k) = \text{take-bit } n \ (\text{nat } k) \rangle$
if $\langle k \geq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-mask-eq*:

$\langle \text{nat } (\text{mask } n) = \text{mask } n \rangle$
 $\langle \text{proof} \rangle$

68.6 Symbolic computations on numeral expressions

context *semiring-bits*

begin

lemma *bit-1-0* [*simp*]:

$\langle \text{bit } 1 \ 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *not-bit-1-Suc* [*simp*]:

$\langle \neg \text{bit } 1 \ (\text{Suc } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *not-bit-1-numeral* [*simp*]:

$\langle \neg \text{bit } 1 \ (\text{numeral } m) \rangle$
 $\langle \text{proof} \rangle$

lemma *not-bit-numeral-Bit0-0* [*simp*]:

$\langle \neg \text{bit } (\text{numeral } (\text{Num.Bit0 } m)) \ 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-Bit1-0* [*simp*]:

$\langle \text{bit } (\text{numeral } (\text{Num.Bit1 } m)) \ 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-Bit0-iff*:

$\langle \text{bit } (\text{numeral } (\text{num.Bit0 } m)) \ n$
 $\longleftrightarrow \text{possible-bit TYPE('a)} \ n \wedge n > 0 \wedge \text{bit } (\text{numeral } m) \ (n - 1) \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-Bit1-Suc-iff*:

$\langle \text{bit } (\text{numeral } (\text{num.Bit1 } m)) \ (\text{Suc } n)$
 $\longleftrightarrow \text{possible-bit TYPE('a)} \ (\text{Suc } n) \wedge \text{bit } (\text{numeral } m) \ n \rangle$
 $\langle \text{proof} \rangle$

end

context *ring-bit-operations*

begin

lemma *not-bit-minus-numeral-Bit0-0 [simp]*:

$\langle \neg \text{bit } (\neg \text{numeral } (\text{Num.Bit0 } m)) \ 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-minus-numeral-Bit1-0 [simp]*:

$\langle \text{bit } (\neg \text{numeral } (\text{Num.Bit1 } m)) \ 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-minus-numeral-Bit0-Suc-iff*:

$\langle \text{bit } (\neg \text{numeral } (\text{num.Bit0 } m)) \ (\text{Suc } n)$
 $\longleftrightarrow \text{possible-bit TYPE('a)} \ (\text{Suc } n) \wedge \text{bit } (\neg \text{numeral } m) \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-minus-numeral-Bit1-Suc-iff*:

$\langle \text{bit } (\neg \text{numeral } (\text{num.Bit1 } m)) \ (\text{Suc } n)$
 $\longleftrightarrow \text{possible-bit TYPE('a)} \ (\text{Suc } n) \wedge \neg \text{bit } (\text{numeral } m) \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-BitM-0 [simp]*:

$\langle \text{bit } (\text{numeral } (\text{Num.BitM } m)) \ 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-BitM-Suc-iff*:

$\langle \text{bit } (\text{numeral } (\text{Num.BitM } m)) \ (\text{Suc } n) \longleftrightarrow \text{possible-bit TYPE('a)} \ (\text{Suc } n) \wedge \neg$
 $\text{bit } (\neg \text{numeral } m) \ n \rangle$
 $\langle \text{proof} \rangle$

end

context *linordered-euclidean-semiring-bit-operations*

begin

lemma *bit-numeral-iff*:

$\langle \text{bit } (\text{numeral } m) \ n \longleftrightarrow \text{bit } (\text{numeral } m :: \text{nat}) \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-Bit0-Suc-iff* [simp]:
 $\langle \text{bit } (\text{numeral } (\text{Num.Bit0 } m)) \ (\text{Suc } n) \longleftrightarrow \text{bit } (\text{numeral } m) \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-Bit1-Suc-iff* [simp]:
 $\langle \text{bit } (\text{numeral } (\text{Num.Bit1 } m)) \ (\text{Suc } n) \longleftrightarrow \text{bit } (\text{numeral } m) \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-rec*:
 $\langle \text{bit } (\text{numeral } (\text{Num.Bit0 } w)) \ n \longleftrightarrow (\text{case } n \text{ of } 0 \Rightarrow \text{False} \mid \text{Suc } m \Rightarrow \text{bit } (\text{numeral } w) \ m) \rangle$
 $\langle \text{bit } (\text{numeral } (\text{Num.Bit1 } w)) \ n \longleftrightarrow (\text{case } n \text{ of } 0 \Rightarrow \text{True} \mid \text{Suc } m \Rightarrow \text{bit } (\text{numeral } w) \ m) \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-simps* [simp]:
 $\langle \text{bit } (\text{numeral } (\text{Num.Bit0 } w)) \ (\text{numeral } n) \longleftrightarrow \text{bit } (\text{numeral } w) \ (\text{pred-numeral } n) \rangle$
 $\langle \text{bit } (\text{numeral } (\text{Num.Bit1 } w)) \ (\text{numeral } n) \longleftrightarrow \text{bit } (\text{numeral } w) \ (\text{pred-numeral } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *and-numerals* [simp]:
 $\langle 1 \ \text{AND} \ \text{numeral } (\text{Num.Bit0 } y) = 0 \rangle$
 $\langle 1 \ \text{AND} \ \text{numeral } (\text{Num.Bit1 } y) = 1 \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \ \text{AND} \ \text{numeral } (\text{Num.Bit0 } y) = 2 * (\text{numeral } x \ \text{AND} \ \text{numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \ \text{AND} \ \text{numeral } (\text{Num.Bit1 } y) = 2 * (\text{numeral } x \ \text{AND} \ \text{numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \ \text{AND} \ 1 = 0 \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \ \text{AND} \ \text{numeral } (\text{Num.Bit0 } y) = 2 * (\text{numeral } x \ \text{AND} \ \text{numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \ \text{AND} \ \text{numeral } (\text{Num.Bit1 } y) = 1 + 2 * (\text{numeral } x \ \text{AND} \ \text{numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \ \text{AND} \ 1 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *or-numerals* [simp]:
 $\langle 1 \ \text{OR} \ \text{numeral } (\text{Num.Bit0 } y) = \text{numeral } (\text{Num.Bit1 } y) \rangle$
 $\langle 1 \ \text{OR} \ \text{numeral } (\text{Num.Bit1 } y) = \text{numeral } (\text{Num.Bit1 } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \ \text{OR} \ \text{numeral } (\text{Num.Bit0 } y) = 2 * (\text{numeral } x \ \text{OR} \ \text{numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \ \text{OR} \ \text{numeral } (\text{Num.Bit1 } y) = 1 + 2 * (\text{numeral } x \ \text{OR} \ \text{numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \ \text{OR} \ 1 = \text{numeral } (\text{Num.Bit1 } x) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \ \text{OR} \ \text{numeral } (\text{Num.Bit0 } y) = 1 + 2 * (\text{numeral } x \ \text{OR} \ \text{numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \ \text{OR} \ \text{numeral } (\text{Num.Bit1 } y) = 1 + 2 * (\text{numeral } x \ \text{OR} \ \text{numeral } y) \rangle$

numeral y)›
 ‹numeral (Num.Bit1 x) OR 1 = numeral (Num.Bit1 x)›
 ‹proof›

lemma xor-numerals [simp]:

‹1 XOR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)›
 ‹1 XOR numeral (Num.Bit1 y) = numeral (Num.Bit0 y)›
 ‹numeral (Num.Bit0 x) XOR numeral (Num.Bit0 y) = 2 * (numeral x XOR
 numeral y)›
 ‹numeral (Num.Bit0 x) XOR numeral (Num.Bit1 y) = 1 + 2 * (numeral x XOR
 numeral y)›
 ‹numeral (Num.Bit0 x) XOR 1 = numeral (Num.Bit1 x)›
 ‹numeral (Num.Bit1 x) XOR numeral (Num.Bit0 y) = 1 + 2 * (numeral x XOR
 numeral y)›
 ‹numeral (Num.Bit1 x) XOR numeral (Num.Bit1 y) = 2 * (numeral x XOR
 numeral y)›
 ‹numeral (Num.Bit1 x) XOR 1 = numeral (Num.Bit0 x)›
 ‹proof›

end

lemma drop-bit-Suc-minus-bit0 [simp]:

‹drop-bit (Suc n) (– numeral (Num.Bit0 k)) = drop-bit n (– numeral k :: int)›
 ‹proof›

lemma drop-bit-Suc-minus-bit1 [simp]:

‹drop-bit (Suc n) (– numeral (Num.Bit1 k)) = drop-bit n (– numeral (Num.inc
 k) :: int)›
 ‹proof›

lemma drop-bit-numeral-minus-bit0 [simp]:

‹drop-bit (numeral l) (– numeral (Num.Bit0 k)) = drop-bit (pred-numeral l) (–
 numeral k :: int)›
 ‹proof›

lemma drop-bit-numeral-minus-bit1 [simp]:

‹drop-bit (numeral l) (– numeral (Num.Bit1 k)) = drop-bit (pred-numeral l) (–
 numeral (Num.inc k) :: int)›
 ‹proof›

lemma take-bit-Suc-minus-bit0:

‹take-bit (Suc n) (– numeral (Num.Bit0 k)) = take-bit n (– numeral k) * (2 ::
 int)›
 ‹proof›

lemma take-bit-Suc-minus-bit1:

‹take-bit (Suc n) (– numeral (Num.Bit1 k)) = take-bit n (– numeral (Num.inc
 k)) * 2 + (1 :: int)›
 ‹proof›

lemma *take-bit-numeral-minus-bit0*:

$\langle \text{take-bit } (\text{numeral } l) \text{ } (- \text{ numeral } (\text{Num.Bit0 } k)) = \text{take-bit } (\text{pred-numeral } l) \text{ } (- \text{ numeral } k) * (2 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-numeral-minus-bit1*:

$\langle \text{take-bit } (\text{numeral } l) \text{ } (- \text{ numeral } (\text{Num.Bit1 } k)) = \text{take-bit } (\text{pred-numeral } l) \text{ } (- \text{ numeral } (\text{Num.inc } k)) * 2 + (1 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *and-nat-numerals [simp]*:

$\langle \text{Suc } 0 \text{ AND numeral } (\text{Num.Bit0 } y) = 0 \rangle$
 $\langle \text{Suc } 0 \text{ AND numeral } (\text{Num.Bit1 } y) = 1 \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \text{ AND Suc } 0 = 0 \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \text{ AND Suc } 0 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *or-nat-numerals [simp]*:

$\langle \text{Suc } 0 \text{ OR numeral } (\text{Num.Bit0 } y) = \text{numeral } (\text{Num.Bit1 } y) \rangle$
 $\langle \text{Suc } 0 \text{ OR numeral } (\text{Num.Bit1 } y) = \text{numeral } (\text{Num.Bit1 } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \text{ OR Suc } 0 = \text{numeral } (\text{Num.Bit1 } x) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \text{ OR Suc } 0 = \text{numeral } (\text{Num.Bit1 } x) \rangle$
 $\langle \text{proof} \rangle$

lemma *xor-nat-numerals [simp]*:

$\langle \text{Suc } 0 \text{ XOR numeral } (\text{Num.Bit0 } y) = \text{numeral } (\text{Num.Bit1 } y) \rangle$
 $\langle \text{Suc } 0 \text{ XOR numeral } (\text{Num.Bit1 } y) = \text{numeral } (\text{Num.Bit0 } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \text{ XOR Suc } 0 = \text{numeral } (\text{Num.Bit1 } x) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \text{ XOR Suc } 0 = \text{numeral } (\text{Num.Bit0 } x) \rangle$
 $\langle \text{proof} \rangle$

context *ring-bit-operations*

begin

lemma *minus-numeral-inc-eq*:

$\langle - \text{ numeral } (\text{Num.inc } n) = \text{NOT } (\text{numeral } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *sub-one-eq-not-neg*:

$\langle \text{Num.sub } n \text{ num.One} = \text{NOT } (- \text{ numeral } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-numeral-eq-not-sub-one*:

$\langle - \text{ numeral } n = \text{NOT } (\text{Num.sub } n \text{ num.One}) \rangle$
 $\langle \text{proof} \rangle$

lemma *not-numeral-eq [simp]*:

$\langle \text{NOT } (\text{numeral } n) = - \text{ numeral } (\text{Num.inc } n) \rangle$

⟨proof⟩

lemma *not-minus-numeral-eq* [simp]:

⟨NOT (− numeral n) = Num.sub n num.One⟩

⟨proof⟩

lemma *minus-not-numeral-eq* [simp]:

⟨− (NOT (numeral n)) = numeral (Num.inc n)⟩

⟨proof⟩

lemma *not-numeral-BitM-eq*:

⟨NOT (numeral (Num.BitM n)) = − numeral (num.Bit0 n)⟩

⟨proof⟩

lemma *not-numeral-Bit0-eq*:

⟨NOT (numeral (Num.Bit0 n)) = − numeral (num.Bit1 n)⟩

⟨proof⟩

end

lemma *bit-minus-numeral-int* [simp]:

⟨bit (− numeral (num.Bit0 w) :: int) (numeral n) ⟷ bit (− numeral w :: int)

(pred-numeral n)⟩

⟨bit (− numeral (num.Bit1 w) :: int) (numeral n) ⟷ ¬ bit (numeral w :: int)

(pred-numeral n)⟩

⟨proof⟩

lemma *bit-minus-numeral-Bit0-Suc-iff* [simp]:

⟨bit (− numeral (num.Bit0 w) :: int) (Suc n) ⟷ bit (− numeral w :: int) n⟩

⟨proof⟩

lemma *bit-minus-numeral-Bit1-Suc-iff* [simp]:

⟨bit (− numeral (num.Bit1 w) :: int) (Suc n) ⟷ ¬ bit (numeral w :: int) n⟩

⟨proof⟩

lemma *and-not-numerals*:

⟨1 AND NOT 1 = (0 :: int)⟩

⟨1 AND NOT (numeral (Num.Bit0 n)) = (1 :: int)⟩

⟨1 AND NOT (numeral (Num.Bit1 n)) = (0 :: int)⟩

⟨numeral (Num.Bit0 m) AND NOT (1 :: int) = numeral (Num.Bit0 m)⟩

⟨numeral (Num.Bit0 m) AND NOT (numeral (Num.Bit0 n)) = (2 :: int) *
(numeral m AND NOT (numeral n))⟩

⟨numeral (Num.Bit0 m) AND NOT (numeral (Num.Bit1 n)) = (2 :: int) *
(numeral m AND NOT (numeral n))⟩

⟨numeral (Num.Bit1 m) AND NOT (1 :: int) = numeral (Num.Bit0 m)⟩

⟨numeral (Num.Bit1 m) AND NOT (numeral (Num.Bit0 n)) = 1 + (2 :: int) *
(numeral m AND NOT (numeral n))⟩

⟨numeral (Num.Bit1 m) AND NOT (numeral (Num.Bit1 n)) = (2 :: int) *
(numeral m AND NOT (numeral n))⟩

⟨proof⟩

fun *and-not-num* :: ⟨num ⇒ num ⇒ num option⟩

where

⟨*and-not-num* num.One num.One = None⟩
 | ⟨*and-not-num* num.One (num.Bit0 n) = Some num.One⟩
 | ⟨*and-not-num* num.One (num.Bit1 n) = None⟩
 | ⟨*and-not-num* (num.Bit0 m) num.One = Some (num.Bit0 m)⟩
 | ⟨*and-not-num* (num.Bit0 m) (num.Bit0 n) = map-option num.Bit0 (*and-not-num* m n)⟩
 | ⟨*and-not-num* (num.Bit0 m) (num.Bit1 n) = map-option num.Bit0 (*and-not-num* m n)⟩
 | ⟨*and-not-num* (num.Bit1 m) num.One = Some (num.Bit0 m)⟩
 | ⟨*and-not-num* (num.Bit1 m) (num.Bit0 n) = (case *and-not-num* m n of None ⇒ Some num.One | Some n' ⇒ Some (num.Bit1 n'))⟩
 | ⟨*and-not-num* (num.Bit1 m) (num.Bit1 n) = map-option num.Bit0 (*and-not-num* m n)⟩

lemma *int-numeral-and-not-num*:

⟨numeral m AND NOT (numeral n) = (case *and-not-num* m n of None ⇒ 0 :: int | Some n' ⇒ numeral n')⟩
 ⟨proof⟩

lemma *int-numeral-not-and-num*:

⟨NOT (numeral m) AND numeral n = (case *and-not-num* n m of None ⇒ 0 :: int | Some n' ⇒ numeral n')⟩
 ⟨proof⟩

lemma *and-not-num-eq-None-iff*:

⟨*and-not-num* m n = None ⟷ numeral m AND NOT (numeral n) = (0 :: int)⟩
 ⟨proof⟩

lemma *and-not-num-eq-Some-iff*:

⟨*and-not-num* m n = Some q ⟷ numeral m AND NOT (numeral n) = (numeral q :: int)⟩
 ⟨proof⟩

lemma *and-minus-numerals [simp]*:

⟨1 AND − (numeral (num.Bit0 n)) = (0 :: int)⟩
 ⟨1 AND − (numeral (num.Bit1 n)) = (1 :: int)⟩
 ⟨numeral m AND − (numeral (num.Bit0 n)) = (case *and-not-num* m (Num.BitM n) of None ⇒ 0 :: int | Some n' ⇒ numeral n')⟩
 ⟨numeral m AND − (numeral (num.Bit1 n)) = (case *and-not-num* m (Num.Bit0 n) of None ⇒ 0 :: int | Some n' ⇒ numeral n')⟩
 ⟨− (numeral (num.Bit0 n)) AND 1 = (0 :: int)⟩
 ⟨− (numeral (num.Bit1 n)) AND 1 = (1 :: int)⟩
 ⟨− (numeral (num.Bit0 n)) AND numeral m = (case *and-not-num* m (Num.BitM n) of None ⇒ 0 :: int | Some n' ⇒ numeral n')⟩
 ⟨− (numeral (num.Bit1 n)) AND numeral m = (case *and-not-num* m (Num.Bit0 n) of None ⇒ 0 :: int | Some n' ⇒ numeral n')⟩

n) of $\text{None} \Rightarrow 0 :: \text{int} \mid \text{Some } n' \Rightarrow \text{numeral } n'\rangle$
 $\langle \text{proof} \rangle$

lemma *and-minus-minus-numerals* [simp]:

$\langle - (\text{numeral } m :: \text{int}) \text{ AND } - (\text{numeral } n :: \text{int}) = \text{NOT } ((\text{numeral } m - 1) \text{ OR } (\text{numeral } n - 1)) \rangle$
 $\langle \text{proof} \rangle$

lemma *or-not-numerals*:

$\langle 1 \text{ OR NOT } 1 = \text{NOT } (0 :: \text{int}) \rangle$
 $\langle 1 \text{ OR NOT } (\text{numeral } (\text{Num.Bit0 } n)) = \text{NOT } (\text{numeral } (\text{Num.Bit0 } n) :: \text{int}) \rangle$
 $\langle 1 \text{ OR NOT } (\text{numeral } (\text{Num.Bit1 } n)) = \text{NOT } (\text{numeral } (\text{Num.Bit0 } n) :: \text{int}) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } m) \text{ OR NOT } (1 :: \text{int}) = \text{NOT } (1 :: \text{int}) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } m) \text{ OR NOT } (\text{numeral } (\text{Num.Bit0 } n)) = 1 + (2 :: \text{int}) * (\text{numeral } m \text{ OR NOT } (\text{numeral } n)) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } m) \text{ OR NOT } (\text{numeral } (\text{Num.Bit1 } n)) = (2 :: \text{int}) * (\text{numeral } m \text{ OR NOT } (\text{numeral } n)) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } m) \text{ OR NOT } (1 :: \text{int}) = \text{NOT } (0 :: \text{int}) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } m) \text{ OR NOT } (\text{numeral } (\text{Num.Bit0 } n)) = 1 + (2 :: \text{int}) * (\text{numeral } m \text{ OR NOT } (\text{numeral } n)) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } m) \text{ OR NOT } (\text{numeral } (\text{Num.Bit1 } n)) = 1 + (2 :: \text{int}) * (\text{numeral } m \text{ OR NOT } (\text{numeral } n)) \rangle$
 $\langle \text{proof} \rangle$

fun *or-not-num-neg* :: $\langle \text{num} \Rightarrow \text{num} \Rightarrow \text{num} \rangle$

where

$\langle \text{or-not-num-neg } \text{num.One } \text{num.One} = \text{num.One} \rangle$
 $\mid \langle \text{or-not-num-neg } \text{num.One } (\text{num.Bit0 } m) = \text{num.Bit1 } m \rangle$
 $\mid \langle \text{or-not-num-neg } \text{num.One } (\text{num.Bit1 } m) = \text{num.Bit1 } m \rangle$
 $\mid \langle \text{or-not-num-neg } (\text{num.Bit0 } n) \text{ num.One} = \text{num.Bit0 } \text{num.One} \rangle$
 $\mid \langle \text{or-not-num-neg } (\text{num.Bit0 } n) (\text{num.Bit0 } m) = \text{Num.BitM } (\text{or-not-num-neg } n \text{ } m) \rangle$
 $\mid \langle \text{or-not-num-neg } (\text{num.Bit0 } n) (\text{num.Bit1 } m) = \text{num.Bit0 } (\text{or-not-num-neg } n \text{ } m) \rangle$
 $\mid \langle \text{or-not-num-neg } (\text{num.Bit1 } n) \text{ num.One} = \text{num.One} \rangle$
 $\mid \langle \text{or-not-num-neg } (\text{num.Bit1 } n) (\text{num.Bit0 } m) = \text{Num.BitM } (\text{or-not-num-neg } n \text{ } m) \rangle$
 $\mid \langle \text{or-not-num-neg } (\text{num.Bit1 } n) (\text{num.Bit1 } m) = \text{Num.BitM } (\text{or-not-num-neg } n \text{ } m) \rangle$

lemma *int-numeral-or-not-num-neg*:

$\langle \text{numeral } m \text{ OR NOT } (\text{numeral } n :: \text{int}) = - \text{numeral } (\text{or-not-num-neg } m \text{ } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *int-numeral-not-or-num-neg*:

$\langle \text{NOT } (\text{numeral } m) \text{ OR } (\text{numeral } n :: \text{int}) = - \text{numeral } (\text{or-not-num-neg } n \text{ } m) \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-or-not-num-eq*:

$\langle \text{numeral } (\text{or-not-num-neg } m \text{ } n) = - (\text{numeral } m \text{ OR NOT } (\text{numeral } n :: \text{int})) \rangle$

⟨proof⟩

lemma *or-minus-numerals* [simp]:

⟨1 OR − (numeral (num.Bit0 n)) = − (numeral (or-not-num-neg num.One (Num.BitM n)) :: int)⟩
 ⟨1 OR − (numeral (num.Bit1 n)) = − (numeral (num.Bit1 n) :: int)⟩
 ⟨numeral m OR − (numeral (num.Bit0 n)) = − (numeral (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 (or-not-num-neg m (Num.Bit0 n)) :: int)⟩
 ⟨proof⟩

lemma *or-minus-minus-numerals* [simp]:

⟨− (numeral m :: int) OR − (numeral n :: int) = NOT ((numeral m − 1) AND (numeral n − 1))⟩
 ⟨proof⟩

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
 ⟨proof⟩

definition *take-bit-num* :: ⟨nat ⇒ num ⇒ num option⟩

where ⟨take-bit-num n m =

(if take-bit n (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) =

$\text{Some } (\text{case take-bit-num } (\text{pred-numeral } r) \text{ } m \text{ of } \text{None} \Rightarrow \text{Num.One} \mid \text{Some } q \Rightarrow \text{Num.Bit1 } q) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-num-code* [code]:

— Ocaml-style pattern matching is more robust wrt. different representations of *nat*

$\langle \text{take-bit-num } n \text{ } m = (\text{case } (n, m)$
 $\text{of } (0, -) \Rightarrow \text{None}$
 $\mid (\text{Suc } n, \text{Num.One}) \Rightarrow \text{Some Num.One}$
 $\mid (\text{Suc } n, \text{Num.Bit0 } m) \Rightarrow (\text{case take-bit-num } n \text{ } m \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } q$
 $\Rightarrow \text{Some } (\text{Num.Bit0 } q))$
 $\mid (\text{Suc } n, \text{Num.Bit1 } m) \Rightarrow \text{Some } (\text{case take-bit-num } n \text{ } m \text{ of } \text{None} \Rightarrow \text{Num.One}$
 $\mid \text{Some } q \Rightarrow \text{Num.Bit1 } q)) \rangle$
 $\langle \text{proof} \rangle$

context *semiring-bit-operations*

begin

lemma *take-bit-num-eq-None-imp*:

$\langle \text{take-bit } m \text{ } (\text{numeral } n) = 0 \rangle \text{ if } \langle \text{take-bit-num } m \text{ } n = \text{None} \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-num-eq-Some-imp*:

$\langle \text{take-bit } m \text{ } (\text{numeral } n) = \text{numeral } q \rangle \text{ if } \langle \text{take-bit-num } m \text{ } n = \text{Some } q \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-numeral-numeral*:

$\langle \text{take-bit } (\text{numeral } m) \text{ } (\text{numeral } n) =$
 $(\text{case take-bit-num } (\text{numeral } m) \text{ } n \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } q \Rightarrow \text{numeral } q) \rangle$
 $\langle \text{proof} \rangle$

end

lemma *take-bit-numeral-minus-numeral-int*:

$\langle \text{take-bit } (\text{numeral } m) \text{ } (- \text{ numeral } n :: \text{int}) =$
 $(\text{case take-bit-num } (\text{numeral } m) \text{ } n \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } q \Rightarrow \text{take-bit } (\text{numeral } m) \text{ } (2 \wedge \text{ numeral } m - \text{ numeral } q)) \rangle$ (is $\langle ?lhs = ?rhs \rangle$)
 $\langle \text{proof} \rangle$

declare *take-bit-num-simps* [simp]

take-bit-numeral-numeral [simp]

take-bit-numeral-minus-numeral-int [simp]

68.7 Symbolic computations for code generation

lemma *bit-int-code* [code]:

$\langle \text{bit } (0 :: \text{int}) \text{ } n \quad \longleftrightarrow \text{False} \rangle$
 $\langle \text{bit } (\text{Int.Neg num.One}) \text{ } n \quad \longleftrightarrow \text{True} \rangle$

```

⟨bit (Int.Pos num.One) 0 ⟷ True⟩
⟨bit (Int.Pos (num.Bit0 m)) 0 ⟷ False⟩
⟨bit (Int.Pos (num.Bit1 m)) 0 ⟷ True⟩
⟨bit (Int.Neg (num.Bit0 m)) 0 ⟷ False⟩
⟨bit (Int.Neg (num.Bit1 m)) 0 ⟷ True⟩
⟨bit (Int.Pos num.One) (Suc n) ⟷ False⟩
⟨bit (Int.Pos (num.Bit0 m)) (Suc n) ⟷ bit (Int.Pos m) n⟩
⟨bit (Int.Pos (num.Bit1 m)) (Suc n) ⟷ bit (Int.Pos m) n⟩
⟨bit (Int.Neg (num.Bit0 m)) (Suc n) ⟷ bit (Int.Neg m) n⟩
⟨bit (Int.Neg (num.Bit1 m)) (Suc n) ⟷ bit (Int.Neg (Num.inc m)) n⟩
⟨proof⟩

```

lemma *not-int-code* [code]:

```

⟨NOT (0 :: int) = - 1⟩
⟨NOT (Int.Pos n) = Int.Neg (Num.inc n)⟩
⟨NOT (Int.Neg n) = Num.sub n num.One⟩
⟨proof⟩

```

fun *and-num* :: ⟨num ⇒ num ⇒ num option⟩

where

```

⟨and-num num.One num.One = Some num.One⟩
| ⟨and-num num.One (num.Bit0 n) = None⟩
| ⟨and-num num.One (num.Bit1 n) = Some num.One⟩
| ⟨and-num (num.Bit0 m) num.One = None⟩
| ⟨and-num (num.Bit0 m) (num.Bit0 n) = map-option num.Bit0 (and-num m n)⟩
| ⟨and-num (num.Bit0 m) (num.Bit1 n) = map-option num.Bit0 (and-num m n)⟩
| ⟨and-num (num.Bit1 m) num.One = Some num.One⟩
| ⟨and-num (num.Bit1 m) (num.Bit0 n) = map-option num.Bit0 (and-num m n)⟩
| ⟨and-num (num.Bit1 m) (num.Bit1 n) = (case and-num m n of None ⇒ Some
num.One | Some n' ⇒ Some (num.Bit1 n'))⟩

```

context *linordered-euclidean-semiring-bit-operations*

begin

lemma *numeral-and-num*:

```

⟨numeral m AND numeral n = (case and-num m n of None ⇒ 0 | Some n' ⇒
numeral n')⟩
⟨proof⟩

```

lemma *and-num-eq-None-iff*:

```

⟨and-num m n = None ⟷ numeral m AND numeral n = 0⟩
⟨proof⟩

```

lemma *and-num-eq-Some-iff*:

```

⟨and-num m n = Some q ⟷ numeral m AND numeral n = numeral q⟩
⟨proof⟩

```

end

lemma *and-int-code* [code]:
fixes $i\ j :: \text{int}$ **shows**
 $\langle 0 \text{ AND } j = 0 \rangle$
 $\langle i \text{ AND } 0 = 0 \rangle$
 $\langle \text{Int.Pos } n \text{ AND } \text{Int.Pos } m = (\text{case and-num } n \text{ m of None} \Rightarrow 0 \mid \text{Some } n' \Rightarrow \text{Int.Pos } n') \rangle$
 $\langle \text{Int.Neg } n \text{ AND } \text{Int.Neg } m = \text{NOT } (\text{Num.sub } n \text{ num.One OR Num.sub } m \text{ num.One}) \rangle$
 $\langle \text{Int.Pos } n \text{ AND } \text{Int.Neg num.One} = \text{Int.Pos } n \rangle$
 $\langle \text{Int.Pos } n \text{ AND } \text{Int.Neg } (\text{num.Bit0 } m) = \text{Num.sub } (\text{or-not-num-neg } (\text{Num.BitM } m) \text{ } n) \text{ num.One} \rangle$
 $\langle \text{Int.Pos } n \text{ AND } \text{Int.Neg } (\text{num.Bit1 } m) = \text{Num.sub } (\text{or-not-num-neg } (\text{num.Bit0 } m) \text{ } n) \text{ num.One} \rangle$
 $\langle \text{Int.Neg num.One AND } \text{Int.Pos } m = \text{Int.Pos } m \rangle$
 $\langle \text{Int.Neg } (\text{num.Bit0 } n) \text{ AND } \text{Int.Pos } m = \text{Num.sub } (\text{or-not-num-neg } (\text{Num.BitM } n) \text{ } m) \text{ num.One} \rangle$
 $\langle \text{Int.Neg } (\text{num.Bit1 } n) \text{ AND } \text{Int.Pos } m = \text{Num.sub } (\text{or-not-num-neg } (\text{num.Bit0 } n) \text{ } m) \text{ num.One} \rangle$
 $\langle \text{proof} \rangle$

context *linordered-euclidean-semiring-bit-operations*
begin

fun *or-num* :: $\langle \text{num} \Rightarrow \text{num} \Rightarrow \text{num} \rangle$
where
 $\langle \text{or-num num.One num.One} = \text{num.One} \rangle$
 $\mid \langle \text{or-num num.One } (\text{num.Bit0 } n) = \text{num.Bit1 } n \rangle$
 $\mid \langle \text{or-num num.One } (\text{num.Bit1 } n) = \text{num.Bit1 } n \rangle$
 $\mid \langle \text{or-num } (\text{num.Bit0 } m) \text{ num.One} = \text{num.Bit1 } m \rangle$
 $\mid \langle \text{or-num } (\text{num.Bit0 } m) (\text{num.Bit0 } n) = \text{num.Bit0 } (\text{or-num } m \text{ } n) \rangle$
 $\mid \langle \text{or-num } (\text{num.Bit0 } m) (\text{num.Bit1 } n) = \text{num.Bit1 } (\text{or-num } m \text{ } n) \rangle$
 $\mid \langle \text{or-num } (\text{num.Bit1 } m) \text{ num.One} = \text{num.Bit1 } m \rangle$
 $\mid \langle \text{or-num } (\text{num.Bit1 } m) (\text{num.Bit0 } n) = \text{num.Bit1 } (\text{or-num } m \text{ } n) \rangle$
 $\mid \langle \text{or-num } (\text{num.Bit1 } m) (\text{num.Bit1 } n) = \text{num.Bit1 } (\text{or-num } m \text{ } n) \rangle$

lemma *numeral-or-num*:
 $\langle \text{numeral } m \text{ OR numeral } n = \text{numeral } (\text{or-num } m \text{ } n) \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral-or-num-eq*:
 $\langle \text{numeral } (\text{or-num } m \text{ } n) = \text{numeral } m \text{ OR numeral } n \rangle$
 $\langle \text{proof} \rangle$

end

lemma *or-int-code* [code]:
fixes $i\ j :: \text{int}$ **shows**
 $\langle 0 \text{ OR } j = j \rangle$
 $\langle i \text{ OR } 0 = i \rangle$

$\langle \text{Int.Pos } n \text{ OR Int.Pos } m = \text{Int.Pos } (\text{or-num } n \ m) \rangle$
 $\langle \text{Int.Neg } n \text{ OR Int.Neg } m = \text{NOT } (\text{Num.sub } n \ \text{num.One AND Num.sub } m \ \text{num.One}) \rangle$
 $\langle \text{Int.Pos } n \text{ OR Int.Neg num.One} = \text{Int.Neg num.One} \rangle$
 $\langle \text{Int.Pos } n \text{ OR Int.Neg } (\text{num.Bit0 } m) = (\text{case and-not-num } (\text{Num.BitM } m) \ n \text{ of } \text{None} \Rightarrow -1 \mid \text{Some } n' \Rightarrow \text{Int.Neg } (\text{Num.inc } n')) \rangle$
 $\langle \text{Int.Pos } n \text{ OR Int.Neg } (\text{num.Bit1 } m) = (\text{case and-not-num } (\text{num.Bit0 } m) \ n \text{ of } \text{None} \Rightarrow -1 \mid \text{Some } n' \Rightarrow \text{Int.Neg } (\text{Num.inc } n')) \rangle$
 $\langle \text{Int.Neg num.One OR Int.Pos } m = \text{Int.Neg num.One} \rangle$
 $\langle \text{Int.Neg } (\text{num.Bit0 } n) \text{ OR Int.Pos } m = (\text{case and-not-num } (\text{Num.BitM } n) \ m \text{ of } \text{None} \Rightarrow -1 \mid \text{Some } n' \Rightarrow \text{Int.Neg } (\text{Num.inc } n')) \rangle$
 $\langle \text{Int.Neg } (\text{num.Bit1 } n) \text{ OR Int.Pos } m = (\text{case and-not-num } (\text{num.Bit0 } n) \ m \text{ of } \text{None} \Rightarrow -1 \mid \text{Some } n' \Rightarrow \text{Int.Neg } (\text{Num.inc } n')) \rangle$
 $\langle \text{proof} \rangle$

fun xor-num :: $\langle \text{num} \Rightarrow \text{num} \Rightarrow \text{num option} \rangle$

where

$\langle \text{xor-num num.One num.One} = \text{None} \rangle$
 $\mid \langle \text{xor-num num.One } (\text{num.Bit0 } n) = \text{Some } (\text{num.Bit1 } n) \rangle$
 $\mid \langle \text{xor-num num.One } (\text{num.Bit1 } n) = \text{Some } (\text{num.Bit0 } n) \rangle$
 $\mid \langle \text{xor-num } (\text{num.Bit0 } m) \ \text{num.One} = \text{Some } (\text{num.Bit1 } m) \rangle$
 $\mid \langle \text{xor-num } (\text{num.Bit0 } m) \ (\text{num.Bit0 } n) = \text{map-option num.Bit0 } (\text{xor-num } m \ n) \rangle$
 $\mid \langle \text{xor-num } (\text{num.Bit0 } m) \ (\text{num.Bit1 } n) = \text{Some } (\text{case xor-num } m \ n \text{ of } \text{None} \Rightarrow \text{num.One} \mid \text{Some } n' \Rightarrow \text{num.Bit1 } n') \rangle$
 $\mid \langle \text{xor-num } (\text{num.Bit1 } m) \ \text{num.One} = \text{Some } (\text{num.Bit0 } m) \rangle$
 $\mid \langle \text{xor-num } (\text{num.Bit1 } m) \ (\text{num.Bit0 } n) = \text{Some } (\text{case xor-num } m \ n \text{ of } \text{None} \Rightarrow \text{num.One} \mid \text{Some } n' \Rightarrow \text{num.Bit1 } n') \rangle$
 $\mid \langle \text{xor-num } (\text{num.Bit1 } m) \ (\text{num.Bit1 } n) = \text{map-option num.Bit0 } (\text{xor-num } m \ n) \rangle$

context linordered-euclidean-semiring-bit-operations

begin

lemma numeral-xor-num:

$\langle \text{numeral } m \text{ XOR numeral } n = (\text{case xor-num } m \ n \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } n' \Rightarrow \text{numeral } n') \rangle$
 $\langle \text{proof} \rangle$

lemma xor-num-eq-None-iff:

$\langle \text{xor-num } m \ n = \text{None} \iff \text{numeral } m \text{ XOR numeral } n = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma xor-num-eq-Some-iff:

$\langle \text{xor-num } m \ n = \text{Some } q \iff \text{numeral } m \text{ XOR numeral } n = \text{numeral } q \rangle$
 $\langle \text{proof} \rangle$

end

context semiring-bit-operations

begin

lemma *push-bit-eq-pow*:

$\langle \text{push-bit } (\text{numeral } n) \ 1 = \text{numeral } (\text{Num.pow } (\text{Num.Bit0 } \text{Num.One}) \ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *set-bit-of-0* [simp]:

$\langle \text{set-bit } n \ 0 = 2 \wedge n \rangle$
 $\langle \text{proof} \rangle$

lemma *unset-bit-of-0* [simp]:

$\langle \text{unset-bit } n \ 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *flip-bit-of-0* [simp]:

$\langle \text{flip-bit } n \ 0 = 2 \wedge n \rangle$
 $\langle \text{proof} \rangle$

lemma *set-bit-0-numeral-eq* [simp]:

$\langle \text{set-bit } 0 \ (\text{numeral } \text{Num.One}) = 1 \rangle$
 $\langle \text{set-bit } 0 \ (\text{numeral } (\text{Num.Bit0 } m)) = \text{numeral } (\text{Num.Bit1 } m) \rangle$
 $\langle \text{set-bit } 0 \ (\text{numeral } (\text{Num.Bit1 } m)) = \text{numeral } (\text{Num.Bit1 } m) \rangle$
 $\langle \text{proof} \rangle$

lemma *set-bit-numeral-eq-or* [simp]:

$\langle \text{set-bit } (\text{numeral } n) \ (\text{numeral } m) = \text{numeral } m \text{ OR } \text{push-bit } (\text{numeral } n) \ 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *unset-bit-0-numeral-eq-and-not'* [simp]:

$\langle \text{unset-bit } 0 \ (\text{numeral } \text{Num.One}) = 0 \rangle$
 $\langle \text{unset-bit } 0 \ (\text{numeral } (\text{Num.Bit0 } m)) = \text{numeral } (\text{Num.Bit0 } m) \rangle$
 $\langle \text{unset-bit } 0 \ (\text{numeral } (\text{Num.Bit1 } m)) = \text{numeral } (\text{Num.Bit0 } m) \rangle$
 $\langle \text{proof} \rangle$

lemma *unset-bit-numeral-eq-or* [simp]:

$\langle \text{unset-bit } (\text{numeral } n) \ (\text{numeral } m) =$
 $(\text{case and-not-num } m \ (\text{Num.pow } (\text{Num.Bit0 } \text{Num.One}) \ n)$
 $\text{of None} \Rightarrow 0$
 $\mid \text{Some } q \Rightarrow \text{numeral } q) \rangle \ (\text{is } \langle ?lhs = - \rangle)$
 $\langle \text{proof} \rangle$

lemma *flip-bit-0-numeral-eq-or* [simp]:

$\langle \text{flip-bit } 0 \ (\text{numeral } \text{Num.One}) = 0 \rangle$
 $\langle \text{flip-bit } 0 \ (\text{numeral } (\text{Num.Bit0 } m)) = \text{numeral } (\text{Num.Bit1 } m) \rangle$
 $\langle \text{flip-bit } 0 \ (\text{numeral } (\text{Num.Bit1 } m)) = \text{numeral } (\text{Num.Bit0 } m) \rangle$
 $\langle \text{proof} \rangle$

lemma *flip-bit-numeral-eq-xor* [simp]:

$\langle \text{flip-bit } (\text{numeral } n) \ (\text{numeral } m) = \text{numeral } m \text{ XOR } \text{push-bit } (\text{numeral } n) \ 1 \rangle$
 $\langle \text{proof} \rangle$

end

context *ring-bit-operations*
begin

lemma *set-bit-minus-numeral-eq-or* [simp]:

$\langle \text{set-bit } (\text{numeral } n) \text{ } (- \text{ numeral } m) = - \text{ numeral } m \text{ OR push-bit } (\text{numeral } n) \text{ } 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *unset-bit-minus-numeral-eq-and-not* [simp]:

$\langle \text{unset-bit } (\text{numeral } n) \text{ } (- \text{ numeral } m) = - \text{ numeral } m \text{ AND NOT } (\text{push-bit } (\text{numeral } n) \text{ } 1) \rangle$
 $\langle \text{proof} \rangle$

lemma *flip-bit-minus-numeral-eq-xor* [simp]:

$\langle \text{flip-bit } (\text{numeral } n) \text{ } (- \text{ numeral } m) = - \text{ numeral } m \text{ XOR push-bit } (\text{numeral } n) \text{ } 1 \rangle$
 $\langle \text{proof} \rangle$

end

lemma *xor-int-code* [code]:

fixes $i \ j :: \text{int}$ **shows**

$\langle 0 \text{ XOR } j = j \rangle$

$\langle i \text{ XOR } 0 = i \rangle$

$\langle \text{Int.Pos } n \text{ XOR Int.Pos } m = (\text{case xor-num } n \ m \text{ of None } \Rightarrow 0 \mid \text{Some } n' \Rightarrow \text{Int.Pos } n') \rangle$

$\langle \text{Int.Neg } n \text{ XOR Int.Neg } m = \text{Num.sub } n \text{ num.One XOR Num.sub } m \text{ num.One} \rangle$

$\langle \text{Int.Neg } n \text{ XOR Int.Pos } m = \text{NOT } (\text{Num.sub } n \text{ num.One XOR Int.Pos } m) \rangle$

$\langle \text{Int.Pos } n \text{ XOR Int.Neg } m = \text{NOT } (\text{Int.Pos } n \text{ XOR Num.sub } m \text{ num.One}) \rangle$

$\langle \text{proof} \rangle$

lemma *push-bit-int-code* [code]:

$\langle \text{push-bit } 0 \ i = i \rangle$

$\langle \text{push-bit } (\text{Suc } n) \ i = \text{push-bit } n \ (\text{Int.dup } i) \rangle$

$\langle \text{proof} \rangle$

lemma *drop-bit-int-code* [code]:

fixes $i :: \text{int}$ **shows**

$\langle \text{drop-bit } 0 \ i = i \rangle$

$\langle \text{drop-bit } (\text{Suc } n) \ 0 = (0 :: \text{int}) \rangle$

$\langle \text{drop-bit } (\text{Suc } n) \ (\text{Int.Pos num.One}) = 0 \rangle$

$\langle \text{drop-bit } (\text{Suc } n) \ (\text{Int.Pos } (\text{num.Bit0 } m)) = \text{drop-bit } n \ (\text{Int.Pos } m) \rangle$

$\langle \text{drop-bit } (\text{Suc } n) \ (\text{Int.Pos } (\text{num.Bit1 } m)) = \text{drop-bit } n \ (\text{Int.Pos } m) \rangle$

$\langle \text{drop-bit } (\text{Suc } n) \ (\text{Int.Neg num.One}) = - \ 1 \rangle$

$\langle \text{drop-bit } (\text{Suc } n) \ (\text{Int.Neg } (\text{num.Bit0 } m)) = \text{drop-bit } n \ (\text{Int.Neg } m) \rangle$

$\langle \text{drop-bit } (\text{Suc } n) \ (\text{Int.Neg } (\text{num.Bit1 } m)) = \text{drop-bit } n \ (\text{Int.Neg } (\text{Num.inc } m)) \rangle$

$\langle \text{proof} \rangle$

68.8 More properties

lemma *take-bit-eq-mask-iff*:

$\langle \text{take-bit } n \ k = \text{mask } n \iff \text{take-bit } n \ (k + 1) = 0 \rangle$ (**is** $\langle ?P \iff ?Q \rangle$)
for $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *take-bit-eq-mask-iff-exp-dvd*:

$\langle \text{take-bit } n \ k = \text{mask } n \iff 2 \wedge n \text{ dvd } k + 1 \rangle$
for $k :: \text{int}$
 $\langle \text{proof} \rangle$

68.9 Bit concatenation

definition *concat-bit* :: $\langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$

where $\langle \text{concat-bit } n \ k \ l = \text{take-bit } n \ k \text{ OR } \text{push-bit } n \ l \rangle$

lemma *bit-concat-bit-iff* [*bit-simps*]:

$\langle \text{bit } (\text{concat-bit } m \ k \ l) \ n \iff n < m \wedge \text{bit } k \ n \vee m \leq n \wedge \text{bit } l \ (n - m) \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-eq*:

$\langle \text{concat-bit } n \ k \ l = \text{take-bit } n \ k + \text{push-bit } n \ l \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-0* [*simp*]:

$\langle \text{concat-bit } 0 \ k \ l = l \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-Suc*:

$\langle \text{concat-bit } (\text{Suc } n) \ k \ l = k \bmod 2 + 2 * \text{concat-bit } n \ (k \text{ div } 2) \ l \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-of-zero-1* [*simp*]:

$\langle \text{concat-bit } n \ 0 \ l = \text{push-bit } n \ l \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-of-zero-2* [*simp*]:

$\langle \text{concat-bit } n \ k \ 0 = \text{take-bit } n \ k \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-nonnegative-iff* [*simp*]:

$\langle \text{concat-bit } n \ k \ l \geq 0 \iff l \geq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-negative-iff* [*simp*]:

$\langle \text{concat-bit } n \ k \ l < 0 \iff l < 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-assoc*:

$\langle \text{concat-bit } n \ k \ (\text{concat-bit } m \ l \ r) = \text{concat-bit } (m + n) \ (\text{concat-bit } n \ k \ l) \ r \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-assoc-sym*:

$\langle \text{concat-bit } m \ (\text{concat-bit } n \ k \ l) \ r = \text{concat-bit } (\min m \ n) \ k \ (\text{concat-bit } (m - n) \ l \ r) \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-eq-iff*:

$\langle \text{concat-bit } n \ k \ l = \text{concat-bit } n \ r \ s$
 $\longleftrightarrow \text{take-bit } n \ k = \text{take-bit } n \ r \wedge l = s \rangle$ **(is** $\langle ?P \longleftrightarrow ?Q \rangle$ **)**
 $\langle \text{proof} \rangle$

lemma *take-bit-concat-bit-eq*:

$\langle \text{take-bit } m \ (\text{concat-bit } n \ k \ l) = \text{concat-bit } (\min m \ n) \ k \ (\text{take-bit } (m - n) \ l) \rangle$
 $\langle \text{proof} \rangle$

lemma *concat-bit-take-bit-eq*:

$\langle \text{concat-bit } n \ (\text{take-bit } n \ b) = \text{concat-bit } n \ b \rangle$
 $\langle \text{proof} \rangle$

68.10 Taking bits with sign propagation

context *ring-bit-operations*

begin

definition *signed-take-bit* :: $\langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle$

where $\langle \text{signed-take-bit } n \ a = \text{take-bit } n \ a \ \text{OR} \ (\text{of-bool } (\text{bit } a \ n) * \text{NOT } (\text{mask } n)) \rangle$

lemma *signed-take-bit-eq-if-positive*:

$\langle \text{signed-take-bit } n \ a = \text{take-bit } n \ a \rangle$ **if** $\langle \neg \text{bit } a \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-eq-if-negative*:

$\langle \text{signed-take-bit } n \ a = \text{take-bit } n \ a \ \text{OR} \ \text{NOT } (\text{mask } n) \rangle$ **if** $\langle \text{bit } a \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *even-signed-take-bit-iff*:

$\langle \text{even } (\text{signed-take-bit } m \ a) \longleftrightarrow \text{even } a \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-signed-take-bit-iff* [*bit-simps*]:

$\langle \text{bit } (\text{signed-take-bit } m \ a) \ n \longleftrightarrow \text{possible-bit } \text{TYPE}('a) \ n \wedge \text{bit } a \ (\min m \ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-0* [*simp*]:

$\langle \text{signed-take-bit } 0 \ a = - (a \bmod 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-Suc*:

$\langle \text{signed-take-bit } (\text{Suc } n) \ a = a \bmod 2 + 2 * \text{signed-take-bit } n \ (a \text{ div } 2) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-of-0* [simp]:

$\langle \text{signed-take-bit } n \ 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-of-minus-1* [simp]:

$\langle \text{signed-take-bit } n \ (-1) = -1 \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-Suc-1* [simp]:

$\langle \text{signed-take-bit } (\text{Suc } n) \ 1 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-numeral-of-1* [simp]:

$\langle \text{signed-take-bit } (\text{numeral } k) \ 1 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-rec*:

$\langle \text{signed-take-bit } n \ a = (\text{if } n = 0 \text{ then } - (a \bmod 2) \text{ else } a \bmod 2 + 2 * \text{signed-take-bit } (n - 1) \ (a \text{ div } 2)) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-eq-iff-take-bit-eq*:

$\langle \text{signed-take-bit } n \ a = \text{signed-take-bit } n \ b \longleftrightarrow \text{take-bit } (\text{Suc } n) \ a = \text{take-bit } (\text{Suc } n) \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-signed-take-bit* [simp]:

$\langle \text{signed-take-bit } m \ (\text{signed-take-bit } n \ a) = \text{signed-take-bit } (\min m \ n) \ a \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-take-bit*:

$\langle \text{signed-take-bit } m \ (\text{take-bit } n \ a) = (\text{if } n \leq m \text{ then } \text{take-bit } n \text{ else } \text{signed-take-bit } m) \ a \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-signed-take-bit*:

$\langle \text{take-bit } m \ (\text{signed-take-bit } n \ a) = \text{take-bit } m \ a \rangle \text{ if } \langle m \leq \text{Suc } n \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-eq-take-bit-add*:

$\langle \text{signed-take-bit } n \ k = \text{take-bit } (\text{Suc } n) \ k + \text{NOT } (\text{mask } (\text{Suc } n)) * \text{of-bool } (\text{bit } k \ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-eq-take-bit-minus*:

$\langle \text{signed-take-bit } n \ k = \text{take-bit } (\text{Suc } n) \ k - 2^{\wedge} \text{Suc } n * \text{of-bool } (\text{bit } k \ n) \rangle$

$\langle \text{proof} \rangle$

end

Modulus centered around 0

lemma *signed-take-bit-eq-concat-bit*:

$\langle \text{signed-take-bit } n \ k = \text{concat-bit } n \ k \ (- \ \text{of-bool } (\text{bit } k \ n)) \rangle$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-add*:

$\langle \text{signed-take-bit } n \ (\text{signed-take-bit } n \ k + \text{signed-take-bit } n \ l) = \text{signed-take-bit } n \ (k + l) \rangle$

for $k \ l :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-diff*:

$\langle \text{signed-take-bit } n \ (\text{signed-take-bit } n \ k - \text{signed-take-bit } n \ l) = \text{signed-take-bit } n \ (k - l) \rangle$

for $k \ l :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-minus*:

$\langle \text{signed-take-bit } n \ (- \ \text{signed-take-bit } n \ k) = \text{signed-take-bit } n \ (- \ k) \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-mult*:

$\langle \text{signed-take-bit } n \ (\text{signed-take-bit } n \ k * \text{signed-take-bit } n \ l) = \text{signed-take-bit } n \ (k * l) \rangle$

for $k \ l :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-eq-take-bit-shift*:

$\langle \text{signed-take-bit } n \ k = \text{take-bit } (\text{Suc } n) \ (k + 2^{\wedge} n) - 2^{\wedge} n \rangle \text{ (is } \langle ?lhs = ?rhs \rangle)$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-nonnegative-iff [simp]*:

$\langle 0 \leq \text{signed-take-bit } n \ k \longleftrightarrow \neg \text{bit } k \ n \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-negative-iff [simp]*:

$\langle \text{signed-take-bit } n \ k < 0 \longleftrightarrow \text{bit } k \ n \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-int-greater-eq-minus-exp* [simp]:

$\langle -(2^n) \leq \text{signed-take-bit } n \ k \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-int-less-exp* [simp]:

$\langle \text{signed-take-bit } n \ k < 2^n \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-int-eq-self-iff*:

$\langle \text{signed-take-bit } n \ k = k \longleftrightarrow -(2^n) \leq k \wedge k < 2^n \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-int-eq-self*:

$\langle \text{signed-take-bit } n \ k = k \rangle \text{ if } \langle -(2^n) \leq k \rangle \langle k < 2^n \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

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 $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-int-less-self-iff*:

$\langle \text{signed-take-bit } n \ k < k \longleftrightarrow 2^n \leq k \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-int-greater-self-iff*:

$\langle k < \text{signed-take-bit } n \ k \longleftrightarrow k < -(2^n) \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-int-greater-eq-self-iff*:

$\langle k \leq \text{signed-take-bit } n \ k \longleftrightarrow k < 2^n \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-int-greater-eq*:

$\langle k + 2^n \text{ Suc } n \leq \text{signed-take-bit } n \ k \rangle \text{ if } \langle k < -(2^n) \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-int-less-eq*:

$\langle \text{signed-take-bit } n \ k \leq k - 2^n \text{ Suc } n \rangle \text{ if } \langle k \geq 2^n \rangle$

for $k :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-take-bit-Suc-sgn-eq* [simp]:

$\langle \text{signed-take-bit } (\text{Suc } n) (\text{sgn } k) = \text{sgn } k \rangle \text{ for } k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-Suc-bit0* [simp]:

$\langle \text{signed-take-bit } (\text{Suc } n) (\text{numeral } (\text{Num.Bit0 } k)) = \text{signed-take-bit } n (\text{numeral } k) \rangle$
 $\langle * (2 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-Suc-bit1* [simp]:

$\langle \text{signed-take-bit } (\text{Suc } n) (\text{numeral } (\text{Num.Bit1 } k)) = \text{signed-take-bit } n (\text{numeral } k) \rangle$
 $\langle * 2 + (1 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-Suc-minus-bit0* [simp]:

$\langle \text{signed-take-bit } (\text{Suc } n) (- \text{numeral } (\text{Num.Bit0 } k)) = \text{signed-take-bit } n (- \text{numeral } k) \rangle$
 $\langle * (2 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-Suc-minus-bit1* [simp]:

$\langle \text{signed-take-bit } (\text{Suc } n) (- \text{numeral } (\text{Num.Bit1 } k)) = \text{signed-take-bit } n (- \text{numeral } k - 1) \rangle$
 $\langle * 2 + (1 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-numeral-bit0* [simp]:

$\langle \text{signed-take-bit } (\text{numeral } l) (\text{numeral } (\text{Num.Bit0 } k)) = \text{signed-take-bit } (\text{pred-numeral } l) (\text{numeral } k) \rangle$
 $\langle * (2 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-numeral-bit1* [simp]:

$\langle \text{signed-take-bit } (\text{numeral } l) (\text{numeral } (\text{Num.Bit1 } k)) = \text{signed-take-bit } (\text{pred-numeral } l) (\text{numeral } k) \rangle$
 $\langle * 2 + (1 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-numeral-minus-bit0* [simp]:

$\langle \text{signed-take-bit } (\text{numeral } l) (- \text{numeral } (\text{Num.Bit0 } k)) = \text{signed-take-bit } (\text{pred-numeral } l) (- \text{numeral } k) \rangle$
 $\langle * (2 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-numeral-minus-bit1* [simp]:

$\langle \text{signed-take-bit } (\text{numeral } l) (- \text{numeral } (\text{Num.Bit1 } k)) = \text{signed-take-bit } (\text{pred-numeral } l) (- \text{numeral } k - 1) \rangle$
 $\langle * 2 + (1 :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-code* [code]:

$\langle \text{signed-take-bit } n a =$
 $(\text{let } l = \text{take-bit } (\text{Suc } n) a$
 $\text{in if bit } l n \text{ then } l + \text{push-bit } (\text{Suc } n) (- 1) \text{ else } l) \rangle$

<proof>

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 *int*, bounded bit values by quotient types over *int*.
- (A special case are idealized unbounded bit values ending in 0 which can be represented by type *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 *not* beyond that boundary.
- The projection on a single bit is then *bit a n = odd (a div 2ⁿ)*.
- This leads to the most fundamental properties of bit values:
 - Equality rule: $(\bigwedge n. \text{possible-bit TYPE(int)}\ n \implies \text{bit a } n = \text{bit b } n) \implies a = b$
 - Induction rule: $\llbracket \bigwedge a. a \text{ div } 2 = a \implies P\ a; \bigwedge a\ b. \llbracket P\ a; (\text{of-bool } b + 2 * a) \text{ div } 2 = a \rrbracket \implies P\ (\text{of-bool } b + 2 * a) \rrbracket \implies P\ a$
- 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: $\text{push-bit } n\ a = a * 2^n$
 - Right shift: $\text{drop-bit } n\ a = a \text{ div } 2^n$
 - Truncation: $\text{take-bit } n\ a = a \text{ mod } 2^n$

- Negation: $\text{bit } (\text{NOT } a) \ n = (\text{possible-bit } \text{TYPE}(\text{int}) \ n \wedge \neg \text{bit } a \ n)$
- And: $\text{bit } (a \ \text{AND } b) \ n = (\text{bit } a \ n \wedge \text{bit } b \ n)$
- Or: $\text{bit } (a \ \text{OR } b) \ n = (\text{bit } a \ n \vee \text{bit } b \ n)$
- Xor: $\text{bit } (a \ \text{XOR } b) \ n = (\text{bit } a \ n \neq \text{bit } b \ n)$
- Set a single bit: $\text{set-bit } n \ a = a \ \text{OR } \text{push-bit } n \ 1$
- Unset a single bit: $\text{unset-bit } n \ a = a \ \text{AND } \text{NOT } (\text{push-bit } n \ 1)$
- Flip a single bit: $\text{flip-bit } n \ a = a \ \text{XOR } \text{push-bit } n \ 1$
- Signed truncation, or modulus centered around 0: $\text{signed-take-bit } n \ a = \text{take-bit } n \ a \ \text{OR } \text{of-bool } (\text{bit } a \ n) * \text{NOT } (\text{mask } n)$
- Bit concatenation: $\text{concat-bit } n \ k \ l = \text{take-bit } n \ k \ \text{OR } \text{push-bit } n \ l$
- (Bounded) conversion from and to a list of bits: $\text{horner-sum of-bool } 2 \ (\text{map } (\text{bit } a) \ [0..<n]) = \text{take-bit } n \ a$

68.12 Lemma duplicates and other

context *semiring-bits*

begin

lemma *exp-div-exp-eq*:

$\langle 2 \wedge^m \text{div } 2 \wedge^n = \text{of-bool } (2 \wedge^m \neq 0 \wedge m \geq n) * 2 \wedge^{(m-n)} \rangle$
 $\langle \text{proof} \rangle$

lemma *bits-1-div-2*:

$\langle 1 \text{div } 2 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *bits-1-div-exp*:

$\langle 1 \text{div } 2 \wedge^n = \text{of-bool } (n = 0) \rangle$
 $\langle \text{proof} \rangle$

lemma *exp-add-not-zero-imp*:

$\langle 2 \wedge^m \neq 0 \rangle \ \text{and} \ \langle 2 \wedge^n \neq 0 \rangle \ \text{if} \ \langle 2 \wedge^{(m+n)} \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma

exp-add-not-zero-imp-left: $\langle 2 \wedge^m \neq 0 \rangle$
and *exp-add-not-zero-imp-right*: $\langle 2 \wedge^n \neq 0 \rangle$
if $\langle 2 \wedge^{(m+n)} \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *exp-not-zero-imp-exp-diff-not-zero*:

$\langle 2 \wedge^{(n-m)} \neq 0 \rangle \ \text{if} \ \langle 2 \wedge^n \neq 0 \rangle$

$\langle \text{proof} \rangle$

lemma *exp-eq-0-imp-not-bit*:

$\langle \neg \text{bit } a \ n \rangle \text{ if } \langle 2^n = 0 \rangle$

$\langle \text{proof} \rangle$

lemma *bit-disjunctive-add-iff*:

$\langle \text{bit } (a + b) \ n \longleftrightarrow \text{bit } a \ n \vee \text{bit } b \ n \rangle$

if $\langle \bigwedge n. \neg \text{bit } a \ n \vee \neg \text{bit } b \ n \rangle$

$\langle \text{proof} \rangle$

end

context *semiring-bit-operations*

begin

lemma *even-mask-div-iff*:

$\langle \text{even } ((2^m - 1) \text{ div } 2^n) \longleftrightarrow 2^n = 0 \vee m \leq n \rangle$

$\langle \text{proof} \rangle$

lemma *mod-exp-eq*:

$\langle a \text{ mod } 2^m \text{ mod } 2^n = a \text{ mod } 2^{\min m \ n} \rangle$

$\langle \text{proof} \rangle$

lemma *mult-exp-mod-exp-eq*:

$\langle m \leq n \implies (a * 2^m) \text{ mod } (2^n) = (a \text{ mod } 2^{(n - m)}) * 2^m \rangle$

$\langle \text{proof} \rangle$

lemma *div-exp-mod-exp-eq*:

$\langle a \text{ div } 2^n \text{ mod } 2^m = a \text{ mod } (2^{(n + m)}) \text{ div } 2^n \rangle$

$\langle \text{proof} \rangle$

lemma *even-mult-exp-div-exp-iff*:

$\langle \text{even } (a * 2^m \text{ div } 2^n) \longleftrightarrow m > n \vee 2^n = 0 \vee (m \leq n \wedge \text{even } (a \text{ div } 2^{(n - m)})) \rangle$

$\langle \text{proof} \rangle$

lemma *mod-exp-div-exp-eq-0*:

$\langle a \text{ mod } 2^n \text{ div } 2^n = 0 \rangle$

$\langle \text{proof} \rangle$

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*:

$\langle a + b = a \text{ OR } b \rangle$ **if** $\langle \bigwedge n. \neg \text{bit } a \ n \vee \neg \text{bit } b \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *even-mod-exp-div-exp-iff*:

$\langle \text{even } (a \bmod 2^m \div 2^n) \longleftrightarrow m \leq n \vee \text{even } (a \div 2^n) \rangle$
 $\langle \text{proof} \rangle$

end

context *ring-bit-operations*

begin

lemma *disjunctive-diff*:

$\langle a - b = a \text{ AND NOT } b \rangle$ **if** $\langle \bigwedge n. \text{bit } b \ n \implies \text{bit } a \ n \rangle$
 $\langle \text{proof} \rangle$

end

lemma *and-nat-rec*:

$\langle m \text{ AND } n = \text{of_bool } (\text{odd } m \wedge \text{odd } n) + 2 * ((m \div 2) \text{ AND } (n \div 2)) \rangle$ **for** m
 $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *or-nat-rec*:

$\langle m \text{ OR } n = \text{of_bool } (\text{odd } m \vee \text{odd } n) + 2 * ((m \div 2) \text{ OR } (n \div 2)) \rangle$ **for** $m \ n$
 $:: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *xor-nat-rec*:

$\langle m \text{ XOR } n = \text{of_bool } (\text{odd } m \neq \text{odd } n) + 2 * ((m \div 2) \text{ XOR } (n \div 2)) \rangle$ **for** m
 $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *bit-push-bit-iff-nat*:

$\langle \text{bit } (\text{push-bit } m \ q) \ n \longleftrightarrow m \leq n \wedge \text{bit } q \ (n - m) \rangle$ **for** $q :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *mask-half-int*:

$\langle \text{mask } n \div 2 = (\text{mask } (n - 1)) :: \text{int} \rangle$
 $\langle \text{proof} \rangle$

lemma *not-int-rec*:

$\langle \text{NOT } k = \text{of_bool } (\text{even } k) + 2 * \text{NOT } (k \div 2) \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *even-not-iff-int*:

$\langle \text{even } (\text{NOT } k) \longleftrightarrow \text{odd } k \rangle$ **for** $k :: \text{int}$
 $\langle \text{proof} \rangle$

```

lemma bit-not-int-iff':
   $\langle \text{bit } (-k - 1) \ n \longleftrightarrow \neg \text{bit } k \ n \rangle$  for  $k :: \text{int}$ 
   $\langle \text{proof} \rangle$ 

lemmas and-int-rec = and-int.rec

lemma even-and-iff-int:
   $\langle \text{even } (k \text{ AND } l) \longleftrightarrow \text{even } k \vee \text{even } l \rangle$  for  $k \ l :: \text{int}$ 
   $\langle \text{proof} \rangle$ 

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:
   $\langle \text{drop-bit } m \ (\text{push-bit } n \ k) = \text{drop-bit } (m - n) \ (\text{push-bit } (n - m) \ k) \rangle$  for  $k :: \text{int}$ 
   $\langle \text{proof} \rangle$ 

lemma bit-push-bit-iff-int:
   $\langle \text{bit } (\text{push-bit } m \ k) \ n \longleftrightarrow m \leq n \wedge \text{bit } k \ (n - m) \rangle$  for  $k :: \text{int}$ 
   $\langle \text{proof} \rangle$ 

bundle bit-operations-syntax
begin
notation
  not ( $\langle \text{NOT} \rangle$ )
  and and (infixr  $\langle \text{AND} \rangle$  64)
  and or (infixr  $\langle \text{OR} \rangle$  59)
  and xor (infixr  $\langle \text{XOR} \rangle$  59)
end

unbundle no bit-operations-syntax

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
  morphisms int-of-integer integer-of-int <proof>
```

```
setup-lifting type-definition-integer
```

```
lemma integer-eq-iff:
   $k = l \longleftrightarrow \text{int-of-integer } k = \text{int-of-integer } l$ 
  <proof>
```

```
lemma integer-eqI:
   $\text{int-of-integer } k = \text{int-of-integer } l \implies k = l$ 
  <proof>
```

```
lemma int-of-integer-integer-of-int [simp]:
   $\text{int-of-integer } (\text{integer-of-int } k) = k$ 
  <proof>
```

```
lemma integer-of-int-int-of-integer [simp]:
   $\text{integer-of-int } (\text{int-of-integer } k) = k$ 
  <proof>
```

```
instantiation integer :: ring-1
begin
```

```
lift-definition zero-integer :: integer
  is 0 :: int
  <proof>
```

```
declare zero-integer.rep-eq [simp]
```

```
lift-definition one-integer :: integer
  is 1 :: int
  <proof>
```

```
declare one-integer.rep-eq [simp]
```

```
lift-definition plus-integer :: integer  $\Rightarrow$  integer  $\Rightarrow$  integer
  is plus :: int  $\Rightarrow$  int  $\Rightarrow$  int
  <proof>
```

```
declare plus-integer.rep-eq [simp]
```

```
lift-definition uminus-integer :: integer  $\Rightarrow$  integer
  is uminus :: int  $\Rightarrow$  int
  <proof>
```

```
declare uminus-integer.rep-eq [simp]
```

lift-definition *minus-integer* :: *integer* \Rightarrow *integer* \Rightarrow *integer*
is *minus* :: *int* \Rightarrow *int* \Rightarrow *int*
 \langle *proof* \rangle

declare *minus-integer.rep-eq* [*simp*]

lift-definition *times-integer* :: *integer* \Rightarrow *integer* \Rightarrow *integer*
is *times* :: *int* \Rightarrow *int* \Rightarrow *int*
 \langle *proof* \rangle

declare *times-integer.rep-eq* [*simp*]

instance \langle *proof* \rangle

end

instance *integer* :: *Rings.dvd* \langle *proof* \rangle

context
includes *lifting-syntax*
notes *transfer-rule-numeral* [*transfer-rule*]
begin

lemma [*transfer-rule*]:
 $(\text{pcr-integer} == => \text{pcr-integer} == => (\longleftrightarrow)) (\text{dvd}) (\text{dvd})$
 \langle *proof* \rangle

lemma [*transfer-rule*]:
 $((\longleftrightarrow) == => \text{pcr-integer}) \text{ of-bool of-bool}$
 \langle *proof* \rangle

lemma [*transfer-rule*]:
 $((=) == => \text{pcr-integer}) \text{ int of-nat}$
 \langle *proof* \rangle

lemma [*transfer-rule*]:
 $((=) == => \text{pcr-integer}) (\lambda k. k) \text{ of-int}$
 \langle *proof* \rangle

lemma [*transfer-rule*]:
 $((=) == => \text{pcr-integer}) \text{ numeral numeral}$
 \langle *proof* \rangle

lemma [*transfer-rule*]:
 $((=) == => (=) == => \text{pcr-integer}) \text{ Num.sub Num.sub}$
 \langle *proof* \rangle

lemma [*transfer-rule*]:
 $(\text{pcr-integer} == => (=) == => \text{pcr-integer}) (\frown) (\frown)$

$\langle \text{proof} \rangle$

end

lemma *int-of-integer-of-nat* [simp]:
 $\text{int-of-integer } (\text{of-nat } n) = \text{of-nat } n$
 $\langle \text{proof} \rangle$

lift-definition *integer-of-nat* :: $\text{nat} \Rightarrow \text{integer}$
is *of-nat* :: $\text{nat} \Rightarrow \text{int}$
 $\langle \text{proof} \rangle$

lemma *integer-of-nat-eq-of-nat* [code]:
 $\text{integer-of-nat} = \text{of-nat}$
 $\langle \text{proof} \rangle$

lemma *int-of-integer-integer-of-nat* [simp]:
 $\text{int-of-integer } (\text{integer-of-nat } n) = \text{of-nat } n$
 $\langle \text{proof} \rangle$

lift-definition *nat-of-integer* :: $\text{integer} \Rightarrow \text{nat}$
is *Int.nat*
 $\langle \text{proof} \rangle$

lemma *nat-of-integer-0* [simp]:
 $\langle \text{nat-of-integer } 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-of-integer-1* [simp]:
 $\langle \text{nat-of-integer } 1 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-of-integer-numeral* [simp]:
 $\langle \text{nat-of-integer } (\text{numeral } n) = \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-of-integer-of-nat* [simp]:
 $\text{nat-of-integer } (\text{of-nat } n) = n$
 $\langle \text{proof} \rangle$

lemma *int-of-integer-of-int* [simp]:
 $\text{int-of-integer } (\text{of-int } k) = k$
 $\langle \text{proof} \rangle$

lemma *nat-of-integer-integer-of-nat* [simp]:
 $\text{nat-of-integer } (\text{integer-of-nat } n) = n$
 $\langle \text{proof} \rangle$

lemma *integer-of-int-eq-of-int* [simp, code-abbrev]:

integer-of-int = *of-int*
 ⟨*proof*⟩

lemma *of-int-integer-of* [*simp*]:
of-int (*int-of-integer* *k*) = (*k* :: *integer*)
 ⟨*proof*⟩

lemma *int-of-integer-numeral* [*simp*]:
int-of-integer (*numeral* *k*) = *numeral* *k*
 ⟨*proof*⟩

lemma *int-of-integer-sub* [*simp*]:
int-of-integer (*Num.sub* *k* *l*) = *Num.sub* *k* *l*
 ⟨*proof*⟩

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)
 ⟨*proof*⟩

lemma *integer-of-num-triv*:
integer-of-num *Num.One* = 1
integer-of-num (*Num.Bit0* *Num.One*) = 2
 ⟨*proof*⟩

instantiation *integer* :: *equal*
begin

lift-definition *equal-integer* :: ⟨*integer* ⇒ *integer* ⇒ *bool*⟩
 is ⟨*HOL.equal* :: *int* ⇒ *int* ⇒ *bool*⟩
 ⟨*proof*⟩

instance
 ⟨*proof*⟩

end

instantiation *integer* :: *linordered-idom*
begin

lift-definition *abs-integer* :: ⟨*integer* ⇒ *integer*⟩
 is ⟨*abs* :: *int* ⇒ *int*⟩
 ⟨*proof*⟩

declare *abs-integer.rep-eq* [*simp*]

```

lift-definition sgn-integer ::  $\langle integer \Rightarrow integer \rangle$ 
  is  $\langle sgn :: int \Rightarrow int \rangle$ 
   $\langle proof \rangle$ 

declare sgn-integer.rep-eq [simp]

lift-definition less-eq-integer ::  $\langle integer \Rightarrow integer \Rightarrow bool \rangle$ 
  is  $\langle less-eq :: int \Rightarrow int \Rightarrow bool \rangle$ 
   $\langle proof \rangle$ 

lemma integer-less-eq-iff:
   $\langle k \leq l \longleftrightarrow int-of-integer\ k \leq int-of-integer\ l \rangle$ 
   $\langle proof \rangle$ 

lift-definition less-integer ::  $\langle integer \Rightarrow integer \Rightarrow bool \rangle$ 
  is  $\langle less :: int \Rightarrow int \Rightarrow bool \rangle$ 
   $\langle proof \rangle$ 

lemma integer-less-iff:
   $\langle k < l \longleftrightarrow int-of-integer\ k < int-of-integer\ l \rangle$ 
   $\langle proof \rangle$ 

instance
   $\langle proof \rangle$ 

end

instance integer :: discrete-linordered-semidom
   $\langle proof \rangle$ 

context
  includes lifting-syntax
begin

lemma [transfer-rule]:
   $\langle (pcr-integer ==> pcr-integer ==> pcr-integer)\ min\ min \rangle$ 
   $\langle proof \rangle$ 

lemma [transfer-rule]:
   $\langle (pcr-integer ==> pcr-integer ==> pcr-integer)\ max\ max \rangle$ 
   $\langle proof \rangle$ 

end

lemma int-of-integer-min [simp]:
   $int-of-integer\ (min\ k\ l) = min\ (int-of-integer\ k)\ (int-of-integer\ l)$ 
   $\langle proof \rangle$ 

```

lemma *int-of-integer-max* [simp]:
 $\text{int-of-integer } (\max k l) = \max (\text{int-of-integer } k) (\text{int-of-integer } l)$
 ⟨proof⟩

lemma *nat-of-integer-non-positive* [simp]:
 $k \leq 0 \implies \text{nat-of-integer } k = 0$
 ⟨proof⟩

lemma *of-nat-of-integer* [simp]:
 $\text{of-nat } (\text{nat-of-integer } k) = \max 0 k$
 ⟨proof⟩

instantiation *integer* :: *unique-euclidean-ring*
begin

lift-definition *divide-integer* :: *integer* \Rightarrow *integer* \Rightarrow *integer*
is *divide* :: *int* \Rightarrow *int* \Rightarrow *int*
 ⟨proof⟩

declare *divide-integer.rep-eq* [simp]

lift-definition *modulo-integer* :: *integer* \Rightarrow *integer* \Rightarrow *integer*
is *modulo* :: *int* \Rightarrow *int* \Rightarrow *int*
 ⟨proof⟩

declare *modulo-integer.rep-eq* [simp]

lift-definition *euclidean-size-integer* :: *integer* \Rightarrow *nat*
is *euclidean-size* :: *int* \Rightarrow *nat*
 ⟨proof⟩

declare *euclidean-size-integer.rep-eq* [simp]

lift-definition *division-segment-integer* :: *integer* \Rightarrow *integer*
is *division-segment* :: *int* \Rightarrow *int*
 ⟨proof⟩

declare *division-segment-integer.rep-eq* [simp]

instance
 ⟨proof⟩

end

lemma [code]:
 $\text{euclidean-size} = \text{nat-of-integer} \circ \text{abs}$
 ⟨proof⟩

lemma [code]:

```

division-segment (k :: integer) = (if k ≥ 0 then 1 else - 1)
⟨proof⟩

instance integer :: linordered-euclidean-semiring
  ⟨proof⟩

instantiation integer :: ring-bit-operations
begin

lift-definition bit-integer :: ⟨integer ⇒ nat ⇒ bool⟩
  is bit ⟨proof⟩

lift-definition not-integer :: ⟨integer ⇒ integer⟩
  is not ⟨proof⟩

lift-definition and-integer :: ⟨integer ⇒ integer ⇒ integer⟩
  is and ⟨proof⟩

lift-definition or-integer :: ⟨integer ⇒ integer ⇒ integer⟩
  is or ⟨proof⟩

lift-definition xor-integer :: ⟨integer ⇒ integer ⇒ integer⟩
  is xor ⟨proof⟩

lift-definition mask-integer :: ⟨nat ⇒ integer⟩
  is mask ⟨proof⟩

lift-definition set-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is set-bit ⟨proof⟩

lift-definition unset-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is unset-bit ⟨proof⟩

lift-definition flip-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is flip-bit ⟨proof⟩

lift-definition push-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is push-bit ⟨proof⟩

lift-definition drop-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is drop-bit ⟨proof⟩

lift-definition take-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is take-bit ⟨proof⟩

instance ⟨proof⟩

end

```

instance *integer* :: *linordered-euclidean-semiring-bit-operations* \langle *proof* \rangle

instantiation *integer* :: *linordered-euclidean-semiring-division*
begin

definition *divmod-integer* :: *num* \Rightarrow *num* \Rightarrow *integer* \times *integer*
where

divmod-integer'-def: *divmod-integer* *m n* = (*numeral m div numeral n*, *numeral m mod numeral n*)

definition *divmod-step-integer* :: *integer* \Rightarrow *integer* \times *integer* \Rightarrow *integer* \times *integer*
where

divmod-step-integer l qr = (*let* (*q, r*) = *qr*
in if $|l| \leq |r|$ *then* ($2 * q + 1$, $r - l$)
else ($2 * q$, *r*))

instance \langle *proof* \rangle

end

lemma *integer-of-nat-0*: *integer-of-nat 0* = 0
 \langle *proof* \rangle

lemma *integer-of-nat-1*: *integer-of-nat 1* = 1
 \langle *proof* \rangle

lemma *integer-of-nat-numeral*:
integer-of-nat (numeral n) = *numeral n*
 \langle *proof* \rangle

69.2 Code theorems for target language integers

Constructors

definition *Pos* :: *num* \Rightarrow *integer*

where

[*simp*, *code-post*]: *Pos* = *numeral*

context

includes *lifting-syntax*

begin

lemma [*transfer-rule*]:
 \langle ((=) \implies *pcr-integer*) *numeral Pos* \rangle
 \langle *proof* \rangle

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)
 ⟨*proof*⟩

definition *Neg* :: *num* \Rightarrow *integer*
where
 [*simp*, *code-abbrev*]: *Neg n = - Pos n*

context
includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
 ⟨*((=) ==> pcr-integer) (λn. - numeral n) Neg*⟩
 ⟨*proof*⟩

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 o numeral*

lemma [*code-computation-unfold*]:
numeral = positive
Pos = positive
Neg = negative
 ⟨*proof*⟩

end

Auxiliary operations

lift-definition *dup* :: *integer* \Rightarrow *integer*
is $\lambda k::int. k + k$
 ⟨*proof*⟩

lemma *dup-code* [*code*]:
dup 0 = 0
dup (Pos n) = Pos (Num.Bit0 n)
dup (Neg n) = Neg (Num.Bit0 n)

$\langle \text{proof} \rangle$

lift-definition $\text{sub} :: \text{num} \Rightarrow \text{num} \Rightarrow \text{integer}$
is $\lambda m n. \text{numeral } m - \text{numeral } n :: \text{int}$
 $\langle \text{proof} \rangle$

lemma $\text{sub-code } [\text{code}]$:
 $\text{sub Num.One Num.One} = 0$
 $\text{sub (Num.Bit0 } m) \text{ Num.One} = \text{Pos (Num.BitM } m)$
 $\text{sub (Num.Bit1 } m) \text{ Num.One} = \text{Pos (Num.Bit0 } m)$
 $\text{sub Num.One (Num.Bit0 } n) = \text{Neg (Num.BitM } n)$
 $\text{sub Num.One (Num.Bit1 } n) = \text{Neg (Num.Bit0 } n)$
 $\text{sub (Num.Bit0 } m) \text{ (Num.Bit0 } n) = \text{dup (sub } m n)$
 $\text{sub (Num.Bit1 } m) \text{ (Num.Bit1 } n) = \text{dup (sub } m n)$
 $\text{sub (Num.Bit1 } m) \text{ (Num.Bit0 } n) = \text{dup (sub } m n) + 1$
 $\text{sub (Num.Bit0 } m) \text{ (Num.Bit1 } n) = \text{dup (sub } m n) - 1$
 $\langle \text{proof} \rangle$

Implementations

lemma $\text{one-integer-code } [\text{code}, \text{code-unfold}]$:
 $1 = \text{Pos Num.One}$
 $\langle \text{proof} \rangle$

lemma $\text{plus-integer-code } [\text{code}]$:
 $k + 0 = (k :: \text{integer})$
 $0 + l = (l :: \text{integer})$
 $\text{Pos } m + \text{Pos } n = \text{Pos } (m + n)$
 $\text{Pos } m + \text{Neg } n = \text{sub } m n$
 $\text{Neg } m + \text{Pos } n = \text{sub } n m$
 $\text{Neg } m + \text{Neg } n = \text{Neg } (m + n)$
 $\langle \text{proof} \rangle$

lemma $\text{uminus-integer-code } [\text{code}]$:
 $\text{uminus } 0 = (0 :: \text{integer})$
 $\text{uminus (Pos } m) = \text{Neg } m$
 $\text{uminus (Neg } m) = \text{Pos } m$
 $\langle \text{proof} \rangle$

lemma $\text{minus-integer-code } [\text{code}]$:
 $k - 0 = (k :: \text{integer})$
 $0 - l = \text{uminus } (l :: \text{integer})$
 $\text{Pos } m - \text{Pos } n = \text{sub } m n$
 $\text{Pos } m - \text{Neg } n = \text{Pos } (m + n)$
 $\text{Neg } m - \text{Pos } n = \text{Neg } (m + n)$
 $\text{Neg } m - \text{Neg } n = \text{sub } n m$
 $\langle \text{proof} \rangle$

lemma $\text{abs-integer-code } [\text{code}]$:
 $|k| = (\text{if } (k :: \text{integer}) < 0 \text{ then } -k \text{ else } k)$

$\langle \text{proof} \rangle$

lemma *sgn-integer-code* [code]:

$\text{sgn } k = (\text{if } k = 0 \text{ then } 0 \text{ else if } (k::\text{integer}) < 0 \text{ then } -1 \text{ else } 1)$

$\langle \text{proof} \rangle$

lemma *times-integer-code* [code]:

$k * 0 = (0::\text{integer})$

$0 * l = (0::\text{integer})$

$\text{Pos } m * \text{Pos } n = \text{Pos } (m * n)$

$\text{Pos } m * \text{Neg } n = \text{Neg } (m * n)$

$\text{Neg } m * \text{Pos } n = \text{Neg } (m * n)$

$\text{Neg } m * \text{Neg } n = \text{Pos } (m * n)$

$\langle \text{proof} \rangle$

definition *divmod-integer* :: $\text{integer} \Rightarrow \text{integer} \Rightarrow \text{integer} \times \text{integer}$

where

$\text{divmod-integer } k \ l = (k \text{ div } l, k \text{ mod } l)$

lemma *fst-divmod-integer* [simp]:

$\text{fst } (\text{divmod-integer } k \ l) = k \text{ div } l$

$\langle \text{proof} \rangle$

lemma *snd-divmod-integer* [simp]:

$\text{snd } (\text{divmod-integer } k \ l) = k \text{ mod } l$

$\langle \text{proof} \rangle$

definition *divmod-abs* :: $\text{integer} \Rightarrow \text{integer} \Rightarrow \text{integer} \times \text{integer}$

where

$\text{divmod-abs } k \ l = (|k| \text{ div } |l|, |k| \text{ mod } |l|)$

lemma *fst-divmod-abs* [simp]:

$\text{fst } (\text{divmod-abs } k \ l) = |k| \text{ div } |l|$

$\langle \text{proof} \rangle$

lemma *snd-divmod-abs* [simp]:

$\text{snd } (\text{divmod-abs } k \ l) = |k| \text{ mod } |l|$

$\langle \text{proof} \rangle$

declare *divmod-algorithm-code* [where ?'a = integer,
folded integer-of-num-def, unfolded integer-of-num-triv,
code]

lemma *divmod-abs-code* [code]:

$\text{divmod-abs } 0 \ j = (0, 0)$

$\text{divmod-abs } j \ 0 = (0, |j|)$

$\text{divmod-abs } (\text{Pos } k) (\text{Pos } l) = \text{divmod } k \ l$

$\text{divmod-abs } (\text{Pos } k) (\text{Neg } l) = \text{divmod } k \ l$

$\text{divmod-abs } (\text{Neg } k) (\text{Pos } l) = \text{divmod } k \ l$

$\text{divmod-abs } (\text{Neg } k) (\text{Neg } l) = \text{divmod } k \ l$
 $\langle \text{proof} \rangle$

lemma *divmod-integer-eq-cases*:

$\text{divmod-integer } k \ l =$
 $(\text{if } k = 0 \text{ then } (0, 0) \text{ else if } l = 0 \text{ then } (0, k) \text{ else}$
 $(\text{apsnd } \circ \text{times } \circ \text{sgn}) \ l \ (\text{if } \text{sgn } k = \text{sgn } l$
 $\text{then } \text{divmod-abs } k \ l$
 $\text{else } (\text{let } (r, s) = \text{divmod-abs } k \ l \text{ in}$
 $\text{if } s = 0 \text{ then } (-r, 0) \text{ else } (-r - 1, |l| - s)))$
 $\langle \text{proof} \rangle$

lemma *divmod-integer-code* [code]:

$\text{divmod-integer } k \ l =$
 $(\text{if } k = 0 \text{ then } (0, 0)$
 $\text{else if } l > 0 \text{ then}$
 $\quad (\text{if } k > 0 \text{ then } \text{divmod-abs } k \ l$
 $\quad \text{else case } \text{divmod-abs } k \ l \text{ of } (r, s) \Rightarrow$
 $\quad \quad \text{if } s = 0 \text{ then } (-r, 0) \text{ else } (-r - 1, l - s))$
 $\text{else if } l = 0 \text{ then } (0, k)$
 $\text{else } \text{apsnd } \text{uminus}$
 $\quad (\text{if } k < 0 \text{ then } \text{divmod-abs } k \ l$
 $\quad \text{else case } \text{divmod-abs } k \ l \text{ of } (r, s) \Rightarrow$
 $\quad \quad \text{if } s = 0 \text{ then } (-r, 0) \text{ else } (-r - 1, -l - s)))$
 $\langle \text{proof} \rangle$

lemma *div-integer-code* [code]:

$k \text{ div } l = \text{fst } (\text{divmod-integer } k \ l)$
 $\langle \text{proof} \rangle$

lemma *mod-integer-code* [code]:

$k \text{ mod } l = \text{snd } (\text{divmod-integer } k \ l)$
 $\langle \text{proof} \rangle$

context

includes *bit-operations-syntax*

begin

lemma *and-integer-code* [code]:

$\langle 0 \text{ AND } k = 0 \rangle$
 $\langle k \text{ AND } 0 = 0 \rangle$
 $\langle \text{Neg Num.One AND } k = k \rangle$
 $\langle k \text{ AND Neg Num.One} = k \rangle$
 $\langle \text{Pos Num.One AND Pos Num.One} = \text{Pos Num.One} \rangle$
 $\langle \text{Pos Num.One AND Pos (Num.Bit0 } n) = 0 \rangle$
 $\langle \text{Pos (Num.Bit0 } m) \text{ AND Pos Num.One} = 0 \rangle$
 $\langle \text{Pos Num.One AND Pos (Num.Bit1 } n) = \text{Pos Num.One} \rangle$
 $\langle \text{Pos (Num.Bit1 } m) \text{ AND Pos Num.One} = \text{Pos Num.One} \rangle$
 $\langle \text{Pos (Num.Bit0 } m) \text{ AND Pos (Num.Bit0 } n) = \text{dup (Pos } m \text{ AND Pos } n) \rangle$

$\langle \text{Pos } (\text{Num.Bit0 } m) \text{ AND Pos } (\text{Num.Bit1 } n) = \text{dup } (\text{Pos } m \text{ AND Pos } n) \rangle$
 $\langle \text{Pos } (\text{Num.Bit1 } m) \text{ AND Pos } (\text{Num.Bit0 } n) = \text{dup } (\text{Pos } m \text{ AND Pos } n) \rangle$
 $\langle \text{Pos } (\text{Num.Bit1 } m) \text{ AND Pos } (\text{Num.Bit1 } n) = \text{Pos Num.One} + \text{dup } (\text{Pos } m \text{ AND Pos } n) \rangle$
 $\langle \text{Pos } m \text{ AND Neg } (\text{num.Bit0 } n) = (\text{case and-not-num } m \text{ (Num.BitM } n) \text{ of None} \Rightarrow 0 \mid \text{Some } n' \Rightarrow \text{Pos } n') \rangle$
 $\langle \text{Neg } (\text{num.Bit0 } m) \text{ AND Pos } n = (\text{case and-not-num } n \text{ (Num.BitM } m) \text{ of None} \Rightarrow 0 \mid \text{Some } n' \Rightarrow \text{Pos } n') \rangle$
 $\langle \text{Pos } m \text{ AND Neg } (\text{num.Bit1 } n) = (\text{case and-not-num } m \text{ (Num.Bit0 } n) \text{ of None} \Rightarrow 0 \mid \text{Some } n' \Rightarrow \text{Pos } n') \rangle$
 $\langle \text{Neg } (\text{num.Bit1 } m) \text{ AND Pos } n = (\text{case and-not-num } n \text{ (Num.Bit0 } m) \text{ of None} \Rightarrow 0 \mid \text{Some } n' \Rightarrow \text{Pos } n') \rangle$
 $\langle \text{Neg } m \text{ AND Neg } n = \text{NOT } (\text{sub } m \text{ Num.One OR sub } n \text{ Num.One}) \rangle$
for $k :: \text{integer}$
 $\langle \text{proof} \rangle$

lemma *or-integer-code* [code]:

$\langle 0 \text{ OR } k = k \rangle$
 $\langle k \text{ OR } 0 = k \rangle$
 $\langle \text{Neg Num.One OR } k = \text{Neg Num.One} \rangle$
 $\langle k \text{ OR Neg Num.One} = \text{Neg Num.One} \rangle$
 $\langle \text{Pos Num.One OR Pos Num.One} = \text{Pos Num.One} \rangle$
 $\langle \text{Pos Num.One OR Pos } (\text{Num.Bit0 } n) = \text{Pos } (\text{Num.Bit1 } n) \rangle$
 $\langle \text{Pos } (\text{Num.Bit0 } m) \text{ OR Pos Num.One} = \text{Pos } (\text{Num.Bit1 } m) \rangle$
 $\langle \text{Pos Num.One OR Pos } (\text{Num.Bit1 } n) = \text{Pos } (\text{Num.Bit1 } n) \rangle$
 $\langle \text{Pos } (\text{Num.Bit1 } m) \text{ OR Pos Num.One} = \text{Pos } (\text{Num.Bit1 } m) \rangle$
 $\langle \text{Pos } (\text{Num.Bit0 } m) \text{ OR Pos } (\text{Num.Bit0 } n) = \text{dup } (\text{Pos } m \text{ OR Pos } n) \rangle$
 $\langle \text{Pos } (\text{Num.Bit0 } m) \text{ OR Pos } (\text{Num.Bit1 } n) = \text{Pos Num.One} + \text{dup } (\text{Pos } m \text{ OR Pos } n) \rangle$
 $\langle \text{Pos } (\text{Num.Bit1 } m) \text{ OR Pos } (\text{Num.Bit0 } n) = \text{Pos Num.One} + \text{dup } (\text{Pos } m \text{ OR Pos } n) \rangle$
 $\langle \text{Pos } (\text{Num.Bit1 } m) \text{ OR Pos } (\text{Num.Bit1 } n) = \text{Pos Num.One} + \text{dup } (\text{Pos } m \text{ OR Pos } n) \rangle$
 $\langle \text{Pos } m \text{ OR Neg } (\text{num.Bit0 } n) = \text{Neg } (\text{or-not-num-neg } m \text{ (Num.BitM } n)) \rangle$
 $\langle \text{Neg } (\text{num.Bit0 } m) \text{ OR Pos } n = \text{Neg } (\text{or-not-num-neg } n \text{ (Num.BitM } m)) \rangle$
 $\langle \text{Pos } m \text{ OR Neg } (\text{num.Bit1 } n) = \text{Neg } (\text{or-not-num-neg } m \text{ (Num.Bit0 } n)) \rangle$
 $\langle \text{Neg } (\text{num.Bit1 } m) \text{ OR Pos } n = \text{Neg } (\text{or-not-num-neg } n \text{ (Num.Bit0 } m)) \rangle$
 $\langle \text{Neg } m \text{ OR Neg } n = \text{NOT } (\text{sub } m \text{ Num.One AND sub } n \text{ Num.One}) \rangle$
for $k :: \text{integer}$
 $\langle \text{proof} \rangle$

lemma *xor-integer-code* [code]:

$\langle 0 \text{ XOR } k = k \rangle$
 $\langle k \text{ XOR } 0 = k \rangle$
 $\langle \text{Neg Num.One XOR } k = \text{NOT } k \rangle$
 $\langle k \text{ XOR Neg Num.One} = \text{NOT } k \rangle$
 $\langle \text{Neg } m \text{ XOR } k = \text{NOT } (\text{sub } m \text{ num.One XOR } k) \rangle$
 $\langle k \text{ XOR Neg } n = \text{NOT } (k \text{ XOR } (\text{sub } n \text{ num.One})) \rangle$
 $\langle \text{Pos Num.One XOR Pos Num.One} = 0 \rangle$

$\langle \text{Pos Num.One XOR Pos (Num.Bit0 } n) = \text{Pos (Num.Bit1 } n) \rangle$
 $\langle \text{Pos (Num.Bit0 } m) \text{ XOR Pos Num.One} = \text{Pos (Num.Bit1 } m) \rangle$
 $\langle \text{Pos Num.One XOR Pos (Num.Bit1 } n) = \text{Pos (Num.Bit0 } n) \rangle$
 $\langle \text{Pos (Num.Bit1 } m) \text{ XOR Pos Num.One} = \text{Pos (Num.Bit0 } m) \rangle$
 $\langle \text{Pos (Num.Bit0 } m) \text{ XOR Pos (Num.Bit0 } n) = \text{dup (Pos } m \text{ XOR Pos } n) \rangle$
 $\langle \text{Pos (Num.Bit0 } m) \text{ XOR Pos (Num.Bit1 } n) = \text{Pos Num.One} + \text{dup (Pos } m \text{ XOR Pos } n) \rangle$
 $\langle \text{Pos (Num.Bit1 } m) \text{ XOR Pos (Num.Bit0 } n) = \text{Pos Num.One} + \text{dup (Pos } m \text{ XOR Pos } n) \rangle$
 $\langle \text{Pos (Num.Bit1 } m) \text{ XOR Pos (Num.Bit1 } n) = \text{dup (Pos } m \text{ XOR Pos } n) \rangle$
for $k :: \text{integer}$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{NOT } k = -k - 1 \rangle$ **for** $k :: \text{integer}$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{bit } k \ n \longleftrightarrow k \text{ AND push-bit } n \ 1 \neq (0 :: \text{integer}) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{mask } n = \text{push-bit } n \ 1 - (1 :: \text{integer}) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{set-bit } n \ k = k \text{ OR push-bit } n \ 1 \rangle$ **for** $k :: \text{integer}$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{unset-bit } n \ k = k \text{ AND NOT (push-bit } n \ 1) \rangle$ **for** $k :: \text{integer}$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{flip-bit } n \ k = k \text{ XOR push-bit } n \ 1 \rangle$ **for** $k :: \text{integer}$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{take-bit } n \ k = k \text{ AND mask } n \rangle$ **for** $k :: \text{integer}$
 $\langle \text{proof} \rangle$

end

definition *bit-cut-integer* :: $\text{integer} \Rightarrow \text{integer} \times \text{bool}$
where *bit-cut-integer* $k = (k \text{ div } 2, \text{ odd } k)$

lemma *bit-cut-integer-code* [code]:
 $\text{bit-cut-integer } k = (\text{if } k = 0 \text{ then } (0, \text{ False})$
 $\text{else let } (r, s) = \text{Code-Numeral.divmod-abs } k \ 2$

$\text{in } (\text{if } k > 0 \text{ then } r \text{ else } -r - s, s = 1))$
 $\langle \text{proof} \rangle$

lemma *equal-integer-code* [code]:
 $HOL.equal\ 0\ (0::integer) \longleftrightarrow True$
 $HOL.equal\ 0\ (Pos\ l) \longleftrightarrow False$
 $HOL.equal\ 0\ (Neg\ l) \longleftrightarrow False$
 $HOL.equal\ (Pos\ k)\ 0 \longleftrightarrow False$
 $HOL.equal\ (Pos\ k)\ (Pos\ l) \longleftrightarrow HOL.equal\ k\ l$
 $HOL.equal\ (Pos\ k)\ (Neg\ l) \longleftrightarrow False$
 $HOL.equal\ (Neg\ k)\ 0 \longleftrightarrow False$
 $HOL.equal\ (Neg\ k)\ (Pos\ l) \longleftrightarrow False$
 $HOL.equal\ (Neg\ k)\ (Neg\ l) \longleftrightarrow HOL.equal\ k\ l$
 $\langle \text{proof} \rangle$

lemma *equal-integer-refl* [code nbe]:
 $HOL.equal\ (k::integer)\ k \longleftrightarrow True$
 $\langle \text{proof} \rangle$

lemma *less-eq-integer-code* [code]:
 $0 \leq (0::integer) \longleftrightarrow True$
 $0 \leq Pos\ l \longleftrightarrow True$
 $0 \leq Neg\ l \longleftrightarrow False$
 $Pos\ k \leq 0 \longleftrightarrow False$
 $Pos\ k \leq Pos\ l \longleftrightarrow k \leq l$
 $Pos\ k \leq Neg\ l \longleftrightarrow False$
 $Neg\ k \leq 0 \longleftrightarrow True$
 $Neg\ k \leq Pos\ l \longleftrightarrow True$
 $Neg\ k \leq Neg\ l \longleftrightarrow l \leq k$
 $\langle \text{proof} \rangle$

lemma *less-integer-code* [code]:
 $0 < (0::integer) \longleftrightarrow False$
 $0 < Pos\ l \longleftrightarrow True$
 $0 < Neg\ l \longleftrightarrow False$
 $Pos\ k < 0 \longleftrightarrow False$
 $Pos\ k < Pos\ l \longleftrightarrow k < l$
 $Pos\ k < Neg\ l \longleftrightarrow False$
 $Neg\ k < 0 \longleftrightarrow True$
 $Neg\ k < Pos\ l \longleftrightarrow True$
 $Neg\ k < Neg\ l \longleftrightarrow l < k$
 $\langle \text{proof} \rangle$

lift-definition *num-of-integer* :: *integer* \Rightarrow *num*
is *num-of-nat* \circ *nat*
 $\langle \text{proof} \rangle$

lemma *num-of-integer-code* [code]:
 $\text{num-of-integer}\ k = (\text{if } k \leq 1 \text{ then } Num.One$

```

    else let
      (l, j) = divmod-integer k 2;
      l' = num-of-integer l;
      l'' = l' + l'
    in if j = 0 then l'' else l'' + Num.One)
⟨proof⟩

```

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⟩

lemma *int-of-integer-code-nbe* [code nbe]:
int-of-integer 0 = 0
int-of-integer (Pos n) = Int.Pos n
int-of-integer (Neg n) = Int.Neg n
 ⟨proof⟩

lemma *int-of-integer-code* [code]:
 ⟨*int-of-integer* k = (
 if k = 0 then 0
 else if k = − 1 then − 1
 else
 let
 (l, j) = divmod-integer k 2;
 l' = 2 * *int-of-integer* l
 in if j = 0 then l' else l' + 1)⟩
 ⟨proof⟩

lemma *integer-of-int-code-nbe* [code nbe]:
integer-of-int 0 = 0
integer-of-int (Int.Pos n) = Pos n
integer-of-int (Int.Neg n) = Neg n
 ⟨proof⟩

lemma *integer-of-int-code* [code]:
 ⟨*integer-of-int* k = (
 if k = 0 then 0
 else if k = − 1 then − 1
 else
 let
 l = 2 * *integer-of-int* (k div 2);
 j = k mod 2
 in if j = 0 then l else l + 1)⟩
 ⟨proof⟩

hide-const (**open**) *Pos Neg sub dup divmod-abs*

context
begin

qualified definition *push-bit* :: $\langle \text{integer} \Rightarrow \text{integer} \Rightarrow \text{integer} \rangle$
where $\langle \text{push-bit } i \ k = \text{Bit-Operations.push-bit } (\text{nat-of-integer } |i|) \ k \rangle$

qualified lemma *push-bit-code* [*code*]:
 $\langle \text{push-bit } i \ k = k * 2 \wedge \text{nat-of-integer } |i| \rangle$
 $\langle \text{proof} \rangle$

lemma *push-bit-integer-code* [*code*]:
 $\langle \text{Bit-Operations.push-bit } n \ k = \text{push-bit } (\text{of-nat } n) \ k \rangle$
 $\langle \text{proof} \rangle$ **definition** *drop-bit* :: $\langle \text{integer} \Rightarrow \text{integer} \Rightarrow \text{integer} \rangle$
where $\langle \text{drop-bit } i \ k = \text{Bit-Operations.drop-bit } (\text{nat-of-integer } |i|) \ k \rangle$

qualified lemma *drop-bit-code* [*code*]:
 $\langle \text{drop-bit } i \ k = k \text{ div } 2 \wedge \text{nat-of-integer } |i| \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-integer-code* [*code*]:
 $\langle \text{Bit-Operations.drop-bit } n \ k = \text{drop-bit } (\text{of-nat } n) \ k \rangle$
 $\langle \text{proof} \rangle$

end

69.3 Serializer setup for target language integers

code-printing

type-constructor *integer* \rightarrow
 (*SML*) *IntInf.int*
and (*OCaml*) *Z.t*
and (*Haskell*) *Integer*
and (*Scala*) *BigInt*
and (*Eval*) *int*
| class-instance *integer* :: *equal* \rightarrow
 (*Haskell*) $-$

code-reserved

(*Eval*) *int Integer*

code-printing

constant *0::integer* \rightarrow
 (*SML*) $!(0 / :/ \text{IntInf.int})$
and (*OCaml*) *Z.zero*
and (*Haskell*) $!(0 / ::/ \text{Integer})$
and (*Scala*) *BigInt(0)*

$\langle ML \rangle$

code-printing

```

constant plus :: integer ⇒ - ⇒ - →
  (SML) IntInf.+ ((-), (-))
  and (OCaml) Z.add
  and (Haskell) infixl 6 +
  and (Scala) infixl 7 +
  and (Eval) infixl 8 +
| constant uminus :: 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) !(raise / Fail / sub)
  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
| constant Bit-Operations.and :: integer ⇒ integer ⇒ integer →
  (SML) IntInf.andb ((-),/ (-))
    and (OCaml) Z.logand
    and (Haskell) infixl 7 .&.
    and (Scala) infixl 3 &
| constant Bit-Operations.or :: integer ⇒ integer ⇒ integer →
  (SML) IntInf.orb ((-),/ (-))
    and (OCaml) Z.logor
    and (Haskell) infixl 5 .|.
    and (Scala) infixl 1 |
| constant Bit-Operations.xor :: integer ⇒ integer ⇒ integer →
  (SML) IntInf.xorb ((-),/ (-))
    and (OCaml) Z.logxor
    and (Haskell) infixl 6 .^.
    and (Scala) infixl 2 ^
| constant Bit-Operations.not :: integer ⇒ integer →
  (SML) IntInf.notb
    and (OCaml) Z.lognot
    and (Haskell) Data.Bits.complement
    and (Scala) -.unary'~

```

code-reserved

(Eval) abs

code-printing code-module *Bit-Shifts* →

(SML) ‹

structure Bit-Shifts : sig

```

    type int = IntInf.int
    val push : int -> int -> int
    val drop : int -> int -> int
end = struct

open IntInf;

fun fold - [] y = y
  | fold f (x :: xs) y = fold f xs (f x y);

fun replicate n x = (if n <= 0 then [] else x :: replicate (n - 1) x);

val max-index = pow (fromInt 2, Word.wordSize) - fromInt 1; (*largest possible word*)

val word-of-int = Word.fromLargeInt o toLarge;

val word-max-index = word-of-int max-index;

fun words-of-int k = case divMod (k, max-index)
  of (b, s) => word-of-int s :: replicate b word-max-index;

fun push' i k = << (k, i);

fun drop' i k = ~>> (k, i);

(* The implementations are formally total, though indices >~ max-index will produce heavy computation load *)

fun push i = fold push' (words-of-int (abs i));

fun drop i = fold drop' (words-of-int (abs i));

end;> for constant Code-Numeral.push-bit Code-Numeral.drop-bit
  and (OCaml) <
module Bit-Shifts : sig
  val push : Z.t -> Z.t -> Z.t
  val drop : Z.t -> Z.t -> Z.t
end = struct

let rec fold f xs y = match xs with
  [] -> y
  | (x :: xs) -> fold f xs (f x y);

let rec replicate n x = (if Z.leq n Z.zero then [] else x :: replicate (Z.pred n) x);

let max-index = Z.of-int max-int;;

let splitIndex i = let (b, s) = Z.div-rem i max-index

```

```

    in Z.to-int s :: replicate b max-int;;

let push' i k = Z.shift-left k i;;

let drop' i k = Z.shift-right k i;;

(* The implementations are formally total, though indices  $\gtrsim$  max-index will produce heavy computation load *)

let push i = fold push' (splitIndex (Z.abs i));;

let drop i = fold drop' (splitIndex (Z.abs i));;

end;;
› for constant Code-Numeral.push-bit Code-Numeral.drop-bit
  and (Haskell) ‹
module Bit-Shifts (push, drop, push', drop') where

import Prelude (Int, Integer, toInteger, fromInteger, maxBound, divMod, (-), (<=),
abs, flip)
import GHC.Bits (Bits)
import Data.Bits (shiftL, shiftR)

fold :: (a -> b -> b) -> [a] -> b -> b
fold - [] y = y
fold f (x : xs) y = fold f xs (f x y)

replicate :: Integer -> a -> [a]
replicate k x = if k <= 0 then [] else x : replicate (k - 1) x

maxIndex :: Integer
maxIndex = toInteger (maxBound :: Int)

splitIndex :: Integer -> [Int]
splitIndex i = fromInteger s : replicate (fromInteger b) maxBound
  where (b, s) = i `divMod` maxIndex

{– The implementations are formally total, though indices  $\gtrsim$  maxIndex will produce heavy computation load –}

push :: Integer -> Integer -> Integer
push i = fold (flip shiftL) (splitIndex (abs i))

drop :: Integer -> Integer -> Integer
drop i = fold (flip shiftR) (splitIndex (abs i))

push' :: Int -> Int -> Int
push' i = flip shiftL (abs i)

```

```

drop' :: Int -> Int -> Int
drop' i = flip shiftR (abs i)
› for constant Code-Numeral.push-bit Code-Numeral.drop-bit
  and (Scala) ‹
object Bit-Shifts {

private val maxIndex : BigInt = BigInt(Int.MaxValue);

private def replicate[A](i : BigInt, x : A) : List[A] =
  i <= 0 match {
    case true => Nil
    case false => x :: replicate[A](i - 1, x)
  }

private def splitIndex(i : BigInt) : List[Int] = {
  val (b, s) = i /% maxIndex
  return s.intValue :: replicate(b, Int.MaxValue)
}

/* The implementations are formally total, though indices >~ maxIndex will pro-
duce heavy computation load */

def push(i: BigInt, k: BigInt) : BigInt =
  splitIndex(i).foldLeft(k) { (l, j) => l << j }

def drop(i: BigInt, k: BigInt) : BigInt =
  splitIndex(i).foldLeft(k) { (l, j) => l >> j }

}
› for constant Code-Numeral.push-bit Code-Numeral.drop-bit

code-reserved
(SML) Bit-Shifts
and (Haskell) Bit-Shifts
and (Scala) Bit-Shifts

code-printing
constant Code-Numeral.push-bit ↪
(SML) Bit'-Shifts.push
and (OCaml) Bit'-Shifts.push
and (Haskell) Bit'-Shifts.push
and (Scala) Bit'-Shifts.push
| constant Code-Numeral.drop-bit ↪
(SML) Bit'-Shifts.drop
and (OCaml) Bit'-Shifts.drop
and (Haskell) Bit'-Shifts.drop
and (Scala) Bit'-Shifts.drop

code-identifier

```

code-module *Code-Numeral* \rightarrow (*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* \langle *proof* \rangle

setup-lifting *type-definition-natural*

lemma *natural-eq-iff* [*termination-simp*]:
 $m = n \longleftrightarrow \text{nat-of-natural } m = \text{nat-of-natural } n$
 \langle *proof* \rangle

lemma *natural-eqI*:
 $\text{nat-of-natural } m = \text{nat-of-natural } n \implies m = n$
 \langle *proof* \rangle

lemma *nat-of-natural-of-nat-inverse* [*simp*]:
 $\text{nat-of-natural } (\text{natural-of-nat } n) = n$
 \langle *proof* \rangle

lemma *natural-of-nat-of-natural-inverse* [*simp*]:
 $\text{natural-of-nat } (\text{nat-of-natural } n) = n$
 \langle *proof* \rangle

instantiation *natural* :: {*comm-monoid-diff*, *semiring-1*}
begin

lift-definition *zero-natural* :: *natural*
is *0* :: *nat*
 \langle *proof* \rangle

declare *zero-natural.rep-eq* [*simp*]

lift-definition *one-natural* :: *natural*
is *1* :: *nat*
 \langle *proof* \rangle

declare *one-natural.rep-eq* [*simp*]

lift-definition *plus-natural* :: *natural* \Rightarrow *natural* \Rightarrow *natural*
is *plus* :: *nat* \Rightarrow *nat* \Rightarrow *nat*
 \langle *proof* \rangle

declare *plus-natural.rep-eq* [*simp*]

lift-definition *minus-natural* :: *natural* \Rightarrow *natural* \Rightarrow *natural*
is *minus* :: *nat* \Rightarrow *nat* \Rightarrow *nat*

```

  <proof>

declare minus-natural.rep-eq [simp]

lift-definition times-natural :: natural  $\Rightarrow$  natural  $\Rightarrow$  natural
  is times :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat
  <proof>

declare times-natural.rep-eq [simp]

instance <proof>

end

instance natural :: Rings.dvd <proof>

context
  includes lifting-syntax
begin

lemma [transfer-rule]:
  <(pcr-natural ==> pcr-natural ==> ( $\longleftrightarrow$ )) (dvd) (dvd)>
  <proof>

lemma [transfer-rule]:
  <(( $\longleftrightarrow$ ) ==> pcr-natural) of-bool of-bool>
  <proof>

lemma [transfer-rule]:
  <((=) ==> pcr-natural) ( $\lambda n. n$ ) of-nat>
  <proof>

lemma [transfer-rule]:
  <((=) ==> pcr-natural) numeral numeral>
  <proof>

lemma [transfer-rule]:
  <(pcr-natural ==> (=) ==> pcr-natural) ( $\bigwedge$ ) ( $\bigwedge$ )>
  <proof>

end

lemma nat-of-natural-of-nat [simp]:
  nat-of-natural (of-nat n) = n
  <proof>

lemma natural-of-nat-of-nat [simp, code-abbrev]:
  natural-of-nat = of-nat
  <proof>

```

lemma *of-nat-of-natural* [*simp*]:

of-nat (*nat-of-natural* *n*) = *n*
 ⟨*proof*⟩

lemma *nat-of-natural-numeral* [*simp*]:

nat-of-natural (*numeral* *k*) = *numeral* *k*
 ⟨*proof*⟩

instantiation *natural* :: {*linordered-semiring*, *equal*}
begin

lift-definition *less-eq-natural* :: *natural* \Rightarrow *natural* \Rightarrow *bool*

is *less-eq* :: *nat* \Rightarrow *nat* \Rightarrow *bool*
 ⟨*proof*⟩

declare *less-eq-natural.rep-eq* [*termination-simp*]

lift-definition *less-natural* :: *natural* \Rightarrow *natural* \Rightarrow *bool*

is *less* :: *nat* \Rightarrow *nat* \Rightarrow *bool*
 ⟨*proof*⟩

declare *less-natural.rep-eq* [*termination-simp*]

lift-definition *equal-natural* :: *natural* \Rightarrow *natural* \Rightarrow *bool*

is *HOL.equal* :: *nat* \Rightarrow *nat* \Rightarrow *bool*
 ⟨*proof*⟩

instance ⟨*proof*⟩

end

context

includes *lifting-syntax*

begin

lemma [*transfer-rule*]:

⟨(*pcr-natural* \implies *pcr-natural* \implies *pcr-natural*) *min min*⟩
 ⟨*proof*⟩

lemma [*transfer-rule*]:

⟨(*pcr-natural* \implies *pcr-natural* \implies *pcr-natural*) *max max*⟩
 ⟨*proof*⟩

end

lemma *nat-of-natural-min* [*simp*]:

nat-of-natural (*min* *k* *l*) = *min* (*nat-of-natural* *k*) (*nat-of-natural* *l*)
 ⟨*proof*⟩

lemma *nat-of-natural-max* [*simp*]:
 $\text{nat-of-natural } (\max k l) = \max (\text{nat-of-natural } k) (\text{nat-of-natural } l)$
 ⟨*proof*⟩

instantiation *natural* :: *unique-euclidean-semiring*
begin

lift-definition *divide-natural* :: *natural* \Rightarrow *natural* \Rightarrow *natural*
is *divide* :: *nat* \Rightarrow *nat* \Rightarrow *nat*
 ⟨*proof*⟩

declare *divide-natural.rep-eq* [*simp*]

lift-definition *modulo-natural* :: *natural* \Rightarrow *natural* \Rightarrow *natural*
is *modulo* :: *nat* \Rightarrow *nat* \Rightarrow *nat*
 ⟨*proof*⟩

declare *modulo-natural.rep-eq* [*simp*]

lift-definition *euclidean-size-natural* :: *natural* \Rightarrow *nat*
is *euclidean-size* :: *nat* \Rightarrow *nat*
 ⟨*proof*⟩

declare *euclidean-size-natural.rep-eq* [*simp*]

lift-definition *division-segment-natural* :: *natural* \Rightarrow *natural*
is *division-segment* :: *nat* \Rightarrow *nat*
 ⟨*proof*⟩

declare *division-segment-natural.rep-eq* [*simp*]

instance
 ⟨*proof*⟩

end

lemma [*code*]:
 $\text{euclidean-size} = \text{nat-of-natural}$
 ⟨*proof*⟩

lemma [*code*]:
 $\text{division-segment } (n::\text{natural}) = 1$
 ⟨*proof*⟩

instance *natural* :: *discrete-linordered-semidom*
 ⟨*proof*⟩

instance *natural* :: *linordered-euclidean-semiring*


```

  ⟨proof⟩

instantiation natural :: semiring-bit-operations
begin

lift-definition bit-natural :: ⟨natural ⇒ nat ⇒ bool⟩
  is bit ⟨proof⟩

lift-definition and-natural :: ⟨natural ⇒ natural ⇒ natural⟩
  is and ⟨proof⟩

lift-definition or-natural :: ⟨natural ⇒ natural ⇒ natural⟩
  is or ⟨proof⟩

lift-definition xor-natural :: ⟨natural ⇒ natural ⇒ natural⟩
  is xor ⟨proof⟩

lift-definition mask-natural :: ⟨nat ⇒ natural⟩
  is mask ⟨proof⟩

lift-definition set-bit-natural :: ⟨nat ⇒ natural ⇒ natural⟩
  is set-bit ⟨proof⟩

lift-definition unset-bit-natural :: ⟨nat ⇒ natural ⇒ natural⟩
  is unset-bit ⟨proof⟩

lift-definition flip-bit-natural :: ⟨nat ⇒ natural ⇒ natural⟩
  is flip-bit ⟨proof⟩

lift-definition push-bit-natural :: ⟨nat ⇒ natural ⇒ natural⟩
  is push-bit ⟨proof⟩

lift-definition drop-bit-natural :: ⟨nat ⇒ natural ⇒ natural⟩
  is drop-bit ⟨proof⟩

lift-definition take-bit-natural :: ⟨nat ⇒ natural ⇒ natural⟩
  is take-bit ⟨proof⟩

instance ⟨proof⟩

end

instance natural :: linordered-euclidean-semiring-bit-operations ⟨proof⟩

lift-definition natural-of-integer :: integer ⇒ natural
  is nat :: int ⇒ nat
  ⟨proof⟩

lift-definition integer-of-natural :: natural ⇒ integer

```

is *of-nat* :: *nat* \Rightarrow *int*
 \langle *proof* \rangle

lemma *natural-of-integer-of-natural* [*simp*]:
natural-of-integer (*integer-of-natural* *n*) = *n*
 \langle *proof* \rangle

lemma *integer-of-natural-of-integer* [*simp*]:
integer-of-natural (*natural-of-integer* *k*) = *max 0 k*
 \langle *proof* \rangle

lemma *int-of-integer-of-natural* [*simp*]:
int-of-integer (*integer-of-natural* *n*) = *of-nat* (*nat-of-natural* *n*)
 \langle *proof* \rangle

lemma *integer-of-natural-of-nat* [*simp*]:
integer-of-natural (*of-nat* *n*) = *of-nat* *n*
 \langle *proof* \rangle

lemma [*measure-function*]:
is-measure *nat-of-natural*
 \langle *proof* \rangle

69.5 Inductive representation of target language naturals

lift-definition *Suc* :: *natural* \Rightarrow *natural*
is *Nat.Suc*
 \langle *proof* \rangle

declare *Suc.rep-eq* [*simp*]

old-rep-datatype *0::natural* *Suc*
 \langle *proof* \rangle

lemma *natural-cases* [*case-names nat, cases type: natural*]:
fixes *m* :: *natural*
assumes $\bigwedge n. m = \text{of-nat } n \implies P$
shows *P*
 \langle *proof* \rangle

instantiation *natural* :: *size*
begin

definition *size-nat* **where** [*simp, code*]: *size-nat* = *nat-of-natural*

instance \langle *proof* \rangle

end

lemma *natural-decr* [*termination-simp*]:
 $n \neq 0 \implies \text{nat-of-natural } n - \text{Nat.Suc } 0 < \text{nat-of-natural } n$
 ⟨*proof*⟩

lemma *natural-zero-minus-one*: $(0::\text{natural}) - 1 = 0$
 ⟨*proof*⟩

lemma *Suc-natural-minus-one*: $\text{Suc } n - 1 = n$
 ⟨*proof*⟩

hide-const (**open**) *Suc*

69.6 Code refinement for target language naturals

lift-definition *Nat* :: *integer* \Rightarrow *natural*
is *nat*
 ⟨*proof*⟩

lemma [*code-post*]:
 $\text{Nat } 0 = 0$
 $\text{Nat } 1 = 1$
 $\text{Nat } (\text{numeral } k) = \text{numeral } k$
 ⟨*proof*⟩

lemma [*code abstype*]:
 $\text{Nat } (\text{integer-of-natural } n) = n$
 ⟨*proof*⟩

lemma [*code*]:
 $\text{natural-of-nat } n = \text{natural-of-integer } (\text{integer-of-nat } n)$
 ⟨*proof*⟩

lemma [*code abstract*]:
 $\text{integer-of-natural } (\text{natural-of-integer } k) = \max 0 k$
 ⟨*proof*⟩

lemma [*code*]:
 $\text{integer-of-natural } (\text{mask } n) = \text{mask } n$
 ⟨*proof*⟩

lemma [*code-abbrev*]:
 $\text{natural-of-integer } (\text{Code-Numeral.Pos } k) = \text{numeral } k$
 ⟨*proof*⟩

lemma [*code abstract*]:
 $\text{integer-of-natural } 0 = 0$
 ⟨*proof*⟩

lemma [*code abstract*]:

integer-of-natural 1 = 1
<proof>

lemma [*code abstract*]:
integer-of-natural (Code-Numeral.Suc n) = integer-of-natural n + 1
<proof>

lemma [*code*]:
nat-of-natural = nat-of-integer ∘ integer-of-natural
<proof>

lemma [*code, code-unfold*]:
case-natural f g n = (if n = 0 then f else g (n - 1))
<proof>

declare *natural.rec* [*code del*]

lemma [*code abstract*]:
integer-of-natural (m + n) = integer-of-natural m + integer-of-natural n
<proof>

lemma [*code abstract*]:
integer-of-natural (m - n) = max 0 (integer-of-natural m - integer-of-natural n)
<proof>

lemma [*code abstract*]:
*integer-of-natural (m * n) = integer-of-natural m * integer-of-natural n*
<proof>

lemma [*code abstract*]:
integer-of-natural (m div n) = integer-of-natural m div integer-of-natural n
<proof>

lemma [*code abstract*]:
integer-of-natural (m mod n) = integer-of-natural m mod integer-of-natural n
<proof>

lemma [*code nbe*]: *HOL.equal n (n::natural) ⟷ True*
<proof>

lemma [*code*]:
HOL.equal m n ⟷ HOL.equal (integer-of-natural m) (integer-of-natural n)
<proof>

lemma [*code*]: *m ≤ n ⟷ integer-of-natural m ≤ integer-of-natural n*
<proof>

lemma [*code*]: *m < n ⟷ integer-of-natural m < integer-of-natural n*

$\langle proof \rangle$

context

includes *bit-operations-syntax*

begin

lemma [*code*]:

$\langle bit\ m\ n \longleftrightarrow bit\ (integer-of-natural\ m)\ n \rangle$

$\langle proof \rangle$

lemma [*code abstract*]:

$\langle integer-of-natural\ (m\ AND\ n) = integer-of-natural\ m\ AND\ integer-of-natural\ n \rangle$

$\langle proof \rangle$

lemma [*code abstract*]:

$\langle integer-of-natural\ (m\ OR\ n) = integer-of-natural\ m\ OR\ integer-of-natural\ n \rangle$

$\langle proof \rangle$

lemma [*code abstract*]:

$\langle integer-of-natural\ (m\ XOR\ n) = integer-of-natural\ m\ XOR\ integer-of-natural\ n \rangle$

$\langle proof \rangle$

lemma [*code abstract*]:

$\langle integer-of-natural\ (mask\ n) = mask\ n \rangle$

$\langle proof \rangle$

lemma [*code abstract*]:

$\langle integer-of-natural\ (set-bit\ n\ m) = set-bit\ n\ (integer-of-natural\ m) \rangle$

$\langle proof \rangle$

lemma [*code abstract*]:

$\langle integer-of-natural\ (unset-bit\ n\ m) = unset-bit\ n\ (integer-of-natural\ m) \rangle$

$\langle proof \rangle$

lemma [*code abstract*]:

$\langle integer-of-natural\ (flip-bit\ n\ m) = flip-bit\ n\ (integer-of-natural\ m) \rangle$

$\langle proof \rangle$

lemma [*code abstract*]:

$\langle integer-of-natural\ (push-bit\ n\ m) = push-bit\ n\ (integer-of-natural\ m) \rangle$

$\langle proof \rangle$

lemma [*code abstract*]:

$\langle integer-of-natural\ (drop-bit\ n\ m) = drop-bit\ n\ (integer-of-natural\ m) \rangle$

$\langle proof \rangle$

lemma [*code abstract*]:

$\langle integer-of-natural\ (take-bit\ n\ m) = take-bit\ n\ (integer-of-natural\ m) \rangle$

$\langle proof \rangle$

```

end

hide-const (open) 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  $\Rightarrow$  -
    integer-of-natural natural-of-integer

lifting-update integer.lifting
lifting-forget integer.lifting

lifting-update natural.lifting
lifting-forget natural.lifting

end

```

70 A HOL random engine

```

theory Random
imports List Groups-List Code-Numeral
begin

```

70.1 Auxiliary functions

```

fun log :: natural  $\Rightarrow$  natural  $\Rightarrow$  natural where
  log b i = (if b  $\leq$  1  $\vee$  i < b then 1 else 1 + log b (i div b))

definition inc-shift :: natural  $\Rightarrow$  natural  $\Rightarrow$  natural where
  inc-shift v k = (if v = k then 1 else k + 1)

definition minus-shift :: natural  $\Rightarrow$  natural  $\Rightarrow$  natural  $\Rightarrow$  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  $\times$  natural

primrec next :: seed  $\Rightarrow$  natural  $\times$  seed where
  next (v, w) = (let
    k = v div 53668;
    v' = minus-shift 2147483563 ((v mod 53668) * 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* \Rightarrow *seed* \times *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* \Rightarrow (*'b* \Rightarrow *'a* \Rightarrow *'b* \times *'a*) \Rightarrow *'b* \Rightarrow *'a* \Rightarrow *'b* \times *'a* **where**

iterate *k* *f* *x* = (*if* *k* = 0 *then* *Pair* *x* *else* *f* *x* $\circ\rightarrow$ *iterate* (*k* − 1) *f*)

definition *range* :: *natural* \Rightarrow *seed* \Rightarrow *natural* \times *seed* **where**

range *k* = *iterate* (*log* 2147483561 *k*)
 ($\lambda l. \text{next} \circ\rightarrow (\lambda v. \text{Pair } (v + l * 2147483561)))$ 1
 $\circ\rightarrow (\lambda v. \text{Pair } (v \bmod k))$

lemma *range*:

k > 0 \implies *fst* (*range* *k* *s*) < *k*
 <proof>

definition *select* :: *'a* *list* \Rightarrow *seed* \Rightarrow *'a* \times *seed* **where**

select *xs* = *range* (*natural-of-nat* (*length* *xs*))
 $\circ\rightarrow (\lambda k. \text{Pair } (\text{nth } xs \ (\text{nat-of-natural } k)))$

lemma *select*:

assumes *xs* $\neq []$
shows *fst* (*select* *xs* *s*) \in *set* *xs*
 <proof>

primrec *pick* :: (*natural* \times *'a*) *list* \Rightarrow *natural* \Rightarrow *'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*) \implies *pick* *xs* *i* \in *set* (*map* *snd* *xs*)
 <proof>

lemma *pick-drop-zero*:

pick (*filter* ($\lambda(k, -). k > 0$) *xs*) = *pick* *xs*
 <proof>

lemma *pick-same*:

$l < \text{length } xs \implies \text{Random.pick } (\text{map } (\text{Pair } 1) \text{ } xs) \text{ } (\text{natural-of-nat } l) = \text{nth } xs \text{ } l$
 $\langle \text{proof} \rangle$

definition *select-weight* :: $(\text{natural} \times 'a) \text{ list} \Rightarrow \text{seed} \Rightarrow 'a \times \text{seed}$ **where**
select-weight $xs = \text{range } (\text{sum-list } (\text{map fst } xs))$
 $\circ \rightarrow (\lambda k. \text{Pair } (\text{pick } xs \text{ } k))$

lemma *select-weight-member*:
assumes $0 < \text{sum-list } (\text{map fst } xs)$
shows $\text{fst } (\text{select-weight } xs \text{ } s) \in \text{set } (\text{map snd } xs)$
 $\langle \text{proof} \rangle$

lemma *select-weight-cons-zero*:
select-weight $((0, x) \# xs) = \text{select-weight } xs$
 $\langle \text{proof} \rangle$

lemma *select-weight-drop-zero*:
select-weight $(\text{filter } (\lambda(k, -). k > 0) \text{ } xs) = \text{select-weight } xs$
 $\langle \text{proof} \rangle$

lemma *select-weight-select*:
assumes $xs \neq []$
shows *select-weight* $(\text{map } (\text{Pair } 1) \text{ } xs) = \text{select } xs$
 $\langle \text{proof} \rangle$

end

70.4 ML interface

code-reflect *Random-Engine*
functions *range select select-weight*

$\langle \text{ML} \rangle$

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 $(= = \subseteq_m$
begin

type-synonym $('a, 'b) \text{ map} = 'a \Rightarrow 'b \text{ option}$ **(infixr** $\langle \mapsto \rangle$ $0)$

abbreviation (*input*)

$empty :: 'a \rightarrow 'b$ **where**
 $empty \equiv \lambda x. None$

definition

$map-comp :: ('b \rightarrow 'c) \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'c)$ (**infixl** $\langle \circ_m \rangle$ 55) **where**
 $f \circ_m g = (\lambda k. \text{case } g \text{ of } None \Rightarrow None \mid Some \ v \Rightarrow f \ v)$

definition

$map-add :: ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$ (**infixl** $\langle ++ \rangle$ 100) **where**
 $m1 ++ m2 = (\lambda x. \text{case } m2 \text{ of } None \Rightarrow m1 \ x \mid Some \ y \Rightarrow Some \ y)$

definition

$restrict-map :: ('a \rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow ('a \rightarrow 'b)$ (**infixl** $\langle | \cdot \rangle$ 110) **where**
 $m | A = (\lambda x. \text{if } x \in A \text{ then } m \ x \text{ else } None)$

notation (*latex output*)

$restrict-map \ (\langle \cdot | \cdot \rangle \ [111,110] \ 110)$

definition

$dom :: ('a \rightarrow 'b) \Rightarrow 'a \text{ set}$ **where**
 $dom \ m = \{a. \ m \ a \neq None\}$

definition

$ran :: ('a \rightarrow 'b) \Rightarrow 'b \text{ set}$ **where**
 $ran \ m = \{b. \ \exists a. \ m \ a = Some \ b\}$

definition

$graph :: ('a \rightarrow 'b) \Rightarrow ('a \times 'b) \text{ set}$ **where**
 $graph \ m = \{(a, b) \mid a \ b. \ m \ a = Some \ b\}$

definition

$map-le :: ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b) \Rightarrow bool$ (**infix** $\langle \subseteq_m \rangle$ 50) **where**
 $(m_1 \subseteq_m m_2) \longleftrightarrow (\forall a \in dom \ m_1. \ m_1 \ a = m_2 \ a)$

Function update syntax $f(x := y, \dots)$ is extended with $x \mapsto y$, which is short for $x := Some \ y. :=$ and \mapsto can be mixed freely. The syntax $[x \mapsto y, \dots]$ is short for $Map.empty(x \mapsto y, \dots)$ but must only contain \mapsto , not $:=$, because $[x:=y]$ clashes with the list update syntax $xs[i:=x]$.

nonterminal *maplet and maplets***open-bundle** *maplet-syntax***begin****syntax**

$-maplet :: ['a, 'a] \Rightarrow maplet \ (\langle \langle open-block \ notation = \langle mixfix \ maplet \rangle \rangle - / \mapsto / - \rangle)$
 $:: maplet \Rightarrow updbind \ (\langle \cdot \rangle)$
 $:: maplet \Rightarrow maplets \ (\langle \cdot \rangle)$

```
-Maplets :: [maplet, maplets] ⇒ maplets  (⟨-,/ -⟩)
-Map      :: maplets ⇒ 'a → 'b  (⟨⟨indent=1 notation=⟨mixfix maplet⟩⟩[-]⟩)
```

syntax (*ASCII*)

```
-maplet :: ['a, 'a] ⇒ maplet  (⟨⟨open-block notation=⟨mixfix maplet⟩⟩- /|->/
-⟩)⟩
```

syntax-consts

```
-maplet -Maplets -Map ⇒ fun-upd
```

translations

```
-Update f (-maplet x y) ⇒ f(x := CONST Some y)
-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)
```

end

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)
⟨proof⟩
```

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 $xs \mapsto ys$ for lists xs and ys .

open-bundle list-maplet-syntax
begin

syntax

```
-maplets :: ['a, 'a] ⇒ maplet  (⟨⟨open-block notation=⟨mixfix maplet⟩⟩- /|↦|/
-⟩)⟩
```

syntax (*ASCII*)

```
-maplets :: ['a, 'a] ⇒ maplet  (⟨⟨open-block notation=⟨mixfix maplet⟩⟩- /|->|/
-⟩)⟩
```

syntax-consts

$\text{-maplets} \Rightarrow \text{map-upds}$

translations

$\text{-Update } m \ (\text{-maplets } xs \ ys) \Rightarrow \text{CONST map-upds } m \ xs \ ys$

$\text{-Map } (\text{-maplets } xs \ ys) \leftarrow \text{-Update } (\lambda u. \text{CONST None}) \ (\text{-maplets } xs \ ys)$

$\text{-Map } (\text{-updbinds } m \ (\text{-maplets } xs \ ys)) \leftarrow \text{-Update } (\text{-Map } m) \ (\text{-maplets } xs \ ys)$

end

71.1 empty

lemma *empty-upd-none* [simp]: $\text{empty}(x := \text{None}) = \text{empty}$
 $\langle \text{proof} \rangle$

71.2 map-upd

lemma *map-upd-triv*: $t \ k = \text{Some } x \implies t(k \mapsto x) = t$
 $\langle \text{proof} \rangle$

lemma *map-upd-nonempty* [simp]: $t(k \mapsto x) \neq \text{empty}$
 $\langle \text{proof} \rangle$

lemma *map-upd-eqD1*:
assumes $m(a \mapsto x) = n(a \mapsto y)$
shows $x = y$
 $\langle \text{proof} \rangle$

lemma *map-upd-Some-unfold*:
 $((m(a \mapsto b)) \ x = \text{Some } y) = (x = a \wedge b = y \vee x \neq a \wedge m \ x = \text{Some } y)$
 $\langle \text{proof} \rangle$

lemma *image-map-upd* [simp]: $x \notin A \implies m(x \mapsto y) \restriction A = m \restriction A$
 $\langle \text{proof} \rangle$

lemma *finite-range-updI*:
assumes *finite* (range f) **shows** *finite* (range ($f(a \mapsto b)$))
 $\langle \text{proof} \rangle$

71.3 map-of

lemma *map-of-eq-empty-iff* [simp]:
 $\text{map-of } xys = \text{empty} \longleftrightarrow xys = []$
 $\langle \text{proof} \rangle$

lemma *empty-eq-map-of-iff* [simp]:
 $\text{empty} = \text{map-of } xys \longleftrightarrow xys = []$
 $\langle \text{proof} \rangle$

lemma *map-of-eq-None-iff*:

$(\text{map-of } xys \ x = \text{None}) = (x \notin \text{fst } (set \ xys))$
 $\langle \text{proof} \rangle$

lemma *map-of-eq-Some-iff* [simp]:
 $\text{distinct}(\text{map } \text{fst } xys) \implies (\text{map-of } xys \ x = \text{Some } y) = ((x,y) \in \text{set } xys)$
 $\langle \text{proof} \rangle$

lemma *Some-eq-map-of-iff* [simp]:
 $\text{distinct}(\text{map } \text{fst } xys) \implies (\text{Some } y = \text{map-of } xys \ x) = ((x,y) \in \text{set } xys)$
 $\langle \text{proof} \rangle$

lemma *map-of-is-SomeI* [simp]:
 $\llbracket \text{distinct}(\text{map } \text{fst } xys); (x,y) \in \text{set } xys \rrbracket \implies \text{map-of } xys \ x = \text{Some } y$
 $\langle \text{proof} \rangle$

lemma *map-of-zip-is-None* [simp]:
 $\text{length } xs = \text{length } ys \implies (\text{map-of } (\text{zip } xs \ ys) \ x = \text{None}) = (x \notin \text{set } xs)$
 $\langle \text{proof} \rangle$

lemma *map-of-zip-is-Some*:
assumes $\text{length } xs = \text{length } ys$
shows $x \in \text{set } xs \longleftrightarrow (\exists y. \text{map-of } (\text{zip } xs \ ys) \ x = \text{Some } y)$
 $\langle \text{proof} \rangle$

lemma *map-of-zip-upd*:
fixes $x :: 'a$ **and** $xs :: 'a \text{ list}$ **and** $ys \ zs :: 'b \text{ list}$
assumes $\text{length } ys = \text{length } xs$
and $\text{length } zs = \text{length } xs$
and $x \notin \text{set } xs$
and $(\text{map-of } (\text{zip } xs \ ys))(x \mapsto y) = (\text{map-of } (\text{zip } xs \ zs))(x \mapsto z)$
shows $\text{map-of } (\text{zip } xs \ ys) = \text{map-of } (\text{zip } xs \ zs)$
 $\langle \text{proof} \rangle$

lemma *map-of-zip-inject*:
assumes $\text{length } ys = \text{length } xs$
and $\text{length } zs = \text{length } xs$
and $\text{dist: distinct } xs$
and $\text{map-of: map-of } (\text{zip } xs \ ys) = \text{map-of } (\text{zip } xs \ zs)$
shows $ys = zs$
 $\langle \text{proof} \rangle$

lemma *map-of-zip-nth*:
assumes $\text{length } xs = \text{length } ys$
assumes $\text{distinct } xs$
assumes $i < \text{length } ys$
shows $\text{map-of } (\text{zip } xs \ ys) \ (xs ! i) = \text{Some } (ys ! i)$
 $\langle \text{proof} \rangle$

lemma *map-of-zip-map*:

$\text{map-of } (\text{zip } xs \ (\text{map } f \ xs)) = (\lambda x. \text{ if } x \in \text{set } xs \text{ then } \text{Some } (f \ x) \text{ else } \text{None})$
 $\langle \text{proof} \rangle$

lemma *finite-range-map-of*: $\text{finite } (\text{range } (\text{map-of } xys))$
 $\langle \text{proof} \rangle$

lemma *map-of-SomeD*: $\text{map-of } xs \ k = \text{Some } y \implies (k, y) \in \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *map-of-mapk-SomeI*:
 $\text{inj } f \implies \text{map-of } t \ k = \text{Some } x \implies$
 $\text{map-of } (\text{map } (\text{case-prod } (\lambda k. \text{Pair } (f \ k))) \ t) \ (f \ k) = \text{Some } x$
 $\langle \text{proof} \rangle$

lemma *weak-map-of-SomeI*: $(k, x) \in \text{set } l \implies \exists x. \text{map-of } l \ k = \text{Some } x$
 $\langle \text{proof} \rangle$

lemma *map-of-filter-in*:
 $\text{map-of } xs \ k = \text{Some } z \implies P \ k \ z \implies \text{map-of } (\text{filter } (\text{case-prod } P) \ xs) \ k = \text{Some } z$
 $\langle \text{proof} \rangle$

lemma *map-of-map*:
 $\text{map-of } (\text{map } (\lambda(k, v). (k, f \ v)) \ xs) = \text{map-option } f \circ \text{map-of } xs$
 $\langle \text{proof} \rangle$

lemma *map-of-filter*:
 $\text{map-of } (\text{filter } (\lambda x. P \ (\text{fst } x)) \ xs) = \text{map-of } xs \mid \text{'Collect } P$
 $\langle \text{proof} \rangle$

lemma *dom-map-option*:
 $\text{dom } (\lambda k. \text{map-option } (f \ k) \ (m \ k)) = \text{dom } m$
 $\langle \text{proof} \rangle$

lemma *dom-map-option-comp [simp]*:
 $\text{dom } (\text{map-option } g \circ m) = \text{dom } m$
 $\langle \text{proof} \rangle$

71.4 map-option related

lemma *map-option-o-empty [simp]*: $\text{map-option } f \circ \text{empty} = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *map-option-o-map-upd [simp]*:
 $\text{map-option } f \circ m(a \mapsto b) = (\text{map-option } f \circ m)(a \mapsto f \ b)$
 $\langle \text{proof} \rangle$

71.5 map-comp related

lemma *map-comp-empty [simp]*:

$m \circ_m \text{empty} = \text{empty}$
 $\text{empty} \circ_m m = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *map-comp-simps* [*simp*]:
 $m2\ k = \text{None} \implies (m1 \circ_m m2)\ k = \text{None}$
 $m2\ k = \text{Some } k' \implies (m1 \circ_m m2)\ k = m1\ k'$
 $\langle \text{proof} \rangle$

lemma *map-comp-Some-iff*:
 $((m1 \circ_m m2)\ k = \text{Some } v) = (\exists k'.\ m2\ k = \text{Some } k' \wedge m1\ k' = \text{Some } v)$
 $\langle \text{proof} \rangle$

lemma *map-comp-None-iff*:
 $((m1 \circ_m m2)\ k = \text{None}) = (m2\ k = \text{None} \vee (\exists k'.\ m2\ k = \text{Some } k' \wedge m1\ k' = \text{None}))$
 $\langle \text{proof} \rangle$

71.6 ++

lemma *map-add-empty*[*simp*]: $m\ ++\ \text{empty} = m$
 $\langle \text{proof} \rangle$

lemma *empty-map-add*[*simp*]: $\text{empty}\ ++\ m = m$
 $\langle \text{proof} \rangle$

lemma *map-add-assoc*[*simp*]: $m1\ ++\ (m2\ ++\ m3) = (m1\ ++\ m2)\ ++\ m3$
 $\langle \text{proof} \rangle$

lemma *map-add-Some-iff*:
 $((m\ ++\ n)\ k = \text{Some } x) = (n\ k = \text{Some } x \vee n\ k = \text{None} \wedge m\ k = \text{Some } x)$
 $\langle \text{proof} \rangle$

lemma *map-add-SomeD* [*dest!*]:
 $(m\ ++\ n)\ k = \text{Some } x \implies n\ k = \text{Some } x \vee n\ k = \text{None} \wedge m\ k = \text{Some } x$
 $\langle \text{proof} \rangle$

lemma *map-add-find-right* [*simp*]: $n\ k = \text{Some } xx \implies (m\ ++\ n)\ k = \text{Some } xx$
 $\langle \text{proof} \rangle$

lemma *map-add-None* [*iff*]: $((m\ ++\ n)\ k = \text{None}) = (n\ k = \text{None} \wedge m\ k = \text{None})$
 $\langle \text{proof} \rangle$

lemma *map-add-upd*[*simp*]: $f\ ++\ g(x \mapsto y) = (f\ ++\ g)(x \mapsto y)$
 $\langle \text{proof} \rangle$

lemma *map-add-upds*[*simp*]: $m1\ ++\ (m2(xs[\mapsto]ys)) = (m1\ ++\ m2)(xs[\mapsto]ys)$
 $\langle \text{proof} \rangle$

lemma *map-add-upd-left*: $m \notin \text{dom } e2 \implies e1(m \mapsto u1) ++ e2 = (e1 ++ e2)(m \mapsto u1)$
 $\langle \text{proof} \rangle$

lemma *map-of-append[simp]*: $\text{map-of } (xs @ ys) = \text{map-of } ys ++ \text{map-of } xs$
 $\langle \text{proof} \rangle$

lemma *finite-range-map-of-map-add*:
 $\text{finite } (\text{range } f) \implies \text{finite } (\text{range } (f ++ \text{map-of } l))$
 $\langle \text{proof} \rangle$

lemma *inj-on-map-add-dom [iff]*:
 $\text{inj-on } (m ++ m') (\text{dom } m') = \text{inj-on } m' (\text{dom } m')$
 $\langle \text{proof} \rangle$

lemma *map-upds-fold-map-upd*:
 $m(\text{ks}[\mapsto] \text{vs}) = \text{foldl } (\lambda m (k, v). m(k \mapsto v)) m (\text{zip } \text{ks } \text{vs})$
 $\langle \text{proof} \rangle$

lemma *map-add-map-of-foldr*:
 $m ++ \text{map-of } ps = \text{foldr } (\lambda(k, v) m. m(k \mapsto v)) ps m$
 $\langle \text{proof} \rangle$

71.7 restrict-map

lemma *restrict-map-to-empty [simp]*: $m|'\{\} = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *restrict-map-insert*: $f|'(\text{insert } a A) = (f|'A)(a := f a)$
 $\langle \text{proof} \rangle$

lemma *restrict-map-empty [simp]*: $\text{empty}|'D = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *restrict-in [simp]*: $x \in A \implies (m|'A) x = m x$
 $\langle \text{proof} \rangle$

lemma *restrict-out [simp]*: $x \notin A \implies (m|'A) x = \text{None}$
 $\langle \text{proof} \rangle$

lemma *ran-restrictD*: $y \in \text{ran } (m|'A) \implies \exists x \in A. m x = \text{Some } y$
 $\langle \text{proof} \rangle$

lemma *dom-restrict [simp]*: $\text{dom } (m|'A) = \text{dom } m \cap A$
 $\langle \text{proof} \rangle$

lemma *restrict-upd-same [simp]*: $m(x \mapsto y)|'(-\{x\}) = m|'(-\{x\})$
 $\langle \text{proof} \rangle$

lemma *restrict-restrict* [simp]: $m|'A|'B = m|(A \cap B)$
 ⟨proof⟩

lemma *restrict-fun-upd* [simp]:
 $m(x := y)|'D = (\text{if } x \in D \text{ then } (m|'(D - \{x\}))(x := y) \text{ else } m|'D)$
 ⟨proof⟩

lemma *fun-upd-None-restrict* [simp]:
 $(m|'D)(x := \text{None}) = (\text{if } x \in D \text{ then } m|(D - \{x\}) \text{ else } m|'D)$
 ⟨proof⟩

lemma *fun-upd-restrict*: $(m|'D)(x := y) = (m|'(D - \{x\}))(x := y)$
 ⟨proof⟩

lemma *fun-upd-restrict-conv* [simp]:
 $x \in D \implies (m|'D)(x := y) = (m|'(D - \{x\}))(x := y)$
 ⟨proof⟩

lemma *map-of-map-restrict*:
 $\text{map-of } (\text{map } (\lambda k. (k, f k)) \text{ } ks) = (\text{Some} \circ f) |' \text{ set } ks$
 ⟨proof⟩

lemma *restrict-complement-singleton-eq*:
 $f |' (- \{x\}) = f(x := \text{None})$
 ⟨proof⟩

71.8 map-upds

lemma *map-upds-Nil1* [simp]: $m(\square \mapsto bs) = m$
 ⟨proof⟩

lemma *map-upds-Nil2* [simp]: $m(as \mapsto \square) = m$
 ⟨proof⟩

lemma *map-upds-Cons* [simp]: $m(a \# as \mapsto b \# bs) = (m(a \mapsto b))(as \mapsto bs)$
 ⟨proof⟩

lemma *map-upds-append1* [simp]:
 $\text{size } xs < \text{size } ys \implies m(xs @ [x] \mapsto ys) = m(xs \mapsto ys, x \mapsto ys! \text{size } xs)$
 ⟨proof⟩

lemma *map-upds-list-update2-drop* [simp]:
 $\text{size } xs \leq i \implies m(xs \mapsto ys[i := y]) = m(xs \mapsto ys)$
 ⟨proof⟩

Something weirdly sensitive about this proof, which needs only four lines in apply style

lemma *map-upd-upds-conv-if*:
 $(f(x \mapsto y))(xs \mapsto ys) =$

$(\text{if } x \in \text{set}(\text{take } (\text{length } ys) \text{ } xs) \text{ then } f(xs \text{ } [\mapsto] \text{ } ys)$
 $\text{else } (f(xs \text{ } [\mapsto] \text{ } ys))(x \mapsto y))$
 $\langle \text{proof} \rangle$

lemma *map-upds-twist* [simp]:
 $a \notin \text{set } as \implies m(a \mapsto b, as[\mapsto] bs) = m(as[\mapsto] bs, a \mapsto b)$
 $\langle \text{proof} \rangle$

lemma *map-upds-apply-nontin* [simp]:
 $x \notin \text{set } xs \implies (f(xs[\mapsto] ys)) \ x = f \ x$
 $\langle \text{proof} \rangle$

lemma *fun-upds-append-drop* [simp]:
 $\text{size } xs = \text{size } ys \implies m(xs @ zs[\mapsto] ys) = m(xs[\mapsto] ys)$
 $\langle \text{proof} \rangle$

lemma *fun-upds-append2-drop* [simp]:
 $\text{size } xs = \text{size } ys \implies m(xs[\mapsto] ys @ zs) = m(xs[\mapsto] ys)$
 $\langle \text{proof} \rangle$

lemma *restrict-map-upds* [simp]:
 $\llbracket \text{length } xs = \text{length } ys; \text{set } xs \subseteq D \rrbracket$
 $\implies m(xs \text{ } [\mapsto] \text{ } ys) \upharpoonright D = (m \upharpoonright (D - \text{set } xs))(xs \text{ } [\mapsto] \text{ } ys)$
 $\langle \text{proof} \rangle$

71.9 dom

lemma *dom-eq-empty-conv* [simp]: $\text{dom } f = \{\} \longleftrightarrow f = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *domI*: $m \ a = \text{Some } b \implies a \in \text{dom } m$
 $\langle \text{proof} \rangle$

lemma *domD*: $a \in \text{dom } m \implies \exists b. m \ a = \text{Some } b$
 $\langle \text{proof} \rangle$

lemma *domIff* [iff, simp del, code-unfold]: $a \in \text{dom } m \longleftrightarrow m \ a \neq \text{None}$
 $\langle \text{proof} \rangle$

lemma *dom-empty* [simp]: $\text{dom } \text{empty} = \{\}$
 $\langle \text{proof} \rangle$

lemma *dom-fun-upd* [simp]:
 $\text{dom}(f(x := y)) = (\text{if } y = \text{None} \text{ then } \text{dom } f - \{x\} \text{ else } \text{insert } x \ (\text{dom } f))$
 $\langle \text{proof} \rangle$

lemma *dom-if*:

$\text{dom } (\lambda x. \text{ if } P \ x \text{ then } f \ x \text{ else } g \ x) = \text{dom } f \cap \{x. P \ x\} \cup \text{dom } g \cap \{x. \neg P \ x\}$
 ⟨proof⟩

lemma *dom-map-of-conv-image-fst*:

$\text{dom } (\text{map-of } xys) = \text{fst } \text{‘ set } xys$
 ⟨proof⟩

lemma *dom-map-of-zip [simp]*: $\text{length } xs = \text{length } ys \implies \text{dom } (\text{map-of } (\text{zip } xs \ ys))$
 $= \text{set } xs$
 ⟨proof⟩

lemma *finite-dom-map-of*: $\text{finite } (\text{dom } (\text{map-of } l))$
 ⟨proof⟩

lemma *dom-map-upds [simp]*:

$\text{dom}(m(xs[\mapsto]ys)) = \text{set}(\text{take } (\text{length } ys) \ xs) \cup \text{dom } m$
 ⟨proof⟩

lemma *dom-map-add [simp]*: $\text{dom } (m ++ n) = \text{dom } n \cup \text{dom } m$
 ⟨proof⟩

lemma *dom-override-on [simp]*:

$\text{dom } (\text{override-on } f \ g \ A) =$
 $(\text{dom } f - \{a. a \in A - \text{dom } g\}) \cup \{a. a \in A \cap \text{dom } g\}$
 ⟨proof⟩

lemma *map-add-comm*: $\text{dom } m1 \cap \text{dom } m2 = \{\} \implies m1 ++ m2 = m2 ++ m1$
 ⟨proof⟩

lemma *map-add-dom-app-simps*:

$m \in \text{dom } l2 \implies (l1 ++ l2) \ m = l2 \ m$
 $m \notin \text{dom } l1 \implies (l1 ++ l2) \ m = l2 \ m$
 $m \notin \text{dom } l2 \implies (l1 ++ l2) \ m = l1 \ m$
 ⟨proof⟩

lemma *dom-const [simp]*:

$\text{dom } (\lambda x. \text{ Some } (f \ x)) = \text{UNIV}$
 ⟨proof⟩

lemma *finite-map-freshness*:

$\text{finite } (\text{dom } (f :: 'a \multimap 'b)) \implies \neg \text{finite } (\text{UNIV} :: 'a \text{ set}) \implies$
 $\exists x. f \ x = \text{None}$
 ⟨proof⟩

lemma *dom-minus*:

$f \ x = \text{None} \implies \text{dom } f - \text{insert } x \ A = \text{dom } f - A$
 ⟨proof⟩

lemma *insert-dom*:

$f\ x = \text{Some } y \implies \text{insert } x\ (\text{dom } f) = \text{dom } f$
 $\langle \text{proof} \rangle$

lemma *map-of-map-keys*:

$\text{set } xs = \text{dom } m \implies \text{map-of } (\text{map } (\lambda k. (k, \text{the } (m\ k))))\ xs = m$
 $\langle \text{proof} \rangle$

lemma *map-of-eqI*:

assumes *set-eq*: $\text{set } (\text{map } \text{fst } xs) = \text{set } (\text{map } \text{fst } ys)$
assumes *map-eq*: $\forall k \in \text{set } (\text{map } \text{fst } xs). \text{map-of } xs\ k = \text{map-of } ys\ k$
shows $\text{map-of } xs = \text{map-of } ys$
 $\langle \text{proof} \rangle$

lemma *map-of-eq-dom*:

assumes $\text{map-of } xs = \text{map-of } ys$
shows $\text{fst } ` \text{set } xs = \text{fst } ` \text{set } ys$
 $\langle \text{proof} \rangle$

lemma *finite-set-of-finite-maps*:

assumes *finite A finite B*
shows $\text{finite } \{m. \text{dom } m = A \wedge \text{ran } m \subseteq B\}$ (**is finite** ?*S*)
 $\langle \text{proof} \rangle$

71.10 *ran*

lemma *ranI*: $m\ a = \text{Some } b \implies b \in \text{ran } m$
 $\langle \text{proof} \rangle$

lemma *ran-empty* [*simp*]: $\text{ran } \text{empty} = \{\}$
 $\langle \text{proof} \rangle$

lemma *ran-map-upd* [*simp*]: $m\ a = \text{None} \implies \text{ran}(m(a \mapsto b)) = \text{insert } b\ (\text{ran } m)$
 $\langle \text{proof} \rangle$

lemma *fun-upd-None-if-notin-dom* [*simp*]: $k \notin \text{dom } m \implies m(k := \text{None}) = m$
 $\langle \text{proof} \rangle$

lemma *ran-map-upd-Some*:

$\llbracket m\ x = \text{Some } y; \text{inj-on } m\ (\text{dom } m); z \notin \text{ran } m \rrbracket \implies \text{ran}(m(x := \text{Some } z)) =$
 $\text{ran } m - \{y\} \cup \{z\}$
 $\langle \text{proof} \rangle$

lemma *ran-map-add*:

assumes $\text{dom } m1 \cap \text{dom } m2 = \{\}$
shows $\text{ran } (m1 ++ m2) = \text{ran } m1 \cup \text{ran } m2$
 $\langle \text{proof} \rangle$

lemma *finite-ran*:

assumes *finite* (*dom p*)

shows *finite* (*ran p*)

<proof>

lemma *ran-distinct*:

assumes *dist*: *distinct* (*map fst al*)

shows *ran* (*map-of al*) = *snd* ‘ *set al*

<proof>

lemma *ran-map-of-zip*:

assumes *length xs* = *length ys* *distinct xs*

shows *ran* (*map-of (zip xs ys)*) = *set ys*

<proof>

lemma *ran-map-option*: *ran* ($\lambda x. \text{map-option } f \ (m \ x)$) = *f* ‘ *ran m*

<proof>

71.11 *graph*

lemma *graph-empty[simp]*: *graph empty* = {}

<proof>

lemma *in-graphI*: *m k* = *Some v* \implies (*k*, *v*) \in *graph m*

<proof>

lemma *in-graphD*: (*k*, *v*) \in *graph m* \implies *m k* = *Some v*

<proof>

lemma *graph-map-upd[simp]*: *graph* (*m*(*k* \mapsto *v*)) = *insert* (*k*, *v*) (*graph* (*m*(*k* := *None*)))

<proof>

lemma *graph-fun-upd-None*: *graph* (*m*(*k* := *None*)) = {*e* \in *graph m*. *fst e* \neq *k*}

<proof>

lemma *graph-restrictD*:

assumes (*k*, *v*) \in *graph* (*m* | ‘ *A*)

shows *k* \in *A* **and** *m k* = *Some v*

<proof>

lemma *graph-map-comp[simp]*: *graph* (*m1* \circ_m *m2*) = *graph m2* *O* *graph m1*

<proof>

lemma *graph-map-add*: *dom m1* \cap *dom m2* = {} \implies *graph* (*m1* ++ *m2*) = *graph m1* \cup *graph m2*

<proof>

lemma *graph-eq-to-snd-dom*: $\text{graph } m = (\lambda x. (x, \text{the } (m \ x))) \text{ ‘ dom } m$
 $\langle \text{proof} \rangle$

lemma *fst-graph-eq-dom*: $\text{fst ‘ graph } m = \text{dom } m$
 $\langle \text{proof} \rangle$

lemma *graph-domD*: $x \in \text{graph } m \implies \text{fst } x \in \text{dom } m$
 $\langle \text{proof} \rangle$

lemma *snd-graph-ran*: $\text{snd ‘ graph } m = \text{ran } m$
 $\langle \text{proof} \rangle$

lemma *graph-ranD*: $x \in \text{graph } m \implies \text{snd } x \in \text{ran } m$
 $\langle \text{proof} \rangle$

lemma *finite-graph-map-of*: $\text{finite } (\text{graph } (\text{map-of } al))$
 $\langle \text{proof} \rangle$

lemma *graph-map-of-if-distinct-dom*: $\text{distinct } (\text{map fst } al) \implies \text{graph } (\text{map-of } al)$
 $= \text{set } al$
 $\langle \text{proof} \rangle$

lemma *finite-graph-iff-finite-dom[simp]*: $\text{finite } (\text{graph } m) = \text{finite } (\text{dom } m)$
 $\langle \text{proof} \rangle$

lemma *inj-on-fst-graph*: $\text{inj-on fst } (\text{graph } m)$
 $\langle \text{proof} \rangle$

71.12 map-le

lemma *map-le-empty [simp]*: $\text{empty} \subseteq_m g$
 $\langle \text{proof} \rangle$

lemma *upd-None-map-le [simp]*: $f(x := \text{None}) \subseteq_m f$
 $\langle \text{proof} \rangle$

lemma *map-le-upd[simp]*: $f \subseteq_m g \implies f(a := b) \subseteq_m g(a := b)$
 $\langle \text{proof} \rangle$

lemma *map-le-imp-upd-le [simp]*: $m1 \subseteq_m m2 \implies m1(x := \text{None}) \subseteq_m m2(x \mapsto y)$
 $\langle \text{proof} \rangle$

lemma *map-le-upds [simp]*:
 $f \subseteq_m g \implies f(as \mapsto bs) \subseteq_m g(as \mapsto bs)$
 $\langle \text{proof} \rangle$

lemma *map-le-implies-dom-le*: $(f \subseteq_m g) \implies (\text{dom } f \subseteq \text{dom } g)$
 $\langle \text{proof} \rangle$

lemma *map-le-refl* [*simp*]: $f \subseteq_m f$
 ⟨*proof*⟩

lemma *map-le-trans*[*trans*]: $\llbracket m1 \subseteq_m m2; m2 \subseteq_m m3 \rrbracket \Longrightarrow m1 \subseteq_m m3$
 ⟨*proof*⟩

lemma *map-le-antisym*: $\llbracket f \subseteq_m g; g \subseteq_m f \rrbracket \Longrightarrow f = g$
 ⟨*proof*⟩

lemma *map-le-map-add* [*simp*]: $f \subseteq_m g \mathrel{++} f$
 ⟨*proof*⟩

lemma *map-le-iff-map-add-commute*: $f \subseteq_m f \mathrel{++} g \longleftrightarrow f \mathrel{++} g = g \mathrel{++} f$
 ⟨*proof*⟩

lemma *map-add-le-mapE*: $f \mathrel{++} g \subseteq_m h \Longrightarrow g \subseteq_m h$
 ⟨*proof*⟩

lemma *map-add-le-mapI*: $\llbracket f \subseteq_m h; g \subseteq_m h \rrbracket \Longrightarrow f \mathrel{++} g \subseteq_m h$
 ⟨*proof*⟩

lemma *map-add-subsumed1*: $f \subseteq_m g \Longrightarrow f \mathrel{++} g = g$
 ⟨*proof*⟩

lemma *map-add-subsumed2*: $f \subseteq_m g \Longrightarrow g \mathrel{++} f = g$
 ⟨*proof*⟩

lemma *dom-eq-singleton-conv*: $\text{dom } f = \{x\} \longleftrightarrow (\exists v. f = [x \mapsto v])$
 (is ?lhs \longleftrightarrow ?rhs)
 ⟨*proof*⟩

lemma *map-add-eq-empty-iff*[*simp*]:
 $(f \mathrel{++} g = \text{empty}) \longleftrightarrow f = \text{empty} \wedge g = \text{empty}$
 ⟨*proof*⟩

lemma *empty-eq-map-add-iff*[*simp*]:
 $(\text{empty} = f \mathrel{++} g) \longleftrightarrow f = \text{empty} \wedge g = \text{empty}$
 ⟨*proof*⟩

71.13 Various

lemma *set-map-of-compr*:
 assumes *distinct*: *distinct* (map fst *xs*)
 shows $\text{set } xs = \{(k, v). \text{map-of } xs \ k = \text{Some } v\}$
 ⟨*proof*⟩

lemma *eq-key-imp-eq-value*:
 $v1 = v2$

if *distinct* (*map fst xs*) (*k*, *v1*) ∈ *set xs* (*k*, *v2*) ∈ *set xs*
 ⟨*proof*⟩

lemma *map-of-inject-set*:
 assumes *distinct*: *distinct* (*map fst xs*) *distinct* (*map fst ys*)
 shows *map-of xs* = *map-of ys* \longleftrightarrow *set xs* = *set ys* (is ?*lhs* \longleftrightarrow ?*rhs*)
 ⟨*proof*⟩

lemma *finite-Map-induct*[*consumes 1, case-names empty update*]:
 assumes *finite* (*dom m*)
 assumes *P Map.empty*
 assumes $\bigwedge k v m. \text{finite } (\text{dom } m) \implies k \notin \text{dom } m \implies P m \implies P (m(k \mapsto v))$
 shows *P m*
 ⟨*proof*⟩

hide-const (open) *Map.empty 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 bool) \Rightarrow bool
 fixes *enum-ex* :: ('a \Rightarrow bool) \Rightarrow bool
 assumes *UNIV-enum*: *UNIV* = *set enum*
 and *enum-distinct*: *distinct enum*
 assumes *enum-all-UNIV*: *enum-all P* \longleftrightarrow *Ball UNIV P*
 assumes *enum-ex-UNIV*: *enum-ex P* \longleftrightarrow *Bex UNIV P*
 — tailored towards simple instantiation
begin

subclass *finite* ⟨*proof*⟩

lemma *enum-UNIV*:
set enum = *UNIV*
 ⟨*proof*⟩

lemma *in-enum*: *x* ∈ *set enum*
 ⟨*proof*⟩

lemma *enum-eq-I*:
 assumes $\bigwedge x. x \in \text{set } xs$

shows $set\ enum = set\ xs$
 $\langle proof \rangle$

lemma *card-UNIV-length-enum*:
 $card\ (UNIV :: 'a\ set) = length\ enum$
 $\langle proof \rangle$

lemma *enum-all [simp]*:
 $enum-all = HOL.All$
 $\langle proof \rangle$

lemma *enum-ex [simp]*:
 $enum-ex = HOL.Ex$
 $\langle proof \rangle$

end

72.2 Implementations using *enum*

72.2.1 Unbounded operations and quantifiers

lemma *Collect-code [code]*:
 $Collect\ P = set\ (filter\ P\ enum)$
 $\langle proof \rangle$

lemma *vimage-code [code]*:
 $f - ' B = set\ (filter\ (\lambda x. f\ x \in B)\ enum-class.enum)$
 $\langle proof \rangle$

definition *card-UNIV* :: $'a\ itself \Rightarrow nat$
where
 $card-UNIV\ TYPE('a) = card\ (UNIV :: 'a\ set)$

lemma *[code]*:
 $card-UNIV\ TYPE('a :: enum) = card\ (set\ (Enum.enum :: 'a\ list))$
 $\langle proof \rangle$

lemma *all-code [code]*: $(\forall x. P\ x) \longleftrightarrow enum-all\ P$
 $\langle proof \rangle$

lemma *exists-code [code]*: $(\exists x. P\ x) \longleftrightarrow enum-ex\ P$
 $\langle proof \rangle$

lemma *exists1-code [code]*: $(\exists! x. P\ x) \longleftrightarrow list-ex1\ P\ enum$
 $\langle proof \rangle$

72.2.2 An executable choice operator

definition
 $enum-the = The$

lemma [code]:
The $P = (\text{case filter } P \text{ enum of } [x] \Rightarrow x \mid - \Rightarrow \text{enum-the } P)$
 <proof>

declare [[code abort: enum-the]]

code-printing
constant enum-the \rightarrow (Eval) (fn '- => raise Match)

72.2.3 Equality and order on functions

instantiation fun :: (enum, equal) equal
begin

definition
HOL.equal $f\ g \longleftrightarrow (\forall x \in \text{set enum. } f\ x = g\ x)$

instance <proof>

end

lemma [code]:
HOL.equal $f\ g \longleftrightarrow \text{enum-all } (\%x. f\ x = g\ x)$
 <proof>

lemma [code nbe]:
HOL.equal $(f :: - \Rightarrow -) \longleftrightarrow \text{True}$
 <proof>

lemma order-fun [code]:
fixes $f\ g :: 'a::\text{enum} \Rightarrow 'b::\text{order}$
shows $f \leq g \longleftrightarrow \text{enum-all } (\lambda x. f\ x \leq g\ x)$
and $f < g \longleftrightarrow f \leq g \wedge \text{enum-ex } (\lambda x. f\ x \neq g\ x)$
 <proof>

72.2.4 Operations on relations

lemma [code]:
Id = image $(\lambda x. (x, x))$ (set Enum.enum)
 <proof>

lemma tranclp-unfold [code]:
tranclp $r\ a\ b \longleftrightarrow (a, b) \in \text{trancl } \{(x, y). r\ x\ y\}$
 <proof>

lemma rtranclp-rtrancl-eq [code]:
rtranclp $r\ x\ y \longleftrightarrow (x, y) \in \text{rtrancl } \{(x, y). r\ x\ y\}$
 <proof>

lemma *max-ext-eq* [code]:

$\text{max-ext } R = \{(X, Y). \text{finite } X \wedge \text{finite } Y \wedge Y \neq \{\} \wedge (\forall x. x \in X \longrightarrow (\exists xa \in Y. (x, xa) \in R))\}$
 ⟨proof⟩

lemma *max-extp-eq* [code]:

$\text{max-extp } r \ x \ y \longleftrightarrow (x, y) \in \text{max-ext } \{(x, y). r \ x \ y\}$
 ⟨proof⟩

lemma *mlex-eq* [code]:

$f <^* \text{mlex}^* R = \{(x, y). f \ x < f \ y \vee (f \ x \leq f \ y \wedge (x, y) \in R)\}$
 ⟨proof⟩

72.2.5 Bounded accessible part

primrec *bacc* :: ('a × 'a) set ⇒ nat ⇒ 'a set

where

$\text{bacc } r \ 0 = \{x. \forall y. (y, x) \notin r\}$
 $\mid \text{bacc } r \ (\text{Suc } n) = (\text{bacc } r \ n \cup \{x. \forall y. (y, x) \in r \longrightarrow y \in \text{bacc } r \ n\})$

lemma *bacc-subseteq-acc*:

$\text{bacc } r \ n \subseteq \text{Wellfounded.acc } r$
 ⟨proof⟩

lemma *bacc-mono*:

$n \leq m \implies \text{bacc } r \ n \subseteq \text{bacc } r \ m$
 ⟨proof⟩

lemma *bacc-upper-bound*:

$\text{bacc } (r :: ('a \times 'a) \text{ set}) \ (\text{card } (\text{UNIV} :: 'a::\text{finite set})) = (\bigcup n. \text{bacc } r \ n)$
 ⟨proof⟩

lemma *acc-subseteq-bacc*:

assumes *finite* *r*

shows $\text{Wellfounded.acc } r \subseteq (\bigcup n. \text{bacc } r \ n)$

⟨proof⟩

lemma *acc-bacc-eq*:

fixes *A* :: ('a :: finite × 'a) set

assumes *finite* *A*

shows $\text{Wellfounded.acc } A = \text{bacc } A \ (\text{card } (\text{UNIV} :: 'a \text{ set}))$

⟨proof⟩

lemma [code]:

fixes *xs* :: ('a::finite × 'a) list

shows $\text{Wellfounded.acc } (\text{set } xs) = \text{bacc } (\text{set } xs) \ (\text{card-UNIV TYPE('a)})$

⟨proof⟩

72.3 Default instances for *enum*

lemma *map-of-zip-enum-is-Some*:

assumes *length ys = length (enum :: 'a::enum list)*

shows $\exists y. \text{map-of } (\text{zip } (\text{enum} :: 'a::\text{enum list}) \text{ ys}) \ x = \text{Some } y$

<proof>

lemma *map-of-zip-enum-inject*:

fixes *xs ys :: 'b::enum list*

assumes *length: length xs = length (enum :: 'a::enum list)*

length ys = length (enum :: 'a::enum list)

and *map-of: the \circ map-of (zip (enum :: 'a::enum list) xs) = the \circ map-of (zip (enum :: 'a::enum list) ys)*

shows *xs = ys*

<proof>

definition *all-n-lists* :: $((\text{'a} :: \text{enum}) \text{ list} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

all-n-lists P n $\longleftrightarrow (\forall xs \in \text{set } (\text{List.n-lists } n \text{ enum}). P \ xs)$

lemma [*code*]:

all-n-lists P n $\longleftrightarrow (\text{if } n = 0 \text{ then } P \ [] \text{ else enum-all } (\%x. \text{all-n-lists } (\%xs. P \ (x \# xs)) \ (n - 1)))$

<proof>

definition *ex-n-lists* :: $((\text{'a} :: \text{enum}) \text{ list} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

ex-n-lists P n $\longleftrightarrow (\exists xs \in \text{set } (\text{List.n-lists } n \text{ enum}). P \ xs)$

lemma [*code*]:

ex-n-lists P n $\longleftrightarrow (\text{if } n = 0 \text{ then } P \ [] \text{ else enum-ex } (\%x. \text{ex-n-lists } (\%xs. P \ (x \# xs)) \ (n - 1)))$

<proof>

instantiation *fun* :: $(\text{enum}, \text{enum}) \text{ enum}$

begin

definition

enum = map ($\lambda ys. \text{the} \circ \text{map-of } (\text{zip } (\text{enum}::'\text{a list}) \text{ ys})$) (List.n-lists (length (enum::'\text{a::enum list})) enum)

definition

enum-all P = all-n-lists ($\lambda bs. P \ (\text{the} \circ \text{map-of } (\text{zip } \text{enum } bs))$) (length (enum :: '\text{a list}))

definition

enum-ex P = ex-n-lists ($\lambda bs. P \ (\text{the} \circ \text{map-of } (\text{zip } \text{enum } bs))$) (length (enum :: '\text{a list}))

instance *<proof>*

end

lemma *enum-fun-code* [code]: *enum* = (let *enum-a* = (*enum* :: 'a::{*enum*, *equal*} list)
 in map ($\lambda y s. the \circ map-of (zip\ enum-a\ ys)$) (*List.n-lists* (*length enum-a*) *enum*))
 ⟨*proof*⟩

lemma *enum-all-fun-code* [code]:
enum-all *P* = (let *enum-a* = (*enum* :: 'a::{*enum*, *equal*} list)
 in *all-n-lists* ($\lambda bs. P (the \circ map-of (zip\ enum-a\ bs))$) (*length enum-a*))
 ⟨*proof*⟩

lemma *enum-ex-fun-code* [code]:
enum-ex *P* = (let *enum-a* = (*enum* :: 'a::{*enum*, *equal*} list)
 in *ex-n-lists* ($\lambda bs. P (the \circ map-of (zip\ enum-a\ bs))$) (*length enum-a*))
 ⟨*proof*⟩

instantiation *set* :: (*enum*) *enum*
begin

definition
enum = map *set* (*subseqs enum*)

definition
enum-all *P* $\longleftrightarrow (\forall A \in set\ enum. P (A :: 'a\ set))$

definition
enum-ex *P* $\longleftrightarrow (\exists A \in set\ enum. P (A :: 'a\ set))$

instance ⟨*proof*⟩

end

instantiation *unit* :: *enum*
begin

definition
enum = [()]

definition
enum-all *P* = *P* ()

definition
enum-ex *P* = *P* ()

instance ⟨*proof*⟩

end

instantiation *bool* :: *enum*
begin

definition
 $enum = [False, True]$

definition
 $enum-all\ P \longleftrightarrow P\ False \wedge P\ True$

definition
 $enum-ex\ P \longleftrightarrow P\ False \vee P\ True$

instance $\langle proof \rangle$

end

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
 $\langle proof \rangle$

end

instantiation *sum* :: (*enum*, *enum*) *enum*
begin

definition
 $enum = map\ Inl\ enum\ @\ map\ Inr\ enum$

definition
 $enum-all\ P \longleftrightarrow enum-all\ (\lambda x.\ P\ (Inl\ x)) \wedge enum-all\ (\lambda x.\ P\ (Inr\ x))$

definition
 $enum-ex\ P \longleftrightarrow enum-ex\ (\lambda x.\ P\ (Inl\ x)) \vee enum-ex\ (\lambda x.\ P\ (Inr\ x))$

instance $\langle proof \rangle$

end

instantiation *option* :: (*enum*) *enum*
begin

definition
 $enum = None \# \text{map } Some \text{ enum}$

definition
 $enum\text{-}all \ P \longleftrightarrow P \ None \wedge enum\text{-}all \ (\lambda x. P \ (Some \ x))$

definition
 $enum\text{-}ex \ P \longleftrightarrow P \ None \vee enum\text{-}ex \ (\lambda x. P \ (Some \ x))$

instance $\langle proof \rangle$

end

72.4 Small finite types

We define small finite types for use in Quickcheck

datatype (*plugins only: code quickcheck extraction*) *finite-1* =
 a_1

notation (**output**) $a_1 \ (\langle a_1 \rangle)$

lemma *UNIV-finite-1*:
 $UNIV = \{a_1\}$
 $\langle proof \rangle$

instantiation *finite-1* :: *enum*
begin

definition
 $enum = [a_1]$

definition
 $enum\text{-}all \ P = P \ a_1$

definition
 $enum\text{-}ex \ P = P \ a_1$

instance $\langle proof \rangle$

end

instantiation *finite-1* :: *linorder*
begin

definition *less-finite-1* :: *finite-1* \Rightarrow *finite-1* \Rightarrow *bool*

where

$x < (y :: \text{finite-1}) \longleftrightarrow \text{False}$

definition *less-eq-finite-1* :: *finite-1* \Rightarrow *finite-1* \Rightarrow *bool*

where

$x \leq (y :: \text{finite-1}) \longleftrightarrow \text{True}$

instance

$\langle \text{proof} \rangle$

end

instance *finite-1* :: {*dense-linorder*, *wellorder*}

$\langle \text{proof} \rangle$

instantiation *finite-1* :: *complete-lattice*

begin

definition [*simp*]: *Inf* = ($\lambda \cdot. a_1$)

definition [*simp*]: *Sup* = ($\lambda \cdot. a_1$)

definition [*simp*]: *bot* = a_1

definition [*simp*]: *top* = a_1

definition [*simp*]: *inf* = ($\lambda \cdot \cdot. a_1$)

definition [*simp*]: *sup* = ($\lambda \cdot \cdot. a_1$)

instance $\langle \text{proof} \rangle$

end

instance *finite-1* :: *complete-distrib-lattice*

$\langle \text{proof} \rangle$

instance *finite-1* :: *complete-linorder* $\langle \text{proof} \rangle$

lemma *finite-1-eq*: $x = a_1$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

instantiation *finite-1* :: *complete-boolean-algebra*

begin

definition [*simp*]: $(-)$ = ($\lambda \cdot \cdot. a_1$)

definition [*simp*]: *uminus* = ($\lambda \cdot. a_1$)

instance $\langle \text{proof} \rangle$

end

instantiation *finite-1* ::

{*linordered-ring-strict*, *linordered-comm-semiring-strict*, *ordered-comm-ring*,
ordered-cancel-comm-monoid-diff, *comm-monoid-mult*, *ordered-ring-abs*,

```

    one, modulo, sgn, inverse}
begin
  definition [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 ⟨proof⟩
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}
  ⟨proof⟩

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⟩

end

instantiation finite-2 :: linorder
begin

  definition less-finite-2 :: finite-2 ⇒ finite-2 ⇒ bool
  where

```


$$x < y \longleftrightarrow x = a_1 \wedge y = a_2$$

definition *less-eq-finite-2* :: *finite-2* \Rightarrow *finite-2* \Rightarrow *bool*

where

$$x \leq y \longleftrightarrow x = y \vee x < (y :: \text{finite-2})$$

instance

$\langle \text{proof} \rangle$

end

instance *finite-2* :: *wellorder*

$\langle \text{proof} \rangle$

instantiation *finite-2* :: *complete-lattice*

begin

definition $\sqcap A = (\text{if } a_1 \in A \text{ then } a_1 \text{ else } a_2)$

definition $\sqcup A = (\text{if } a_2 \in A \text{ then } a_2 \text{ else } a_1)$

definition $[simp]$: *bot* = *a*₁

definition $[simp]$: *top* = *a*₂

definition $x \sqcap y = (\text{if } x = a_1 \vee y = a_1 \text{ then } a_1 \text{ else } a_2)$

definition $x \sqcup y = (\text{if } x = a_2 \vee y = a_2 \text{ then } a_2 \text{ else } a_1)$

lemma *neq-finite-2-a₁-iff* $[simp]$: $x \neq a_1 \longleftrightarrow x = a_2$

$\langle \text{proof} \rangle$

lemma *neq-finite-2-a₁-iff'* $[simp]$: $a_1 \neq x \longleftrightarrow x = a_2$

$\langle \text{proof} \rangle$

lemma *neq-finite-2-a₂-iff* $[simp]$: $x \neq a_2 \longleftrightarrow x = a_1$

$\langle \text{proof} \rangle$

lemma *neq-finite-2-a₂-iff'* $[simp]$: $a_2 \neq x \longleftrightarrow x = a_1$

$\langle \text{proof} \rangle$

instance

$\langle \text{proof} \rangle$

end

instance *finite-2* :: *complete-linorder* $\langle \text{proof} \rangle$

instance *finite-2* :: *complete-distrib-lattice* $\langle \text{proof} \rangle$

instantiation *finite-2* :: {*field*, *idom-abs-sgn*, *idom-modulo*} **begin**

definition $[simp]$: *0* = *a*₁

definition $[simp]$: *1* = *a*₂

definition $x + y = (\text{case } (x, y) \text{ of } (a_1, a_1) \Rightarrow a_1 \mid (a_2, a_2) \Rightarrow a_1 \mid - \Rightarrow a_2)$

definition *uminus* = $(\lambda x :: \text{finite-2}. x)$

```

definition  $(-) = ((+) :: \text{finite-2} \Rightarrow -)$ 
definition  $x * y = (\text{case } (x, y) \text{ of } (a_2, a_2) \Rightarrow a_2 \mid - \Rightarrow a_1)$ 
definition  $\text{inverse} = (\lambda x :: \text{finite-2}. x)$ 
definition  $\text{divide} = ((* :: \text{finite-2} \Rightarrow -)$ 
definition  $x \bmod y = (\text{case } (x, y) \text{ of } (a_2, a_1) \Rightarrow a_2 \mid - \Rightarrow a_1)$ 
definition  $\text{abs} = (\lambda x :: \text{finite-2}. x)$ 
definition  $\text{sgn} = (\lambda x :: \text{finite-2}. x)$ 
instance
   $\langle \text{proof} \rangle$ 
end

lemma  $\text{two-finite-2}$   $[\text{simp}]$ :
   $2 = a_1$ 
   $\langle \text{proof} \rangle$ 

lemma  $\text{dvd-finite-2-unfold}$ :
   $x \text{ dvd } y \longleftrightarrow x = a_2 \vee y = a_1$ 
   $\langle \text{proof} \rangle$ 

instantiation  $\text{finite-2} :: \{\text{normalization-semidom}, \text{unique-euclidean-semiring}\}$  be-
gin
definition  $[\text{simp}]$ :  $\text{normalize} = (\text{id} :: \text{finite-2} \Rightarrow -)$ 
definition  $[\text{simp}]$ :  $\text{unit-factor} = (\text{id} :: \text{finite-2} \Rightarrow -)$ 
definition  $[\text{simp}]$ :  $\text{euclidean-size } x = (\text{case } x \text{ of } a_1 \Rightarrow 0 \mid a_2 \Rightarrow 1)$ 
definition  $[\text{simp}]$ :  $\text{division-segment } (x :: \text{finite-2}) = 1$ 
instance
   $\langle \text{proof} \rangle$ 
end

hide-const (open)  $a_1 \ a_2$ 

datatype (plugins only: code quickcheck extraction)  $\text{finite-3} =$ 
   $a_1 \mid a_2 \mid a_3$ 

notation (output)  $a_1 \ (\langle a_1 \rangle)$ 
notation (output)  $a_2 \ (\langle a_2 \rangle)$ 
notation (output)  $a_3 \ (\langle a_3 \rangle)$ 

lemma  $\text{UNIV-finite-3}$ :
   $\text{UNIV} = \{a_1, a_2, a_3\}$ 
   $\langle \text{proof} \rangle$ 

instantiation  $\text{finite-3} :: \text{enum}$ 
begin

definition
   $\text{enum} = [a_1, a_2, a_3]$ 

```

definition

$$\text{enum-all } P \longleftrightarrow P \ a_1 \wedge P \ a_2 \wedge P \ a_3$$
definition

$$\text{enum-ex } P \longleftrightarrow P \ a_1 \vee P \ a_2 \vee P \ a_3$$
instance $\langle \text{proof} \rangle$ **end****lemma** *finite-3-not-eq-unfold*:
$$x \neq a_1 \longleftrightarrow x \in \{a_2, a_3\}$$

$$x \neq a_2 \longleftrightarrow x \in \{a_1, a_3\}$$

$$x \neq a_3 \longleftrightarrow x \in \{a_1, a_2\}$$
 $\langle \text{proof} \rangle$ **instantiation** *finite-3 :: linorder***begin****definition** *less-finite-3 :: finite-3 \Rightarrow finite-3 \Rightarrow bool***where**

$$x < y = (\text{case } x \text{ of } a_1 \Rightarrow y \neq a_1 \mid a_2 \Rightarrow y = a_3 \mid a_3 \Rightarrow \text{False})$$
definition *less-eq-finite-3 :: finite-3 \Rightarrow finite-3 \Rightarrow bool***where**

$$x \leq y \longleftrightarrow x = y \vee x < (y :: \text{finite-3})$$
instance $\langle \text{proof} \rangle$ **end****instance** *finite-3 :: wellorder* $\langle \text{proof} \rangle$ **class** *finite-lattice = finite + lattice + Inf + Sup + bot + top +***assumes** *Inf-finite-empty*: $\text{Inf } \{\} = \text{Sup } \text{UNIV}$ **assumes** *Inf-finite-insert*: $\text{Inf } (\text{insert } a \ A) = a \sqcap \text{Inf } A$ **assumes** *Sup-finite-empty*: $\text{Sup } \{\} = \text{Inf } \text{UNIV}$ **assumes** *Sup-finite-insert*: $\text{Sup } (\text{insert } a \ A) = a \sqcup \text{Sup } A$ **assumes** *bot-finite-def*: $\text{bot} = \text{Inf } \text{UNIV}$ **assumes** *top-finite-def*: $\text{top} = \text{Sup } \text{UNIV}$ **begin****subclass** *complete-lattice* $\langle \text{proof} \rangle$ **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⟩

lemma *finite-Inf-Sup*: $\sqcap (\text{Sup } A) \leq \sqcup (\text{Inf } \{f \mid f \in A. f Y \in Y\})$
 ⟨proof⟩

subclass *complete-distrib-lattice*
 ⟨proof⟩
end

instantiation *finite-3* :: *finite-lattice*
begin

definition $\sqcap A = (\text{if } a_1 \in A \text{ then } a_1 \text{ else if } a_2 \in A \text{ then } a_2 \text{ else } a_3)$
definition $\sqcup A = (\text{if } a_3 \in A \text{ then } a_3 \text{ else if } a_2 \in A \text{ then } a_2 \text{ else } a_1)$
definition [simp]: $\text{bot} = a_1$
definition [simp]: $\text{top} = a_3$
definition [simp]: $\text{inf} = (\text{min} :: \text{finite-3} \Rightarrow -)$
definition [simp]: $\text{sup} = (\text{max} :: \text{finite-3} \Rightarrow -)$

instance
 ⟨proof⟩
end

instance *finite-3* :: *complete-lattice* ⟨proof⟩

instance *finite-3* :: *finite-distrib-lattice*
 ⟨proof⟩

instance *finite-3* :: *complete-distrib-lattice* ⟨proof⟩

instance *finite-3* :: *complete-linorder* ⟨proof⟩

instantiation *finite-3* :: {*field*, *idom-abs-sgn*, *idom-modulo*} **begin**

definition [simp]: $0 = a_1$

definition [simp]: $1 = a_2$

definition

$x + y = (\text{case } (x, y) \text{ of}$
 $(a_1, a_1) \Rightarrow a_1 \mid (a_2, a_3) \Rightarrow a_1 \mid (a_3, a_2) \Rightarrow a_1$
 $\mid (a_1, a_2) \Rightarrow a_2 \mid (a_2, a_1) \Rightarrow a_2 \mid (a_3, a_3) \Rightarrow a_2$
 $\mid - \Rightarrow a_3)$

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 $x - y = x + (- y :: \text{finite-3})$

definition $x * y = (\text{case } (x, y) \text{ of } (a_2, a_2) \Rightarrow a_2 \mid (a_3, a_3) \Rightarrow a_2 \mid (a_2, a_3) \Rightarrow a_3$
 $\mid (a_3, a_2) \Rightarrow a_3 \mid - \Rightarrow a_1)$

definition $\text{inverse} = (\lambda x :: \text{finite-3}. x)$

definition $x \text{ div } y = x * \text{inverse } (y :: \text{finite-3})$

definition $x \text{ mod } y = (\text{case } y \text{ of } a_1 \Rightarrow x \mid - \Rightarrow a_1)$

definition $\text{abs} = (\lambda x. \text{case } x \text{ of } a_3 \Rightarrow a_2 \mid - \Rightarrow x)$

definition $sgn = (\lambda x :: \text{finite-3}. x)$

instance

$\langle \text{proof} \rangle$

end

lemma $\text{two-finite-3} \text{ [simp]}:$

$2 = a_3$

$\langle \text{proof} \rangle$

lemma $\text{dvd-finite-3-unfold}:$

$x \text{ dvd } y \longleftrightarrow x = a_2 \vee x = a_3 \vee y = a_1$

$\langle \text{proof} \rangle$

instantiation $\text{finite-3} :: \{\text{normalization-semidom}, \text{unique-euclidean-semiring}\}$ **begin**

definition $\text{[simp]}: \text{normalize } x = (\text{case } x \text{ of } a_3 \Rightarrow a_2 \mid - \Rightarrow x)$

definition $\text{[simp]}: \text{unit-factor} = (\text{id} :: \text{finite-3} \Rightarrow -)$

definition $\text{[simp]}: \text{euclidean-size } x = (\text{case } x \text{ of } a_1 \Rightarrow 0 \mid - \Rightarrow 1)$

definition $\text{[simp]}: \text{division-segment } (x :: \text{finite-3}) = 1$

instance

$\langle \text{proof} \rangle$

end

hide-const (**open**) $a_1 \ a_2 \ a_3$

datatype (*plugins only: code quickcheck extraction*) $\text{finite-4} =$

$a_1 \mid a_2 \mid a_3 \mid a_4$

notation (**output**) $a_1 \ (\langle a_1 \rangle)$

notation (**output**) $a_2 \ (\langle a_2 \rangle)$

notation (**output**) $a_3 \ (\langle a_3 \rangle)$

notation (**output**) $a_4 \ (\langle a_4 \rangle)$

lemma $\text{UNIV-finite-4}:$

$\text{UNIV} = \{a_1, a_2, a_3, a_4\}$

$\langle \text{proof} \rangle$

instantiation $\text{finite-4} :: \text{enum}$

begin

definition

$\text{enum} = [a_1, a_2, a_3, a_4]$

definition

$\text{enum-all } P \longleftrightarrow P \ a_1 \wedge P \ a_2 \wedge P \ a_3 \wedge P \ a_4$

definition

$\text{enum-ex } P \longleftrightarrow P \ a_1 \vee P \ a_2 \vee P \ a_3 \vee P \ a_4$

instance $\langle proof \rangle$

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

$$\begin{aligned} x < y &\longleftrightarrow (\text{case } (x, y) \text{ of} \\ &\quad (a_1, a_1) \Rightarrow \text{False} \mid (a_1, -) \Rightarrow \text{True} \\ &\quad \mid (a_2, a_4) \Rightarrow \text{True} \\ &\quad \mid (a_3, a_4) \Rightarrow \text{True} \mid - \Rightarrow \text{False}) \end{aligned}$$

definition

$$\begin{aligned} x \leq y &\longleftrightarrow (\text{case } (x, y) \text{ of} \\ &\quad (a_1, -) \Rightarrow \text{True} \\ &\quad \mid (a_2, a_2) \Rightarrow \text{True} \mid (a_2, a_4) \Rightarrow \text{True} \\ &\quad \mid (a_3, a_3) \Rightarrow \text{True} \mid (a_3, a_4) \Rightarrow \text{True} \\ &\quad \mid (a_4, a_4) \Rightarrow \text{True} \mid - \Rightarrow \text{False}) \end{aligned}$$

definition

$\sqcap A = (\text{if } a_1 \in A \vee a_2 \in A \wedge a_3 \in A \text{ then } a_1 \text{ else if } a_2 \in A \text{ then } a_2 \text{ else if } a_3 \in A \text{ then } a_3 \text{ else } a_4)$

definition

$\sqcup A = (\text{if } a_4 \in A \vee a_2 \in A \wedge a_3 \in A \text{ then } a_4 \text{ else if } a_2 \in A \text{ then } a_2 \text{ else if } a_3 \in A \text{ then } a_3 \text{ else } a_1)$

definition [*simp*]: $\text{bot} = a_1$

definition [*simp*]: $\text{top} = a_4$

definition

$$\begin{aligned} x \sqcap y &= (\text{case } (x, y) \text{ of} \\ &\quad (a_1, -) \Rightarrow a_1 \mid (-, a_1) \Rightarrow a_1 \mid (a_2, a_3) \Rightarrow a_1 \mid (a_3, a_2) \Rightarrow a_1 \\ &\quad \mid (a_2, -) \Rightarrow a_2 \mid (-, a_2) \Rightarrow a_2 \\ &\quad \mid (a_3, -) \Rightarrow a_3 \mid (-, a_3) \Rightarrow a_3 \\ &\quad \mid - \Rightarrow a_4) \end{aligned}$$

definition

$$\begin{aligned} x \sqcup y &= (\text{case } (x, y) \text{ of} \\ &\quad (a_4, -) \Rightarrow a_4 \mid (-, a_4) \Rightarrow a_4 \mid (a_2, a_3) \Rightarrow a_4 \mid (a_3, a_2) \Rightarrow a_4 \\ &\quad \mid (a_2, -) \Rightarrow a_2 \mid (-, a_2) \Rightarrow a_2 \\ &\quad \mid (a_3, -) \Rightarrow a_3 \mid (-, a_3) \Rightarrow a_3 \\ &\quad \mid - \Rightarrow a_1) \end{aligned}$$

instance

$\langle proof \rangle$

end

instance *finite-4* :: *complete-lattice* $\langle proof \rangle$

instance *finite-4* :: *complete-distrib-lattice* $\langle proof \rangle$

instantiation *finite-4* :: *complete-boolean-algebra* **begin**
definition $- x = (\text{case } x \text{ of } a_1 \Rightarrow a_4 \mid a_2 \Rightarrow a_3 \mid a_3 \Rightarrow a_2 \mid a_4 \Rightarrow a_1)$
definition $x - y = x \sqcap - (y :: \text{finite-4})$
instance
 $\langle \text{proof} \rangle$
end

hide-const (**open**) $a_1 \ a_2 \ a_3 \ a_4$

datatype (*plugins only: code quickcheck extraction*) *finite-5* =
 $a_1 \mid a_2 \mid a_3 \mid a_4 \mid a_5$

notation (**output**) $a_1 \ (\langle a_1 \rangle)$
notation (**output**) $a_2 \ (\langle a_2 \rangle)$
notation (**output**) $a_3 \ (\langle a_3 \rangle)$
notation (**output**) $a_4 \ (\langle a_4 \rangle)$
notation (**output**) $a_5 \ (\langle a_5 \rangle)$

lemma *UNIV-finite-5*:
 $UNIV = \{a_1, a_2, a_3, a_4, a_5\}$
 $\langle \text{proof} \rangle$

instantiation *finite-5* :: *enum*
begin

definition
 $enum = [a_1, a_2, a_3, a_4, a_5]$

definition
 $enum\text{-}all \ P \longleftrightarrow P \ a_1 \wedge P \ a_2 \wedge P \ a_3 \wedge P \ a_4 \wedge P \ a_5$

definition
 $enum\text{-}ex \ P \longleftrightarrow P \ a_1 \vee P \ a_2 \vee P \ a_3 \vee P \ a_4 \vee P \ a_5$

instance $\langle \text{proof} \rangle$

end

instantiation *finite-5* :: *finite-lattice*
begin

The non-distributive pentagon lattice N_5

definition
 $x < y \longleftrightarrow (\text{case } (x, y) \text{ of}$
 $(a_1, a_1) \Rightarrow \text{False} \mid (a_1, -) \Rightarrow \text{True}$
 $\mid (a_2, a_3) \Rightarrow \text{True} \mid (a_2, a_5) \Rightarrow \text{True}$
 $\mid (a_3, a_5) \Rightarrow \text{True}$
 $\mid (a_4, a_5) \Rightarrow \text{True} \mid - \Rightarrow \text{False})$

definition

$$\begin{aligned}
x \leq y &\longleftrightarrow (\text{case } (x, y) \text{ of} \\
&\quad (a_1, -) \Rightarrow \text{True} \\
&\quad | (a_2, a_2) \Rightarrow \text{True} \mid (a_2, a_3) \Rightarrow \text{True} \mid (a_2, a_5) \Rightarrow \text{True} \\
&\quad | (a_3, a_3) \Rightarrow \text{True} \mid (a_3, a_5) \Rightarrow \text{True} \\
&\quad | (a_4, a_4) \Rightarrow \text{True} \mid (a_4, a_5) \Rightarrow \text{True} \\
&\quad | (a_5, a_5) \Rightarrow \text{True} \mid - \Rightarrow \text{False})
\end{aligned}$$
definition

$$\begin{aligned}
\sqcap A &= \\
&(\text{if } a_1 \in A \vee a_4 \in A \wedge (a_2 \in A \vee a_3 \in A) \text{ then } a_1 \\
&\quad \text{else if } a_2 \in A \text{ then } a_2 \\
&\quad \text{else if } a_3 \in A \text{ then } a_3 \\
&\quad \text{else if } a_4 \in A \text{ then } a_4 \\
&\quad \text{else } a_5)
\end{aligned}$$
definition

$$\begin{aligned}
\sqcup A &= \\
&(\text{if } a_5 \in A \vee a_4 \in A \wedge (a_2 \in A \vee a_3 \in A) \text{ then } a_5 \\
&\quad \text{else if } a_3 \in A \text{ then } a_3 \\
&\quad \text{else if } a_2 \in A \text{ then } a_2 \\
&\quad \text{else if } a_4 \in A \text{ then } a_4 \\
&\quad \text{else } a_1)
\end{aligned}$$
definition [simp]: bot = a_1 **definition** [simp]: top = a_5 **definition**

$$\begin{aligned}
x \sqcap y &= (\text{case } (x, y) \text{ of} \\
&\quad (a_1, -) \Rightarrow a_1 \mid (-, a_1) \Rightarrow a_1 \mid (a_2, a_4) \Rightarrow a_1 \mid (a_4, a_2) \Rightarrow a_1 \mid (a_3, a_4) \Rightarrow a_1 \mid \\
&\quad (a_4, a_3) \Rightarrow a_1 \\
&\quad | (a_2, -) \Rightarrow a_2 \mid (-, a_2) \Rightarrow a_2 \\
&\quad | (a_3, -) \Rightarrow a_3 \mid (-, a_3) \Rightarrow a_3 \\
&\quad | (a_4, -) \Rightarrow a_4 \mid (-, a_4) \Rightarrow a_4 \\
&\quad | - \Rightarrow a_5)
\end{aligned}$$
definition

$$\begin{aligned}
x \sqcup y &= (\text{case } (x, y) \text{ of} \\
&\quad (a_5, -) \Rightarrow a_5 \mid (-, a_5) \Rightarrow a_5 \mid (a_2, a_4) \Rightarrow a_5 \mid (a_4, a_2) \Rightarrow a_5 \mid (a_3, a_4) \Rightarrow a_5 \mid \\
&\quad (a_4, a_3) \Rightarrow a_5 \\
&\quad | (a_3, -) \Rightarrow a_3 \mid (-, a_3) \Rightarrow a_3 \\
&\quad | (a_2, -) \Rightarrow a_2 \mid (-, a_2) \Rightarrow a_2 \\
&\quad | (a_4, -) \Rightarrow a_4 \mid (-, a_4) \Rightarrow a_4 \\
&\quad | - \Rightarrow a_1)
\end{aligned}$$
instance $\langle \text{proof} \rangle$ **end****instance** finite-5 :: complete-lattice $\langle \text{proof} \rangle$


```
hide-const (open) a1 a2 a3 a4 a5
```

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 ntranc1
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)

context comm-semiring-1
begin

definition of-char :: ⟨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]⟩
  ⟨proof⟩

end

lemma (in comm-semiring-1) of-nat-of-char:
```

$\langle \text{of-nat } (\text{of-char } c) = \text{of-char } c \rangle$
 $\langle \text{proof} \rangle$

lemma (in *comm-ring-1*) *of-int-of-char*:
 $\langle \text{of-int } (\text{of-char } c) = \text{of-char } c \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-of-char [simp]*:
 $\langle \text{nat } (\text{of-char } c) = \text{of-char } c \rangle$
 $\langle \text{proof} \rangle$

context *linordered-euclidean-semiring-bit-operations*
begin

definition *char-of* :: $\langle 'a \Rightarrow \text{char} \rangle$
where $\langle \text{char-of } n = \text{Char } (\text{bit } n \ 0) (\text{bit } n \ 1) (\text{bit } n \ 2) (\text{bit } n \ 3) (\text{bit } n \ 4) (\text{bit } n \ 5) (\text{bit } n \ 6) (\text{bit } n \ 7) \rangle$

lemma *char-of-take-bit-eq*:
 $\langle \text{char-of } (\text{take-bit } n \ m) = \text{char-of } m \rangle$ **if** $\langle n \geq 8 \rangle$
 $\langle \text{proof} \rangle$

lemma *char-of-char [simp]*:
 $\langle \text{char-of } (\text{of-char } c) = c \rangle$
 $\langle \text{proof} \rangle$

lemma *char-of-comp-of-char [simp]*:
 $\text{char-of} \circ \text{of-char} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *inj-of-char*:
 $\langle \text{inj of-char} \rangle$
 $\langle \text{proof} \rangle$

lemma *of-char-eqI*:
 $\langle c = d \rangle$ **if** $\langle \text{of-char } c = \text{of-char } d \rangle$
 $\langle \text{proof} \rangle$

lemma *of-char-eq-iff [simp]*:
 $\langle \text{of-char } c = \text{of-char } d \iff c = d \rangle$
 $\langle \text{proof} \rangle$

lemma *of-char-of [simp]*:
 $\langle \text{of-char } (\text{char-of } a) = a \text{ mod } 256 \rangle$
 $\langle \text{proof} \rangle$

lemma *char-of-mod-256 [simp]*:
 $\langle \text{char-of } (n \text{ mod } 256) = \text{char-of } n \rangle$

⟨proof⟩

lemma *of-char-mod-256* [simp]:

⟨of-char $c \bmod 256 = \text{of-char } c$ ⟩

⟨proof⟩

lemma *char-of-quasi-inj* [simp]:

⟨char-of $m = \text{char-of } n \longleftrightarrow m \bmod 256 = n \bmod 256$ ⟩ (is ⟨ $?P \longleftrightarrow ?Q$ ⟩)

⟨proof⟩

lemma *char-of-eq-iff*:

⟨char-of $n = c \longleftrightarrow \text{take-bit } 8 \ n = \text{of-char } c$ ⟩

⟨proof⟩

lemma *char-of-nat* [simp]:

⟨char-of (of-nat n) = char-of n ⟩

⟨proof⟩

end

lemma *inj-on-char-of-nat* [simp]:

inj-on char-of {0::nat.. 256 }

⟨proof⟩

lemma *nat-of-char-less-256* [simp]:

of-char $c < (256 :: \text{nat})$

⟨proof⟩

lemma *range-nat-of-char*:

range of-char = {0::nat.. 256 }

⟨proof⟩

lemma *UNIV-char-of-nat*:

UNIV = char-of ‘ {0::nat.. 256 }’

⟨proof⟩

lemma *card-UNIV-char*:

card (UNIV :: char set) = 256

⟨proof⟩

context

includes *lifting-syntax* **and** *integer.lifting* **and** *natural.lifting*

begin

lemma [transfer-rule]:

⟨(pcr-integer ==> (=)) char-of char-of⟩

⟨proof⟩

lemma [transfer-rule]:

⟨((=) == => pcr-integer) of-char of-char⟩
 ⟨proof⟩

lemma [transfer-rule]:
 ⟨(pcr-natural == => (=)) char-of char-of⟩
 ⟨proof⟩

lemma [transfer-rule]:
 ⟨((=) == => pcr-natural) of-char of-char⟩
 ⟨proof⟩

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
 ⟨proof⟩

lemma size'-char-eq-0 [simp, code]:
 ⟨size-char c = 0⟩
 ⟨proof⟩

syntax
 -Char :: str-position ⇒ char (⟨(⟨open-block notation=⟨literal char⟩⟩ CHR -)⟩)
 -Char-ord :: num-const ⇒ char (⟨(⟨open-block notation=⟨literal char code⟩⟩ CHR
 -)⟩)
syntax-consts
 -Char -Char-ord ⇒ Char

type-synonym string = char list

syntax
 -String :: str-position ⇒ string (⟨(⟨open-block notation=⟨literal string⟩⟩-)⟩)

⟨ML⟩

instantiation char :: enum
begin

definition
 Enum.enum = [
 CHR 0x00, CHR 0x01, CHR 0x02, CHR 0x03,
 CHR 0x04, CHR 0x05, CHR 0x06, CHR 0x07,
 CHR 0x08, CHR 0x09, CHR "⌞", CHR 0x0B,

CHR 0x0C, CHR 0x0D, CHR 0x0E, CHR 0x0F,
 CHR 0x10, CHR 0x11, CHR 0x12, CHR 0x13,
 CHR 0x14, CHR 0x15, CHR 0x16, CHR 0x17,
 CHR 0x18, CHR 0x19, CHR 0x1A, CHR 0x1B,
 CHR 0x1C, CHR 0x1D, CHR 0x1E, CHR 0x1F,
 CHR " ", CHR "!", CHR 0x22, CHR "#",
 CHR "\$", CHR "%", CHR "&", CHR 0x27,
 CHR "(", CHR ")", CHR "*", CHR "+",
 CHR ",", CHR "-", CHR ".", CHR "/",
 CHR "0", CHR "1", CHR "2", CHR "3",
 CHR "4", CHR "5", CHR "6", CHR "7",
 CHR "8", CHR "9", CHR ":", CHR ";",
 CHR "<", CHR "=", CHR ">", CHR "?",
 CHR "@", CHR "A", CHR "B", CHR "C",
 CHR "D", CHR "E", CHR "F", CHR "G",
 CHR "H", CHR "I", CHR "J", CHR "K",
 CHR "L", CHR "M", CHR "N", CHR "O",
 CHR "P", CHR "Q", CHR "R", CHR "S",
 CHR "T", CHR "U", CHR "V", CHR "W",
 CHR "X", CHR "Y", CHR "Z", CHR "[",
 CHR 0x5C, CHR "\", CHR "^", CHR "_",
 CHR 0x60, CHR "a", CHR "b", CHR "c",
 CHR "d", CHR "e", CHR "f", CHR "g",
 CHR "h", CHR "i", CHR "j", CHR "k",
 CHR "l", CHR "m", CHR "n", CHR "o",
 CHR "p", CHR "q", CHR "r", CHR "s",
 CHR "t", CHR "u", CHR "v", CHR "w",
 CHR "x", CHR "y", CHR "z", CHR "{",
 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 $P \longleftrightarrow \text{list-all } P \text{ (Enum.enum :: char list)}$

definition

Enum.enum-ex $P \longleftrightarrow \text{list-ex } P \text{ (Enum.enum :: char list)}$

lemma *enum-char-unfold*:

Enum.enum = *map char-of* [0..*256*]
 ⟨*proof*⟩

instance ⟨*proof*⟩**end****lemma** *linorder-char*:

class.linorder ($\lambda c \text{ d. of-char } c \leq (\text{of-char } d :: \text{nat})) (\lambda c \text{ d. of-char } c < (\text{of-char } d :: \text{nat}))$
 ⟨*proof*⟩

Optimized version for execution

definition *char-of-integer* :: *integer* \Rightarrow *char*

where [*code-abbrev*]: *char-of-integer* = *char-of*

definition *integer-of-char* :: *char* \Rightarrow *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)
 <proof>

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*
 <proof>

73.2 Strings as dedicated type for target language code generation

73.2.1 Logical specification

context
begin

qualified definition *ascii-of* :: *char* \Rightarrow *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
 <proof> **lemma** *digit0-ascii-of-iff* [simp]:
digit0 (String.ascii-of c) \longleftrightarrow digit0 c
 <proof> **lemma** *digit1-ascii-of-iff* [simp]:
digit1 (String.ascii-of c) \longleftrightarrow digit1 c
 <proof> **lemma** *digit2-ascii-of-iff* [simp]:
digit2 (String.ascii-of c) \longleftrightarrow digit2 c
 <proof> **lemma** *digit3-ascii-of-iff* [simp]:
digit3 (String.ascii-of c) \longleftrightarrow digit3 c
 <proof> **lemma** *digit4-ascii-of-iff* [simp]:
digit4 (String.ascii-of c) \longleftrightarrow digit4 c
 <proof> **lemma** *digit5-ascii-of-iff* [simp]:
digit5 (String.ascii-of c) \longleftrightarrow digit5 c
 <proof> **lemma** *digit6-ascii-of-iff* [simp]:
digit6 (String.ascii-of c) \longleftrightarrow digit6 c
 <proof> **lemma** *not-digit7-ascii-of* [simp]:
 \neg *digit7 (ascii-of c)*
 <proof> **lemma** *ascii-of-idem*:
ascii-of c = c if \neg digit7 c
 <proof> **typedef** *literal* = {*cs*. $\forall c \in \text{set } cs. \neg \text{digit7 } c$ }
morphisms *explode Abs-literal*
 <proof> **setup-lifting** *type-definition-literal*

qualified lift-definition *implode* :: *string* \Rightarrow *literal*
is *map ascii-of*
 <proof> **lemma** *implode-explode-eq* [simp]:

```

  String.implode (String.explode s) = s
  ⟨proof⟩ lemma explode-implode-eq [simp]:
    String.explode (String.implode cs) = map ascii-of cs
  ⟨proof⟩

```

end

context *linordered-euclidean-semiring-bit-operations*
begin

context
begin

```

qualified lemma char-of-ascii-of [simp]:
  ⟨of-char (String.ascii-of c) = take-bit 7 (of-char c)⟩
  ⟨proof⟩ lemma ascii-of-char-of:
  ⟨String.ascii-of (char-of a) = char-of (take-bit 7 a)⟩
  ⟨proof⟩

```

end

end

73.2.2 Syntactic representation

Logical ground representations for literals are:

1. \emptyset for the empty literal;
2. *Literal* $b0 \dots b6\ s$ 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*.

```

instantiation String.literal :: zero
begin

```

context
begin

```

qualified lift-definition zero-literal :: String.literal
  is Nil
  ⟨proof⟩

```

```

instance ⟨proof⟩

```


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 $\lambda b0\ b1\ b2\ b3\ b4\ b5\ b6\ cs.$ *Char* *b0 b1 b2 b3 b4 b5 b6 False # cs*
 $\langle proof \rangle$ **lemma** *Literal-eq-iff* [*simp*]:
 Literal *b0 b1 b2 b3 b4 b5 b6 s* = *Literal* *c0 c1 c2 c3 c4 c5 c6 t*
 $\longleftrightarrow (b0 \longleftrightarrow c0) \wedge (b1 \longleftrightarrow c1) \wedge (b2 \longleftrightarrow c2) \wedge (b3 \longleftrightarrow c3)$
 $\wedge (b4 \longleftrightarrow c4) \wedge (b5 \longleftrightarrow c5) \wedge (b6 \longleftrightarrow c6) \wedge s = t$
 $\langle proof \rangle$ **lemma** *empty-neq-Literal* [*simp*]:
 empty-literal \neq *Literal* *b0 b1 b2 b3 b4 b5 b6 s*
 $\langle proof \rangle$ **lemma** *Literal-neq-empty* [*simp*]:
 Literal *b0 b1 b2 b3 b4 b5 b6 s* \neq *empty-literal*
 $\langle proof \rangle$

end

code-datatype 0 :: *String.literal* *String.Literal*

syntax
 -*Literal* :: *str-position* \Rightarrow *String.literal*
 ($\langle \langle open\text{-}block\ notation = \langle literal\ string \rangle \rangle STR\ - \rangle$)
 -*Ascii* :: *num-const* \Rightarrow *String.literal*
 ($\langle \langle open\text{-}block\ notation = \langle literal\ char\ code \rangle \rangle STR\ - \rangle$)
syntax-consts
 -*Literal* -*Ascii* \Rightarrow *String.Literal*

$\langle ML \rangle$

73.2.3 Operations

instantiation *String.literal* :: *plus*
begin

context
begin

qualified lift-definition *plus-literal* :: *String.literal* \Rightarrow *String.literal* \Rightarrow *String.literal*
 is (@)
 $\langle proof \rangle$

```

instance  $\langle proof \rangle$ 

end

end

instance String.literal :: monoid-add
   $\langle proof \rangle$ 

lemma add-Literal-assoc:
   $\langle String.Literal\ b0\ b1\ b2\ b3\ b4\ b5\ b6\ t + s = String.Literal\ b0\ b1\ b2\ b3\ b4\ b5\ b6$ 
   $(t + s) \rangle$ 
   $\langle proof \rangle$ 

instantiation String.literal :: size
begin

context
  includes literal.lifting
begin

lift-definition size-literal :: String.literal  $\Rightarrow$  nat
  is length  $\langle proof \rangle$ 

end

instance  $\langle proof \rangle$ 

end

instantiation String.literal :: equal
begin

context
begin

qualified lift-definition equal-literal :: String.literal  $\Rightarrow$  String.literal  $\Rightarrow$  bool
  is HOL.equal  $\langle proof \rangle$ 

instance
   $\langle proof \rangle$ 

end

end

instantiation String.literal :: linorder
begin

```

context

begin

qualified lift-definition *less-eq-literal* :: *String.literal* \Rightarrow *String.literal* \Rightarrow *bool*
is *ord.lexordp-eq* ($\lambda c d. \text{of-char } c < (\text{of-char } d :: \text{nat})$)
 $\langle \text{proof} \rangle$ **lift-definition** *less-literal* :: *String.literal* \Rightarrow *String.literal* \Rightarrow *bool*
is *ord.lexordp* ($\lambda c d. \text{of-char } c < (\text{of-char } d :: \text{nat})$)
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

end

lemma *infinite-literal*:

infinite (*UNIV* :: *String.literal set*)
 $\langle \text{proof} \rangle$

lemma *add-literal-code* [*code*]:

$\langle \text{STR } "" + s = s \rangle$
 $\langle s + \text{STR } "" = s \rangle$
 $\langle \text{String.Literal } b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ t + s = \text{String.Literal } b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6$
 $(t + s) \rangle$
 $\langle \text{proof} \rangle$

73.2.4 Executable conversions

context

begin

qualified lift-definition *asciis-of-literal* :: *String.literal* \Rightarrow *integer list*
is *map of-char*

$\langle \text{proof} \rangle$ **lemma** *asciis-of-zero* [*simp*, *code*]:

asciis-of-literal 0 = []

$\langle \text{proof} \rangle$ **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*

$\langle \text{proof} \rangle$ **lift-definition** *literal-of-asciis* :: *integer list* \Rightarrow *String.literal*

is *map* (*String.ascii-of* \circ *char-of*)

$\langle \text{proof} \rangle$ **lemma** *literal-of-asciis-Nil* [*simp*, *code*]:

literal-of-asciis [] = 0

$\langle \text{proof} \rangle$ **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* \Rightarrow *String.Literal* *b0 b1 b2 b3 b4 b5 b6*
(literal-of-asciis ks))

$\langle \text{proof} \rangle$ **lemma** *literal-of-asciis-of-literal* [*simp*]:

literal-of-asciis (*asciis-of-literal s*) = *s*

```

⟨proof⟩ lemma explode-code [code]:
  String.explode s = map char-of (asciis-of-literal s)
⟨proof⟩ lemma implode-code [code]:
  String.implode cs = literal-of-asciis (map of-char cs)
⟨proof⟩ 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)
     $\longleftrightarrow$  (b0  $\longleftrightarrow$  a0)  $\wedge$  (b1  $\longleftrightarrow$  a1)  $\wedge$  (b2  $\longleftrightarrow$  a2)  $\wedge$  (b3  $\longleftrightarrow$  a3)
     $\wedge$  (b4  $\longleftrightarrow$  a4)  $\wedge$  (b5  $\longleftrightarrow$  a5)  $\wedge$  (b6  $\longleftrightarrow$  a6)  $\wedge$  (s = r)
⟨proof⟩

```

end

73.2.5 Technical code generation setup

Alternative constructor for generated computations

context

begin

qualified definition *Literal'* :: *bool* \Rightarrow *bool* \Rightarrow *bool* \Rightarrow *bool* \Rightarrow *bool* \Rightarrow *bool* \Rightarrow *bool* \Rightarrow *bool*
 \Rightarrow *String.literal* \Rightarrow *String.literal*
where [*simp*]: *Literal'* = *String.Literal*

```

lemma [code]:
  ⟨Literal' b0 b1 b2 b3 b4 b5 b6 s = String.literal-of-asciis
    [foldr ( $\lambda b k.$  of-bool b + k * 2) [b0, b1, b2, b3, b4, b5, b6] 0] + s⟩
⟨proof⟩

```

```

lemma [code-computation-unfold]:
  String.Literal = Literal'
⟨proof⟩

```

end

code-reserved

```

(SML) string String Char Str-Literal
and (OCaml) string String Char Str-Literal
and (Haskell) Str-Literal
and (Scala) String Str-Literal

```

code-identifier

```

code-module String  $\rightarrow$ 
  (SML) Str and (OCaml) Str and (Haskell) Str and (Scala) Str

```

code-printing

```

type-constructor String.literal  $\rightarrow$ 
  (SML) string
and (OCaml) string
and (Haskell) String

```

```

    and (Scala) String
| constant STR "" ↪
  (SML)
    and (OCaml)
    and (Haskell)
    and (Scala)

```

⟨ML⟩

code-printing

```

code-module Str-Literal ↪
  (SML) ⟨structure Str-Literal : sig
    type int = IntInf.int
    val literal-of-ascii : int list -> string
    val ascii-of-literal : string -> int list
  end = struct

    open IntInf;

    fun map f [] = []
    | map f (x :: xs) = f x :: map f xs; (* deliberate clone not relying on List.- structure *)

    fun check-ascii k =
      if 0 <= k andalso k < 128
      then k
      else raise Fail Non-ASCII character in literal;

    val char-of-ascii = Char.chr o toInt o (fn k => k mod 128);

    val ascii-of-char = check-ascii o fromInt o Char.ord;

    val literal-of-ascii = String.implode o map char-of-ascii;

    val ascii-of-literal = map ascii-of-char o String.explode;

  end;⟩ for constant String.literal-of-ascii String.ascii-of-literal
    and (OCaml) ⟨module Str-Literal : sig
      val literal-of-ascii : Z.t list -> string
      val ascii-of-literal : string -> Z.t list
    end = struct

      (* deliberate clones not relying on List.- module *)

      let rec length xs = match xs with
        [] -> 0
      | x :: xs -> 1 + length xs;;

      let rec nth xs n = match xs with

```

```

(x :: xs) -> if n <= 0 then x else nth xs (n - 1);;

let rec map-range f n =
  if n <= 0
  then []
  else
    let m = n - 1
    in map-range f m @ [f m];;

let implode f xs =
  String.init (length xs) (fun n -> f (nth xs n));;

let explode f s =
  map-range (fun n -> f (String.get s n)) (String.length s);;

let z-128 = Z.of-int 128;;

let check-ascii k =
  if 0 <= k && k < 128
  then k
  else failwith Non-ASCII character in literal;;

let char-of-ascii k = Char.chr (Z.to-int (Z.rem k z-128));;

let ascii-of-char c = Z.of-int (check-ascii (Char.code c));;

let literal-of-asciis ks = implode char-of-ascii ks;;

let asciis-of-literal s = explode ascii-of-char s;;

end;;> for constant String.literal-of-asciis String.asciis-of-literal
and (Haskell) <module Str-Literal(literalOfAsciis, asciisOfLiteral) where

check-ascii :: Int -> Int
check-ascii k
| (0 <= k && k < 128) = k
| otherwise = error Non-ASCII character in literal

charOfAscii :: Integer -> Char
charOfAscii = toEnum . Prelude.fromInteger . (\k -> k `mod` 128)

asciiOfChar :: Char -> Integer
asciiOfChar = toInteger . check-ascii . fromEnum

literalOfAsciis :: [Integer] -> [Char]
literalOfAsciis = map charOfAscii

asciisOfLiteral :: [Char] -> [Integer]
asciisOfLiteral = map asciiOfChar

```

```

› for constant String.literal-of-ascii String.ascii-of-literal
  and (Scala) ›object Str-Literal {

private def checkAscii(k : Int) : Int =
  0 <= k && k < 128 match {
    case true => k
    case false => sys.error(Non-ASCII character in literal)
  }

private def charOfAscii(k : BigInt) : Char =
  (k % 128).charValue

private def asciiOfChar(c : Char) : BigInt =
  BigInt(checkAscii(c.toInt))

def literalOfAscii(ks : List[BigInt]) : String =
  ks.map(charOfAscii).mkString

def asciiOfLiteral(s : String) : List[BigInt] =
  s.toList.map(asciiOfChar)

}
› for constant String.literal-of-ascii String.ascii-of-literal
| constant ›(+) :: String.literal ⇒ String.literal ⇒ String.literal › ↪
  (SML) infixl 18 ^
  and (OCaml) 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.literal'-of'-ascii
  and (Haskell) Str'-Literal.literalOfAscii
  and (Scala) Str'-Literal.literalOfAscii
| constant String.ascii-of-literal › ↪
  (SML) Str'-Literal.ascii'-of'-literal
  and (OCaml) Str'-Literal.ascii'-of'-literal
  and (Haskell) Str'-Literal.asciiOfLiteral
  and (Scala) Str'-Literal.asciiOfLiteral
| 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

⟨ML⟩

```

definition abort :: String.literal ⇒ (unit ⇒ 'a) ⇒ 'a
  where [simp]: abort - f = f ()

```

```

declare [[code drop: Code.abort]]

```

lemma abort-cong:

```

  msg = msg' ⇒ Code.abort msg f = Code.abort msg' f
  ⟨proof⟩

```

⟨ML⟩

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  $\Rightarrow$  typerep
begin

definition typerep-of :: 'a  $\Rightarrow$  typerep where
  [simp]: typerep-of x = typerep TYPE('a)

end

syntax
  -TYPEREP :: type  $\Rightarrow$  logic ( $\langle$ ( $\langle$ indent=1 notation= $\langle$ mixfix TYPEREP $\rangle$  $\rangle$ TYPEREP/(1'(-')) $\rangle$ )
syntax-consts
  -TYPEREP  $\Rightarrow$  typerep

 $\langle$ ML $\rangle$ 

lemma [code]:
  HOL.equal (Typerep tyco1 tys1) (Typerep tyco2 tys2)  $\longleftrightarrow$  HOL.equal tyco1 tyco2
     $\wedge$  list-all2 HOL.equal tys1 tys2
   $\langle$ proof $\rangle$ 

lemma [code nbe]:
  HOL.equal (x :: typerep) x  $\longleftrightarrow$  True
   $\langle$ proof $\rangle$ 

code-printing
  type-constructor typerep  $\rightarrow$  (Eval) Term.typ
| constant Typerep  $\rightarrow$  (Eval) Term.Type/ (-, -)

code-reserved
  (Eval) Term

hide-const (open) typerep Typerep

end

```

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  $\Rightarrow$  bool)

```

```

lemma pred-eqI:
  ( $\bigwedge w. \text{eval } P \ w \longleftrightarrow \text{eval } Q \ w$ )  $\Longrightarrow$  P = Q
   $\langle$ proof $\rangle$ 

```

lemma *pred-eq-iff*:

$$P = Q \implies (\bigwedge w. \text{eval } P \ w \longleftrightarrow \text{eval } Q \ w) \\ \langle \text{proof} \rangle$$

instantiation *pred* :: (type) complete-lattice
begin

definition

$$P \leq Q \longleftrightarrow \text{eval } P \leq \text{eval } Q$$

definition

$$P < Q \longleftrightarrow \text{eval } P < \text{eval } Q$$

definition

$$\perp = \text{Pred } \perp$$

lemma *eval-bot [simp]*:

$$\text{eval } \perp = \perp \\ \langle \text{proof} \rangle$$

definition

$$\top = \text{Pred } \top$$

lemma *eval-top [simp]*:

$$\text{eval } \top = \top \\ \langle \text{proof} \rangle$$

definition

$$P \sqcap Q = \text{Pred } (\text{eval } P \sqcap \text{eval } Q)$$

lemma *eval-inf [simp]*:

$$\text{eval } (P \sqcap Q) = \text{eval } P \sqcap \text{eval } Q \\ \langle \text{proof} \rangle$$

definition

$$P \sqcup Q = \text{Pred } (\text{eval } P \sqcup \text{eval } Q)$$

lemma *eval-sup [simp]*:

$$\text{eval } (P \sqcup Q) = \text{eval } P \sqcup \text{eval } Q \\ \langle \text{proof} \rangle$$

definition

$$\bigcap A = \text{Pred } (\bigcap (\text{eval } ` A))$$

lemma *eval-Inf [simp]*:

$$\text{eval } (\bigcap A) = \bigcap (\text{eval } ` A) \\ \langle \text{proof} \rangle$$

definition

$$\sqcup A = \text{Pred } (\sqcup (eval \text{ ' } A))$$
lemma *eval-Sup* [simp]:
$$eval (\sqcup A) = \sqcup (eval \text{ ' } A)$$

⟨proof⟩

instance ⟨proof⟩**end****lemma** *eval-INF* [simp]:
$$eval (\sqcap (f \text{ ' } A)) = \sqcap ((eval \circ f) \text{ ' } A)$$

⟨proof⟩

lemma *eval-SUP* [simp]:
$$eval (\sqcup (f \text{ ' } A)) = \sqcup ((eval \circ f) \text{ ' } A)$$

⟨proof⟩

instantiation *pred* :: (type) complete-boolean-algebra
begin**definition**

$$- P = \text{Pred } (- \text{ eval } P)$$
lemma *eval-compl* [simp]:
$$eval (- P) = - \text{ eval } P$$

⟨proof⟩

definition

$$P - Q = \text{Pred } (eval P - eval Q)$$
lemma *eval-minus* [simp]:
$$eval (P - Q) = eval P - eval Q$$

⟨proof⟩

instance ⟨proof⟩**end****definition** *single* :: 'a \Rightarrow 'a pred **where**

$$single x = \text{Pred } ((=) x)$$
lemma *eval-single* [simp]:
$$eval (single x) = (=) x$$

⟨proof⟩

definition *bind* :: 'a pred \Rightarrow ('a \Rightarrow 'b pred) \Rightarrow 'b pred (**infixl** $\langle \gg \rangle$ 70) **where**

$$P \gg f = (\sqcup (f \text{ ' } \{x. \text{ eval } P x\}))$$

lemma *eval-bind [simp]*:
 $eval\ (P \ggg f) = eval\ (\bigsqcup (f\ '\ \{x.\ eval\ P\ x\}))$
 $\langle proof \rangle$

lemma *bind-bind*:
 $(P \ggg Q) \ggg R = P \ggg (\lambda x.\ Q\ x \ggg R)$
 $\langle proof \rangle$

lemma *bind-single*:
 $P \ggg single = P$
 $\langle proof \rangle$

lemma *single-bind*:
 $single\ x \ggg P = P\ x$
 $\langle proof \rangle$

lemma *bottom-bind*:
 $\perp \ggg P = \perp$
 $\langle proof \rangle$

lemma *sup-bind*:
 $(P \sqcup Q) \ggg R = P \ggg R \sqcup Q \ggg R$
 $\langle proof \rangle$

lemma *Sup-bind*:
 $(\bigsqcup A \ggg f) = \bigsqcup ((\lambda x.\ x \ggg f)\ '\ A)$
 $\langle proof \rangle$

lemma *pred-iffI*:
assumes $\bigwedge x.\ eval\ A\ x \implies eval\ B\ x$
and $\bigwedge x.\ eval\ B\ x \implies eval\ A\ x$
shows $A = B$
 $\langle proof \rangle$

lemma *singleI*: $eval\ (single\ x)\ x$
 $\langle proof \rangle$

lemma *singleI-unit*: $eval\ (single\ ())\ x$
 $\langle proof \rangle$

lemma *singleE*: $eval\ (single\ x)\ y \implies (y = x \implies P) \implies P$
 $\langle proof \rangle$

lemma *singleE'*: $eval\ (single\ x)\ y \implies (x = y \implies P) \implies P$
 $\langle proof \rangle$

lemma *bindI*: $eval\ P\ x \implies eval\ (Q\ x)\ y \implies eval\ (P \ggg Q)\ y$
 $\langle proof \rangle$

lemma *bindE*: $eval\ (R \gg= Q)\ y \implies (\bigwedge x. eval\ R\ x \implies eval\ (Q\ x)\ y \implies P) \implies P$
 ⟨proof⟩

lemma *botE*: $eval\ \perp\ x \implies P$
 ⟨proof⟩

lemma *supI1*: $eval\ A\ x \implies eval\ (A \sqcup B)\ x$
 ⟨proof⟩

lemma *supI2*: $eval\ B\ x \implies eval\ (A \sqcup B)\ x$
 ⟨proof⟩

lemma *supE*: $eval\ (A \sqcup B)\ x \implies (eval\ A\ x \implies P) \implies (eval\ B\ x \implies P) \implies P$
 ⟨proof⟩

lemma *single-not-bot* [*simp*]:
 $single\ x \neq \perp$
 ⟨proof⟩

lemma *not-bot*:
 assumes $A \neq \perp$
 obtains x **where** $eval\ A\ x$
 ⟨proof⟩

75.2 Emptiness check and definite choice

definition *is-empty* :: $'a\ pred \Rightarrow bool$ **where**
 $is_empty\ A \longleftrightarrow A = \perp$

lemma *is-empty-bot*:
 $is_empty\ \perp$
 ⟨proof⟩

lemma *not-is-empty-single*:
 $\neg is_empty\ (single\ x)$
 ⟨proof⟩

lemma *is-empty-sup*:
 $is_empty\ (A \sqcup B) \longleftrightarrow is_empty\ A \wedge is_empty\ B$
 ⟨proof⟩

definition *singleton* :: $(unit \Rightarrow 'a) \Rightarrow 'a\ pred \Rightarrow 'a$ **where**
 $singleton\ default\ A = (if\ \exists!x. eval\ A\ x\ then\ THE\ x. eval\ A\ x\ else\ default\ ())\ \mathbf{for}\ default$

lemma *singleton-eqI*:
 $\exists!x. eval\ A\ x \implies eval\ A\ x \implies singleton\ default\ A = x$ **for** *default*

<proof>

lemma *eval-singletonI*:

$\exists!x. \text{eval } A \ x \Longrightarrow \text{eval } A \ (\text{singleton default } A) \text{ for default}$
<proof>

lemma *single-singleton*:

$\exists!x. \text{eval } A \ x \Longrightarrow \text{single } (\text{singleton default } A) = A \text{ for default}$
<proof>

lemma *singleton-undefinedI*:

$\neg (\exists!x. \text{eval } A \ x) \Longrightarrow \text{singleton default } A = \text{default } () \text{ for default}$
<proof>

lemma *singleton-bot*:

$\text{singleton default } \perp = \text{default } () \text{ for default}$
<proof>

lemma *singleton-single*:

$\text{singleton default } (\text{single } x) = x \text{ for default}$
<proof>

lemma *singleton-sup-single-single*:

$\text{singleton default } (\text{single } x \sqcup \text{single } y) = (\text{if } x = y \text{ then } x \text{ else default } ()) \text{ for default}$
<proof>

lemma *singleton-sup-aux*:

$\text{singleton default } (A \sqcup B) = (\text{if } A = \perp \text{ then singleton default } B$
 $\text{else if } B = \perp \text{ then singleton default } A$
 $\text{else singleton default}$
 $(\text{single } (\text{singleton default } A) \sqcup \text{single } (\text{singleton default } B))) \text{ for default}$
<proof>

lemma *singleton-sup*:

$\text{singleton default } (A \sqcup B) = (\text{if } A = \perp \text{ then singleton default } B$
 $\text{else if } B = \perp \text{ then singleton default } A$
 $\text{else if singleton default } A = \text{singleton default } B \text{ then singleton default } A \text{ else}$
 $\text{default } ()) \text{ for default}$
<proof>

75.3 Derived operations

definition *if-pred* :: *bool* \Rightarrow *unit pred* **where**

if-pred-eq: *if-pred* *b* = (*if* *b* then *single* () else \perp)

definition *holds* :: *unit pred* \Rightarrow *bool* **where**

holds-eq: *holds* *P* = *eval* *P* ()

definition *not-pred* :: *unit pred* \Rightarrow *unit pred* **where**
not-pred-eq: *not-pred* *P* = (if *eval P* () then \perp else *single* ())

lemma *if-predI*: *P* \Rightarrow *eval (if-pred P)* ()
 ⟨*proof*⟩

lemma *if-predE*: *eval (if-pred b) x* \Rightarrow (*b* \Rightarrow *x* = () \Rightarrow *P*) \Rightarrow *P*
 ⟨*proof*⟩

lemma *not-predI*: \neg *P* \Rightarrow *eval (not-pred (Pred ($\lambda u.$ *P*)))* ()
 ⟨*proof*⟩

lemma *not-predI'*: \neg *eval P* () \Rightarrow *eval (not-pred P)* ()
 ⟨*proof*⟩

lemma *not-predE*: *eval (not-pred (Pred ($\lambda u.$ *P*))) x* \Rightarrow (\neg *P* \Rightarrow *thesis*) \Rightarrow *thesis*
 ⟨*proof*⟩

lemma *not-predE'*: *eval (not-pred P) x* \Rightarrow (\neg *eval P x* \Rightarrow *thesis*) \Rightarrow *thesis*
 ⟨*proof*⟩

lemma *f* () = *False* \vee *f* () = *True*
 ⟨*proof*⟩

lemma *closure-of-bool-cases* [*no-atp*]:
fixes *f* :: *unit* \Rightarrow *bool*
assumes *f* = ($\lambda u.$ *False*) \Rightarrow *P f*
assumes *f* = ($\lambda u.$ *True*) \Rightarrow *P f*
shows *P f*
 ⟨*proof*⟩

lemma *unit-pred-cases*:
assumes *P* \perp
assumes *P (single* ())
shows *P Q*
 ⟨*proof*⟩

lemma *holds-if-pred*:
holds (if-pred b) = *b*
 ⟨*proof*⟩

lemma *if-pred-holds*:
if-pred (holds P) = *P*
 ⟨*proof*⟩

lemma *is-empty-holds*:
is-empty P \longleftrightarrow \neg *holds P*
 ⟨*proof*⟩

definition $map :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \Rightarrow 'b \text{ pred}$ **where**
 $map\ f\ P = P \gg= (single \circ f)$

lemma $eval\text{-}map$ $[simp]$:
 $eval\ (map\ f\ P) = (\bigsqcup x \in \{x. eval\ P\ x\}. (\lambda y. f\ x = y))$
 $\langle proof \rangle$

functor map : map
 $\langle proof \rangle$

75.4 Implementation

datatype (*plugins only: code extraction*) (*dead 'a*) $seq =$
 $Empty$
 $| Insert\ 'a\ 'a\ pred$
 $| Join\ 'a\ pred\ 'a\ seq$

primrec $pred\text{-}of\text{-}seq :: 'a\ seq \Rightarrow 'a\ pred$ **where**
 $pred\text{-}of\text{-}seq\ Empty = \perp$
 $| pred\text{-}of\text{-}seq\ (Insert\ x\ P) = single\ x \sqcup P$
 $| pred\text{-}of\text{-}seq\ (Join\ P\ xq) = P \sqcup pred\text{-}of\text{-}seq\ xq$

definition $Seq :: (unit \Rightarrow 'a\ seq) \Rightarrow 'a\ pred$ **where**
 $Seq\ f = pred\text{-}of\text{-}seq\ (f\ ())$

code-datatype Seq

primrec $member :: 'a\ seq \Rightarrow 'a \Rightarrow bool$ **where**
 $member\ Empty\ x \longleftrightarrow False$
 $| member\ (Insert\ y\ P)\ x \longleftrightarrow x = y \vee eval\ P\ x$
 $| member\ (Join\ P\ xq)\ x \longleftrightarrow eval\ P\ x \vee member\ xq\ x$

lemma $eval\text{-}member$:
 $member\ xq = eval\ (pred\text{-}of\text{-}seq\ xq)$
 $\langle proof \rangle$

lemma $eval\text{-}code$ $[code]$: $eval\ (Seq\ f) = member\ (f\ ())$
 $\langle proof \rangle$

lemma $single\text{-}code$ $[code]$:
 $single\ x = Seq\ (\lambda u. Insert\ x\ \perp)$
 $\langle proof \rangle$

primrec $apply :: ('a \Rightarrow 'b\ pred) \Rightarrow 'a\ seq \Rightarrow 'b\ seq$ **where**
 $apply\ f\ Empty = Empty$
 $| apply\ f\ (Insert\ x\ P) = Join\ (f\ x)\ (Join\ (P \gg= f)\ Empty)$
 $| apply\ f\ (Join\ P\ xq) = Join\ (P \gg= f)\ (apply\ f\ xq)$

lemma $apply\text{-}bind$:

$\text{pred-of-seq } (\text{apply } f \ xq) = \text{pred-of-seq } xq \ggg f$
 $\langle \text{proof} \rangle$

lemma *bind-code* [code]:
 $\text{Seq } g \ggg f = \text{Seq } (\lambda u. \text{apply } f \ (g \ ()))$
 $\langle \text{proof} \rangle$

lemma *bot-set-code* [code]:
 $\perp = \text{Seq } (\lambda u. \text{Empty})$
 $\langle \text{proof} \rangle$

primrec *adjunct* :: 'a pred \Rightarrow 'a seq \Rightarrow 'a seq **where**
 $\text{adjunct } P \ \text{Empty} = \text{Join } P \ \text{Empty}$
 $| \text{adjunct } P \ (\text{Insert } x \ Q) = \text{Insert } x \ (Q \sqcup P)$
 $| \text{adjunct } P \ (\text{Join } Q \ xq) = \text{Join } Q \ (\text{adjunct } P \ xq)$

lemma *adjunct-sup*:
 $\text{pred-of-seq } (\text{adjunct } P \ xq) = P \sqcup \text{pred-of-seq } xq$
 $\langle \text{proof} \rangle$

lemma *sup-code* [code]:
 $\text{Seq } f \sqcup \text{Seq } g = \text{Seq } (\lambda u. \text{case } f \ ($
 $\quad \text{of } \text{Empty} \Rightarrow g \ ($
 $\quad | \text{Insert } x \ P \Rightarrow \text{Insert } x \ (P \sqcup \text{Seq } g)$
 $\quad | \text{Join } P \ xq \Rightarrow \text{adjunct } (\text{Seq } g) \ (\text{Join } P \ xq))$
 $\langle \text{proof} \rangle$

primrec *contained* :: 'a seq \Rightarrow 'a pred \Rightarrow bool **where**
 $\text{contained } \text{Empty} \ Q \longleftrightarrow \text{True}$
 $| \text{contained } (\text{Insert } x \ P) \ Q \longleftrightarrow \text{eval } Q \ x \wedge P \leq Q$
 $| \text{contained } (\text{Join } P \ xq) \ Q \longleftrightarrow P \leq Q \wedge \text{contained } xq \ Q$

lemma *single-less-eq-eval*:
 $\text{single } x \leq P \longleftrightarrow \text{eval } P \ x$
 $\langle \text{proof} \rangle$

lemma *contained-less-eq*:
 $\text{contained } xq \ Q \longleftrightarrow \text{pred-of-seq } xq \leq Q$
 $\langle \text{proof} \rangle$

lemma *less-eq-pred-code* [code]:
 $\text{Seq } f \leq Q = (\text{case } f \ ($
 $\quad \text{of } \text{Empty} \Rightarrow \text{True}$
 $\quad | \text{Insert } x \ P \Rightarrow \text{eval } Q \ x \wedge P \leq Q$
 $\quad | \text{Join } P \ xq \Rightarrow P \leq Q \wedge \text{contained } xq \ Q)$
 $\langle \text{proof} \rangle$

instantiation *pred* :: (type) equal
begin

definition *equal-pred*

where [*simp*]: $HOL.equal\ P\ Q \longleftrightarrow P = (Q :: 'a\ pred)$

instance $\langle proof \rangle$

end

lemma [*code nbe*]:

$HOL.equal\ P\ P \longleftrightarrow True$ **for** $P :: 'a\ pred$

$\langle proof \rangle$

lemma [*code*]:

$HOL.equal\ P\ Q \longleftrightarrow P \leq Q \wedge Q \leq P$ **for** $P\ Q :: 'a\ pred$

$\langle proof \rangle$

lemma [*code*]:

$case-pred\ f\ P = f\ (eval\ P)$

$\langle proof \rangle$

lemma [*code*]:

$rec-pred\ f\ P = f\ (eval\ P)$

$\langle proof \rangle$

inductive $eq :: 'a \Rightarrow 'a \Rightarrow bool$ **where** $eq\ x\ x$

lemma $eq-is-eq: eq\ x\ y \equiv (x = y)$

$\langle proof \rangle$

primrec $null :: 'a\ seq \Rightarrow bool$ **where**

$null\ Empty \longleftrightarrow True$

| $null\ (Insert\ x\ P) \longleftrightarrow False$

| $null\ (Join\ P\ xq) \longleftrightarrow is-empty\ P \wedge null\ xq$

lemma $null-is-empty$:

$null\ xq \longleftrightarrow is-empty\ (pred-of-seq\ xq)$

$\langle proof \rangle$

lemma $is-empty-code$ [*code*]:

$is-empty\ (Seq\ f) \longleftrightarrow null\ (f\ ())$

$\langle proof \rangle$

primrec $the-only :: (unit \Rightarrow 'a) \Rightarrow 'a\ seq \Rightarrow 'a$ **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 $x = \text{singleton default } P$; $y = \text{the-only default } xq$ in
if $x = y$ then x else default $()$) for default

lemma *the-only-singleton:*

the-only default $xq = \text{singleton default } (\text{pred-of-seq } xq)$ for default
 $\langle \text{proof} \rangle$

lemma *singleton-code [code]:*

singleton default $(\text{Seq } f) =$
(case $f ()$ of
Empty \Rightarrow default $()$
| Insert $x P \Rightarrow$ if is-empty P then x
else let $y = \text{singleton default } P$ in
if $x = y$ then x else default $()$
| Join $P xq \Rightarrow$ if is-empty P then the-only default xq
else if null xq then singleton default P
else let $x = \text{singleton default } P$; $y = \text{the-only default } xq$ in
if $x = y$ then x else default $()$) for default
 $\langle \text{proof} \rangle$

definition *the :: 'a pred \Rightarrow 'a where*

the $A = (\text{THE } x. \text{eval } A \ x)$

lemma *the-eqI:*

(THE $x. \text{eval } P \ x) = x \Longrightarrow \text{the } P = x$
 $\langle \text{proof} \rangle$

lemma *the-eq [code]: the $A = \text{singleton } (\lambda x. \text{Code.abort } (\text{STR "not-unique"}) (\lambda -.$*
the $A)) \ A$

$\langle \text{proof} \rangle$

code-reflect *Predicate*

datatypes *pred = Seq and seq = Empty | Insert | Join*

$\langle \text{ML} \rangle$

Conversion from and to sets

definition *pred-of-set :: 'a set \Rightarrow 'a pred where*

pred-of-set = Pred $\circ (\lambda A \ x. x \in A)$

lemma *eval-pred-of-set [simp]:*

eval (pred-of-set A) $x \longleftrightarrow x \in A$
 $\langle \text{proof} \rangle$

definition *set-of-pred :: 'a pred \Rightarrow 'a set where*

set-of-pred = Collect $\circ \text{eval}$

lemma *member-set-of-pred [simp]:*

$x \in \text{set-of-pred } P \longleftrightarrow \text{Predicate.eval } P \ x$

$\langle \text{proof} \rangle$

definition *set-of-seq* :: 'a seq \Rightarrow 'a set **where**
set-of-seq = *set-of-pred* \circ *pred-of-seq*

lemma *member-set-of-seq* [*simp*]:
 $x \in \text{set-of-seq } xq = \text{Predicate.member } xq \ x$
 $\langle \text{proof} \rangle$

lemma *of-pred-code* [*code*]:
 $\text{set-of-pred } (\text{Predicate.Seq } f) = (\text{case } f \ () \ \text{of}$
 $\quad \text{Predicate.Empty} \Rightarrow \{\}$
 $\quad | \text{Predicate.Insert } x \ P \Rightarrow \text{insert } x \ (\text{set-of-pred } P)$
 $\quad | \text{Predicate.Join } P \ xq \Rightarrow \text{set-of-pred } P \cup \text{set-of-seq } xq)$
 $\langle \text{proof} \rangle$

lemma *of-seq-code* [*code*]:
 $\text{set-of-seq } \text{Predicate.Empty} = \{\}$
 $\text{set-of-seq } (\text{Predicate.Insert } x \ P) = \text{insert } x \ (\text{set-of-pred } P)$
 $\text{set-of-seq } (\text{Predicate.Join } P \ xq) = \text{set-of-pred } P \cup \text{set-of-seq } xq$
 $\langle \text{proof} \rangle$

Lazy Evaluation of an indexed function

function *iterate-upto* :: (natural \Rightarrow 'a) \Rightarrow natural \Rightarrow natural \Rightarrow 'a *Predicate.pred*
where
 $\text{iterate-upto } f \ n \ m =$
 $\quad \text{Predicate.Seq } (\%u. \text{if } n > m \text{ then } \text{Predicate.Empty}$
 $\quad \text{else } \text{Predicate.Insert } (f \ n) \ (\text{iterate-upto } f \ (n + 1) \ m))$
 $\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

Misc

declare *Inf-set-fold* [**where** 'a = 'a *Predicate.pred*, *code*]
declare *Sup-set-fold* [**where** 'a = 'a *Predicate.pred*, *code*]

lemma *pred-of-set-fold-sup*:
assumes *finite A*
shows $\text{pred-of-set } A = \text{Finite-Set.fold } \text{sup } \text{bot } (\text{Predicate.single } 'A) \ (\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma *pred-of-set-set-fold-sup*:
 $\text{pred-of-set } (\text{set } xs) = \text{fold } \text{sup } (\text{List.map } \text{Predicate.single } xs) \ \text{bot}$
 $\langle \text{proof} \rangle$

lemma *pred-of-set-set-foldr-sup* [*code*]:

pred-of-set (*set xs*) = *foldr sup (List.map Predicate.single xs) bot*
<proof>

no-notation *bind* (**infixl** $\langle \gg \rangle$ 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* \Rightarrow *'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

<proof>

lemma *lazy-sequence-eqI:*

list-of-lazy-sequence xq = list-of-lazy-sequence yq \implies xq = yq

<proof>

lemma *lazy-sequence-eq-iff:*

xq = yq \longleftrightarrow list-of-lazy-sequence xq = list-of-lazy-sequence yq

<proof>

lemma *case-lazy-sequence [simp]:*

case-lazy-sequence f xq = f (list-of-lazy-sequence xq)

<proof>

lemma *rec-lazy-sequence [simp]:*

rec-lazy-sequence f xq = f (list-of-lazy-sequence xq)

<proof>

definition *Lazy-Sequence* :: (*unit* \Rightarrow (*'a* \times *'a lazy-sequence*) *option*) \Rightarrow *'a lazy-sequence*

where

Lazy-Sequence $f = \text{lazy-sequence-of-list } (\text{case } f \text{ () of}$
 $\text{None} \Rightarrow []$
 $| \text{Some } (x, xq) \Rightarrow x \# \text{list-of-lazy-sequence } xq)$

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
 $\text{None} \Rightarrow []$
 $| \text{Some } (x, xq) \Rightarrow x \# \text{list-of-lazy-sequence } xq)$
 ⟨proof⟩

definition *yield* :: 'a lazy-sequence \Rightarrow ('a \times 'a lazy-sequence) option

where

yield $xq = (\text{case } \text{list-of-lazy-sequence } xq \text{ of}$
 $[] \Rightarrow \text{None}$
 $| x \# xs \Rightarrow \text{Some } (x, \text{lazy-sequence-of-list } xs))$

lemma *yield-Seq* [simp, code]:

yield (*Lazy-Sequence* f) = f ()
 ⟨proof⟩

lemma *case-yield-eq* [simp]: *case-option* g h (*yield* xq) =

case-list g ($\lambda x. \text{curry } h \ x \circ \text{lazy-sequence-of-list}$) (*list-of-lazy-sequence* xq)
 ⟨proof⟩

lemma *equal-lazy-sequence-code* [code]:

HOL.equal $xq \ yq = (\text{case } (\text{yield } xq, \text{yield } yq) \text{ of}$
 $(\text{None}, \text{None}) \Rightarrow \text{True}$
 $| (\text{Some } (x, xq'), \text{Some } (y, yq')) \Rightarrow \text{HOL.equal } x \ y \wedge \text{HOL.equal } xq \ yq$
 $| - \Rightarrow \text{False})$
 ⟨proof⟩

lemma [code nbe]:

HOL.equal ($x :: \text{'a lazy-sequence}$) $x \longleftrightarrow \text{True}$
 ⟨proof⟩

definition *empty* :: 'a lazy-sequence

where

empty = *lazy-sequence-of-list* []

lemma *list-of-lazy-sequence-empty* [simp]:

list-of-lazy-sequence *empty* = []
 ⟨proof⟩

lemma *empty-code* [code]:
 $empty = Lazy-Sequence (\lambda-. None)$
 ⟨proof⟩

definition *single* :: 'a \Rightarrow 'a lazy-sequence
where
 $single\ x = lazy-sequence-of-list\ [x]$

lemma *list-of-lazy-sequence-single* [simp]:
 $list-of-lazy-sequence\ (single\ x) = [x]$
 ⟨proof⟩

lemma *single-code* [code]:
 $single\ x = Lazy-Sequence (\lambda-. Some\ (x,\ empty))$
 ⟨proof⟩

definition *append* :: 'a lazy-sequence \Rightarrow 'a lazy-sequence \Rightarrow '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]:
 $list-of-lazy-sequence\ (append\ xq\ yq) = list-of-lazy-sequence\ xq\ @\ list-of-lazy-sequence\ yq$
 ⟨proof⟩

lemma *append-code* [code]:
 $append\ xq\ yq = Lazy-Sequence (\lambda-. case\ yield\ xq\ of$
 $\quad None \Rightarrow yield\ yq$
 $\quad | Some\ (x,\ xq') \Rightarrow Some\ (x,\ append\ xq'\ yq))$
 ⟨proof⟩

definition *map* :: ('a \Rightarrow 'b) \Rightarrow 'a lazy-sequence \Rightarrow '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]:
 $list-of-lazy-sequence\ (map\ f\ xq) = List.map\ f\ (list-of-lazy-sequence\ xq)$
 ⟨proof⟩

lemma *map-code* [code]:
 $map\ f\ xq =$
 $Lazy-Sequence (\lambda-. map-option\ (\lambda(x,\ xq'). (f\ x,\ map\ f\ xq'))\ (yield\ xq))$
 ⟨proof⟩

definition *flat* :: 'a lazy-sequence lazy-sequence \Rightarrow 'a lazy-sequence
where
 $flat\ xq = lazy-sequence-of-list\ (concat\ (List.map\ list-of-lazy-sequence\ (list-of-lazy-sequence\ xq)))$

$xqq)))$

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))
 ⟨proof⟩

lemma *flat-code* [code]:

flat xqq = *Lazy-Sequence* (λ -. case yield xqq of
 None \Rightarrow None
 | *Some* (xq , xqq') \Rightarrow yield (*append* xq (*flat* xqq')))
 ⟨proof⟩

definition *bind* :: 'a lazy-sequence \Rightarrow ('a \Rightarrow 'b lazy-sequence) \Rightarrow 'b lazy-sequence
where

bind xq f = *flat* (*map* f xq)

definition *if-seq* :: bool \Rightarrow unit lazy-sequence

where

if-seq b = (if b then *single* () else *empty*)

definition *those* :: 'a option lazy-sequence \Rightarrow 'a lazy-sequence option

where

those xq = *map-option lazy-sequence-of-list* (*List.those* (*list-of-lazy-sequence* xq))

function *iterate-upto* :: (natural \Rightarrow 'a) \Rightarrow natural \Rightarrow natural \Rightarrow 'a lazy-sequence

where

iterate-upto f n m =
 Lazy-Sequence (λ -. if $n > m$ then None else *Some* (f n , *iterate-upto* f ($n + 1$)
 m))
 ⟨proof⟩

termination ⟨proof⟩

definition *not-seq* :: unit lazy-sequence \Rightarrow unit lazy-sequence

where

not-seq xq = (case yield xq of
 None \Rightarrow *single* ()
 | *Some* ((), xq) \Rightarrow *empty*)

76.2 Code setup

code-reflect *Lazy-Sequence*

datatypes *lazy-sequence* = *Lazy-Sequence*

⟨ML⟩

76.3 Generator Sequences

76.3.1 General lazy sequence operation

definition $product :: 'a\ lazy-sequence \Rightarrow 'b\ lazy-sequence \Rightarrow ('a \times 'b)\ lazy-sequence$
where
 $product\ s1\ s2 = bind\ s1\ (\lambda a. bind\ s2\ (\lambda b. single\ (a, b)))$

76.3.2 Small lazy typeclasses

class $small-lazy =$
fixes $small-lazy :: natural \Rightarrow 'a\ lazy-sequence$

instantiation $unit :: small-lazy$
begin

definition $small-lazy\ d = single\ ()$

instance $\langle proof \rangle$

end

instantiation $int :: small-lazy$
begin

maybe optimise this expression $\rightarrow append\ (single\ x)\ xs == cons\ x\ xs$ Performance difference?

function $small-lazy' :: int \Rightarrow int \Rightarrow int\ lazy-sequence$
where
 $small-lazy'\ d\ i = (if\ d < i\ then\ empty$
 $\quad else\ append\ (single\ i)\ (small-lazy'\ d\ (i + 1)))$
 $\langle proof \rangle$

termination
 $\langle proof \rangle$

definition
 $small-lazy\ d = small-lazy'\ (int\ (nat-of-natural\ d))\ (-\ (int\ (nat-of-natural\ d)))$

instance $\langle proof \rangle$

end

instantiation $prod :: (small-lazy, small-lazy)\ small-lazy$
begin

definition
 $small-lazy\ d = product\ (small-lazy\ d)\ (small-lazy\ d)$

instance $\langle proof \rangle$

end

instantiation *list* :: (*small-lazy*) *small-lazy*
begin

fun *small-lazy-list* :: *natural* \Rightarrow '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))) ($\lambda(x, xs).$ *single* (*x* # *xs*)) *else empty*)

instance \langle *proof* \rangle

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* ($\lambda-$. *Some* (*None*, *empty*))

lemma *list-of-lazy-sequence-hit-bound* [*simp*]:

list-of-lazy-sequence hit-bound = [*None*]

\langle *proof* \rangle

definition *hb-single* :: 'a \Rightarrow 'a *hit-bound-lazy-sequence*

where

hb-single *x* = *Lazy-Sequence* ($\lambda-$. *Some* (*Some* *x*, *empty*))

definition *hb-map* :: ('a \Rightarrow 'b) \Rightarrow 'a *hit-bound-lazy-sequence* \Rightarrow '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 ($\lambda-$. *map-option* ($\lambda(x, xq').$ (*map-option* *f* *x*, *hb-map* *f* *xq'*)) (*yield* *xq*))

\langle *proof* \rangle

definition *hb-flat* :: 'a *hit-bound-lazy-sequence hit-bound-lazy-sequence* \Rightarrow 'a *hit-bound-lazy-sequence*

where

hb-flat *xqq* = *lazy-sequence-of-list* (*concat*

 (*List.map* (($\lambda x.$ *case* *x* *of* *None* \Rightarrow [*None*] | *Some* *xs* \Rightarrow *xs*) \circ *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* ((λx . *case* *x* of *None* \Rightarrow [*None*] | *Some* *xs* \Rightarrow *xs*) \circ *map-option*
list-of-lazy-sequence) (*list-of-lazy-sequence* *xqq*))
 ⟨*proof*⟩

lemma *hb-flat-code* [code]:
hb-flat *xqq* = *Lazy-Sequence* (λ -. *case* *yield* *xqq* of
None \Rightarrow *None*
 | *Some* (*xq*, *xqq'*) \Rightarrow *yield*
 (*append* (*case* *xq* of *None* \Rightarrow *hit-bound* | *Some* *xq* \Rightarrow *xq*) (*hb-flat* *xqq'*)))
 ⟨*proof*⟩

definition *hb-bind* :: 'a *hit-bound-lazy-sequence* \Rightarrow ('a \Rightarrow 'b *hit-bound-lazy-sequence*)
 \Rightarrow 'b *hit-bound-lazy-sequence*

where
hb-bind *xq* *f* = *hb-flat* (*hb-map* *f* *xq*)

definition *hb-if-seq* :: bool \Rightarrow unit *hit-bound-lazy-sequence*

where
hb-if-seq *b* = (*if* *b* then *hb-single* () else *empty*)

definition *hb-not-seq* :: unit *hit-bound-lazy-sequence* \Rightarrow unit *lazy-sequence*

where
hb-not-seq *xq* = (*case* *yield* *xq* of
None \Rightarrow *single* ()
 | *Some* (*x*, *xq*) \Rightarrow *empty*)

hide-const (**open**) *yield* *empty* *single* *append* *flat* *map* *bind*
if-seq *those* *iterate-upto* *not-seq* *product*

hide-fact (**open**) *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 'a *dseq* = *natural* \Rightarrow bool \Rightarrow 'a *lazy-sequence* *option*

definition *empty* :: 'a *dseq*
where
empty = (λ -. *Some* *Lazy-Sequence.empty*)

definition *single* :: 'a \Rightarrow 'a dseq

where

single *x* = (λ - -. *Some* (*Lazy-Sequence.single* *x*))

definition *eval* :: 'a dseq \Rightarrow natural \Rightarrow bool \Rightarrow 'a lazy-sequence option

where

[simp]: *eval* *f* *i* *pol* = *f* *i* *pol*

definition *yield* :: 'a dseq \Rightarrow natural \Rightarrow bool \Rightarrow ('a \times 'a dseq) option

where

yield *f* *i* *pol* = (case *eval* *f* *i* *pol* of

None \Rightarrow *None*

| *Some* *s* \Rightarrow (*map-option* \circ *apsnd*) (λ *r* - -. *Some* *r*) (*Lazy-Sequence.yield* *s*))

definition *map-seq* :: ('a \Rightarrow 'b dseq) \Rightarrow 'a lazy-sequence \Rightarrow 'b dseq

where

map-seq *f* *xq* *i* *pol* = *map-option* *Lazy-Sequence.flat*

(*Lazy-Sequence.those* (*Lazy-Sequence.map* (λ *x*. *f* *x* *i* *pol*) *xq*))

lemma *map-seq-code* [code]:

map-seq *f* *xq* *i* *pol* = (case *Lazy-Sequence.yield* *xq* of

None \Rightarrow *Some* *Lazy-Sequence.empty*

| *Some* (*x*, *xq'*) \Rightarrow (case *eval* (*f* *x*) *i* *pol* of

None \Rightarrow *None*

| *Some* *yq* \Rightarrow (case *map-seq* *f* *xq'* *i* *pol* of

None \Rightarrow *None*

| *Some* *zq* \Rightarrow *Some* (*Lazy-Sequence.append* *yq* *zq*)))

(*proof*)

definition *bind* :: 'a dseq \Rightarrow ('a \Rightarrow 'b dseq) \Rightarrow 'b dseq

where

bind *x* *f* = (λ *i* *pol*.

if *i* = 0 then

(if *pol* then *Some* *Lazy-Sequence.empty* else *None*)

else

(case *x* (*i* - 1) *pol* of

None \Rightarrow *None*

| *Some* *xq* \Rightarrow *map-seq* *f* *xq* *i* *pol*))

definition *union* :: 'a dseq \Rightarrow 'a dseq \Rightarrow 'a dseq

where

union *x* *y* = (λ *i* *pol*. case (*x* *i* *pol*, *y* *i* *pol*) of

(*Some* *xq*, *Some* *yq*) \Rightarrow *Some* (*Lazy-Sequence.append* *xq* *yq*)

| - \Rightarrow *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* = (λi *pol.* *case* *x* *i* (\neg *pol*) *of*
 None \Rightarrow *Some Lazy-Sequence.empty*
 | *Some* *xq* \Rightarrow (*case* *Lazy-Sequence.yield* *xq* *of*
 None \Rightarrow *Some (Lazy-Sequence.single ())*
 | *Some* - \Rightarrow *Some (Lazy-Sequence.empty)*))

definition *map* :: (*'a* \Rightarrow *'b*) \Rightarrow *'a dseq* \Rightarrow *'b dseq*

where

map *f* *g* = (λi *pol.* *case* *g* *i* *pol* *of*
 None \Rightarrow *None*
 | *Some* *xq* \Rightarrow *Some (Lazy-Sequence.map* *f* *xq*)

77.2 Positive Depth-Limited Sequence

type-synonym *'a pos-dseq* = *natural* \Rightarrow *'a Lazy-Sequence.lazy-sequence*

definition *pos-empty* :: *'a pos-dseq*

where

pos-empty = ($\lambda i.$ *Lazy-Sequence.empty*)

definition *pos-single* :: *'a* \Rightarrow *'a pos-dseq*

where

pos-single *x* = ($\lambda i.$ *Lazy-Sequence.single* *x*)

definition *pos-bind* :: *'a pos-dseq* \Rightarrow (*'a* \Rightarrow *'b pos-dseq*) \Rightarrow *'b pos-dseq*

where

pos-bind *x* *f* = ($\lambda i.$ *Lazy-Sequence.bind* (*x* *i*) ($\lambda a.$ *f* *a* *i*))

definition *pos-decr-bind* :: *'a pos-dseq* \Rightarrow (*'a* \Rightarrow *'b pos-dseq*) \Rightarrow *'b pos-dseq*

where

pos-decr-bind *x* *f* = ($\lambda i.$
 if *i* = 0 *then*
 Lazy-Sequence.empty
 else
 Lazy-Sequence.bind (*x* (*i* - 1)) ($\lambda a.$ *f* *a* *i*))

definition *pos-union* :: *'a pos-dseq* \Rightarrow *'a pos-dseq* \Rightarrow *'a pos-dseq*

where

pos-union *xq* *yq* = ($\lambda i.$ *Lazy-Sequence.append* (*xq* *i*) (*yq* *i*))

definition *pos-if-seq* :: *bool* \Rightarrow *unit pos-dseq*

where

pos-if-seq *b* = (*if* *b* *then* *pos-single* () *else* *pos-empty*)

definition *pos-iterate-upto* :: (*natural* \Rightarrow *'a*) \Rightarrow *natural* \Rightarrow *natural* \Rightarrow *'a pos-dseq*

where

pos-iterate-upto $f\ n\ m = (\lambda i. \text{Lazy-Sequence.iterate-upto } f\ n\ m)$

definition *pos-map* $:: ('a \Rightarrow 'b) \Rightarrow 'a\ \text{pos-dseq} \Rightarrow 'b\ \text{pos-dseq}$
where
pos-map $f\ xq = (\lambda i. \text{Lazy-Sequence.map } f\ (xq\ i))$

77.3 Negative Depth-Limited Sequence

type-synonym *'a neg-dseq* $= \text{natural} \Rightarrow 'a\ \text{Lazy-Sequence.hit-bound-lazy-sequence}$

definition *neg-empty* $:: 'a\ \text{neg-dseq}$
where
neg-empty $= (\lambda i. \text{Lazy-Sequence.empty})$

definition *neg-single* $:: 'a \Rightarrow 'a\ \text{neg-dseq}$
where
neg-single $x = (\lambda i. \text{Lazy-Sequence.hb-single } x)$

definition *neg-bind* $:: 'a\ \text{neg-dseq} \Rightarrow ('a \Rightarrow 'b\ \text{neg-dseq}) \Rightarrow 'b\ \text{neg-dseq}$
where
neg-bind $x\ f = (\lambda i. \text{hb-bind } (x\ i)\ (\lambda a. f\ a\ i))$

definition *neg-decr-bind* $:: 'a\ \text{neg-dseq} \Rightarrow ('a \Rightarrow 'b\ \text{neg-dseq}) \Rightarrow 'b\ \text{neg-dseq}$
where
neg-decr-bind $x\ f = (\lambda i. \text{if } i = 0 \text{ then } \text{Lazy-Sequence.hit-bound} \text{ else } \text{hb-bind } (x\ (i - 1))\ (\lambda a. f\ a\ i))$

definition *neg-union* $:: 'a\ \text{neg-dseq} \Rightarrow 'a\ \text{neg-dseq} \Rightarrow 'a\ \text{neg-dseq}$
where
neg-union $x\ y = (\lambda i. \text{Lazy-Sequence.append } (x\ i)\ (y\ i))$

definition *neg-if-seq* $:: \text{bool} \Rightarrow \text{unit}\ \text{neg-dseq}$
where
neg-if-seq $b = (\text{if } b \text{ then } \text{neg-single } () \text{ else } \text{neg-empty})$

definition *neg-iterate-upto*
where
neg-iterate-upto $f\ n\ m = (\lambda i. \text{Lazy-Sequence.iterate-upto } (\lambda i. \text{Some } (f\ i))\ n\ m)$

definition *neg-map* $:: ('a \Rightarrow 'b) \Rightarrow 'a\ \text{neg-dseq} \Rightarrow 'b\ \text{neg-dseq}$
where
neg-map $f\ xq = (\lambda i. \text{Lazy-Sequence.hb-map } f\ (xq\ i))$

77.4 Negation

definition *pos-not-seq* $:: \text{unit}\ \text{neg-dseq} \Rightarrow \text{unit}\ \text{pos-dseq}$
where

pos-not-seq $xq = (\lambda i. \text{Lazy-Sequence.hb-not-seq } (xq \ (3 * i)))$

definition *neg-not-seq* :: *unit pos-dseq* \Rightarrow *unit neg-dseq*

where

neg-not-seq $x = (\lambda i. \text{case Lazy-Sequence.yield } (x \ i) \text{ of}$
 None $\Rightarrow \text{Lazy-Sequence.hb-single } ()$
 | Some $((), xq) \Rightarrow \text{Lazy-Sequence.empty})$

$\langle ML \rangle$

code-reserved

(Eval) Limited-Sequence

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-empty-def pos-single-def pos-bind-def pos-union-def pos-if-seq-def pos-iterate-upto-def
pos-not-seq-def pos-map-def
 neg-empty-def neg-single-def neg-bind-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* \Rightarrow *typerep* \Rightarrow *term* **where**

Const - - = *dummy-term*

definition *App* :: *term* \Rightarrow *term* \Rightarrow *term* **where**

App - - = *dummy-term*

definition *Abs* :: *String.literal* \Rightarrow *typerep* \Rightarrow *term* \Rightarrow *term* **where**
Abs - - - = *dummy-term*

definition *Free* :: *String.literal* \Rightarrow *typerep* \Rightarrow *term* **where**
Free - - = *dummy-term*

code-datatype *Const App Abs Free*

class *term-of* = *typerep* +
fixes *term-of* :: 'a \Rightarrow *term*

lemma *term-of-anything*: *term-of* *x* \equiv *t*
 <proof>

definition *valapp* :: ('a \Rightarrow 'b) \times (*unit* \Rightarrow *term*)
 \Rightarrow 'a \times (*unit* \Rightarrow *term*) \Rightarrow 'b \times (*unit* \Rightarrow *term*) **where**
valapp *f* *x* = (*fst* *f* (*fst* *x*), $\lambda u.$ *App* (*snd* *f* ()) (*snd* *x* ()))

lemma *valapp-code* [*code*, *code-unfold*]:
valapp (*f*, *tf*) (*x*, *tx*) = (*f* *x*, $\lambda u.$ *App* (*tf* ()) (*tx* ()))
 <proof>

78.1.2 Syntax

definition *termify* :: 'a \Rightarrow *term* **where**
 [*code del*]: *termify* *x* = *dummy-term*

abbreviation *valtermify* :: 'a \Rightarrow 'a \times (*unit* \Rightarrow *term*) **where**
valtermify *x* \equiv (*x*, $\lambda u.$ *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 \Rightarrow -, *TYPE*('a))

qualified definition *TERM-OF-EQUAL* :: 'a::*term-of itself*

where

TERM-OF-EQUAL = *snd* ($\lambda(a::'a).$ (*Code-Evaluation.term-of* *a*, *HOL.eq* *a*), *TYPE*('a))

end

lemma *eq-eq-TrueD*:
fixes $x\ y :: 'a::\{\}$
assumes $(x \equiv y) \equiv \text{Trueprop True}$
shows $x \equiv y$
 $\langle \text{proof} \rangle$

code-printing

type-constructor $\text{term} \rightarrow (\text{Eval})\ \text{Term.term}$
| **constant** $\text{Const} \rightarrow (\text{Eval})\ \text{Term.Const}/ ((-), (-))$
| **constant** $\text{App} \rightarrow (\text{Eval})\ \text{Term.\$}/ ((-), (-))$
| **constant** $\text{Abs} \rightarrow (\text{Eval})\ \text{Term.Abs}/ ((-), (-), (-))$
| **constant** $\text{Free} \rightarrow (\text{Eval})\ \text{Term.Free}/ ((-), (-))$

$\langle \text{ML} \rangle$

code-reserved

$(\text{Eval})\ \text{Code-Evaluation}$

$\langle \text{ML} \rangle$

78.3 Dedicated *term-of* instances

instantiation $\text{fun} :: (\text{typerep}, \text{typerep})\ \text{term-of}$
begin

definition

$\text{term-of}\ (f :: 'a \Rightarrow 'b) =$
 $\text{Const}\ (\text{STR}\ \text{"Pure.dummy-pattern"})$
 $(\text{Typerep.Typeprep}\ (\text{STR}\ \text{"fun"})\ [\text{Typerep.typerep}\ \text{TYPE}('a),\ \text{Typerep.typerep}\ \text{TYPE}('b)])$

instance $\langle \text{proof} \rangle$

end

declare $[[\text{code drop}:$

$\text{term-of} :: \text{typerep} \Rightarrow -$
 $\text{term-of} :: \text{term} \Rightarrow -$
 $\text{term-of} :: \text{integer} \Rightarrow -$
 $\text{term-of} :: \text{String.literal} \Rightarrow -$
 $\text{term-of} :: -\ \text{Predicate.pred} \Rightarrow -$
 $\text{term-of} :: -\ \text{Predicate.seq} \Rightarrow -]]$

code-printing

constant $\text{term-of} :: \text{integer} \Rightarrow \text{term} \rightarrow (\text{Eval})\ \text{HOLogic.mk}'\text{-number}/\ \text{HOLogic.code}'\text{-integerT}$
| **constant** $\text{term-of} :: \text{String.literal} \Rightarrow \text{term} \rightarrow (\text{Eval})\ \text{HOLogic.mk}'\text{-literal}$

lemma $\text{term-of-integer}\ [\text{unfolded}\ \text{typerep-fun-def}\ \text{typerep-num-def}\ \text{typerep-integer-def},$

```

code]:
  term-of (i :: integer) =
    (if i > 0 then
      App (Const (STR "Num.numeral-class.numeral") (TYPEREP(num ⇒ inte-
ger)))
      (term-of (num-of-integer i))
    else if i = 0 then Const (STR "Groups.zero-class.zero") TYPEREP(integer)
    else
      App (Const (STR "Groups.uminus-class.uminus") TYPEREP(integer ⇒ in-
teger))
      (term-of (- i)))
    ⟨proof⟩

```

```

code-reserved
  (Eval) HOLogic

```

78.4 Generic 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

```

⟨ML⟩

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* \Rightarrow *Random.seed* \Rightarrow (*a* \times (*unit* \Rightarrow *term*)) \times *Random.seed*

79.3 Fundamental and numeric types

instantiation *bool* :: *random*
begin

context
includes *state-combinator-syntax*
begin

definition
random i = *Random.range 2* $\circ \rightarrow$
 $(\lambda k. \text{Pair } (\text{if } k = 0 \text{ then } \text{Code-Evaluation.valtermify False else } \text{Code-Evaluation.valtermify True}))$

instance $\langle \text{proof} \rangle$

end

end

instantiation *itself* :: (*typerep*) *random*
begin

definition
random-itself :: *natural* \Rightarrow *Random.seed* \Rightarrow (*a itself* \times (*unit* \Rightarrow *term*)) \times *Random.seed*
where *random-itself* - = *Pair* (*Code-Evaluation.valtermify TYPE('a)*)

instance $\langle \text{proof} \rangle$

end

instantiation *char* :: *random*
begin

context
includes *state-combinator-syntax*
begin

definition

random - = *Random.select* (*Enum.enum* :: *char list*) $\circ\rightarrow$ ($\lambda c.$ *Pair* (*c*, $\lambda u.$ *Code-Evaluation.term-of* *c*))

instance $\langle proof \rangle$

end

end

instantiation *String.literal* :: *random*
begin

definition

random - = *Pair* (*STR* "", $\lambda u.$ *Code-Evaluation.term-of* (*STR* ""))

instance $\langle proof \rangle$

end

instantiation *nat* :: *random*
begin

context

includes *state-combinator-syntax*

begin

definition *random-nat* :: *natural* \Rightarrow *Random.seed*

\Rightarrow (*nat* \times (*unit* \Rightarrow *Code-Evaluation.term*)) \times *Random.seed*

where

random-nat *i* = *Random.range* (*i* + 1) $\circ\rightarrow$ ($\lambda k.$ *Pair* (
 let *n* = *nat-of-natural* *k*
 in (*n*, $\lambda-. \text{Code-Evaluation.term-of } n$)))

instance $\langle proof \rangle$

end

end

instantiation *int* :: *random*
begin

context

includes *state-combinator-syntax*

begin

definition

random *i* = *Random.range* ($2 * i + 1$) $\circ\rightarrow$ ($\lambda k.$ *Pair* (
 let *j* = (*if* *k* \geq *i* *then* *int* (*nat-of-natural* (*k* - *i*)) *else* - (*int* (*nat-of-natural* (*i*

```

- k))))
  in (j, λ-. Code-Evaluation.term-of j)))

instance ⟨proof⟩

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) ○→ (λn. Pair (n, λ-. Code-Evaluation.term-of
n))

instance ⟨proof⟩

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) ○→ (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 ⟨proof⟩

end

end

```

79.4 Complex generators

Towards $'a \Rightarrow 'b$

axiomatization *random-fun-aux* :: *typerep* \Rightarrow *typerep* \Rightarrow (*'a* \Rightarrow *'a* \Rightarrow *bool*) \Rightarrow (*'a* \Rightarrow *term*)
 \Rightarrow (*Random.seed* \Rightarrow (*'b* \times (*unit* \Rightarrow *term*)) \times *Random.seed*)
 \Rightarrow (*Random.seed* \Rightarrow *Random.seed* \times *Random.seed*)
 \Rightarrow *Random.seed* \Rightarrow ((*'a* \Rightarrow *'b*) \times (*unit* \Rightarrow *term*)) \times *Random.seed*

definition *random-fun-lift* :: (*Random.seed* \Rightarrow (*'b* \times (*unit* \Rightarrow *term*))) \times *Random.seed*
 \Rightarrow *Random.seed* \Rightarrow ((*'a*::*term-of* \Rightarrow *'b*::*typerep*) \times (*unit* \Rightarrow *term*)) \times *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* \Rightarrow *Random.seed* \Rightarrow ((*'a* \Rightarrow *'b*) \times (*unit* \Rightarrow *term*)) \times *Random.seed*
where *random i* = *random-fun-lift* (*random i*)

instance \langle *proof* \rangle

end

Towards type copies and datatypes

context
includes *state-combinator-syntax*
begin

definition *collapse* :: (*'a* \Rightarrow (*'a* \Rightarrow *'b* \times *'a*) \times *'a*) \Rightarrow *'a* \Rightarrow *'b* \times *'a*
where *collapse* *f* = (*f* $\circ \rightarrow$ *id*)

end

definition *beyond* :: *natural* \Rightarrow *natural* \Rightarrow *natural*
where *beyond* *k l* = (if *l* > *k* then *l* else 0)

lemma *beyond-zero*: *beyond* *k* 0 = 0
 \langle *proof* \rangle

context
includes *term-syntax*
begin

$$\text{valterm-emptyset} = \text{Code-Evaluation.valtermify } (\{\} :: ('a :: \text{typerep}) \text{ set})$$
$$\text{valtermify-insert } x \ s = \text{Code-Evaluation.valtermify insert } \{ \cdot \} \ (x :: ('a :: \text{typerep} * \rightarrow)) \{ \cdot \} \ s$$

```

instantiation set :: (random) random
begin

```

```
includes state-combinator-syntax
begin
```

$$\begin{aligned} & \text{random-aux-set } 0 \ j = \text{collapse } (\text{Random.select-weight } [(1, \text{Pair valterm-emptyset})]) \\ & | \text{ random-aux-set } (\text{Code-Numeral.Suc } i) \ j = \\ & \quad \text{collapse } (\text{Random.select-weight} \\ & \quad \quad [(1, \text{Pair valterm-emptyset}), \\ & \quad \quad (\text{Code-Numeral.Suc } i, \\ & \quad \quad \quad \text{random } j \circ \rightarrow (\%x. \text{random-aux-set } i \ j \circ \rightarrow (\%s. \text{Pair } (\text{valtermify-insert } x \\ & \quad \quad \quad s))))))]) \end{aligned}$$
$$\begin{aligned} & \text{random-aux-set } i \ j = \\ & \quad \text{collapse } (\text{Random.select-weight } [(1, \text{Pair valterm-emptyset}), \\ & \quad (i, \text{random } j \circ \rightarrow (\%x. \text{random-aux-set } (i - 1) \ j \circ \rightarrow (\%s. \text{Pair } (\text{valtermify-insert} \\ & \quad x \ s)))))) \\ & \langle \text{proof} \rangle \end{aligned}$$

end

⟨proof⟩

79.5 Deriving random generators for datatypes

$\langle ML \rangle$

79.6 Code setup

code-printing

constant *random-fun-aux* \rightarrow (*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 \Rightarrow natural \Rightarrow natural \Rightarrow *Random.seed* \Rightarrow ('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 \Rightarrow natural \Rightarrow *Random.seed* \Rightarrow ('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')*)))
 $\langle \text{proof} \rangle$

type-synonym 'a *random-pred* = *Random.seed* \Rightarrow ('a *Predicate.pred* \times *Random.seed*)

definition *empty* :: 'a *random-pred*

where *empty* = *Pair bot*

definition *single* :: 'a \Rightarrow 'a random-pred
where *single* *x* = *Pair* (*Predicate.single* *x*)

definition *bind* :: 'a random-pred \Rightarrow ('a \Rightarrow 'b random-pred) \Rightarrow '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 \Rightarrow 'a random-pred \Rightarrow '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 \Rightarrow unit random-pred
where
if-randompred *b* = (*if* *b* *then single* () *else empty*)

definition *iterate-upto* :: (natural \Rightarrow 'a) \Rightarrow natural \Rightarrow natural \Rightarrow 'a random-pred
where
iterate-upto *f n m* = *Pair* (*Predicate.iterate-upto* *f n m*)

definition *not-randompred* :: unit random-pred \Rightarrow 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 \Rightarrow ('a \times (unit \Rightarrow term)) \times Random.seed) \Rightarrow 'a random-pred
where *Random g* = *scomp g* (*Pair* \circ (*Predicate.single* \circ *fst*))

definition *map* :: ('a \Rightarrow 'b) \Rightarrow 'a random-pred \Rightarrow 'b random-pred
where *map f P* = *bind P* (*single* \circ *f*)

hide-const (**open**) *iter'* *iter empty single bind union if-randompred*
iterate-upto not-randompred Random map

hide-fact *iter'.simps*

hide-fact (**open**) *iter-def empty-def single-def bind-def union-def*
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
```

```
type-synonym 'a random-dseq = natural  $\Rightarrow$  natural  $\Rightarrow$  Random.seed  $\Rightarrow$  ('a Limited-Sequence.dseq  $\times$  Random.seed)
```

```
definition empty :: 'a random-dseq
where
  empty = (%nrandom size. Pair (Limited-Sequence.empty))
```

```
definition single :: 'a  $\Rightarrow$  'a random-dseq
where
  single x = (%nrandom size. Pair (Limited-Sequence.single x))
```

```
definition bind :: 'a random-dseq  $\Rightarrow$  ('a  $\Rightarrow$  'b random-dseq)  $\Rightarrow$  'b random-dseq
where
  bind R f = ( $\lambda$ nrandom size s. let
    (P, s') = R nrandom size s;
    (s1, s2) = Random.split-seed s'
  in (Limited-Sequence.bind P (%a. fst (f a nrandom size s1)), s2))
```

```
definition union :: 'a random-dseq  $\Rightarrow$  'a random-dseq  $\Rightarrow$  'a random-dseq
where
  union R1 R2 = ( $\lambda$ nrandom size s. let
    (S1, s') = R1 nrandom size s; (S2, s'') = R2 nrandom size s'
  in (Limited-Sequence.union S1 S2, s''))
```

```
definition if-random-dseq :: bool  $\Rightarrow$  unit random-dseq
where
  if-random-dseq b = (if b then single () else empty)
```

```
definition not-random-dseq :: unit random-dseq  $\Rightarrow$  unit random-dseq
where
  not-random-dseq R = ( $\lambda$ nrandom size s. let
    (S, s') = R nrandom size s
  in (Limited-Sequence.not-seq S, s'))
```

```
definition map :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a random-dseq  $\Rightarrow$  'b random-dseq
where
  map f P = bind P (single  $\circ$  f)
```

```
fun Random :: (natural  $\Rightarrow$  Random.seed  $\Rightarrow$  (('a  $\times$  (unit  $\Rightarrow$  term))  $\times$  Random.seed))
 $\Rightarrow$  'a random-dseq
where
  Random g nrandom = (%size. if nrandom  $\leq$  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 \Rightarrow natural \Rightarrow Random.seed \Rightarrow '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 \Rightarrow 'a pos-random-dseq

where

pos-single x = (%nrandom size seed. Limited-Sequence.pos-single x)

definition pos-bind :: 'a pos-random-dseq \Rightarrow ('a \Rightarrow 'b pos-random-dseq) \Rightarrow 'b pos-random-dseq

where

pos-bind R f = (λ nrandom size seed. Limited-Sequence.pos-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition pos-decr-bind :: 'a pos-random-dseq \Rightarrow ('a \Rightarrow 'b pos-random-dseq) \Rightarrow 'b pos-random-dseq

where

pos-decr-bind R f = (λ nrandom size seed. Limited-Sequence.pos-decr-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition pos-union :: 'a pos-random-dseq \Rightarrow 'a pos-random-dseq \Rightarrow 'a pos-random-dseq

where

pos-union R1 R2 = (λ nrandom size seed. Limited-Sequence.pos-union (R1 nrandom size seed) (R2 nrandom size seed))

definition pos-if-random-dseq :: bool \Rightarrow unit pos-random-dseq

where

pos-if-random-dseq b = (if b then pos-single () else pos-empty)

definition pos-iterate-upto :: (natural \Rightarrow 'a) \Rightarrow natural \Rightarrow natural \Rightarrow 'a pos-random-dseq

where

pos-iterate-upto f n m = (λ nrandom size seed. Limited-Sequence.pos-iterate-upto f n m)

definition pos-map :: ('a \Rightarrow 'b) \Rightarrow 'a pos-random-dseq \Rightarrow 'b pos-random-dseq

where

pos-map f P = pos-bind P (pos-single \circ f)

fun iter :: (Random.seed \Rightarrow ('a \times (unit \Rightarrow term)) \times Random.seed)

\Rightarrow natural \Rightarrow Random.seed \Rightarrow 'a Lazy-Sequence.lazy-sequence

where

iter random nrandom seed =
(if nrandom = 0 then Lazy-Sequence.empty else Lazy-Sequence.Lazy-Sequence
(%u. let ((x, -), seed') = random seed in Some (x, iter random (nrandom - 1)
seed'))))

definition *pos-Random* :: (*natural* \Rightarrow *Random.seed* \Rightarrow ('*a* \times (*unit* \Rightarrow *term*)) \times *Random.seed*)
 \Rightarrow '*a pos-random-dseq*

where

pos-Random g = (%nrandom size seed depth. iter (g size) nrandom seed)

type-synonym '*a neg-random-dseq* = *natural* \Rightarrow *natural* \Rightarrow *Random.seed* \Rightarrow '*a Limited-Sequence.neg-dseq*

definition *neg-empty* :: '*a neg-random-dseq*

where

neg-empty = (%nrandom size seed. Limited-Sequence.neg-empty)

definition *neg-single* :: '*a* \Rightarrow '*a neg-random-dseq*

where

neg-single x = (%nrandom size seed. Limited-Sequence.neg-single x)

definition *neg-bind* :: '*a neg-random-dseq* \Rightarrow ('*a* \Rightarrow '*b neg-random-dseq*) \Rightarrow '*b neg-random-dseq*

where

neg-bind R f = (λ nrandom size seed. Limited-Sequence.neg-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition *neg-decr-bind* :: '*a neg-random-dseq* \Rightarrow ('*a* \Rightarrow '*b neg-random-dseq*) \Rightarrow '*b neg-random-dseq*

where

neg-decr-bind R f = (λ nrandom size seed. Limited-Sequence.neg-decr-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition *neg-union* :: '*a neg-random-dseq* \Rightarrow '*a neg-random-dseq* \Rightarrow '*a neg-random-dseq*

where

neg-union R1 R2 = (λ nrandom size seed. Limited-Sequence.neg-union (R1 nrandom size seed) (R2 nrandom size seed))

definition *neg-if-random-dseq* :: *bool* \Rightarrow *unit neg-random-dseq*

where

neg-if-random-dseq b = (if b then neg-single () else neg-empty)

definition *neg-iterate-upto* :: (*natural* \Rightarrow '*a*) \Rightarrow *natural* \Rightarrow *natural* \Rightarrow '*a neg-random-dseq*

where

neg-iterate-upto f n m = (λ nrandom 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 = (λn random 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 o f)*

definition *pos-not-random-dseq* :: *unit neg-random-dseq* => *unit pos-random-dseq*
where
pos-not-random-dseq R = (λn random 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-empty-def pos-single-def pos-bind-def pos-decr-bind-def pos-union-def pos-if-random-dseq-def
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

82 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  $\Rightarrow$  (bool  $\times$  term list) option)  $\Rightarrow$  natural  $\Rightarrow$  (bool  $\times$  term list) option
```

```
class full-exhaustive = term-of +  
  fixes full-exhaustive ::  
    ('a  $\times$  (unit  $\Rightarrow$  term)  $\Rightarrow$  (bool  $\times$  term list) option)  $\Rightarrow$  natural  $\Rightarrow$  (bool  $\times$  term list) option
```

```
instantiation natural :: full-exhaustive  
begin
```

```
function full-exhaustive-natural' ::  
  (natural  $\times$  (unit  $\Rightarrow$  term)  $\Rightarrow$  (bool  $\times$  term list) option)  $\Rightarrow$   
  natural  $\Rightarrow$  natural  $\Rightarrow$  (bool  $\times$  term list) option  
  where full-exhaustive-natural' f d i =  
    (if d < i then None  
     else (f (i,  $\lambda$ -. Code-Evaluation.term-of i)) orelse (full-exhaustive-natural' f d  
      (i + 1)))  
  <proof>
```

```
termination  
  <proof>
```

```
definition full-exhaustive f d = full-exhaustive-natural' f d 0
```

```
instance <proof>
```

```
end
```

```
instantiation natural :: exhaustive  
begin
```

```
function exhaustive-natural' ::  
  (natural  $\Rightarrow$  (bool  $\times$  term list) option)  $\Rightarrow$  natural  $\Rightarrow$  natural  $\Rightarrow$  (bool  $\times$  term list) option  
  where exhaustive-natural' f d i =  
    (if d < i then None  
     else (f i orelse exhaustive-natural' f d (i + 1)))  
  <proof>
```

```
termination  
  <proof>
```

```
definition exhaustive f d = exhaustive-natural' f d 0
```

```
instance <proof>
```

```
end
```

instantiation *integer* :: *exhaustive*
begin

function *exhaustive-integer'* ::
 (*integer* \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *integer* \Rightarrow *integer* \Rightarrow (*bool* \times *term list*)
option
where *exhaustive-integer' f d i* =
 (*if d* < *i* *then* *None* *else* (*f i* *orelse* *exhaustive-integer' f d (i + 1)*))
<proof>

termination
<proof>

definition *exhaustive f d* = *exhaustive-integer' f (integer-of-natural d) (– (integer-of-natural d))*

instance *<proof>*

end

instantiation *integer* :: *full-exhaustive*
begin

function *full-exhaustive-integer'* ::
 (*integer* \times (*unit* \Rightarrow *term*) \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow
integer \Rightarrow *integer* \Rightarrow (*bool* \times *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 \Rightarrow *Some t*
 | *None* \Rightarrow *full-exhaustive-integer' f d (i + 1)*))
<proof>

termination
<proof>

definition *full-exhaustive f d* =
full-exhaustive-integer' f (integer-of-natural d) (– (integer-of-natural d))

instance *<proof>*

end

instantiation *nat* :: *exhaustive*
begin

definition *exhaustive f d* = *exhaustive (λx. f (nat-of-natural x)) d*

instance $\langle proof \rangle$

end

instantiation *nat* :: *full-exhaustive*
begin

definition *full-exhaustive* *f d* =
full-exhaustive ($\lambda(x, xt). f \text{ (nat-of-natural } x, \lambda-. \text{ Code-Evaluation.term-of (nat-of-natural } x))$) *d*

instance $\langle proof \rangle$

end

instantiation *int* :: *exhaustive*
begin

function *exhaustive-int'* ::
 (*int* \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *int* \Rightarrow *int* \Rightarrow (*bool* \times *term list*) *option*
where *exhaustive-int' f d i* =
 (*if* *d* < *i* *then* *None* *else* (*f i orelse exhaustive-int' f d (i + 1)*))
 $\langle proof \rangle$

termination
 $\langle proof \rangle$

definition *exhaustive* *f d* =
exhaustive-int' f (int-of-integer (integer-of-natural d))
 ($-$ (*int-of-integer (integer-of-natural d)*))

instance $\langle proof \rangle$

end

instantiation *int* :: *full-exhaustive*
begin

function *full-exhaustive-int'* ::
 (*int* \times (*unit* \Rightarrow *term*) \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow
int \Rightarrow *int* \Rightarrow (*bool* \times *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 \Rightarrow *Some t*
 | *None* \Rightarrow *full-exhaustive-int' f d (i + 1)*))
 $\langle proof \rangle$

termination

$\langle proof \rangle$

definition *full-exhaustive* $f\ d =$

full-exhaustive-int' $f\ (int\text{-of-integer}\ (integer\text{-of-natural}\ d))$
 $(- (int\text{-of-integer}\ (integer\text{-of-natural}\ d)))$

instance $\langle proof \rangle$

end

instantiation *prod* :: (*exhaustive*, *exhaustive*) *exhaustive*
begin

definition *exhaustive* $f\ d = exhaustive\ (\lambda x. exhaustive\ (\lambda y. f\ ((x, y)))\ d)\ d$

instance $\langle proof \rangle$

end

context

includes *term-syntax*

begin

definition

[code-unfold]: *valtermify-pair* $x\ y =$
Code-Evaluation.valtermify (*Pair* :: '*a*::*typerep* \Rightarrow '*b*::*typerep* \Rightarrow '*a* \times '*b*) $\{\cdot\}$ x
 $\{\cdot\}$ y

end

instantiation *prod* :: (*full-exhaustive*, *full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive* $f\ d =$

full-exhaustive $(\lambda x. full\text{-exhaustive}\ (\lambda y. f\ (valtermify\text{-pair}\ x\ y))\ d)\ d$

instance $\langle proof \rangle$

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 ⟨proof⟩
```

```
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 ⟨proof⟩
```

```
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 ⟨proof⟩
```

```
end
```

definition *[code-unfold]*:

```
valtermify-absdummy =
  (λ(v, t).
    (λ::'a. v,
      λu::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'
      (λg. full-exhaustive
        (λa. full-exhaustive (λb. f (valtermify-fun-upd g a b)) d) d) (i - 1) d
  else None)
```

definition *full-exhaustive-fun* ::

```
(('a ⇒ 'b) × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
  natural ⇒ (bool × term list) option
```

where *full-exhaustive-fun* f d = *full-exhaustive-fun'* f d d

instance *<proof>*

end

82.2.1 A smarter enumeration scheme for functions over finite datatypes

class *check-all* = *enum* + *term-of* +

fixes *check-all* :: ('a × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ (bool * term list) option

fixes *enum-term-of* :: 'a itself ⇒ unit ⇒ term list

```

fun check-all-n-lists :: ('a::check-all list  $\times$  (unit  $\Rightarrow$  term list)  $\Rightarrow$ 
  (bool  $\times$  term list) option)  $\Rightarrow$  natural  $\Rightarrow$  (bool * term list) option
where
  check-all-n-lists f n =
    (if n = 0 then f ([], ( $\lambda$ -. []))
     else check-all ( $\lambda$ (x, xt).
       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 =
    (Code-Evaluation.termify
      (fun-upd :: ('a::typerep  $\Rightarrow$  'b::typerep)  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  'a  $\Rightarrow$  'b)  $\langle \cdot \rangle$  g  $\langle \cdot \rangle$  a
       $\langle \cdot \rangle$  b)

end

definition mk-map-term ::
  (unit  $\Rightarrow$  typerep)  $\Rightarrow$  (unit  $\Rightarrow$  typerep)  $\Rightarrow$ 
  (unit  $\Rightarrow$  term list)  $\Rightarrow$  (unit  $\Rightarrow$  term list)  $\Rightarrow$  unit  $\Rightarrow$  term
  where mk-map-term T1 T2 domm rng =
    ( $\lambda$ -.
      let
        T1 = T1 ();
        T2 = T2 ();
        update-term =
          ( $\lambda$ g (a, b).
            Code-Evaluation.App (Code-Evaluation.App (Code-Evaluation.App
              (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 ())
(rng ())))

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-.
 let
 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 ⟨proof⟩

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* (λ*a*. λ*u* :: 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 <proof>

end

instantiation *unit* :: *check-all*
begin

definition *check-all* *f* = *f* (*Code-Evaluation.valtermify* ())

definition *enum-term-of-unit* :: unit itself ⇒ unit ⇒ term list
where *enum-term-of-unit* = (λ- -. [*Code-Evaluation.term-of* ()])

instance <proof>

end

instantiation *bool* :: *check-all*
begin

definition
check-all *f* =
 (*case* *f* (*Code-Evaluation.valtermify* *False*) of
 Some *x'* ⇒ *Some* *x'*
 | *None* ⇒ *f* (*Code-Evaluation.valtermify* *True*))

definition *enum-term-of-bool* :: *bool itself* \Rightarrow *unit* \Rightarrow *term list*

where *enum-term-of-bool* = (λ - *-*. *map* *Code-Evaluation.term-of* (*Enum.enum* :: *bool list*))

instance \langle *proof* \rangle

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*) $\langle \cdot \rangle$ *x*
 $\langle \cdot \rangle$ *y*

end

instantiation *prod* :: (*check-all*, *check-all*) *check-all*

begin

definition *check-all* *f* = *check-all* (λ *x*. *check-all* (λ *y*. *f* (*valtermify-pair* *x y*)))

definition *enum-term-of-prod* :: ('*a* * '*b*) *itself* \Rightarrow *unit* \Rightarrow *term list*

where *enum-term-of-prod* =

(λ - *-*.
map (λ (*x*, *y*). *termify-pair* *TYPE*('a) *TYPE*('b) *x y*)
List.product (*enum-term-of* (*TYPE*('a)) ()) (*enum-term-of* (*TYPE*('b))
 ()))

instance \langle *proof* \rangle

end

context

includes *term-syntax*

begin

definition

[*code-unfold*]: *valtermify-Inl* *x* =

Code-Evaluation.valtermify (*Inl* :: '*a*::*typerep* \Rightarrow '*a* + '*b* :: *typerep*) { \cdot } *x*

definition

[*code-unfold*]: *valtermify-Inr* *x* =

Code-Evaluation.valtermify (*Inr* :: '*b*::*typerep* \Rightarrow '*a*::*typerep* + '*b*) { \cdot } *x*

end

instantiation *sum* :: (*check-all*, *check-all*) *check-all*
begin

definition

check-all *f* = *check-all* ($\lambda a. f$ (*valtermify-Inl* *a*)) *orelse* *check-all* ($\lambda b. f$ (*valtermify-Inr* *b*))

definition *enum-term-of-sum* :: (*'a* + *'b*) *itself* \Rightarrow *unit* \Rightarrow *term list*

where *enum-term-of-sum* =
 ($\lambda -.$
 let
 T1 = *Typerep.typerep* (*TYPE('a)*);
 T2 = *Typerep.typerep* (*TYPE('b)*)
 in
 map
 (*Code-Evaluation.App* (*Code-Evaluation.Const* (*STR "Sum-Type.Inl"*)
 (*Typerep.Typerep* (*STR "fun"*) [*T1*, *Typerep.Typerep* (*STR "Sum-Type.sum"*)
 [*T1*, *T2*]])))
 (*enum-term-of* (*TYPE('a)*) ()) @
 map
 (*Code-Evaluation.App* (*Code-Evaluation.Const* (*STR "Sum-Type.Inr"*)
 (*Typerep.Typerep* (*STR "fun"*) [*T2*, *Typerep.Typerep* (*STR "Sum-Type.sum"*)
 [*T1*, *T2*]])))
 (*enum-term-of* (*TYPE('b)*) ()))

instance $\langle \text{proof} \rangle$

end

instantiation *char* :: *check-all*

begin

primrec *check-all-char'* ::

(*char* \times (*unit* \Rightarrow *term*) \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *char list* \Rightarrow (*bool* \times *term list*) *option*

where *check-all-char'* *f* [] = *None*

| *check-all-char'* *f* (*c* # *cs*) = *f* (*c*, $\lambda -.$ *Code-Evaluation.term-of* *c*)
 orelse *check-all-char'* *f* *cs*

definition *check-all-char* ::

(*char* \times (*unit* \Rightarrow *term*) \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow (*bool* \times *term list*) *option*
where *check-all* *f* = *check-all-char'* *f* *Enum.enum*

definition *enum-term-of-char* :: *char* *itself* \Rightarrow *unit* \Rightarrow *term list*

where

enum-term-of-char = ($\lambda -.$ *map* *Code-Evaluation.term-of* (*Enum.enum* :: *char list*))

instance $\langle proof \rangle$

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 \Rightarrow unit \Rightarrow 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 TYPE('a),
            Typerep.Typerep (STR "Option.option") [Typerep.typerep TYPE('a)]])))
    (enum-term-of (TYPE('a)) ())))

```

instance $\langle proof \rangle$

end

instantiation *Enum.finite-1* :: *check-all*
begin

definition *check-all* f = f (Code-Evaluation.valtermify *Enum.finite-1.a₁*)

definition *enum-term-of-finite-1* :: *Enum.finite-1* itself \Rightarrow unit \Rightarrow term list

where *enum-term-of-finite-1* = (λ- -. [Code-Evaluation.term-of *Enum.finite-1.a₁*])

instance $\langle proof \rangle$

end

instantiation *Enum.finite-2* :: *check-all*
begin

definition

check-all *f* =
 (*f* (*Code-Evaluation.valtermify* *Enum.finite-2.a₁*) *orelse*
f (*Code-Evaluation.valtermify* *Enum.finite-2.a₂*))

definition *enum-term-of-finite-2* :: *Enum.finite-2* *itself* \Rightarrow *unit* \Rightarrow *term list*
where *enum-term-of-finite-2* =
 (λ - *-*. *map* *Code-Evaluation.term-of* (*Enum.enum* :: *Enum.finite-2 list*))

instance \langle *proof* \rangle

end

instantiation *Enum.finite-3* :: *check-all*
begin

definition

check-all *f* =
 (*f* (*Code-Evaluation.valtermify* *Enum.finite-3.a₁*) *orelse*
f (*Code-Evaluation.valtermify* *Enum.finite-3.a₂*) *orelse*
f (*Code-Evaluation.valtermify* *Enum.finite-3.a₃*))

definition *enum-term-of-finite-3* :: *Enum.finite-3* *itself* \Rightarrow *unit* \Rightarrow *term list*
where *enum-term-of-finite-3* =
 (λ - *-*. *map* *Code-Evaluation.term-of* (*Enum.enum* :: *Enum.finite-3 list*))

instance \langle *proof* \rangle

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* \Rightarrow *unit* \Rightarrow *term list*
where *enum-term-of-finite-4* =
 (λ - *-*. *map* *Code-Evaluation.term-of* (*Enum.enum* :: *Enum.finite-4 list*))

instance \langle *proof* \rangle

end

82.3 Bounded universal quantifiers

```
class bounded-forall =
  fixes bounded-forall :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  natural  $\Rightarrow$  bool
```

82.4 Fast exhaustive combinators

```
class fast-exhaustive = term-of +
  fixes fast-exhaustive :: ('a  $\Rightarrow$  unit)  $\Rightarrow$  natural  $\Rightarrow$  unit
```

axiomatization *throw-Counterexample* :: term list \Rightarrow unit

axiomatization *catch-Counterexample* :: unit \Rightarrow term list option

code-printing

```
constant throw-Counterexample  $\rightarrow$ 
  (Quickcheck) raise (Exhaustive'-Generators.Counterexample -)
| constant catch-Counterexample  $\rightarrow$ 
  (Quickcheck) (((-); NONE) handle Exhaustive'-Generators.Counterexample ts
 $\Rightarrow$  SOME ts)
```

82.5 Continuation passing style functions as plus monad

type-synonym 'a cps = ('a \Rightarrow term list option) \Rightarrow term list option

definition *cps-empty* :: 'a cps
 where *cps-empty* = (λ cont. None)

definition *cps-single* :: 'a \Rightarrow 'a cps
 where *cps-single* v = (λ cont. cont v)

definition *cps-bind* :: 'a cps \Rightarrow ('a \Rightarrow 'b cps) \Rightarrow 'b cps
 where *cps-bind* m f = (λ cont. m (λ a. (f a) cont))

definition *cps-plus* :: 'a cps \Rightarrow 'a cps \Rightarrow 'a cps
 where *cps-plus* a b = (λ c. case a c of None \Rightarrow b c | Some x \Rightarrow Some x)

definition *cps-if* :: bool \Rightarrow unit cps
 where *cps-if* b = (if b then *cps-single* () else *cps-empty*)

definition *cps-not* :: unit cps \Rightarrow unit cps
 where *cps-not* n = (λ c. case n (λ u. Some []) of None \Rightarrow c () | Some - \Rightarrow None)

type-synonym 'a pos-bound-cps =
 ('a \Rightarrow (bool * term list) option) \Rightarrow natural \Rightarrow (bool * term list) option

definition *pos-bound-cps-empty* :: 'a pos-bound-cps
 where *pos-bound-cps-empty* = (λ cont i. None)

definition *pos-bound-cps-single* :: 'a \Rightarrow 'a *pos-bound-cps*

where *pos-bound-cps-single* *v* = ($\lambda \text{cont } i.$ *cont v*)

definition *pos-bound-cps-bind* :: 'a *pos-bound-cps* \Rightarrow ('a \Rightarrow 'b *pos-bound-cps*) \Rightarrow 'b *pos-bound-cps*

where *pos-bound-cps-bind* *m f* = ($\lambda \text{cont } i.$ *if* *i* = 0 *then* None *else* (*m* ($\lambda a.$ (*f a*) *cont i*) (*i* - 1)))

definition *pos-bound-cps-plus* :: 'a *pos-bound-cps* \Rightarrow 'a *pos-bound-cps* \Rightarrow 'a *pos-bound-cps*

where *pos-bound-cps-plus* *a b* = (λc *i.* *case a c i of* None \Rightarrow *b c i* | *Some x* \Rightarrow *Some x*)

definition *pos-bound-cps-if* :: *bool* \Rightarrow *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* \Rightarrow *term list three-valued*) \Rightarrow *natural* \Rightarrow *term list three-valued*

definition *neg-bound-cps-empty* :: 'a *neg-bound-cps*

where *neg-bound-cps-empty* = ($\lambda \text{cont } i.$ *No-value*)

definition *neg-bound-cps-single* :: 'a \Rightarrow 'a *neg-bound-cps*

where *neg-bound-cps-single* *v* = ($\lambda \text{cont } i.$ *cont* (*Known v*))

definition *neg-bound-cps-bind* :: 'a *neg-bound-cps* \Rightarrow ('a \Rightarrow 'b *neg-bound-cps*) \Rightarrow 'b *neg-bound-cps*

where *neg-bound-cps-bind* *m f* =

($\lambda \text{cont } i.$

if *i* = 0 *then* *cont Unknown*

else *m* ($\lambda a.$ *case a of* *Unknown* \Rightarrow *cont Unknown* | *Known a'* \Rightarrow *f a' cont i*)

(*i* - 1))

definition *neg-bound-cps-plus* :: 'a *neg-bound-cps* \Rightarrow 'a *neg-bound-cps* \Rightarrow 'a *neg-bound-cps*

where *neg-bound-cps-plus* *a b* =

(λc *i.*

case a c i of

No-value \Rightarrow *b c i*

| *Value x* \Rightarrow *Value x*

| *Unknown-value* \Rightarrow

(*case b c i of*

No-value \Rightarrow *Unknown-value*

| *Value x* \Rightarrow *Value x*)

| *Unknown-value* \Rightarrow *Unknown-value*))

definition *neg-bound-cps-if* :: *bool* \Rightarrow *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* \Rightarrow *unit neg-bound-cps*
where *neg-bound-cps-not* *n* =
 (λc *i*. *case* *n* (λu . *Some* (*True*, [])) *i* *of* *None* \Rightarrow *c* (*Known* ()) | *Some* - \Rightarrow *No-value*)

definition *pos-bound-cps-not* :: *unit neg-bound-cps* \Rightarrow *unit pos-bound-cps*
where *pos-bound-cps-not* *n* =
 (λc *i*. *case* *n* (λu . *Value* []) *i* *of* *No-value* \Rightarrow *c* () | *Value* - \Rightarrow *None* | *Unknown-value* \Rightarrow *None*)

82.6 Defining generators for any first-order data type

axiomatization *unknown* :: 'a

notation (**output**) *unknown* ($\langle ? \rangle$)

$\langle ML \rangle$

declare [[*quickcheck-batch-tester* = *exhaustive*]]

82.7 Defining generators for abstract types

$\langle ML \rangle$

hide-fact (**open**) *orelse-def*

no-notation *orelse* (**infixr** $\langle \text{orelse} \rangle$ 55)

hide-const *valtermify-absdummy valtermify-fun-upd*
valterm-emptyset valtermify-insert
valtermify-pair valtermify-Inl valtermify-Inr
termify-fun-upd term-emptyset termify-insert termify-pair setify

hide-const (**open**)

exhaustive full-exhaustive
exhaustive-int' full-exhaustive-int'
exhaustive-integer' full-exhaustive-integer'
exhaustive-natural' full-exhaustive-natural'
throw-Counterexample catch-Counterexample
check-all enum-term-of
orelse unknown mk-map-term check-all-n-lists check-all-subsets

hide-type (**open**) *cps pos-bound-cps neg-bound-cps unknown three-valued*

hide-const (**open**) *cps-empty cps-single cps-bind cps-plus cps-if cps-not*
pos-bound-cps-empty pos-bound-cps-single pos-bound-cps-bind

```

pos-bound-cps-plus pos-bound-cps-if pos-bound-cps-not
neg-bound-cps-empty neg-bound-cps-single neg-bound-cps-bind
neg-bound-cps-plus neg-bound-cps-if neg-bound-cps-not
Unknown Known Unknown-value Value No-value

```

end

83 A compiler for predicates defined by introduction rules

```

theory Predicate-Compile
imports Random-Sequence Quickcheck-Exhaustive
keywords
  code-pred :: thy-goal and
  values :: diag
begin

```

⟨ML⟩

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  $\longleftrightarrow$   $x \in A$ 

```

```

definition contains-pred :: 'a set => 'a => unit Predicate.pred
where contains-pred A x = (if  $x \in A$  then Predicate.single () else bot)

```

```

lemma pred-of-setE:
  assumes Predicate.eval (pred-of-set A) x
  obtains contains A x
⟨proof⟩

```

```

lemma pred-of-setI: contains A x ==> Predicate.eval (pred-of-set A) x
⟨proof⟩

```

```

lemma pred-of-set-eq: pred-of-set  $\equiv$   $\lambda A.$  Predicate.Pred (contains A)
⟨proof⟩

```

```

lemma containsI:  $x \in A ==>$  contains A x
⟨proof⟩

```

```

lemma containsE: assumes contains A x
  obtains A' x' where A = A' x = x'  $x \in A$ 
⟨proof⟩

```

```

lemma contains-predI: contains A x ==> Predicate.eval (contains-pred A x) ()

```

⟨proof⟩

lemma *contains-predE*:

assumes *Predicate.eval* (*contains-pred* *A* *x*) *y*

obtains *contains* *A* *x*

⟨proof⟩

lemma *contains-pred-eq*: *contains-pred* $\equiv \lambda A\ x.\ \text{Predicate.Pred } (\lambda y.\ \text{contains } A\ x)$

⟨proof⟩

lemma *contains-pred-notI*:

$\neg \text{contains } A\ x \implies \text{Predicate.eval } (\text{Predicate.not-pred } (\text{contains-pred } A\ x))\ ()$

⟨proof⟩

⟨ML⟩

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

⟨ML⟩

code-printing

code-module *Typerep* \rightarrow (*Haskell-Quickcheck*) \langle

module *Typerep*(*Typerep*(..)) *where*

data *Typerep* = *Typerep* *String* [*Typerep*]

\rangle **for type-constructor** *typerep* **constant** *Typerep.Typep*

 | **type-constructor** *typerep* \rightarrow (*Haskell-Quickcheck*) *Typerep.Typep*

 | **constant** *Typerep.Typep* \rightarrow (*Haskell-Quickcheck*) *Typerep.Typep*

code-reserved

 (*Haskell-Quickcheck*) *Typerep*

code-printing

```

type-constructor integer  $\rightarrow$  (Haskell-Quickcheck) Prelude.Int
| constant 0::integer  $\rightarrow$ 
  (Haskell-Quickcheck) !(0 / :: / Prelude.Int)

⟨ML⟩

```

```

code-printing
  constant Code-Numeral.push-bit  $\rightarrow$ 
    (Haskell-Quickcheck) Bit'-Shifts.drop'
| constant Code-Numeral.drop-bit  $\rightarrow$ 
  (Haskell-Quickcheck) Bit'-Shifts.push'

```

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  $\Rightarrow$  'a) list

```

```

primrec map-cons :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a narrowing-cons  $\Rightarrow$  'b narrowing-cons
where
  map-cons f (Narrowing-cons ty cs) = Narrowing-cons ty (map ( $\lambda c. f \circ c$ ) 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  $\Rightarrow$  narrowing-term  $\Rightarrow$  Code-Evaluation.term

```

```

lemma partial-term-of-anything: partial-term-of x nt  $\equiv$  t
  ⟨proof⟩

```

84.1.4 Auxiliary functions for Narrowing

```

consts nth :: 'a list  $\Rightarrow$  integer  $\Rightarrow$  'a

```

```

code-printing constant nth  $\rightarrow$  (Haskell-Quickcheck) infixl 9 !!

```

```

consts error :: char list  $\Rightarrow$  'a

```

```

code-printing constant error  $\rightarrow$  (Haskell-Quickcheck) error

```

```

consts toEnum :: integer  $\Rightarrow$  char

```

```

code-printing constant toEnum  $\rightarrow$  (Haskell-Quickcheck) Prelude.toEnum

```


consts *marker* :: *char*

code-printing constant *marker* \rightarrow (*Haskell-Quickcheck*) "\0"

84.1.5 Narrowing’s basic operations

type-synonym *'a narrowing* = *integer* \Rightarrow *'a narrowing-cons*

definition *cons* :: *'a* \Rightarrow *'a narrowing*

where

cons a d = (*Narrowing-cons* (*Narrowing-sum-of-products* []) [(λ -. *a*)])

fun *conv* :: (*narrowing-term list* \Rightarrow *'a*) *list* \Rightarrow *narrowing-term* \Rightarrow *'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* \Rightarrow *bool*

where

non-empty (*Narrowing-sum-of-products ps*) = (\neg (*List.null ps*))

definition *apply* :: (*'a* \Rightarrow *'b*) *narrowing* \Rightarrow *'a narrowing* \Rightarrow *'b narrowing*

where

apply f a d = (*if d* > 0 *then*

(*case f d of Narrowing-cons* (*Narrowing-sum-of-products ps*) *cfs* \Rightarrow

case a (*d* - 1) *of Narrowing-cons ta cas* \Rightarrow

let

shallow = *non-empty ta*;

cs = [(λ (*x* # *xs*) \Rightarrow *cf xs* (*conv cas x*)). *shallow*, *cf* \leftarrow *cfs*]

in Narrowing-cons (*Narrowing-sum-of-products* [*ta* # *p. shallow*, *p* \leftarrow *ps*])

cs)

else Narrowing-cons (*Narrowing-sum-of-products* [] [])

definition *sum* :: *'a narrowing* \Rightarrow *'a narrowing* \Rightarrow *'a narrowing*

where

sum a b d =

(*case a d of Narrowing-cons* (*Narrowing-sum-of-products ssa*) *ca* \Rightarrow

case b d of Narrowing-cons (*Narrowing-sum-of-products ssb*) *cb* \Rightarrow

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'*

<proof>

lemma [*fundef-cong*]:

assumes *f d* = *f' d* ($\bigwedge d'. 0 \leq d' \wedge d' < d \implies a d' = a' d'$)

assumes *d* = *d'*

shows $\text{apply } f \ a \ d = \text{apply } f' \ a' \ d'$
 $\langle \text{proof} \rangle$

84.1.6 Narrowing generator type class

```
class narrowing =
  fixes narrowing :: integer => 'a narrowing-cons

datatype (plugins only: code extraction) property =
  Universal narrowing-type (narrowing-term => property) narrowing-term =>
  Code-Evaluation.term
| Existential narrowing-type (narrowing-term => property) narrowing-term =>
  Code-Evaluation.term
| Property bool
```

definition $\text{exists} :: ('a :: \{\text{narrowing, partial-term-of}\} \Rightarrow \text{property}) \Rightarrow \text{property}$
where
 $\text{exists } f = (\text{case narrowing } (100 :: \text{integer}) \text{ of } \text{Narrowing-cons } \text{ty } \text{cs} \Rightarrow \text{Existential } \text{ty } (\lambda t. f (\text{conv } \text{cs } t)) (\text{partial-term-of } (\text{TYPE}('a))))$

definition $\text{all} :: ('a :: \{\text{narrowing, partial-term-of}\} \Rightarrow \text{property}) \Rightarrow \text{property}$
where
 $\text{all } f = (\text{case narrowing } (100 :: \text{integer}) \text{ of } \text{Narrowing-cons } \text{ty } \text{cs} \Rightarrow \text{Universal } \text{ty } (\lambda t. f (\text{conv } \text{cs } t)) (\text{partial-term-of } (\text{TYPE}('a))))$

84.1.7 class is-testable

The class *is-testable* ensures that all necessary type instances are generated.

```
class is-testable
```

```
instance bool :: is-testable  $\langle \text{proof} \rangle$ 
```

```
instance fun :: ({term-of, narrowing, partial-term-of}, is-testable) is-testable  $\langle \text{proof} \rangle$ 
```

definition $\text{ensure-testable} :: 'a :: \text{is-testable} \Rightarrow 'a :: \text{is-testable}$
where

$\text{ensure-testable } f = f$

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 $\text{eval-ffun} :: ('a, 'b) \text{ffun} \Rightarrow 'a \Rightarrow 'b$
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 <~~/src/HOL/Tools/Quickcheck/Narrowing-Engine.hs>
external-file <~~/src/HOL/Tools/Quickcheck/PNF-Narrowing-Engine.hs>
<ML>

```

```

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) narrow-
ing-cons
where
  narrowing-dummy-narrowing = narrowing

```

```

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)
<proof>

```

84.2 Narrowing for sets

```

instantiation set :: (narrowing) narrowing
begin

```

```

definition narrowing-set = Quickcheck-Narrowing.apply (Quickcheck-Narrowing.cons
set) narrowing

```

```

instance <proof>

```

end

84.3 Narrowing for integers

definition *drawn-from* :: 'a list \Rightarrow 'a narrowing-cons

where

drawn-from xs =
 Narrowing-cons (Narrowing-sum-of-products (map (λ -. []) xs)) (map (λ x -. x)
 xs)

function *around-zero* :: int \Rightarrow int list

where

around-zero i = (if i < 0 then [] else (if i = 0 then [0] else *around-zero* (i - 1)
 @ [i, -i]))
 <proof>

termination <proof>

declare *around-zero.simps* [simp del]

lemma *length-around-zero*:

assumes i \geq 0

shows length (*around-zero* i) = 2 * nat i + 1
 <proof>

instantiation int :: narrowing

begin

definition

narrowing-int d = (let (u :: - \Rightarrow - \Rightarrow unit) = conv; i = int-of-integer d
 in drawn-from (around-zero i))

instance <proof>

end

lemma [code]:

partial-term-of (ty :: int itself) (Narrowing-variable p t) \equiv
 Code-Evaluation.Free (STR "-") (Typerep.Typerep (STR "Int.int") [])
partial-term-of (ty :: int itself) (Narrowing-constructor i []) \equiv
 (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))
 <proof>

instantiation integer :: narrowing

begin

definition

narrowing-integer $d = (\text{let } (u :: - \Rightarrow - \Rightarrow \text{unit}) = \text{conv}; i = \text{int-of-integer } d$
in drawn-from (*map integer-of-int* (*around-zero* i)))

instance $\langle \text{proof} \rangle$

end

lemma $[\text{code}]$:

partial-term-of ($ty :: \text{integer itself}$) (*Narrowing-variable* $p\ t$) \equiv
Code-Evaluation.Free (*STR* $''-''$) (*Typerep.Typerep* (*STR* $''\text{Code-Numeral.integer}''$)
 $[]$)
partial-term-of ($ty :: \text{integer itself}$) (*Narrowing-constructor* $i\ []$) \equiv
 (*if* $i \bmod 2 = 0$
then *Code-Evaluation.term-of* ($- i \text{ div } 2$)
else *Code-Evaluation.term-of* ($(i + 1) \text{ div } 2$))
 $\langle \text{proof} \rangle$

code-printing constant *Code-Evaluation.term-of* $:: \text{integer} \Rightarrow \text{term} \rightarrow (\text{Haskell-Quickcheck})$

(*let* { $t = \text{Typerep.Typerep Code'-Numeral.integer } []$;
 $\text{mkFunT } s\ t = \text{Typerep.Typerep fun } [s, t]$;
 $\text{numT} = \text{Typerep.Typerep Num.num } []$;
 $\text{mkBit } 0 = \text{Generated'-Code.Const Num.num.Bit0 } (\text{mkFunT numT numT})$;
 $\text{mkBit } 1 = \text{Generated'-Code.Const Num.num.Bit1 } (\text{mkFunT numT numT})$;
 $\text{mkNumeral } 1 = \text{Generated'-Code.Const Num.num.One numT}$;
 $\text{mkNumeral } i = \text{let } \{ q = i \text{ 'Prelude.div' } 2; r = i \text{ 'Prelude.mod' } 2 \}$
in *Generated'-Code.App* (*mkBit* r) (*mkNumeral* q);
 $\text{mkNumber } 0 = \text{Generated'-Code.Const Groups.zero'-class.zero } t$;
 $\text{mkNumber } 1 = \text{Generated'-Code.Const Groups.one'-class.one } t$;
 $\text{mkNumber } i = \text{if } i > 0 \text{ then}$
 Generated'-Code.App
 (*Generated'-Code.Const Num.numeral'-class.numeral*
 (*mkFunT numT t*))
 (*mkNumeral i*)
else
 Generated'-Code.App
 (*Generated'-Code.Const Groups.uminus'-class.uminus* (*mkFunT t t*))
 (*mkNumber* ($- i$)); } *in* *mkNumber*)

84.4 The *find-unused-assms* command

$\langle \text{ML} \rangle$

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.simps non-empty.simps apply-def sum-def ensure-testable-def all-def exists-def*

end

theory *Mirabelle*

imports *Sledgehammer Predicate-Compile Presburger*

begin

$\langle ML \rangle$

end

85 Program extraction for HOL

theory *Extraction*

imports *Option*

begin

85.1 Setup

$\langle ML \rangle$

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

typeof (*Trueprop* *P*) \equiv *typeof* *P*

typeof *P* \equiv *Type* (*TYPE*(*Null*)) \implies *typeof* *Q* \equiv *Type* (*TYPE*('Q)) \implies
typeof (*P* \longrightarrow *Q*) \equiv *Type* (*TYPE*('Q))

typeof *Q* \equiv *Type* (*TYPE*(*Null*)) \implies *typeof* (*P* \longrightarrow *Q*) \equiv *Type* (*TYPE*(*Null*))

$$\begin{aligned}
& \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\
& \quad \text{typeof } (P \longrightarrow Q) \equiv \text{Type } (\text{TYPE}('P \Rightarrow 'Q)) \\
& (\lambda x. \text{typeof } (P x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& \quad \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 \\
& \quad \text{typeof } (\forall x::'a. P x) \equiv \text{Type } (\text{TYPE}('a \Rightarrow 'P)) \\
& (\lambda x. \text{typeof } (P x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& \quad \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 \\
& \quad \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 \\
& \quad \text{typeof } (P \vee 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 \\
& \quad \text{typeof } (P \vee 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 \\
& \quad \text{typeof } (P \vee 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 \\
& \quad \text{typeof } (P \vee 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 \\
& \quad \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('Q)) \\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
& \quad \text{typeof } (P \wedge 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}('Q)) \implies \\
& \quad \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('P \times 'Q)) \\
& \text{typeof } (P = Q) \equiv \text{typeof } ((P \longrightarrow Q) \wedge (Q \longrightarrow P)) \\
& \text{typeof } (x \in P) \equiv \text{typeof } P
\end{aligned}$$

85.3 Realizability

realizability

$$(\text{realizes } t \text{ (Trueprop } P)) \equiv (\text{Trueprop } (\text{realizes } t P))$$

$$\begin{aligned}
& (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \text{ (} P \longrightarrow Q)) \equiv (\text{realizes } \text{Null } P \longrightarrow \text{realizes } t Q)
\end{aligned}$$

$$\begin{aligned}
& (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}('P))) \implies \\
& (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (P \longrightarrow Q)) \equiv (\forall x::'P. \text{realizes } x \ P \longrightarrow \text{realizes } \text{Null } Q) \\
\\
& (\text{realizes } t \ (P \longrightarrow Q)) \equiv (\forall x. \text{realizes } x \ P \longrightarrow \text{realizes } (t \ x) \ Q) \\
\\
& (\lambda x. \text{typeof } (P \ x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (\forall x. P \ x)) \equiv (\forall x. \text{realizes } \text{Null } (P \ x)) \\
\\
& (\text{realizes } t \ (\forall x. P \ x)) \equiv (\forall x. \text{realizes } (t \ x) \ (P \ x)) \\
\\
& (\lambda x. \text{typeof } (P \ x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (\exists x. P \ x)) \equiv (\text{realizes } \text{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 \\
& (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (P \vee Q)) \equiv \\
& (\text{case } t \text{ of 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 \\
& (\text{realizes } t \ (P \vee Q)) \equiv \\
& (\text{case } t \text{ of 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 \\
& (\text{realizes } t \ (P \vee Q)) \equiv \\
& (\text{case } t \text{ of None } \Rightarrow \text{realizes } \text{Null } Q \mid \text{Some } p \Rightarrow \text{realizes } p \ P) \\
\\
& (\text{realizes } t \ (P \vee Q)) \equiv \\
& (\text{case } t \text{ of 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 \\
& (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } \text{Null } P \wedge \text{realizes } t \ Q) \\
\\
& (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } t \ P \wedge \text{realizes } \text{Null } Q) \\
\\
& (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } (\text{fst } t) \ P \wedge \text{realizes } (\text{snd } t) \ Q) \\
\\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
& \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::'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
\end{aligned}$$

$$(\text{realizes } t \ (P = Q)) \equiv (\text{realizes } t \ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$$

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 $r1$: $\bigwedge p. P \ p \Longrightarrow R \ (f \ p)$ **and** $r2$: $\bigwedge q. Q \ q \Longrightarrow R \ (g \ q)$
shows $R \ (\text{case } x \text{ of } \text{Inl } p \Rightarrow f \ p \mid \text{Inr } q \Rightarrow g \ q)$
 $\langle \text{proof} \rangle$

theorem *disjE-realizer2*:

assumes r : $\text{case } x \text{ of } \text{None} \Rightarrow P \mid \text{Some } q \Rightarrow Q \ q$
and $r1$: $P \Longrightarrow R \ f$ **and** $r2$: $\bigwedge q. Q \ q \Longrightarrow R \ (g \ q)$
shows $R \ (\text{case } x \text{ of } \text{None} \Rightarrow f \mid \text{Some } q \Rightarrow g \ q)$
 $\langle \text{proof} \rangle$

theorem *disjE-realizer3*:

assumes r : $\text{case } x \text{ of } \text{Left} \Rightarrow P \mid \text{Right} \Rightarrow Q$
and $r1$: $P \Longrightarrow R \ f$ **and** $r2$: $Q \Longrightarrow R \ g$
shows $R \ (\text{case } x \text{ of } \text{Left} \Rightarrow f \mid \text{Right} \Rightarrow g)$
 $\langle \text{proof} \rangle$

theorem *conjI-realizer*:

$P \ p \Longrightarrow Q \ q \Longrightarrow P \ (fst \ (p, \ q)) \wedge Q \ (snd \ (p, \ q))$
 $\langle \text{proof} \rangle$

theorem *exI-realizer*:

$P \ y \ x \Longrightarrow P \ (snd \ (x, \ y)) \ (fst \ (x, \ y)) \ \langle \text{proof} \rangle$

theorem *exE-realizer*: $P \ (snd \ p) \ (fst \ p) \Longrightarrow$

$(\bigwedge x \ y. P \ y \ x \Longrightarrow Q \ (f \ x \ y)) \Longrightarrow Q \ (\text{let } (x, \ y) = p \text{ in } f \ x \ y)$
 $\langle \text{proof} \rangle$

theorem *exE-realizer'*: $P \ (snd \ p) \ (fst \ p) \Longrightarrow$

$(\bigwedge x \ y. P \ y \ x \Longrightarrow Q) \Longrightarrow Q \ \langle \text{proof} \rangle$

realizers

$\text{impI} \ (P, \ Q): \lambda pq. \ pq$
 $\lambda(c: -) \ (d: -) \ P \ Q \ pq \ (h: -). \ \text{allI} \ \dots \ c \cdot (\lambda x. \ \text{impI} \ \dots \ (h \cdot x))$

$\text{impI} \ (P): \text{Null}$
 $\lambda(c: -) \ P \ Q \ (h: -). \ \text{allI} \ \dots \ c \cdot (\lambda x. \ \text{impI} \ \dots \ (h \cdot x))$

$\text{impI} \ (Q): \lambda q. \ q \ \lambda(c: -) \ P \ Q \ q. \ \text{impI} \ \dots$

$\text{impI}: \text{Null} \ \text{impI}$

$\text{mp} \ (P, \ Q): \lambda pq. \ pq$

$$\lambda(c: -) (d: -) P Q pq (h: -) p. mp \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot c \cdot h)$$

$$mp (P): Null$$

$$\lambda(c: -) P Q (h: -) p. mp \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot c \cdot h)$$

$$mp (Q): \lambda q. q \lambda(c: -) P Q q. mp \cdot - \cdot -$$

$$mp: Null mp$$

$$allI (P): \lambda p. p \lambda(c: -) P (d: -) p. allI \cdot - \cdot d$$

$$allI: Null allI$$

$$spec (P): \lambda x p. p x \lambda(c: -) P x (d: -) p. spec \cdot - \cdot x \cdot d$$

$$spec: Null spec$$

$$exI (P): \lambda x p. (x, p) \lambda(c: -) P x (d: -) p. exI\text{-realizer} \cdot P \cdot p \cdot x \cdot c \cdot d$$

$$exI: \lambda x. x \lambda P x (c: -) (h: -). h$$

$$exE (P, Q): \lambda p pq. let (x, y) = p in pq x y$$

$$\lambda(c: -) (d: -) P Q (e: -) p (h: -) pq. exE\text{-realizer} \cdot P \cdot p \cdot Q \cdot pq \cdot c \cdot e \cdot d \cdot h$$

$$exE (P): Null$$

$$\lambda(c: -) P Q (d: -) p. exE\text{-realizer}' \cdot - \cdot - \cdot - \cdot c \cdot d$$

$$exE (Q): \lambda x pq. pq x$$

$$\lambda(c: -) P Q (d: -) x (h1: -) pq (h2: -). h2 \cdot x \cdot h1$$

$$exE: Null$$

$$\lambda P Q (c: -) x (h1: -) (h2: -). h2 \cdot x \cdot h1$$

$$conjI (P, Q): Pair$$

$$\lambda(c: -) (d: -) P Q p (h: -) q. conjI\text{-realizer} \cdot P \cdot p \cdot Q \cdot q \cdot c \cdot d \cdot h$$

$$conjI (P): \lambda p. p$$

$$\lambda(c: -) P Q p. conjI \cdot - \cdot -$$

$$conjI (Q): \lambda q. q$$

$$\lambda(c: -) P Q (h: -) q. conjI \cdot - \cdot - \cdot h$$

$$conjI: Null conjI$$

$$conjunct1 (P, Q): fst$$

$$\lambda(c: -) (d: -) P Q pq. conjunct1 \cdot - \cdot -$$

$$conjunct1 (P): \lambda p. p$$

$$\lambda(c: -) P Q p. conjunct1 \cdot - \cdot -$$

conjunct1 (*Q*): *Null*
 $\lambda(c: -) P Q q. \text{conjunct1} \cdot \cdot \cdot$

conjunct1: *Null conjunct1*

conjunct2 (*P*, *Q*): *snd*
 $\lambda(c: -) (d: -) P Q pq. \text{conjunct2} \cdot \cdot \cdot$

conjunct2 (*P*): *Null*
 $\lambda(c: -) P Q p. \text{conjunct2} \cdot \cdot \cdot$

conjunct2 (*Q*): $\lambda p. p$
 $\lambda(c: -) P Q p. \text{conjunct2} \cdot \cdot \cdot$

conjunct2: *Null conjunct2*

disjI1 (*P*, *Q*): *Inl*
 $\lambda(c: -) (d: -) P Q p. \text{iffD2} \cdot \cdot \cdot \cdot (\text{sum.case-1} \cdot P \cdot \cdot \cdot p \cdot \text{arity-type-bool} \cdot c \cdot$
d)

disjI1 (*P*): *Some*
 $\lambda(c: -) P Q p. \text{iffD2} \cdot \cdot \cdot \cdot (\text{option.case-2} \cdot \cdot \cdot P \cdot p \cdot \text{arity-type-bool} \cdot c)$

disjI1 (*Q*): *None*
 $\lambda(c: -) P Q. \text{iffD2} \cdot \cdot \cdot \cdot (\text{option.case-1} \cdot \cdot \cdot \cdot \text{arity-type-bool} \cdot c)$

disjI1: *Left*
 $\lambda P Q. \text{iffD2} \cdot \cdot \cdot \cdot (\text{sumbool.case-1} \cdot \cdot \cdot \cdot \text{arity-type-bool})$

disjI2 (*P*, *Q*): *Inr*
 $\lambda(d: -) (c: -) Q P q. \text{iffD2} \cdot \cdot \cdot \cdot (\text{sum.case-2} \cdot \cdot \cdot Q \cdot q \cdot \text{arity-type-bool} \cdot c \cdot$
d)

disjI2 (*P*): *None*
 $\lambda(c: -) Q P. \text{iffD2} \cdot \cdot \cdot \cdot (\text{option.case-1} \cdot \cdot \cdot \cdot \text{arity-type-bool} \cdot c)$

disjI2 (*Q*): *Some*
 $\lambda(c: -) Q P q. \text{iffD2} \cdot \cdot \cdot \cdot (\text{option.case-2} \cdot \cdot \cdot Q \cdot q \cdot \text{arity-type-bool} \cdot c)$

disjI2: *Right*
 $\lambda Q P. \text{iffD2} \cdot \cdot \cdot \cdot (\text{sumbool.case-2} \cdot \cdot \cdot \cdot \text{arity-type-bool})$

disjE (*P*, *Q*, *R*): $\lambda pq pr qr.$
 $(\text{case } pq \text{ of } \text{Inl } p \Rightarrow pr \ p \mid \text{Inr } q \Rightarrow qr \ q)$
 $\lambda(c: -) (d: -) (e: -) P Q R pq (h1: -) pr (h2: -) qr.$
 $\text{disjE-realizer} \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot e \cdot h1 \cdot h2$

disjE (*Q*, *R*): $\lambda pq pr qr.$

(*case* pq of *None* $\Rightarrow pr$ | *Some* $q \Rightarrow qr$ q)
 $\lambda(c: -) (d: -) P Q R pq (h1: -) pr (h2: -) qr.$
 $disjE\text{-realizer2} \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot h1 \cdot h2$

$disjE (P, R): \lambda pq pr qr.$
 (*case* pq of *None* $\Rightarrow qr$ | *Some* $p \Rightarrow pr$ p)
 $\lambda(c: -) (d: -) P Q R pq (h1: -) pr (h2: -) qr (h3: -).$
 $disjE\text{-realizer2} \cdot \cdot \cdot \cdot pq \cdot R \cdot qr \cdot pr \cdot c \cdot d \cdot h1 \cdot h3 \cdot h2$

$disjE (R): \lambda pq pr qr.$
 (*case* pq of *Left* $\Rightarrow pr$ | *Right* $\Rightarrow qr$)
 $\lambda(c: -) P Q R pq (h1: -) pr (h2: -) qr.$
 $disjE\text{-realizer3} \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot c \cdot h1 \cdot h2$

$disjE (P, Q): \text{Null}$
 $\lambda(c: -) (d: -) P Q R pq. disjE\text{-realizer} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot d \cdot$
 $arity\text{-type}\text{-bool}$

$disjE (Q): \text{Null}$
 $\lambda(c: -) P Q R pq. disjE\text{-realizer2} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot arity\text{-type}\text{-bool}$

$disjE (P): \text{Null}$
 $\lambda(c: -) P Q R pq (h1: -) (h2: -) (h3: -).$
 $disjE\text{-realizer2} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot arity\text{-type}\text{-bool} \cdot h1 \cdot h3 \cdot h2$

$disjE: \text{Null}$
 $\lambda P Q R pq. disjE\text{-realizer3} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot arity\text{-type}\text{-bool}$

$FalseE (P): \text{default}$
 $\lambda(c: -) P. FalseE \cdot -$

$FalseE: \text{Null FalseE}$

$notI (P): \text{Null}$
 $\lambda(c: -) P (h: -). allI \cdot \cdot \cdot \cdot c \cdot (\lambda x. notI \cdot \cdot \cdot \cdot (h \cdot x))$

$notI: \text{Null notI}$

$notE (P, R): \lambda p. \text{default}$
 $\lambda(c: -) (d: -) P R (h: -) p. notE \cdot \cdot \cdot \cdot (spec \cdot \cdot \cdot \cdot p \cdot c \cdot h)$

$notE (P): \text{Null}$
 $\lambda(c: -) P R (h: -) p. notE \cdot \cdot \cdot \cdot (spec \cdot \cdot \cdot \cdot p \cdot c \cdot h)$

$notE (R): \text{default}$
 $\lambda(c: -) P R. notE \cdot \cdot \cdot \cdot$

$notE: \text{Null notE}$

subst (*P*): $\lambda s \ t \ ps. \ ps$
 $\lambda(c: -) \ s \ t \ P \ (d: -) \ (h: -) \ ps. \ subst \cdot s \cdot t \cdot P \ ps \cdot d \cdot h$

subst: *Null subst*

iffD1 (*P*, *Q*): *fst*
 $\lambda(d: -) \ (c: -) \ Q \ P \ pq \ (h: -) \ p.$
 $mp \cdot - \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot d \cdot (conjunct1 \cdot - \cdot - \cdot h))$

iffD1 (*P*): $\lambda p. \ p$
 $\lambda(c: -) \ Q \ P \ p \ (h: -). \ mp \cdot - \cdot - \cdot - \cdot (conjunct1 \cdot - \cdot - \cdot h)$

iffD1 (*Q*): *Null*
 $\lambda(c: -) \ Q \ P \ q1 \ (h: -) \ q2.$
 $mp \cdot - \cdot - \cdot - \cdot (spec \cdot - \cdot q2 \cdot c \cdot (conjunct1 \cdot - \cdot - \cdot h))$

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*
 $\lambda(c: -) \ (d: -) \ P \ Q \ pq \ (h: -) \ q.$
 $mp \cdot - \cdot - \cdot - \cdot (spec \cdot - \cdot q \cdot d \cdot (conjunct2 \cdot - \cdot - \cdot h))$

iffD2 (*P*): $\lambda p. \ p$
 $\lambda(c: -) \ P \ Q \ p \ (h: -). \ mp \cdot - \cdot - \cdot - \cdot (conjunct2 \cdot - \cdot - \cdot h)$

iffD2 (*Q*): *Null*
 $\lambda(c: -) \ P \ Q \ q1 \ (h: -) \ q2.$
 $mp \cdot - \cdot - \cdot - \cdot (spec \cdot - \cdot q2 \cdot c \cdot (conjunct2 \cdot - \cdot - \cdot h))$

iffD2: *Null iffD2*

iffI (*P*, *Q*): *Pair*
 $\lambda(c: -) \ (d: -) \ P \ Q \ pq \ (h1 : -) \ qp \ (h2 : -). \ conjI\text{-realizer} \cdot$
 $(\lambda pq. \ \forall x. \ P \ x \longrightarrow Q \ (pq \ x)) \cdot pq \cdot$
 $(\lambda qp. \ \forall x. \ Q \ x \longrightarrow P \ (qp \ x)) \cdot qp \cdot$
 $(arity\text{-type}\text{-fun} \cdot c \cdot d) \cdot$
 $(arity\text{-type}\text{-fun} \cdot d \cdot c) \cdot$
 $(allI \cdot - \cdot - \cdot c \cdot (\lambda x. \ impI \cdot - \cdot - \cdot (h1 \cdot x))) \cdot$
 $(allI \cdot - \cdot - \cdot d \cdot (\lambda x. \ impI \cdot - \cdot - \cdot (h2 \cdot x)))$

iffI (*P*): $\lambda p. \ p$
 $\lambda(c: -) \ P \ Q \ (h1 : -) \ p \ (h2 : -). \ conjI \cdot - \cdot - \cdot - \cdot$
 $(allI \cdot - \cdot - \cdot c \cdot (\lambda x. \ impI \cdot - \cdot - \cdot (h1 \cdot x))) \cdot$
 $(impI \cdot - \cdot - \cdot h2)$

iffI (*Q*): $\lambda q. \ q$
 $\lambda(c: -) \ P \ Q \ q \ (h1 : -) \ (h2 : -). \ conjI \cdot - \cdot - \cdot - \cdot$
 $(impI \cdot - \cdot - \cdot h1) \cdot$
 $(allI \cdot - \cdot - \cdot c \cdot (\lambda x. \ impI \cdot - \cdot - \cdot (h2 \cdot x)))$

```

    iffI: Null iffI
end

```

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 $\alpha (beta\text{-update } f \text{ } rec) = \alpha \text{ } rec$ for distinct fields α and β of some record rec with n fields. There are $n \wedge 2$ 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 $O(\log(n) \wedge 2)$ 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 $'a \times ('b \times ('c \times 'd))$. If we balance the tuple tree to $('a \times 'b) \times ('c \times 'd)$ then accessors can be defined by converting to the underlying type then using $O(\log(n))$ *fst* or *snd* operations. Updaters 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 \Rightarrow 'b \times 'c 'b \times 'c \Rightarrow 'a

primrec

repr :: (*'a, 'b, 'c*) *tuple-isomorphism* \Rightarrow *'a \Rightarrow 'b \times 'c* **where**
repr (*Tuple-Isomorphism r a*) = *r*

primrec

abst :: (*'a, 'b, 'c*) *tuple-isomorphism* \Rightarrow *'b \times 'c \Rightarrow 'a* **where**
abst (*Tuple-Isomorphism r a*) = *a*

definition

iso-tuple-fst :: (*'a, 'b, 'c*) *tuple-isomorphism* \Rightarrow *'a \Rightarrow 'b* **where**
iso-tuple-fst isom = *fst \circ repr isom*

definition

iso-tuple-snd :: (*'a, 'b, 'c*) *tuple-isomorphism* \Rightarrow *'a \Rightarrow 'c* **where**
iso-tuple-snd isom = *snd \circ repr isom*

definition

iso-tuple-fst-update ::
(*'a, 'b, 'c*) *tuple-isomorphism* \Rightarrow (*'b \Rightarrow 'b*) \Rightarrow (*'a \Rightarrow 'a*) **where**
iso-tuple-fst-update isom f = *abst isom \circ apfst f \circ repr isom*

definition

iso-tuple-snd-update ::
(*'a, 'b, 'c*) *tuple-isomorphism* \Rightarrow (*'c \Rightarrow 'c*) \Rightarrow (*'a \Rightarrow 'a*) **where**
iso-tuple-snd-update isom f = *abst isom \circ apsnd f \circ repr isom*

definition

iso-tuple-cons ::
 (*'a*, *'b*, *'c*) *tuple-isomorphism* \Rightarrow *'b* \Rightarrow *'c* \Rightarrow *'a* **where**
iso-tuple-cons isom = *curry (abst isom)*

86.3 Logical infrastructure for records**definition**

iso-tuple-surjective-proof-assist :: *'a* \Rightarrow *'b* \Rightarrow (*'a* \Rightarrow *'b*) \Rightarrow *bool* **where**
iso-tuple-surjective-proof-assist *x y f* \longleftrightarrow *f x = y*

definition

iso-tuple-update-accessor-cong-assist ::
 ((*'b* \Rightarrow *'b*) \Rightarrow (*'a* \Rightarrow *'a*)) \Rightarrow (*'a* \Rightarrow *'b*) \Rightarrow *bool* **where**
iso-tuple-update-accessor-cong-assist upd ac \longleftrightarrow
 ($\forall f v. \text{upd } (\lambda x. f (ac v)) v = \text{upd } f v$) \wedge ($\forall v. \text{upd } id v = v$)

definition

iso-tuple-update-accessor-eq-assist ::
 ((*'b* \Rightarrow *'b*) \Rightarrow (*'a* \Rightarrow *'a*)) \Rightarrow (*'a* \Rightarrow *'b*) \Rightarrow *'a* \Rightarrow (*'b* \Rightarrow *'b*) \Rightarrow *'a* \Rightarrow *'b* \Rightarrow *bool*
where
iso-tuple-update-accessor-eq-assist upd ac v f v' x \longleftrightarrow
upd f v = v' \wedge ac v = x \wedge 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*: $\bigwedge v. v' = v \implies f v = f' v$
shows *upd f r = upd f' r'*
<proof>

lemma *update-accessor-congruence-unfoldE*:

iso-tuple-update-accessor-cong-assist upd ac \implies
 $r = r' \implies ac r' = v' \implies (\bigwedge v. v = v' \implies f v = f' v) \implies$
 $upd f r = upd f' r'$
<proof>

lemma *iso-tuple-update-accessor-cong-assist-id*:

iso-tuple-update-accessor-cong-assist upd ac $\implies upd id = id$
<proof>

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*
<proof>

lemma *update-accessor-noop-compE*:

assumes *uac*: *iso-tuple-update-accessor-cong-assist upd ac*

and $ac: f (ac\ x) = ac\ x$
shows $upd\ (g \circ f)\ x = upd\ g\ x$
 $\langle proof \rangle$

lemma *update-accessor-cong-assist-idI*:
iso-tuple-update-accessor-cong-assist id id
 $\langle proof \rangle$

lemma *update-accessor-cong-assist-triv*:
iso-tuple-update-accessor-cong-assist upd ac \implies
iso-tuple-update-accessor-cong-assist upd ac
 $\langle proof \rangle$

lemma *update-accessor-accessor-eqE*:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies ac v = x
 $\langle proof \rangle$

lemma *update-accessor-updator-eqE*:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies upd f v = v'
 $\langle proof \rangle$

lemma *iso-tuple-update-accessor-eq-assist-idI*:
v' = f v \implies iso-tuple-update-accessor-eq-assist id id v f v' v
 $\langle proof \rangle$

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
 $\langle proof \rangle$

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
 $\langle proof \rangle$

lemma *iso-tuple-surjective-proof-assistI*:
f x = y \implies iso-tuple-surjective-proof-assist x y f
 $\langle proof \rangle$

lemma *iso-tuple-surjective-proof-assist-idE*:
iso-tuple-surjective-proof-assist x y id \implies x = y
 $\langle proof \rangle$

locale *isomorphic-tuple* =
fixes *isom* :: ('a, 'b, 'c) *tuple-isomorphism*
assumes *repr-inv*: $\bigwedge x. abst\ isom\ (repr\ isom\ x) = x$
and *abst-inv*: $\bigwedge y. repr\ isom\ (abst\ isom\ y) = y$
begin

lemma *repr-inj*: *repr isom* $x = \text{repr isom } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *abst-inj*: *abst isom* $x = \text{abst isom } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemmas *simps* = *Let-def repr-inv abst-inv repr-inj abst-inj*

lemma *iso-tuple-access-update-fst-fst*:
 $f \circ h \ g = j \circ f \implies$
 $(f \circ \text{iso-tuple-fst isom}) \circ (\text{iso-tuple-fst-update isom} \circ h) \ g =$
 $j \circ (f \circ \text{iso-tuple-fst isom})$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-access-update-snd-snd*:
 $f \circ h \ g = j \circ f \implies$
 $(f \circ \text{iso-tuple-snd isom}) \circ (\text{iso-tuple-snd-update isom} \circ h) \ g =$
 $j \circ (f \circ \text{iso-tuple-snd isom})$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-access-update-fst-snd*:
 $(f \circ \text{iso-tuple-fst isom}) \circ (\text{iso-tuple-snd-update isom} \circ h) \ g =$
 $\text{id} \circ (f \circ \text{iso-tuple-fst isom})$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-access-update-snd-fst*:
 $(f \circ \text{iso-tuple-snd isom}) \circ (\text{iso-tuple-fst-update isom} \circ h) \ g =$
 $\text{id} \circ (f \circ \text{iso-tuple-snd isom})$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-swap-fst-fst*:
 $h \ f \circ j \ g = j \ g \circ h \ f \implies$
 $(\text{iso-tuple-fst-update isom} \circ h) \ f \circ (\text{iso-tuple-fst-update isom} \circ j) \ g =$
 $(\text{iso-tuple-fst-update isom} \circ j) \ g \circ (\text{iso-tuple-fst-update isom} \circ h) \ f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-swap-snd-snd*:
 $h \ f \circ j \ g = j \ g \circ h \ f \implies$
 $(\text{iso-tuple-snd-update isom} \circ h) \ f \circ (\text{iso-tuple-snd-update isom} \circ j) \ g =$
 $(\text{iso-tuple-snd-update isom} \circ j) \ g \circ (\text{iso-tuple-snd-update isom} \circ h) \ f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-swap-fst-snd*:
 $(\text{iso-tuple-snd-update isom} \circ h) \ f \circ (\text{iso-tuple-fst-update isom} \circ j) \ g =$
 $(\text{iso-tuple-fst-update isom} \circ j) \ g \circ (\text{iso-tuple-snd-update isom} \circ h) \ f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-swap-snd-fst*:
 $(\text{iso-tuple-fst-update isom} \circ h) \ f \circ (\text{iso-tuple-snd-update isom} \circ j) \ g =$

$(\text{iso-tuple-snd-update isom} \circ j) \ g \circ (\text{iso-tuple-fst-update isom} \circ h) \ f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-compose-fst-fst:*

$h \ f \circ j \ g = k \ (f \circ g) \implies$
 $(\text{iso-tuple-fst-update isom} \circ h) \ f \circ (\text{iso-tuple-fst-update isom} \circ j) \ g =$
 $(\text{iso-tuple-fst-update isom} \circ k) \ (f \circ g)$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-compose-snd-snd:*

$h \ f \circ j \ g = k \ (f \circ g) \implies$
 $(\text{iso-tuple-snd-update isom} \circ h) \ f \circ (\text{iso-tuple-snd-update isom} \circ j) \ g =$
 $(\text{iso-tuple-snd-update isom} \circ k) \ (f \circ g)$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-surjective-proof-assist-step:*

$\text{iso-tuple-surjective-proof-assist } v \ a \ (\text{iso-tuple-fst isom} \circ f) \implies$
 $\text{iso-tuple-surjective-proof-assist } v \ b \ (\text{iso-tuple-snd isom} \circ f) \implies$
 $\text{iso-tuple-surjective-proof-assist } v \ (\text{iso-tuple-cons isom } a \ b) \ f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-fst-update-accessor-cong-assist:*

assumes *iso-tuple-update-accessor-cong-assist* $f \ g$
shows *iso-tuple-update-accessor-cong-assist*
 $(\text{iso-tuple-fst-update isom} \circ f) \ (g \circ \text{iso-tuple-fst isom})$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-snd-update-accessor-cong-assist:*

assumes *iso-tuple-update-accessor-cong-assist* $f \ g$
shows *iso-tuple-update-accessor-cong-assist*
 $(\text{iso-tuple-snd-update isom} \circ f) \ (g \circ \text{iso-tuple-snd isom})$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-fst-update-accessor-eq-assist:*

assumes *iso-tuple-update-accessor-eq-assist* $f \ g \ a \ u \ a' \ v$
shows *iso-tuple-update-accessor-eq-assist*
 $(\text{iso-tuple-fst-update isom} \circ f) \ (g \circ \text{iso-tuple-fst isom})$
 $(\text{iso-tuple-cons isom } a \ b) \ u \ (\text{iso-tuple-cons isom } a' \ b) \ v$
 $\langle \text{proof} \rangle$

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*
 $(\text{iso-tuple-snd-update isom} \circ f) \ (g \circ \text{iso-tuple-snd isom})$
 $(\text{iso-tuple-cons isom } a \ b) \ u \ (\text{iso-tuple-cons isom } a \ b') \ v$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-cons-conj-eqI:*

$a = c \wedge b = d \wedge P \longleftrightarrow Q \implies$

iso-tuple-cons isom a b = iso-tuple-cons isom c d \wedge P \longleftrightarrow Q
<proof>

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-fst-snd
iso-tuple-update-swap-snd-fst
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: $\bigwedge x y. \text{repr } x = \text{repr } y \longleftrightarrow x = y$*
and *abst-inv: $\bigwedge z. \text{repr } (\text{abst } z) = z$*
and *v: v \equiv Tuple-Isomorphism repr abst*
shows *isomorphic-tuple v*
<proof>

definition

tuple-iso-tuple \equiv Tuple-Isomorphism id id

lemma *tuple-iso-tuple:*
isomorphic-tuple tuple-iso-tuple
<proof>

lemma *refl-conj-eq: Q = R \implies P \wedge Q \longleftrightarrow P \wedge R*
<proof>

lemma *iso-tuple-UNIV-I: x \in UNIV \equiv True*
<proof>

lemma *iso-tuple-True-simp: (True \implies PROP P) \equiv PROP P*
<proof>

lemma *prop-subst: s = t \implies PROP P t \implies PROP P s*
<proof>

lemma *K-record-comp*: $(\lambda x. c) \circ f = (\lambda x. c)$
 $\langle \text{proof} \rangle$

86.4 Concrete record syntax

nonterminal

ident **and**
field-type **and**
field-types **and**
field **and**
fields **and**
field-update **and**
field-updates

open-bundle *record-syntax*
begin

syntax

-constify $:: id \Rightarrow ident$ $(\langle - \rangle)$
-constify $:: longid \Rightarrow ident$ $(\langle - \rangle)$

-field-type $:: ident \Rightarrow type \Rightarrow field\text{-}type$ $(\langle (\langle indent=2 notation=\langle infix$
*field type \rangle \rangle - :: / - \rangle)
 $:: field\text{-}type \Rightarrow field\text{-}types$ $(\langle - \rangle)$
-field-types $:: field\text{-}type \Rightarrow field\text{-}types \Rightarrow field\text{-}types$ $(\langle -, / - \rangle)$
-record-type $:: field\text{-}types \Rightarrow type$ $(\langle (\langle indent=3 notation=\langle mixfix$
*record type \rangle \rangle \langle - \rangle \rangle)
-record-type-scheme $:: field\text{-}types \Rightarrow type \Rightarrow type$ $(\langle (\langle indent=3 nota-$
 $tion=\langle mixfix record type \rangle \rangle \langle -, / (\langle indent=2 notation=\langle infix more type \rangle \rangle \dots :: / - \rangle) \rangle)$**

-field $:: ident \Rightarrow 'a \Rightarrow field$ $(\langle (\langle indent=2 notation=\langle infix$
*field value \rangle \rangle (\langle open-block markup=\langle const \rangle \rangle -) := / - \rangle)
 $:: field \Rightarrow fields$ $(\langle - \rangle)$
-fields $:: field \Rightarrow fields \Rightarrow fields$ $(\langle -, / - \rangle)$
-record $:: fields \Rightarrow 'a$ $(\langle (\langle indent=3 notation=\langle mixfix$
*record value \rangle \rangle \langle - \rangle \rangle)
-record-scheme $:: fields \Rightarrow 'a \Rightarrow 'a$ $(\langle (\langle indent=3 notation=\langle mixfix$
*record value \rangle \rangle \langle -, / (\langle indent=2 notation=\langle infix more value \rangle \rangle \dots = / - \rangle) \rangle)***

-field-update $:: ident \Rightarrow 'a \Rightarrow field\text{-}update$ $(\langle (\langle indent=2 notation=\langle infix$
*field update \rangle \rangle (\langle open-block markup=\langle const \rangle \rangle -) := / - \rangle)
 $:: field\text{-}update \Rightarrow field\text{-}updates$ $(\langle - \rangle)$
-field-updates $:: field\text{-}update \Rightarrow field\text{-}updates \Rightarrow field\text{-}updates$ $(\langle -, / - \rangle)$
-record-update $:: 'a \Rightarrow field\text{-}updates \Rightarrow 'b$ $(\langle (\langle open-block nota-$
 $tion=\langle mixfix record update \rangle \rangle - / (3 \langle - \rangle) \rangle) [900, 0] 900)$*

syntax (*ASCII*)

-record-type $:: field\text{-}types \Rightarrow type$ $(\langle (\langle indent=3 notation=\langle mixfix$

```

record type >>'(| - |')>>
  -record-type-scheme :: field-types => type => type      (⟨⟨indent=3 notation=⟨mixfix record type>>'(| -,/ (⟨indent=2 notation=⟨infix more type>>... ::/ -) |')>>⟩)
  -record              :: fields => 'a                    (⟨⟨indent=3 notation=⟨mixfix record value>>'(| - |')>>⟩)
  -record-scheme      :: fields => 'a => 'a                (⟨⟨indent=3 notation=⟨mixfix record value>>'(| -,/ (⟨indent=2 notation=⟨infix more value>>... =/ -) |')>>⟩)
  -record-update      :: 'a => field-updates => 'b         (⟨⟨open-block notation=⟨mixfix record update>>-/(3'(| - |'))> [900, 0] 900)

end

```

86.5 Record package

⟨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 * a = normalize a
  and normalize-left-idem [simp]: normalize a * b = a * b
  and normalize-idem [simp]: normalize (a * b) = a * b
  and normalize-top [simp]: normalize ⊤ = ⊤
  and normalize-bottom [simp]: normalize ⊥ = ⊥
  and top-left-normalize [simp]: ⊤ * a = normalize a
  and bottom-left-bottom [simp]: ⊥ * a = ⊥
begin

```

```

lemma left-idem [simp]:
  a * (a * b) = a * b

```

$\langle proof \rangle$

lemma *right-idem* [simp]:

$(a * b) * b = a * b$

$\langle proof \rangle$

lemma *comp-fun-idem*: *comp-fun-idem* f

$\langle proof \rangle$

interpretation *comp-fun-idem* f

$\langle proof \rangle$

lemma *top-right-normalize* [simp]:

$a * \top = \text{normalize } a$

$\langle proof \rangle$

lemma *bottom-right-bottom* [simp]:

$a * \perp = \perp$

$\langle proof \rangle$

lemma *normalize-right-idem* [simp]:

$a * \text{normalize } b = a * b$

$\langle proof \rangle$

end

locale *bounded-quasi-semilattice-set* = *bounded-quasi-semilattice*

begin

interpretation *comp-fun-idem* f

$\langle proof \rangle$

definition $F :: 'a \text{ set} \Rightarrow 'a$

where

$eq\text{-fold}: F \ A = (\text{if } finite \ A \text{ then } Finite\text{-Set.fold } f \ \top \ A \text{ else } \perp)$

lemma *infinite* [simp]:

$infinite \ A \Longrightarrow F \ A = \perp$

$\langle proof \rangle$

lemma *set-eq-fold* [code]:

$F \ (\text{set } xs) = \text{fold } f \ xs \ \top$

$\langle proof \rangle$

lemma *empty* [simp]:

$F \ \{\} = \top$

$\langle proof \rangle$

lemma *insert* [simp]:

$F \text{ (insert } a \text{ } A) = a * F \text{ } A$
 $\langle \text{proof} \rangle$

lemma *normalize* [simp]:
 $\text{normalize } (F \text{ } A) = F \text{ } A$
 $\langle \text{proof} \rangle$

lemma *in-idem*:
assumes $a \in A$
shows $a * F \text{ } A = F \text{ } A$
 $\langle \text{proof} \rangle$

lemma *union*:
 $F \text{ } (A \cup B) = F \text{ } A * F \text{ } B$
 $\langle \text{proof} \rangle$

lemma *remove*:
assumes $a \in A$
shows $F \text{ } A = a * F \text{ } (A - \{a\})$
 $\langle \text{proof} \rangle$

lemma *insert-remove*:
 $F \text{ (insert } a \text{ } A) = a * F \text{ } (A - \{a\})$
 $\langle \text{proof} \rangle$

lemma *subset*:
assumes $B \subseteq A$
shows $F \text{ } B * F \text{ } A = F \text{ } A$
 $\langle \text{proof} \rangle$

end

87.2 Abstract GCD and LCM

class *gcd* = *zero* + *one* + *dvd* +
fixes $\text{gcd} :: 'a \Rightarrow 'a \Rightarrow 'a$
and $\text{lcm} :: 'a \Rightarrow 'a \Rightarrow 'a$

class *Gcd* = *gcd* +
fixes $\text{Gcd} :: 'a \text{ set} \Rightarrow 'a$
and $\text{Lcm} :: 'a \text{ set} \Rightarrow 'a$

syntax
 $\text{-GCD1} \quad :: \text{pttrns} \Rightarrow 'b \Rightarrow 'b \quad (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } \text{GCD} \rangle \rangle \text{GCD}$
 $\text{-./ -} \rangle [0, 10] 10)$
 $\text{-GCD} \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b \quad (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } \text{GCD} \rangle \rangle \text{GCD}$
 $\text{-\in -./ -} \rangle [0, 0, 10] 10)$
 $\text{-LCM1} \quad :: \text{pttrns} \Rightarrow 'b \Rightarrow 'b \quad (\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder } \text{LCM} \rangle \rangle \text{LCM}$
 $\text{-./ -} \rangle [0, 10] 10)$

-LCM :: pttm \Rightarrow 'a set \Rightarrow 'b \Rightarrow 'b (\langle (\langle indent=3 notation= \langle binder LCM \rangle \rangle LCM
 \in ./ -) \rangle [0, 0, 10] 10)

syntax-consts

-GCD1 -GCD \Rightarrow Gcd and
 -LCM1 -LCM \Rightarrow Lcm

translations

GCD x y. f \Rightarrow GCD x. GCD y. f
 GCD x. f \Rightarrow CONST Gcd (CONST range (λ x. f))
 GCD x \in A. f \Rightarrow CONST Gcd ((λ x. f) ' A)
 LCM x y. f \Rightarrow LCM x. LCM y. f
 LCM x. f \Rightarrow CONST Lcm (CONST range (λ x. f))
 LCM x \in A. f \Rightarrow CONST Lcm ((λ x. f) ' A)

class semiring-gcd = normalization-semidom + gcd +
assumes gcd-dvd1 [iff]: gcd a b dvd a
and gcd-dvd2 [iff]: gcd a b dvd b
and gcd-greatest: c dvd a \implies c dvd b \implies c dvd gcd a b
and normalize-gcd [simp]: normalize (gcd a b) = gcd a b
and lcm-gcd: lcm a b = normalize (a * b div gcd a b)
begin

lemma gcd-greatest-iff [simp]: a dvd gcd b c \longleftrightarrow a dvd b \wedge a dvd c
 \langle proof \rangle

lemma gcd-dvdI1: a dvd c \implies gcd a b dvd c
 \langle proof \rangle

lemma gcd-dvdI2: b dvd c \implies gcd a b dvd c
 \langle proof \rangle

lemma dvd-gcdD1: a dvd gcd b c \implies a dvd b
 \langle proof \rangle

lemma dvd-gcdD2: a dvd gcd b c \implies a dvd c
 \langle proof \rangle

lemma gcd-0-left [simp]: gcd 0 a = normalize a
 \langle proof \rangle

lemma gcd-0-right [simp]: gcd a 0 = normalize a
 \langle proof \rangle

lemma gcd-eq-0-iff [simp]: gcd a b = 0 \longleftrightarrow a = 0 \wedge b = 0
 (is ?P \longleftrightarrow ?Q)
 \langle proof \rangle

lemma unit-factor-gcd: unit-factor (gcd a b) = (if a = 0 \wedge b = 0 then 0 else 1)

$\langle proof \rangle$

lemma *is-unit-gcd-iff* [simp]:

is-unit (gcd *a b*) \longleftrightarrow gcd *a b* = 1

$\langle proof \rangle$

sublocale gcd: *abel-semigroup gcd*

$\langle proof \rangle$

sublocale gcd: *bounded-quasi-semilattice gcd 0 1 normalize*

$\langle proof \rangle$

lemma *gcd-self*: gcd *a a* = *normalize a*

$\langle proof \rangle$

lemma *gcd-left-idem*: gcd *a* (gcd *a b*) = gcd *a b*

$\langle proof \rangle$

lemma *gcd-right-idem*: gcd (gcd *a b*) *b* = gcd *a b*

$\langle proof \rangle$

lemma *gcdI*:

assumes *c dvd a* **and** *c dvd b*

and greatest: $\bigwedge d. d \text{ dvd } a \implies d \text{ dvd } b \implies d \text{ dvd } c$

and *normalize c* = *c*

shows *c* = gcd *a b*

$\langle proof \rangle$

lemma *gcd-unique*:

d dvd a \wedge *d dvd b* \wedge *normalize d* = *d* \wedge ($\forall e. e \text{ dvd } a \wedge e \text{ dvd } b \implies e \text{ dvd } d$) \longleftrightarrow
d = gcd *a b*

$\langle proof \rangle$

lemma *gcd-dvd-prod*: gcd *a b* dvd *k* * *b*

$\langle proof \rangle$

lemma *gcd-proj2-if-dvd*: *b dvd a* \implies gcd *a b* = *normalize b*

$\langle proof \rangle$

lemma *gcd-proj1-if-dvd*: *a dvd b* \implies gcd *a b* = *normalize a*

$\langle proof \rangle$

lemma *gcd-proj1-iff*: gcd *m n* = *normalize m* \longleftrightarrow *m dvd n*

$\langle proof \rangle$

lemma *gcd-proj2-iff*: gcd *m n* = *normalize n* \longleftrightarrow *n dvd m*

$\langle proof \rangle$

lemma *gcd-mult-left*: gcd (*c* * *a*) (*c* * *b*) = *normalize* (*c* * gcd *a b*)

$\langle proof \rangle$

lemma *gcd-mult-right*: $gcd\ (a * c)\ (b * c) = normalize\ (gcd\ b\ a * c)$
 $\langle proof \rangle$

lemma *dvd-lcm1* [iff]: $a\ dvd\ lcm\ a\ b$
 $\langle proof \rangle$

lemma *dvd-lcm2* [iff]: $b\ dvd\ lcm\ a\ b$
 $\langle proof \rangle$

lemma *dvd-lcmI1*: $a\ dvd\ b \implies a\ dvd\ lcm\ b\ c$
 $\langle proof \rangle$

lemma *dvd-lcmI2*: $a\ dvd\ c \implies a\ dvd\ lcm\ b\ c$
 $\langle proof \rangle$

lemma *lcm-dvdD1*: $lcm\ a\ b\ dvd\ c \implies a\ dvd\ c$
 $\langle proof \rangle$

lemma *lcm-dvdD2*: $lcm\ a\ b\ dvd\ c \implies b\ dvd\ c$
 $\langle proof \rangle$

lemma *lcm-least*:
 assumes $a\ dvd\ c$ and $b\ dvd\ c$
 shows $lcm\ a\ b\ dvd\ c$
 $\langle proof \rangle$

lemma *lcm-least-iff* [simp]: $lcm\ a\ b\ dvd\ c \longleftrightarrow a\ dvd\ c \wedge b\ dvd\ c$
 $\langle proof \rangle$

lemma *normalize-lcm* [simp]: $normalize\ (lcm\ a\ b) = lcm\ a\ b$
 $\langle proof \rangle$

lemma *lcm-0-left* [simp]: $lcm\ 0\ a = 0$
 $\langle proof \rangle$

lemma *lcm-0-right* [simp]: $lcm\ a\ 0 = 0$
 $\langle proof \rangle$

lemma *lcm-eq-0-iff*: $lcm\ a\ b = 0 \longleftrightarrow a = 0 \vee b = 0$
 (is $?P \longleftrightarrow ?Q$)
 $\langle proof \rangle$

lemma *zero-eq-lcm-iff*: $0 = lcm\ a\ b \longleftrightarrow a = 0 \vee b = 0$
 $\langle proof \rangle$

lemma *lcm-eq-1-iff* [simp]: $lcm\ a\ b = 1 \longleftrightarrow is-unit\ a \wedge is-unit\ b$
 $\langle proof \rangle$

lemma *unit-factor-lcm*: *unit-factor* (*lcm a b*) = (*if a = 0 \vee b = 0 then 0 else 1*)
 ⟨*proof*⟩

sublocale *lcm*: *abel-semigroup lcm*
 ⟨*proof*⟩

sublocale *lcm*: *bounded-quasi-semilattice lcm 1 0 normalize*
 ⟨*proof*⟩

lemma *lcm-self*: *lcm a a = normalize a*
 ⟨*proof*⟩

lemma *lcm-left-idem*: *lcm a (lcm a b) = lcm a b*
 ⟨*proof*⟩

lemma *lcm-right-idem*: *lcm (lcm a b) b = lcm a b*
 ⟨*proof*⟩

lemma *gcd-lcm*:
assumes *a \neq 0 and b \neq 0*
shows *gcd a b = normalize (a * b div lcm a b)*
 ⟨*proof*⟩

lemma *lcm-1-left*: *lcm 1 a = normalize a*
 ⟨*proof*⟩

lemma *lcm-1-right*: *lcm a 1 = normalize a*
 ⟨*proof*⟩

lemma *lcm-mult-left*: *lcm (c * a) (c * b) = normalize (c * lcm a b)*
 ⟨*proof*⟩

lemma *lcm-mult-right*: *lcm (a * c) (b * c) = normalize (lcm b a * c)*
 ⟨*proof*⟩

lemma *lcm-mult-unit1*: *is-unit a \implies lcm (b * a) c = lcm b c*
 ⟨*proof*⟩

lemma *lcm-mult-unit2*: *is-unit a \implies lcm b (c * a) = lcm b c*
 ⟨*proof*⟩

lemma *lcm-div-unit1*:
is-unit a \implies lcm (b div a) c = lcm b c
 ⟨*proof*⟩

lemma *lcm-div-unit2*: *is-unit a \implies lcm b (c div a) = lcm b c*
 ⟨*proof*⟩

lemma *normalize-lcm-left*: $\text{lcm } (\text{normalize } a) \ b = \text{lcm } a \ b$
 $\langle \text{proof} \rangle$

lemma *normalize-lcm-right*: $\text{lcm } a \ (\text{normalize } b) = \text{lcm } a \ b$
 $\langle \text{proof} \rangle$

lemma *comp-fun-idem-gcd*: *comp-fun-idem gcd*
 $\langle \text{proof} \rangle$

lemma *comp-fun-idem-lcm*: *comp-fun-idem lcm*
 $\langle \text{proof} \rangle$

lemma *gcd-dvd-antisym*: $\text{gcd } a \ b \ \text{dvd } \text{gcd } c \ d \implies \text{gcd } c \ d \ \text{dvd } \text{gcd } a \ b \implies \text{gcd } a \ b = \text{gcd } c \ d$
 $\langle \text{proof} \rangle$

declare *unit-factor-lcm* [*simp*]

lemma *lcmI*:
assumes $a \ \text{dvd } c$ **and** $b \ \text{dvd } c$ **and** $\bigwedge d. a \ \text{dvd } d \implies b \ \text{dvd } d \implies c \ \text{dvd } d$
and $\text{normalize } c = c$
shows $c = \text{lcm } a \ b$
 $\langle \text{proof} \rangle$

lemma *gcd-dvd-lcm* [*simp*]: $\text{gcd } a \ b \ \text{dvd } \text{lcm } a \ b$
 $\langle \text{proof} \rangle$

lemmas *lcm-0 = lcm-0-right*

lemma *lcm-unique*:
 $a \ \text{dvd } d \wedge b \ \text{dvd } d \wedge \text{normalize } d = d \wedge (\forall e. a \ \text{dvd } e \wedge b \ \text{dvd } e \longrightarrow d \ \text{dvd } e) \longleftrightarrow d = \text{lcm } a \ b$
 $\langle \text{proof} \rangle$

lemma *lcm-proj1-if-dvd*:
assumes $b \ \text{dvd } a$ **shows** $\text{lcm } a \ b = \text{normalize } a$
 $\langle \text{proof} \rangle$

lemma *lcm-proj2-if-dvd*: $a \ \text{dvd } b \implies \text{lcm } a \ b = \text{normalize } b$
 $\langle \text{proof} \rangle$

lemma *lcm-proj1-iff*: $\text{lcm } m \ n = \text{normalize } m \longleftrightarrow n \ \text{dvd } m$
 $\langle \text{proof} \rangle$

lemma *lcm-proj2-iff*: $\text{lcm } m \ n = \text{normalize } n \longleftrightarrow m \ \text{dvd } n$
 $\langle \text{proof} \rangle$

lemma *gcd-mono*: $a \ \text{dvd } c \implies b \ \text{dvd } d \implies \text{gcd } a \ b \ \text{dvd } \text{gcd } c \ d$
 $\langle \text{proof} \rangle$

lemma *lcm-mono*: $a \text{ dvd } c \implies b \text{ dvd } d \implies \text{lcm } a \ b \text{ dvd } \text{lcm } c \ d$
 ⟨proof⟩

lemma *dvd-productE*:
 assumes $p \text{ dvd } a * b$
 obtains $x \ y$ where $p = x * y$ $x \text{ dvd } a$ $y \text{ dvd } b$
 ⟨proof⟩

lemma *gcd-mult-unit1*:
 assumes *is-unit* a shows $\text{gcd } (b * a) \ c = \text{gcd } b \ c$
 ⟨proof⟩

lemma *gcd-mult-unit2*: *is-unit* $a \implies \text{gcd } b \ (c * a) = \text{gcd } b \ c$
 ⟨proof⟩

lemma *gcd-div-unit1*: *is-unit* $a \implies \text{gcd } (b \text{ div } a) \ c = \text{gcd } b \ c$
 ⟨proof⟩

lemma *gcd-div-unit2*: *is-unit* $a \implies \text{gcd } b \ (c \text{ div } a) = \text{gcd } b \ c$
 ⟨proof⟩

lemma *normalize-gcd-left*: $\text{gcd } (\text{normalize } a) \ b = \text{gcd } a \ b$
 ⟨proof⟩

lemma *normalize-gcd-right*: $\text{gcd } a \ (\text{normalize } b) = \text{gcd } a \ b$
 ⟨proof⟩

lemma *gcd-add1* [simp]: $\text{gcd } (m + n) \ n = \text{gcd } m \ n$
 ⟨proof⟩

lemma *gcd-add2* [simp]: $\text{gcd } m \ (m + n) = \text{gcd } m \ n$
 ⟨proof⟩

lemma *gcd-add-mult*: $\text{gcd } m \ (k * m + n) = \text{gcd } m \ n$
 ⟨proof⟩

end

class *ring-gcd* = *comm-ring-1* + *semiring-gcd*
begin

lemma *gcd-neg1* [simp]: $\text{gcd } (-a) \ b = \text{gcd } a \ b$
 ⟨proof⟩

lemma *gcd-neg2* [simp]: $\text{gcd } a \ (-b) = \text{gcd } a \ b$
 ⟨proof⟩

lemma *gcd-neg-numeral-1* [simp]: $\text{gcd } (- \text{numeral } n) \ a = \text{gcd } (\text{numeral } n) \ a$

$\langle proof \rangle$

lemma *gcd-neg-numeral-2* [simp]: $gcd\ a\ (-\ numeral\ n) = gcd\ a\ (numeral\ n)$
 $\langle proof \rangle$

lemma *gcd-diff1*: $gcd\ (m - n)\ n = gcd\ m\ n$
 $\langle proof \rangle$

lemma *gcd-diff2*: $gcd\ (n - m)\ n = gcd\ m\ n$
 $\langle proof \rangle$

lemma *lcm-neg1* [simp]: $lcm\ (-a)\ b = lcm\ a\ b$
 $\langle proof \rangle$

lemma *lcm-neg2* [simp]: $lcm\ a\ (-b) = lcm\ a\ b$
 $\langle proof \rangle$

lemma *lcm-neg-numeral-1* [simp]: $lcm\ (-\ numeral\ n)\ a = lcm\ (numeral\ n)\ a$
 $\langle proof \rangle$

lemma *lcm-neg-numeral-2* [simp]: $lcm\ a\ (-\ numeral\ n) = lcm\ a\ (numeral\ n)$
 $\langle proof \rangle$

end

class *semiring-Gcd* = *semiring-gcd* + *Gcd* +
assumes *Gcd-dvd*: $a \in A \implies Gcd\ A\ dvd\ a$
and *Gcd-greatest*: $(\bigwedge b. b \in A \implies a\ dvd\ b) \implies a\ dvd\ Gcd\ A$
and *normalize-Gcd* [simp]: $normalize\ (Gcd\ A) = Gcd\ A$
assumes *dvd-Lcm*: $a \in A \implies a\ dvd\ Lcm\ A$
and *Lcm-least*: $(\bigwedge b. b \in A \implies b\ dvd\ a) \implies Lcm\ A\ dvd\ a$
and *normalize-Lcm* [simp]: $normalize\ (Lcm\ A) = Lcm\ A$
begin

lemma *Lcm-Gcd*: $Lcm\ A = Gcd\ \{b. \forall a \in A. a\ dvd\ b\}$
 $\langle proof \rangle$

lemma *Gcd-Lcm*: $Gcd\ A = Lcm\ \{b. \forall a \in A. b\ dvd\ a\}$
 $\langle proof \rangle$

lemma *Gcd-empty* [simp]: $Gcd\ \{\} = 0$
 $\langle proof \rangle$

lemma *Lcm-empty* [simp]: $Lcm\ \{\} = 1$
 $\langle proof \rangle$

lemma *Gcd-insert* [simp]: $Gcd\ (insert\ a\ A) = gcd\ a\ (Gcd\ A)$
 $\langle proof \rangle$

lemma *Lcm-insert* [simp]: $Lcm\ (insert\ a\ A) = lcm\ a\ (Lcm\ A)$
 ⟨proof⟩

lemma *LcmI*:
 assumes $\bigwedge a. a \in A \implies a\ dvd\ b$
 and $\bigwedge c. (\bigwedge a. a \in A \implies a\ dvd\ c) \implies b\ dvd\ c$
 and *normalize* $b = b$
 shows $b = Lcm\ A$
 ⟨proof⟩

lemma *Lcm-subset*: $A \subseteq B \implies Lcm\ A\ dvd\ Lcm\ B$
 ⟨proof⟩

lemma *Lcm-Un*: $Lcm\ (A \cup B) = lcm\ (Lcm\ A)\ (Lcm\ B)$
 ⟨proof⟩

lemma *Gcd-0-iff* [simp]: $Gcd\ A = 0 \longleftrightarrow A \subseteq \{0\}$
 (is $?P \longleftrightarrow ?Q$)
 ⟨proof⟩

lemma *Lcm-1-iff* [simp]: $Lcm\ A = 1 \longleftrightarrow (\forall a \in A. is-unit\ a)$
 (is $?P \longleftrightarrow ?Q$)
 ⟨proof⟩

lemma *unit-factor-Lcm*: $unit-factor\ (Lcm\ A) = (if\ Lcm\ A = 0\ then\ 0\ else\ 1)$
 ⟨proof⟩

lemma *unit-factor-Gcd*: $unit-factor\ (Gcd\ A) = (if\ Gcd\ A = 0\ then\ 0\ else\ 1)$
 ⟨proof⟩

lemma *GcdI*:
 assumes $\bigwedge a. a \in A \implies b\ dvd\ a$
 and $\bigwedge c. (\bigwedge a. a \in A \implies c\ dvd\ a) \implies c\ dvd\ b$
 and *normalize* $b = b$
 shows $b = Gcd\ A$
 ⟨proof⟩

lemma *Gcd-eq-1-I*:
 assumes *is-unit* a and $a \in A$
 shows $Gcd\ A = 1$
 ⟨proof⟩

lemma *Lcm-eq-0-I*:
 assumes $0 \in A$
 shows $Lcm\ A = 0$
 ⟨proof⟩

lemma *Gcd-UNIV* [simp]: $Gcd\ UNIV = 1$
 ⟨proof⟩

lemma *Lcm-UNIV* [simp]: $Lcm\ UNIV = 0$
 ⟨proof⟩

lemma *Lcm-0-iff*:
 assumes *finite A*
 shows $Lcm\ A = 0 \longleftrightarrow 0 \in A$
 ⟨proof⟩

lemma *Gcd-image-normalize* [simp]: $Gcd\ (normalize\ 'A) = Gcd\ A$
 ⟨proof⟩

lemma *Gcd-eqI*:
 assumes $normalize\ a = a$
 assumes $\bigwedge b. b \in A \implies a\ dvd\ b$
 and $\bigwedge c. (\bigwedge b. b \in A \implies c\ dvd\ b) \implies c\ dvd\ a$
 shows $Gcd\ A = a$
 ⟨proof⟩

lemma *dvd-GcdD*: $x\ dvd\ Gcd\ A \implies y \in A \implies x\ dvd\ y$
 ⟨proof⟩

lemma *dvd-Gcd-iff*: $x\ dvd\ Gcd\ A \longleftrightarrow (\forall y \in A. x\ dvd\ y)$
 ⟨proof⟩

lemma *Gcd-mult*: $Gcd\ ((*)\ c\ 'A) = normalize\ (c * Gcd\ A)$
 ⟨proof⟩

lemma *Lcm-eqI*:
 assumes $normalize\ a = a$
 and $\bigwedge b. b \in A \implies b\ dvd\ a$
 and $\bigwedge c. (\bigwedge b. b \in A \implies b\ dvd\ c) \implies a\ dvd\ c$
 shows $Lcm\ A = a$
 ⟨proof⟩

lemma *Lcm-dvdD*: $Lcm\ A\ dvd\ x \implies y \in A \implies y\ dvd\ x$
 ⟨proof⟩

lemma *Lcm-dvd-iff*: $Lcm\ A\ dvd\ x \longleftrightarrow (\forall y \in A. y\ dvd\ x)$
 ⟨proof⟩

lemma *Lcm-mult*:
 assumes $A \neq \{\}$
 shows $Lcm\ ((*)\ c\ 'A) = normalize\ (c * Lcm\ A)$
 ⟨proof⟩

lemma *Lcm-no-units*: $Lcm\ A = Lcm\ (A - \{a.\ is_unit\ a\})$
 ⟨proof⟩

lemma *Lcm-0-iff'*: $Lcm\ A = 0 \longleftrightarrow (\nexists l. l \neq 0 \wedge (\forall a \in A. a\ dvd\ l))$
 ⟨proof⟩

lemma *Lcm-no-multiple*: $(\forall m. m \neq 0 \longrightarrow (\exists a \in A. \neg a\ dvd\ m)) \implies Lcm\ A = 0$
 ⟨proof⟩

lemma *Lcm-singleton [simp]*: $Lcm\ \{a\} = normalize\ a$
 ⟨proof⟩

lemma *Lcm-2 [simp]*: $Lcm\ \{a, b\} = lcm\ a\ b$
 ⟨proof⟩

lemma *Gcd-1*: $1 \in A \implies Gcd\ A = 1$
 ⟨proof⟩

lemma *Gcd-singleton [simp]*: $Gcd\ \{a\} = normalize\ a$
 ⟨proof⟩

lemma *Gcd-2 [simp]*: $Gcd\ \{a, b\} = gcd\ a\ b$
 ⟨proof⟩

lemma *Gcd-mono*:
 assumes $\bigwedge x. x \in A \implies f\ x\ dvd\ g\ x$
 shows $(GCD\ x \in A. f\ x)\ dvd\ (GCD\ x \in A. g\ x)$
 ⟨proof⟩

lemma *Lcm-mono*:
 assumes $\bigwedge x. x \in A \implies f\ x\ dvd\ g\ x$
 shows $(LCM\ x \in A. f\ x)\ dvd\ (LCM\ x \in A. g\ x)$
 ⟨proof⟩

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
 $Gcd_fin\ (\langle Gcd_fin \rangle) = Gcd_fin.F :: 'a\ set \Rightarrow 'a\ \langle proof \rangle$

abbreviation *gcd-list* :: $'a\ list \Rightarrow 'a$
where $gcd_list\ xs \equiv Gcd_fin\ (set\ xs)$

sublocale *Lcm-fin*: *bounded-quasi-semilattice-set lcm 1 0 normalize*
defines
 $Lcm_fin\ (\langle Lcm_fin \rangle) = Lcm_fin.F\ \langle proof \rangle$

abbreviation $lcm_list :: 'a\ list \Rightarrow 'a$
where $lcm_list\ xs \equiv Lcm_{fin}\ (set\ xs)$

lemma Gcd_fin_dvd :
 $a \in A \implies Gcd_{fin}\ A\ dvd\ a$
 $\langle proof \rangle$

lemma dvd_Lcm_fin :
 $a \in A \implies a\ dvd\ Lcm_{fin}\ A$
 $\langle proof \rangle$

lemma $Gcd_fin_greatest$:
 $a\ dvd\ Gcd_{fin}\ A$ **if** $finite\ A$ **and** $\bigwedge b. b \in A \implies a\ dvd\ b$
 $\langle proof \rangle$

lemma Lcm_fin_least :
 $Lcm_{fin}\ A\ dvd\ a$ **if** $finite\ A$ **and** $\bigwedge b. b \in A \implies b\ dvd\ a$
 $\langle proof \rangle$

lemma $gcd_list_greatest$:
 $a\ dvd\ gcd_list\ bs$ **if** $\bigwedge b. b \in set\ bs \implies a\ dvd\ b$
 $\langle proof \rangle$

lemma lcm_list_least :
 $lcm_list\ bs\ dvd\ a$ **if** $\bigwedge b. b \in set\ bs \implies b\ dvd\ a$
 $\langle proof \rangle$

lemma $dvd_Gcd_fin_iff$:
 $b\ dvd\ Gcd_{fin}\ A \longleftrightarrow (\forall a \in A. b\ dvd\ a)$ **if** $finite\ A$
 $\langle proof \rangle$

lemma $dvd_gcd_list_iff$:
 $b\ dvd\ gcd_list\ xs \longleftrightarrow (\forall a \in set\ xs. b\ dvd\ a)$
 $\langle proof \rangle$

lemma $Lcm_fin_dvd_iff$:
 $Lcm_{fin}\ A\ dvd\ b \longleftrightarrow (\forall a \in A. a\ dvd\ b)$ **if** $finite\ A$
 $\langle proof \rangle$

lemma $lcm_list_dvd_iff$:
 $lcm_list\ xs\ dvd\ b \longleftrightarrow (\forall a \in set\ xs. a\ dvd\ b)$
 $\langle proof \rangle$

lemma Gcd_fin_mult :
 $Gcd_{fin}\ (image\ (times\ b)\ A) = normalize\ (b * Gcd_{fin}\ A)$ **if** $finite\ A$
 $\langle proof \rangle$

lemma Lcm_fin_mult :

$Lcm_{fin} (image (times\ b)\ A) = normalize\ (b * Lcm_{fin}\ A)$ **if** $A \neq \{\}$
 $\langle proof \rangle$

lemma *unit-factor-Gcd-fin*:
 $unit_factor\ (Gcd_{fin}\ A) = of_bool\ (Gcd_{fin}\ A \neq 0)$
 $\langle proof \rangle$

lemma *unit-factor-Lcm-fin*:
 $unit_factor\ (Lcm_{fin}\ A) = of_bool\ (Lcm_{fin}\ A \neq 0)$
 $\langle proof \rangle$

lemma *is-unit-Gcd-fin-iff* [simp]:
 $is_unit\ (Gcd_{fin}\ A) \longleftrightarrow Gcd_{fin}\ A = 1$
 $\langle proof \rangle$

lemma *is-unit-Lcm-fin-iff* [simp]:
 $is_unit\ (Lcm_{fin}\ A) \longleftrightarrow Lcm_{fin}\ A = 1$
 $\langle proof \rangle$

lemma *Gcd-fin-0-iff*:
 $Gcd_{fin}\ A = 0 \longleftrightarrow A \subseteq \{0\} \wedge finite\ A$
 $\langle proof \rangle$

lemma *Lcm-fin-0-iff*:
 $Lcm_{fin}\ A = 0 \longleftrightarrow 0 \in A$ **if** $finite\ A$
 $\langle proof \rangle$

lemma *Lcm-fin-1-iff*:
 $Lcm_{fin}\ A = 1 \longleftrightarrow (\forall a \in A. is_unit\ a) \wedge finite\ A$
 $\langle proof \rangle$

end

context *semiring-Gcd*
begin

lemma *Gcd-fin-eq-Gcd* [simp]:
 $Gcd_{fin}\ A = Gcd\ A$ **if** $finite\ A$ **for** $A :: 'a\ set$
 $\langle proof \rangle$

lemma *Gcd-set-eq-fold* [code-unfold]:
 $Gcd\ (set\ xs) = fold\ gcd\ xs\ 0$
 $\langle proof \rangle$

lemma *Lcm-fin-eq-Lcm* [simp]:
 $Lcm_{fin}\ A = Lcm\ A$ **if** $finite\ A$ **for** $A :: 'a\ set$
 $\langle proof \rangle$

lemma *Lcm-set-eq-fold* [code-unfold]:

$Lcm\ (set\ xs) = fold\ lcm\ xs\ 1$
 $\langle proof \rangle$

end

87.4 Coprimality

context *semiring-gcd*
begin

lemma *coprime-imp-gcd-eq-1* [*simp*]:
 $gcd\ a\ b = 1\ \text{if}\ coprime\ a\ b$
 $\langle proof \rangle$

lemma *gcd-eq-1-imp-coprime* [*dest!*]:
 $coprime\ a\ b\ \text{if}\ gcd\ a\ b = 1$
 $\langle proof \rangle$

lemma *coprime-iff-gcd-eq-1* [*presburger, code*]:
 $coprime\ a\ b \longleftrightarrow gcd\ a\ b = 1$
 $\langle proof \rangle$

lemma *is-unit-gcd* [*simp*]:
 $is-unit\ (gcd\ a\ b) \longleftrightarrow coprime\ a\ b$
 $\langle proof \rangle$

lemma *coprime-add-one-left* [*simp*]: $coprime\ (a + 1)\ a$
 $\langle proof \rangle$

lemma *coprime-add-one-right* [*simp*]: $coprime\ a\ (a + 1)$
 $\langle proof \rangle$

lemma *coprime-mult-left-iff* [*simp*]:
 $coprime\ (a * b)\ c \longleftrightarrow coprime\ a\ c \wedge coprime\ b\ c$
 $\langle proof \rangle$

lemma *coprime-mult-right-iff* [*simp*]:
 $coprime\ c\ (a * b) \longleftrightarrow coprime\ c\ a \wedge coprime\ c\ b$
 $\langle proof \rangle$

lemma *coprime-power-left-iff* [*simp*]:
 $coprime\ (a \wedge n)\ b \longleftrightarrow coprime\ a\ b \vee n = 0$
 $\langle proof \rangle$

lemma *coprime-power-right-iff* [*simp*]:
 $coprime\ a\ (b \wedge n) \longleftrightarrow coprime\ a\ b \vee n = 0$
 $\langle proof \rangle$

lemma *prod-coprime-left*:

coprime $(\prod_{i \in A}. f\ i)\ a$ **if** $\bigwedge i. i \in A \implies \text{coprime}\ (f\ i)\ a$
 <proof>

lemma *prod-coprime-right*:
coprime $a\ (\prod_{i \in A}. f\ i)$ **if** $\bigwedge i. i \in A \implies \text{coprime}\ a\ (f\ i)$
 <proof>

lemma *prod-list-coprime-left*:
coprime $(\text{prod-list}\ xs)\ a$ **if** $\bigwedge x. x \in \text{set}\ xs \implies \text{coprime}\ x\ a$
 <proof>

lemma *prod-list-coprime-right*:
coprime $a\ (\text{prod-list}\ xs)$ **if** $\bigwedge x. x \in \text{set}\ xs \implies \text{coprime}\ a\ x$
 <proof>

lemma *coprime-dvd-mult-left-iff*:
 $a\ \text{dvd}\ b * c \longleftrightarrow a\ \text{dvd}\ b$ **if** *coprime* $a\ c$
 <proof>

lemma *coprime-dvd-mult-right-iff*:
 $a\ \text{dvd}\ c * b \longleftrightarrow a\ \text{dvd}\ c$ **if** *coprime* $a\ b$
 <proof>

lemma *divides-mult*:
 $a * b\ \text{dvd}\ c$ **if** $a\ \text{dvd}\ c$ **and** $b\ \text{dvd}\ c$ **and** *coprime* $a\ b$
 <proof>

lemma *div-gcd-coprime*:
assumes $a \neq 0 \vee b \neq 0$
shows *coprime* $(a\ \text{div}\ \text{gcd}\ a\ b)\ (b\ \text{div}\ \text{gcd}\ a\ b)$
 <proof>

lemma *gcd-coprime*:
assumes $c: \text{gcd}\ a\ b \neq 0$
and $a: a = a' * \text{gcd}\ a\ b$
and $b: b = b' * \text{gcd}\ a\ b$
shows *coprime* $a'\ b'$
 <proof>

lemma *gcd-coprime-exists*:
assumes $\text{gcd}\ a\ b \neq 0$
shows $\exists a'\ b'. a = a' * \text{gcd}\ a\ b \wedge b = b' * \text{gcd}\ a\ b \wedge \text{coprime}\ a'\ b'$
 <proof>

lemma *pow-divides-pow-iff* [simp]:
 $a^{\wedge} n\ \text{dvd}\ b^{\wedge} n \longleftrightarrow a\ \text{dvd}\ b$ **if** $n > 0$
 <proof>

lemma *coprime-crossproduct*:

fixes $a\ b\ c\ d :: 'a$
assumes $\text{coprime } a\ d$ **and** $\text{coprime } b\ c$
shows $\text{normalize } a * \text{normalize } c = \text{normalize } b * \text{normalize } d \longleftrightarrow$
 $\text{normalize } a = \text{normalize } b \wedge \text{normalize } c = \text{normalize } d$
(is $?lhs \longleftrightarrow ?rhs)$
 $\langle \text{proof} \rangle$

lemma $\text{gcd-mult-left-left-cancel}$:
 $\text{gcd } (c * a)\ b = \text{gcd } a\ b$ **if** $\text{coprime } b\ c$
 $\langle \text{proof} \rangle$

lemma $\text{gcd-mult-left-right-cancel}$:
 $\text{gcd } (a * c)\ b = \text{gcd } a\ b$ **if** $\text{coprime } b\ c$
 $\langle \text{proof} \rangle$

lemma $\text{gcd-mult-right-left-cancel}$:
 $\text{gcd } a\ (c * b) = \text{gcd } a\ b$ **if** $\text{coprime } a\ c$
 $\langle \text{proof} \rangle$

lemma $\text{gcd-mult-right-right-cancel}$:
 $\text{gcd } a\ (b * c) = \text{gcd } a\ b$ **if** $\text{coprime } a\ c$
 $\langle \text{proof} \rangle$

lemma gcd-exp-weak :
 $\text{gcd } (a \wedge^n)\ (b \wedge^n) = \text{normalize } (\text{gcd } a\ b \wedge^n)$
 $\langle \text{proof} \rangle$

lemma division-decomp :
assumes $a\ \text{dvd } b * c$
shows $\exists b'\ c'.\ a = b' * c' \wedge b' \text{ dvd } b \wedge c' \text{ dvd } c$
 $\langle \text{proof} \rangle$

lemma lcm-coprime : $\text{coprime } a\ b \implies \text{lcm } a\ b = \text{normalize } (a * b)$
 $\langle \text{proof} \rangle$

end

context ring-gcd
begin

lemma $\text{coprime-minus-left-iff}$ $[\text{simp}]$:
 $\text{coprime } (-\ a)\ b \longleftrightarrow \text{coprime } a\ b$
 $\langle \text{proof} \rangle$

lemma $\text{coprime-minus-right-iff}$ $[\text{simp}]$:
 $\text{coprime } a\ (-\ b) \longleftrightarrow \text{coprime } a\ b$
 $\langle \text{proof} \rangle$

lemma $\text{coprime-diff-one-left}$ $[\text{simp}]$: $\text{coprime } (a - 1)\ a$

$\langle \text{proof} \rangle$

lemma *coprime-doff-one-right* [simp]: *coprime a (a - 1)*
 $\langle \text{proof} \rangle$

end

context *semiring-Gcd*
begin

lemma *Lcm-coprime*:
 assumes *finite A*
 and $A \neq \{\}$
 and $\bigwedge a b. a \in A \implies b \in A \implies a \neq b \implies \text{coprime } a \ b$
 shows $\text{Lcm } A = \text{normalize } (\prod A)$
 $\langle \text{proof} \rangle$

lemma *Lcm-coprime'*:
 $\text{card } A \neq 0 \implies$
 $(\bigwedge a b. a \in A \implies b \in A \implies a \neq b \implies \text{coprime } a \ b) \implies$
 $\text{Lcm } A = \text{normalize } (\prod A)$
 $\langle \text{proof} \rangle$

end

And some consequences: cancellation modulo *m*

lemma *mult-mod-cancel-right*:
 fixes $m :: 'a :: \{\text{euclidean-ring-cancel}, \text{semiring-gcd}\}$
 assumes $\text{eq}: (a * n) \bmod m = (b * n) \bmod m$ **and** *coprime m n*
 shows $a \bmod m = b \bmod m$
 $\langle \text{proof} \rangle$

lemma *mult-mod-cancel-left*:
 fixes $m :: 'a :: \{\text{euclidean-ring-cancel}, \text{semiring-gcd}\}$
 assumes $(n * a) \bmod m = (n * b) \bmod m$ **and** *coprime m n*
 shows $a \bmod m = b \bmod m$
 $\langle \text{proof} \rangle$

87.5 GCD and LCM for multiplicative normalisation functions

class *semiring-gcd-mult-normalize* = *semiring-gcd* + *normalization-semidom-multiplicative*
begin

lemma *mult-gcd-left*: $c * \text{gcd } a \ b = \text{unit-factor } c * \text{gcd } (c * a) \ (c * b)$
 $\langle \text{proof} \rangle$

lemma *mult-gcd-right*: $\text{gcd } a \ b * c = \text{gcd } (a * c) \ (b * c) * \text{unit-factor } c$
 $\langle \text{proof} \rangle$

lemma *gcd-mult-distrib'*: $\text{normalize } c * \text{gcd } a \ b = \text{gcd } (c * a) \ (c * b)$
 $\langle \text{proof} \rangle$

lemma *gcd-mult-distrib*: $k * \text{gcd } a \ b = \text{gcd } (k * a) \ (k * b) * \text{unit-factor } k$
 $\langle \text{proof} \rangle$

lemma *gcd-mult-lcm [simp]*: $\text{gcd } a \ b * \text{lcm } a \ b = \text{normalize } a * \text{normalize } b$
 $\langle \text{proof} \rangle$

lemma *lcm-mult-gcd [simp]*: $\text{lcm } a \ b * \text{gcd } a \ b = \text{normalize } a * \text{normalize } b$
 $\langle \text{proof} \rangle$

lemma *mult-lcm-left*: $c * \text{lcm } a \ b = \text{unit-factor } c * \text{lcm } (c * a) \ (c * b)$
 $\langle \text{proof} \rangle$

lemma *mult-lcm-right*: $\text{lcm } a \ b * c = \text{lcm } (a * c) \ (b * c) * \text{unit-factor } c$
 $\langle \text{proof} \rangle$

lemma *lcm-gcd-prod*: $\text{lcm } a \ b * \text{gcd } a \ b = \text{normalize } (a * b)$
 $\langle \text{proof} \rangle$

lemma *lcm-mult-distrib'*: $\text{normalize } c * \text{lcm } a \ b = \text{lcm } (c * a) \ (c * b)$
 $\langle \text{proof} \rangle$

lemma *lcm-mult-distrib*: $k * \text{lcm } a \ b = \text{lcm } (k * a) \ (k * b) * \text{unit-factor } k$
 $\langle \text{proof} \rangle$

lemma *coprime-crossproduct'*:
 fixes $a \ b \ c \ d$
 assumes $b \neq 0$
 assumes *unit-factors*: $\text{unit-factor } b = \text{unit-factor } d$
 assumes *coprime*: $\text{coprime } a \ b \ \text{coprime } c \ d$
 shows $a * d = b * c \longleftrightarrow a = c \wedge b = d$
 $\langle \text{proof} \rangle$

lemma *gcd-exp [simp]*:
 $\text{gcd } (a \wedge n) \ (b \wedge n) = \text{gcd } a \ b \wedge n$
 $\langle \text{proof} \rangle$

end

87.6 GCD and LCM on *nat* and *int*

instantiation *nat* :: *gcd*
begin

fun *gcd-nat* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$
 where $\text{gcd-nat } x \ y = (\text{if } y = 0 \text{ then } x \text{ else } \text{gcd } y \ (x \bmod y))$

```

definition lcm-nat :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat
  where lcm-nat x y = x * y div (gcd x y)

instance  $\langle$ proof $\rangle$ 

end

instantiation int :: gcd
begin

definition gcd-int :: int  $\Rightarrow$  int  $\Rightarrow$  int
  where gcd-int x y = int (gcd (nat |x|) (nat |y|))

definition lcm-int :: int  $\Rightarrow$  int  $\Rightarrow$  int
  where lcm-int x y = int (lcm (nat |x|) (nat |y|))

instance  $\langle$ proof $\rangle$ 

end

lemma gcd-int-int-eq [simp]:
  gcd (int m) (int n) = int (gcd m n)
   $\langle$ proof $\rangle$ 

lemma gcd-nat-abs-left-eq [simp]:
  gcd (nat |k|) n = nat (gcd k (int n))
   $\langle$ proof $\rangle$ 

lemma gcd-nat-abs-right-eq [simp]:
  gcd n (nat |k|) = nat (gcd (int n) k)
   $\langle$ proof $\rangle$ 

lemma abs-gcd-int [simp]:
  |gcd x y| = gcd x y
  for x y :: int
   $\langle$ proof $\rangle$ 

lemma gcd-abs1-int [simp]:
  gcd |x| y = gcd x y
  for x y :: int
   $\langle$ proof $\rangle$ 

lemma gcd-abs2-int [simp]:
  gcd x |y| = gcd x y
  for x y :: int
   $\langle$ proof $\rangle$ 

lemma lcm-int-int-eq [simp]:

```

$\text{lcm } (\text{int } m) (\text{int } n) = \text{int } (\text{lcm } m n)$
 $\langle \text{proof} \rangle$

lemma *lcm-nat-abs-left-eq* [simp]:
 $\text{lcm } (\text{nat } |k|) n = \text{nat } (\text{lcm } k (\text{int } n))$
 $\langle \text{proof} \rangle$

lemma *lcm-nat-abs-right-eq* [simp]:
 $\text{lcm } n (\text{nat } |k|) = \text{nat } (\text{lcm } (\text{int } n) k)$
 $\langle \text{proof} \rangle$

lemma *lcm-abs1-int* [simp]:
 $\text{lcm } |x| y = \text{lcm } x y$
for $x y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *lcm-abs2-int* [simp]:
 $\text{lcm } x |y| = \text{lcm } x y$
for $x y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *abs-lcm-int* [simp]: $|\text{lcm } i j| = \text{lcm } i j$
for $i j :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-nat-induct* [case-names base step]:
fixes $m n :: \text{nat}$
assumes $\bigwedge m. P m 0$
and $\bigwedge m n. 0 < n \implies P n (m \bmod n) \implies P m n$
shows $P m n$
 $\langle \text{proof} \rangle$

lemma *gcd-neg1-int* [simp]: $\text{gcd } (- x) y = \text{gcd } x y$
for $x y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-neg2-int* [simp]: $\text{gcd } x (- y) = \text{gcd } x y$
for $x y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-cases-int*:
fixes $x y :: \text{int}$
assumes $x \geq 0 \implies y \geq 0 \implies P (\text{gcd } x y)$
and $x \geq 0 \implies y \leq 0 \implies P (\text{gcd } x (- y))$
and $x \leq 0 \implies y \geq 0 \implies P (\text{gcd } (- x) y)$
and $x \leq 0 \implies y \leq 0 \implies P (\text{gcd } (- x) (- y))$
shows $P (\text{gcd } x y)$
 $\langle \text{proof} \rangle$

lemma *gcd-ge-0-int* [simp]: $\text{gcd } (x::\text{int}) \ y \geq 0$
for $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *lcm-neg1-int*: $\text{lcm } (-x) \ y = \text{lcm } x \ y$
for $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *lcm-neg2-int*: $\text{lcm } x \ (-y) = \text{lcm } x \ y$
for $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *lcm-cases-int*:
fixes $x \ y :: \text{int}$
assumes $x \geq 0 \implies y \geq 0 \implies P \ (\text{lcm } x \ y)$
and $x \geq 0 \implies y \leq 0 \implies P \ (\text{lcm } x \ (-y))$
and $x \leq 0 \implies y \geq 0 \implies P \ (\text{lcm } (-x) \ y)$
and $x \leq 0 \implies y \leq 0 \implies P \ (\text{lcm } (-x) \ (-y))$
shows $P \ (\text{lcm } x \ y)$
 $\langle \text{proof} \rangle$

lemma *lcm-ge-0-int* [simp]: $\text{lcm } x \ y \geq 0$
for $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-0-nat*: $\text{gcd } x \ 0 = x$
for $x :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-0-int* [simp]: $\text{gcd } x \ 0 = |x|$
for $x :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-0-left-nat*: $\text{gcd } 0 \ x = x$
for $x :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-0-left-int* [simp]: $\text{gcd } 0 \ x = |x|$
for $x :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-red-nat*: $\text{gcd } x \ y = \text{gcd } y \ (x \bmod y)$
for $x \ y :: \text{nat}$
 $\langle \text{proof} \rangle$

Weaker, but useful for the simplifier.

lemma *gcd-non-0-nat*: $y \neq 0 \implies \text{gcd } x \ y = \text{gcd } y \ (x \bmod y)$
for $x \ y :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-1-nat* [*simp*]: $\text{gcd } m \ 1 = 1$
for $m :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-Suc-0* [*simp*]: $\text{gcd } m \ (\text{Suc } 0) = \text{Suc } 0$
for $m :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-1-int* [*simp*]: $\text{gcd } m \ 1 = 1$
for $m :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-idem-nat*: $\text{gcd } x \ x = x$
for $x :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-idem-int*: $\text{gcd } x \ x = |x|$
for $x :: \text{int}$
 $\langle \text{proof} \rangle$

declare *gcd-nat.simps* [*simp del*]

gcd m n divides *m* and *n*. The conjunctions don’t seem provable separately.

instance *nat :: semiring-gcd*
 $\langle \text{proof} \rangle$

instance *int :: ring-gcd*
 $\langle \text{proof} \rangle$

lemma *gcd-le1-nat* [*simp*]: $a \neq 0 \implies \text{gcd } a \ b \leq a$
for $a \ b :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-le2-nat* [*simp*]: $b \neq 0 \implies \text{gcd } a \ b \leq b$
for $a \ b :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-le1-int* [*simp*]: $a > 0 \implies \text{gcd } a \ b \leq a$
for $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-le2-int* [*simp*]: $b > 0 \implies \text{gcd } a \ b \leq b$
for $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-pos-nat* [*simp*]: $\text{gcd } m \ n > 0 \longleftrightarrow m \neq 0 \vee n \neq 0$
for $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-pos-int* [simp]: $\text{gcd } m \ n > 0 \longleftrightarrow m \neq 0 \vee n \neq 0$
for $m \ n :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-unique-nat*: $d \text{ dvd } a \wedge d \text{ dvd } b \wedge (\forall e. e \text{ dvd } a \wedge e \text{ dvd } b \longrightarrow e \text{ dvd } d) \longleftrightarrow d = \text{gcd } a \ b$
for $d \ a :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-unique-int*:
 $d \geq 0 \wedge d \text{ dvd } a \wedge d \text{ dvd } b \wedge (\forall e. e \text{ dvd } a \wedge e \text{ dvd } b \longrightarrow e \text{ dvd } d) \longleftrightarrow d = \text{gcd } a \ b$
for $d \ a :: \text{int}$
 $\langle \text{proof} \rangle$

interpretation *gcd-nat*:
semilattice-neutr-order gcd 0::nat Rings.dvd $\lambda m \ n. m \text{ dvd } n \wedge m \neq n$
 $\langle \text{proof} \rangle$

lemma *gcd-proj1-if-dvd-int* [simp]: $x \text{ dvd } y \Longrightarrow \text{gcd } x \ y = |x|$
for $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *gcd-proj2-if-dvd-int* [simp]: $y \text{ dvd } x \Longrightarrow \text{gcd } x \ y = |y|$
for $x \ y :: \text{int}$
 $\langle \text{proof} \rangle$

Multiplication laws.

lemma *gcd-mult-distrib-nat*: $k * \text{gcd } m \ n = \text{gcd } (k * m) (k * n)$
for $k \ m \ n :: \text{nat}$
 — [1, page 27]
 $\langle \text{proof} \rangle$

lemma *gcd-mult-distrib-int*: $|k| * \text{gcd } m \ n = \text{gcd } (k * m) (k * n)$
for $k \ m \ n :: \text{int}$
 $\langle \text{proof} \rangle$

Addition laws.

lemma *gcd-diff1-nat*: $m \geq n \Longrightarrow \text{gcd } (m - n) \ n = \text{gcd } m \ n$
for $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-diff2-nat*: $n \geq m \Longrightarrow \text{gcd } (n - m) \ n = \text{gcd } m \ n$
for $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *gcd-non-0-int*:

```

fixes  $x\ y :: \text{int}$ 
assumes  $y > 0$  shows  $\text{gcd } x\ y = \text{gcd } y\ (x \bmod y)$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma gcd-red-int:  $\text{gcd } x\ y = \text{gcd } y\ (x \bmod y)$ 
for  $x\ y :: \text{int}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma finite-divisors-nat [simp]:
fixes  $m :: \text{nat}$ 
assumes  $m > 0$ 
shows  $\text{finite } \{d. d \text{ dvd } m\}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma finite-divisors-int [simp]:
fixes  $i :: \text{int}$ 
assumes  $i \neq 0$ 
shows  $\text{finite } \{d. d \text{ dvd } i\}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma Max-divisors-self-nat [simp]:  $n \neq 0 \implies \text{Max } \{d::\text{nat}. d \text{ dvd } n\} = n$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma Max-divisors-self-int [simp]:
assumes  $n \neq 0$  shows  $\text{Max } \{d::\text{int}. d \text{ dvd } n\} = |n|$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma gcd-is-Max-divisors-nat:
fixes  $m\ n :: \text{nat}$ 
assumes  $n > 0$  shows  $\text{gcd } m\ n = \text{Max } \{d. d \text{ dvd } m \wedge d \text{ dvd } n\}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma gcd-is-Max-divisors-int:
fixes  $m\ n :: \text{int}$ 
assumes  $n \neq 0$  shows  $\text{gcd } m\ n = \text{Max } \{d. d \text{ dvd } m \wedge d \text{ dvd } n\}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma gcd-code-int [code]:  $\text{gcd } k\ l = \text{if } l = 0 \text{ then } k \text{ else } \text{gcd } l\ (|k| \bmod |l|)$ 
for  $k\ l :: \text{int}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma coprime-Suc-left-nat [simp]:
 $\text{coprime } (\text{Suc } n)\ n$ 
 $\langle \text{proof} \rangle$ 

```

lemma *coprime-Suc-right-nat* [simp]:
 $\text{coprime } n \ (\text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *coprime-diff-one-left-nat* [simp]:
 $\text{coprime } (n - 1) \ n \ \text{if } n > 0 \ \text{for } n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *coprime-diff-one-right-nat* [simp]:
 $\text{coprime } n \ (n - 1) \ \text{if } n > 0 \ \text{for } n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *coprime-crossproduct-nat*:
fixes $a \ b \ c \ d :: \text{nat}$
assumes $\text{coprime } a \ d$ **and** $\text{coprime } b \ c$
shows $a * c = b * d \longleftrightarrow a = b \wedge c = d$
 $\langle \text{proof} \rangle$

lemma *coprime-crossproduct-int*:
fixes $a \ b \ c \ d :: \text{int}$
assumes $\text{coprime } a \ d$ **and** $\text{coprime } b \ c$
shows $|a| * |c| = |b| * |d| \longleftrightarrow |a| = |b| \wedge |c| = |d|$
 $\langle \text{proof} \rangle$

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* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{int} * \text{int}$
where $\text{bezw } x \ y =$
 $(\text{if } y = 0 \text{ then } (1, 0)$
 else
 $(\text{snd } (\text{bezw } y \ (x \bmod y)),$
 $\text{fst } (\text{bezw } y \ (x \bmod y)) - \text{snd } (\text{bezw } y \ (x \bmod y)) * \text{int}(x \text{ div } y)))$

lemma *bezw-0* [simp]: $\text{bezw } x \ 0 = (1, 0)$
 $\langle \text{proof} \rangle$

lemma *bezw-non-0*:
 $y > 0 \implies \text{bezw } x \ y =$
 $(\text{snd } (\text{bezw } y \ (x \bmod y)), \text{fst } (\text{bezw } y \ (x \bmod y)) - \text{snd } (\text{bezw } y \ (x \bmod y)) * \text{int}(x \text{ div } y))$
 $\langle \text{proof} \rangle$

declare *bezw.simps* [simp del]

lemma *bezw-aux*: $\text{int } (\text{gcd } x \ y) = \text{fst } (\text{bezw } x \ y) * \text{int } x + \text{snd } (\text{bezw } x \ y) * \text{int } y$
 $\langle \text{proof} \rangle$

lemma *bezout-int*: $\exists u v. u * x + v * y = \text{gcd } x y$
for $x y :: \text{int}$
 $\langle \text{proof} \rangle$

Versions of Bezout for *nat*, by Amine Chaieb.

lemma *Euclid-induct* [*case-names swap zero add*]:
fixes $P :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$
assumes $c: \bigwedge a b. P a b \longleftrightarrow P b a$
and $z: \bigwedge a. P a 0$
and $\text{add}: \bigwedge a b. P a b \longrightarrow P a (a + b)$
shows $P a b$
 $\langle \text{proof} \rangle$

lemma *bezout-lemma-nat*:
fixes $d :: \text{nat}$
shows $\llbracket d \text{ dvd } a; d \text{ dvd } b; a * x = b * y + d \vee b * x = a * y + d \rrbracket$
 $\implies \exists x y. d \text{ dvd } a \wedge d \text{ dvd } a + b \wedge (a * x = (a + b) * y + d \vee (a + b) * x =$
 $a * y + d)$
 $\langle \text{proof} \rangle$

lemma *bezout-add-nat*:
 $\exists (d :: \text{nat}) x y. d \text{ dvd } a \wedge d \text{ dvd } b \wedge (a * x = b * y + d \vee b * x = a * y + d)$
 $\langle \text{proof} \rangle$

lemma *bezout1-nat*: $\exists (d :: \text{nat}) x y. d \text{ dvd } a \wedge d \text{ dvd } b \wedge (a * x - b * y = d \vee b * x - a * y = d)$
 $\langle \text{proof} \rangle$

lemma *bezout-add-strong-nat*:
fixes $a b :: \text{nat}$
assumes $a: a \neq 0$
shows $\exists d x y. d \text{ dvd } a \wedge d \text{ dvd } b \wedge a * x = b * y + d$
 $\langle \text{proof} \rangle$

lemma *bezout-nat*:
fixes $a :: \text{nat}$
assumes $a: a \neq 0$
shows $\exists x y. a * x = b * y + \text{gcd } a b$
 $\langle \text{proof} \rangle$

87.8 LCM properties on *nat* and *int*

lemma *lcm-altdef-int* [*code*]: $\text{lcm } a b = |a| * |b| \text{ div } \text{gcd } a b$
for $a b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *prod-gcd-lcm-nat*: $m * n = \text{gcd } m n * \text{lcm } m n$

for $m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *prod-gcd-lcm-int*: $|m| * |n| = \text{gcd } m\ n * \text{lcm } m\ n$
for $m\ n :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *lcm-pos-nat*: $m > 0 \implies n > 0 \implies \text{lcm } m\ n > 0$
for $m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *lcm-pos-int*: $m \neq 0 \implies n \neq 0 \implies \text{lcm } m\ n > 0$
for $m\ n :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *dvd-pos-nat*: $n > 0 \implies m\ \text{dvd } n \implies m > 0$
for $m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *lcm-unique-nat*:
 $a\ \text{dvd } d \wedge b\ \text{dvd } d \wedge (\forall e. a\ \text{dvd } e \wedge b\ \text{dvd } e \longrightarrow d\ \text{dvd } e) \longleftrightarrow d = \text{lcm } a\ b$
for $a\ b\ d :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *lcm-unique-int*:
 $d \geq 0 \wedge a\ \text{dvd } d \wedge b\ \text{dvd } d \wedge (\forall e. a\ \text{dvd } e \wedge b\ \text{dvd } e \longrightarrow d\ \text{dvd } e) \longleftrightarrow d = \text{lcm } a\ b$
for $a\ b\ d :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *lcm-proj2-if-dvd-nat* [simp]: $x\ \text{dvd } y \implies \text{lcm } x\ y = y$
for $x\ y :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *lcm-proj2-if-dvd-int* [simp]: $x\ \text{dvd } y \implies \text{lcm } x\ y = |y|$
for $x\ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *lcm-proj1-if-dvd-nat* [simp]: $x\ \text{dvd } y \implies \text{lcm } y\ x = y$
for $x\ y :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *lcm-proj1-if-dvd-int* [simp]: $x\ \text{dvd } y \implies \text{lcm } y\ x = |y|$
for $x\ y :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *lcm-proj1-iff-nat* [simp]: $\text{lcm } m\ n = m \longleftrightarrow n\ \text{dvd } m$
for $m\ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *lcm-proj2-iff-nat* [simp]: $\text{lcm } m \ n = n \longleftrightarrow m \ \text{dvd } n$
for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *lcm-proj1-iff-int* [simp]: $\text{lcm } m \ n = |m| \longleftrightarrow n \ \text{dvd } m$
for $m \ n :: \text{int}$
 ⟨proof⟩

lemma *lcm-proj2-iff-int* [simp]: $\text{lcm } m \ n = |n| \longleftrightarrow m \ \text{dvd } n$
for $m \ n :: \text{int}$
 ⟨proof⟩

lemma *lcm-1-iff-nat* [simp]: $\text{lcm } m \ n = \text{Suc } 0 \longleftrightarrow m = \text{Suc } 0 \wedge n = \text{Suc } 0$
for $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *lcm-1-iff-int* [simp]: $\text{lcm } m \ n = 1 \longleftrightarrow (m = 1 \vee m = -1) \wedge (n = 1 \vee n = -1)$
for $m \ n :: \text{int}$
 ⟨proof⟩

87.9 The complete divisibility lattice on *nat* and *int*

Lifting *gcd* and *lcm* to sets (*Gcd* / *Lcm*). *Gcd* is defined via *Lcm* to facilitate the proof that we have a complete lattice.

instantiation $\text{nat} :: \text{semiring-Gcd}$
begin

interpretation *semilattice-neutr-set lcm 1::nat*
 ⟨proof⟩

definition $\text{Lcm } M = (\text{if finite } M \text{ then } F \ M \text{ else } 0)$ **for** $M :: \text{nat set}$

lemma *Lcm-nat-empty*: $\text{Lcm } \{\} = (1::\text{nat})$
 ⟨proof⟩

lemma *Lcm-nat-insert*: $\text{Lcm } (\text{insert } n \ M) = \text{lcm } n \ (\text{Lcm } M)$ **for** $n :: \text{nat}$
 ⟨proof⟩

lemma *Lcm-nat-infinite*: $\text{infinite } M \implies \text{Lcm } M = 0$ **for** $M :: \text{nat set}$
 ⟨proof⟩

lemma *dvd-Lcm-nat* [simp]:
fixes $M :: \text{nat set}$
assumes $m \in M$
shows $m \ \text{dvd } \text{Lcm } M$
 ⟨proof⟩

lemma *Lcm-dvd-nat* [*simp*]:
fixes $M :: \text{nat set}$
assumes $\forall m \in M. m \text{ dvd } n$
shows $\text{Lcm } M \text{ dvd } n$
 $\langle \text{proof} \rangle$

definition $\text{Gcd } M = \text{Lcm } \{d. \forall m \in M. d \text{ dvd } m\}$ **for** $M :: \text{nat set}$

instance
 $\langle \text{proof} \rangle$

end

lemma *Gcd-nat-eq-one*: $1 \in N \implies \text{Gcd } N = 1$
for $N :: \text{nat set}$
 $\langle \text{proof} \rangle$

instance $\text{nat} :: \text{semiring-gcd-mult-normalize}$
 $\langle \text{proof} \rangle$

Alternative characterizations of Gcd:

lemma *Gcd-eq-Max*:
fixes $M :: \text{nat set}$
assumes $\text{finite } (M :: \text{nat set})$ **and** $M \neq \{\}$ **and** $0 \notin M$
shows $\text{Gcd } M = \text{Max } (\bigcap m \in M. \{d. d \text{ dvd } m\})$
 $\langle \text{proof} \rangle$

lemma *Gcd-remove0-nat*: $\text{Gcd } M = \text{Gcd } (M - \{0\})$
for $M :: \text{nat set}$
 $\langle \text{proof} \rangle$

lemma *Lcm-in-lcm-closed-set-nat*:
fixes $M :: \text{nat set}$
assumes $\text{finite } M$ $M \neq \{\}$ $\bigwedge m n. \llbracket m \in M; n \in M \rrbracket \implies \text{lcm } m n \in M$
shows $\text{Lcm } M \in M$
 $\langle \text{proof} \rangle$

lemma *Lcm-eq-Max-nat*:
fixes $M :: \text{nat set}$
assumes $M: \text{finite } M$ $M \neq \{\}$ $0 \notin M$ **and** $\text{lcm}: \bigwedge m n. \llbracket m \in M; n \in M \rrbracket \implies \text{lcm } m n \in M$
shows $\text{Lcm } M = \text{Max } M$
 $\langle \text{proof} \rangle$

lemma *mult-inj-if-coprime-nat*:
 $\text{inj-on } f A \implies \text{inj-on } g B \implies (\bigwedge a b. \llbracket a \in A; b \in B \rrbracket \implies \text{coprime } (f a) (g b)) \implies$
 $\text{inj-on } (\lambda(a, b). f a * g b) (A \times B)$
for $f :: 'a \Rightarrow \text{nat}$ **and** $g :: 'b \Rightarrow \text{nat}$
 $\langle \text{proof} \rangle$

87.9.1 Setwise GCD and LCM for integers**instantiation** $int :: Gcd$ **begin****definition** $Gcd-int :: int\ set \Rightarrow int$ **where** $Gcd\ K = int\ (GCD\ k \in K. (nat \circ abs)\ k)$ **definition** $Lcm-int :: int\ set \Rightarrow int$ **where** $Lcm\ K = int\ (LCM\ k \in K. (nat \circ abs)\ k)$ **instance** $\langle proof \rangle$ **end****lemma** $Gcd-int-eq\ [simp]:$ $(GCD\ n \in N. int\ n) = int\ (Gcd\ N)$ $\langle proof \rangle$ **lemma** $Gcd-nat-abs-eq\ [simp]:$ $(GCD\ k \in K. nat\ |k|) = nat\ (Gcd\ K)$ $\langle proof \rangle$ **lemma** $abs-Gcd-eq\ [simp]:$ $|Gcd\ K| = Gcd\ K$ **for** $K :: int\ set$ $\langle proof \rangle$ **lemma** $uminus-Gcd-eq\ [simp]:$ **fixes** $K :: int\ set$ **shows** $Gcd\ (uminus\ 'K) = Gcd\ K$ $\langle proof \rangle$ **lemma** $Gcd-int-greater-eq-0\ [simp]:$ $Gcd\ K \geq 0$ **for** $K :: int\ set$ $\langle proof \rangle$ **lemma** $Gcd-abs-eq\ [simp]:$ $(GCD\ k \in K. |k|) = Gcd\ K$ **for** $K :: int\ set$ $\langle proof \rangle$ **lemma** $Lcm-int-eq\ [simp]:$ $(LCM\ n \in N. int\ n) = int\ (Lcm\ N)$ $\langle proof \rangle$ **lemma** $Lcm-nat-abs-eq\ [simp]:$ $(LCM\ k \in K. nat\ |k|) = nat\ (Lcm\ K)$ $\langle proof \rangle$

lemma *abs-Lcm-eq* [simp]:
 $|Lcm\ K| = Lcm\ K$ **for** $K :: int\ set$
 ⟨proof⟩

lemma *Lcm-int-greater-eq-0* [simp]:
 $Lcm\ K \geq 0$
for $K :: int\ set$
 ⟨proof⟩

lemma *Lcm-abs-eq* [simp]:
 $(LCM\ k \in K.\ |k|) = Lcm\ K$
for $K :: int\ set$
 ⟨proof⟩

instance *int* :: *semiring-Gcd*
 ⟨proof⟩

instance *int* :: *semiring-gcd-mult-normalize*
 ⟨proof⟩

87.10 GCD and LCM on *integer*

instantiation *integer* :: *gcd*
begin

context
includes *integer.lifting*
begin

lift-definition *gcd-integer* :: *integer* \Rightarrow *integer* \Rightarrow *integer* **is** *gcd* ⟨proof⟩

lift-definition *lcm-integer* :: *integer* \Rightarrow *integer* \Rightarrow *integer* **is** *lcm* ⟨proof⟩

end

instance ⟨proof⟩

end

lifting-update *integer.lifting*
lifting-forget *integer.lifting*

context
includes *integer.lifting*
begin

lemma *gcd-code-integer* [code]: $gcd\ k\ l = |if\ l = (0::integer)\ then\ k\ else\ gcd\ l\ (|k| \bmod |l|)|$
 ⟨proof⟩

```

lemma lcm-code-integer [code]: lcm a b = |a| * |b| div gcd a b
  for a b :: integer
  ⟨proof⟩

```

end

code-printing

```

constant gcd :: integer ⇒ - ⇒
  (OCaml) !(fun k l -> if Z.equal k Z.zero then/ Z.abs l else if Z.equal/ l Z.zero
then Z.abs k else Z.gcd k l)
and (Haskell) Prelude.gcd
and (Scala) -.gcd'((-)')
  — There is no gcd operation in the SML standard library, so no code setup for
  SML

```

Some code equations

```

lemmas Gcd-nat-set-eq-fold [code] = Gcd-set-eq-fold [where ?'a = nat]
lemmas Lcm-nat-set-eq-fold [code] = Lcm-set-eq-fold [where ?'a = nat]
lemmas Gcd-int-set-eq-fold [code] = Gcd-set-eq-fold [where ?'a = int]
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 ⟷ m = 0 ∨ n = 0
  for m n :: nat
  ⟨proof⟩

```

```

lemma lcm-0-iff-int [simp]: lcm m n = 0 ⟷ m = 0 ∨ n = 0
  for m n :: int
  ⟨proof⟩

```

```

lemma dvd-lcm-I1-nat [simp]: k dvd m ⟹ k dvd lcm m n
  for k m n :: nat
  ⟨proof⟩

```

```

lemma dvd-lcm-I2-nat [simp]: k dvd n ⟹ k dvd lcm m n
  for k m n :: nat
  ⟨proof⟩

```

```

lemma dvd-lcm-I1-int [simp]: i dvd m ⟹ i dvd lcm m n
  for i m n :: int
  ⟨proof⟩

```

```

lemma dvd-lcm-I2-int [simp]: i dvd n ⟹ i dvd lcm m n
  for i m n :: int
  ⟨proof⟩

```

```

lemmas Gcd-dvd-nat [simp] = Gcd-dvd [where ?'a = nat]
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]: $m \in M \implies m \text{ dvd } \text{Lcm } M$
for $M :: \text{int set}$
 <proof>

lemma *gcd-neg-numeral-1-int* [simp]: $\text{gcd } (- \text{numeral } n :: \text{int}) \ x = \text{gcd } (\text{numeral } n) \ x$
 <proof>

lemma *gcd-neg-numeral-2-int* [simp]: $\text{gcd } x \ (- \text{numeral } n :: \text{int}) = \text{gcd } x \ (\text{numeral } n)$
 <proof>

lemma *gcd-proj1-if-dvd-nat* [simp]: $x \text{ dvd } y \implies \text{gcd } x \ y = x$
for $x \ y :: \text{nat}$
 <proof>

lemma *gcd-proj2-if-dvd-nat* [simp]: $y \text{ dvd } x \implies \text{gcd } x \ y = y$
for $x \ y :: \text{nat}$
 <proof>

lemma *Gcd-in*:
fixes $A :: \text{nat set}$
assumes $\bigwedge a \ b. a \in A \implies b \in A \implies \text{gcd } a \ b \in A$
assumes $A \neq \{\}$
shows $\text{Gcd } A \in A$
 <proof>

lemma *bezout-gcd-nat'*:
fixes $a \ b :: \text{nat}$
shows $\exists x \ y. b * y \leq a * x \wedge a * x - b * y = \text{gcd } a \ b \vee a * y \leq b * x \wedge b * x - a * y = \text{gcd } a \ b$
 <proof>

lemmas *Lcm-eq-0-I-nat* [simp] = *Lcm-eq-0-I* [where ?'a = nat]
lemmas *Lcm-0-iff-nat* [simp] = *Lcm-0-iff* [where ?'a = nat]
lemmas *Lcm-least-int* [simp] = *Lcm-least* [where ?'a = int]

87.11 Characteristic of a semiring

definition (in *semiring-1*) *semiring-char* :: 'a itself \Rightarrow nat
where *semiring-char* - = $\text{Gcd } \{n. \text{of-nat } n = (0 :: 'a)\}$

syntax *-type-char* :: type \Rightarrow nat ($\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix } \text{CHAR} \rangle \rangle \text{CHAR} / (1 '(-))) \rangle$)
syntax-consts *-type-char* \Rightarrow *semiring-char*
translations $\text{CHAR}(t) \mapsto \text{CONST } \text{semiring-char } (\text{CONST } \text{Pure.type} :: t \text{ itself})$
 <ML>

context *semiring-1*

begin

lemma *of-nat-CHAR* [*simp*]: *of-nat CHAR('a) = (0 :: 'a)*
 ⟨*proof*⟩

lemma *of-nat-eq-0-iff-char-dvd*: *of-nat n = (0 :: 'a) \longleftrightarrow CHAR('a) dvd n*
 ⟨*proof*⟩

lemma *CHAR-eqI*:
assumes *of-nat c = (0 :: 'a)*
assumes $\bigwedge x. \text{of-nat } x = (0 :: 'a) \implies c \text{ dvd } x$
shows $\text{CHAR}('a) = c$
 ⟨*proof*⟩

lemma *CHAR-eq0-iff*: $\text{CHAR}('a) = 0 \longleftrightarrow (\forall n > 0. \text{of-nat } n \neq (0 :: 'a))$
 ⟨*proof*⟩

lemma *CHAR-pos-iff*: $\text{CHAR}('a) > 0 \longleftrightarrow (\exists n > 0. \text{of-nat } n = (0 :: 'a))$
 ⟨*proof*⟩

lemma *CHAR-eq-posI*:
assumes $c > 0$ *of-nat c = (0 :: 'a)* $\bigwedge x. x > 0 \implies x < c \implies \text{of-nat } x \neq (0 :: 'a)$
shows $\text{CHAR}('a) = c$
 ⟨*proof*⟩

end

lemma (**in** *semiring-char-0*) *CHAR-eq-0* [*simp*]: $\text{CHAR}('a) = 0$
 ⟨*proof*⟩

lemma *CHAR-not-1* [*simp*]: $\text{CHAR}('a :: \{\text{semiring-1}, \text{zero-neq-one}\}) \neq \text{Suc } 0$
 ⟨*proof*⟩

lemma (**in** *idom*) *CHAR-not-1'* [*simp*]: $\text{CHAR}('a) \neq \text{Suc } 0$
 ⟨*proof*⟩

lemma (**in** *ring-1*) *uminus-CHAR-2*:
assumes $\text{CHAR}('a) = 2$
shows $-(x :: 'a) = x$
 ⟨*proof*⟩

lemma (**in** *ring-1*) *minus-CHAR-2*:
assumes $\text{CHAR}('a) = 2$
shows $(x - y :: 'a) = x + y$
 ⟨*proof*⟩

```

lemma (in semiring-1-cancel) of-nat-eq-iff-char-dvd:
  assumes  $m < n$ 
  shows  $\text{of-nat } m = (\text{of-nat } n :: 'a) \longleftrightarrow \text{CHAR}('a) \text{ dvd } (n - m)$ 
  <proof>

```

```

lemma (in ring-1) of-int-eq-0-iff-char-dvd:
   $(\text{of-int } n = (0 :: 'a)) = (\text{int } \text{CHAR}('a) \text{ dvd } n)$ 
  <proof>

```

```

lemma (in semiring-1-cancel) finite-imp-CHAR-pos:
  assumes finite (UNIV :: 'a set)
  shows  $\text{CHAR}('a) > 0$ 
  <proof>

```

```

end

```

88 Nitpick: Yet Another Counterexample Generator for Isabelle/HOL

```

theory Nitpick
imports Record GCD
keywords
  nitpick :: diag and
  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
  unknown :: 'a
  is-unknown :: 'a  $\Rightarrow$  bool
  bisim :: bisim-iterator  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  bisim-iterator-max :: bisim-iterator
  Quot :: 'a  $\Rightarrow$  'b
  safe-The :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a

```

Alternative definitions.

```

lemma Ex1-unfold[nitpick-unfold]:  $\text{Ex1 } P \equiv \exists x. \{x. P x\} = \{x\}$ 
  <proof>

```

```

lemma rtranc1-unfold[nitpick-unfold]:  $r^* \equiv (r^+)^=$ 

```

$\langle \text{proof} \rangle$

lemma *rtrancp-unfold[nitpick-unfold]*: $\text{rtrancp } r \ a \ b \equiv (a = b \vee \text{trancp } r \ a \ b)$
 $\langle \text{proof} \rangle$

lemma *trancp-unfold[nitpick-unfold]*:
 $\text{trancp } r \ a \ b \equiv (a, b) \in \text{trancp } \{(x, y). \ r \ x \ y\}$
 $\langle \text{proof} \rangle$

lemma *[nitpick-simp]*:
 $\text{of-nat } n \equiv (\text{if } n = 0 \text{ then } 0 \text{ else } 1 + \text{of-nat } (n - 1))$
 $\langle \text{proof} \rangle$

definition *prod* :: $'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow ('a \times 'b) \text{ set}$ **where**
 $\text{prod } A \ B = \{(a, b). \ a \in A \wedge b \in B\}$

definition *refl'* :: $('a \times 'a) \text{ set} \Rightarrow \text{bool}$ **where**
 $\text{refl}' \ r \equiv \forall x. \ (x, x) \in r$

definition *wf'* :: $('a \times 'a) \text{ set} \Rightarrow \text{bool}$ **where**
 $\text{wf}' \ r \equiv \text{acyclic } r \wedge (\text{finite } r \vee \text{unknown})$

definition *card'* :: $'a \text{ set} \Rightarrow \text{nat}$ **where**
 $\text{card}' \ A \equiv \text{if finite } A \text{ then length } (\text{SOME } xs. \ \text{set } xs = A \wedge \text{distinct } xs) \text{ else } 0$

definition *sum'* :: $('a \Rightarrow 'b::\text{comm-monoid-add}) \Rightarrow 'a \text{ set} \Rightarrow 'b$ **where**
 $\text{sum}' \ f \ A \equiv \text{if finite } A \text{ then sum-list } (\text{map } f \ (\text{SOME } xs. \ \text{set } xs = A \wedge \text{distinct } xs))$
 $\text{else } 0$

inductive *fold-graph'* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow \text{bool}$ **where**
 $\text{fold-graph}' \ f \ z \ \{\} \ z \mid$
 $\llbracket x \in A; \text{fold-graph}' \ f \ z \ (A - \{x\}) \ y \rrbracket \Longrightarrow \text{fold-graph}' \ f \ z \ A \ (f \ x \ y)$

The following lemmas are not strictly necessary but they help the *specialize* optimization.

lemma *The-psimp[nitpick-psimp]*: $P = (=) \ x \Longrightarrow \text{The } P = x$
 $\langle \text{proof} \rangle$

lemma *Eps-psimp[nitpick-psimp]*:
 $\llbracket P \ x; \neg P \ y; \text{Eps } P = y \rrbracket \Longrightarrow \text{Eps } P = x$
 $\langle \text{proof} \rangle$

lemma *case-unit-unfold[nitpick-unfold]*:
 $\text{case-unit } x \ u \equiv x$
 $\langle \text{proof} \rangle$

declare *unit.case[nitpick-simp del]*

lemma *case-nat-unfold[nitpick-unfold]*:

case-nat $x f n \equiv \text{if } n = 0 \text{ then } x \text{ else } f (n - 1)$
 ⟨proof⟩

declare *nat.case*[*nitpick-simp del*]

lemma *size-list-simp*[*nitpick-simp*]:

size-list $f xs = (\text{if } xs = [] \text{ then } 0 \text{ else } \text{Suc } (f (\text{hd } xs) + \text{size-list } f (\text{tl } xs)))$
size $xs = (\text{if } xs = [] \text{ then } 0 \text{ else } \text{Suc } (\text{size } (\text{tl } xs)))$
 ⟨proof⟩

Auxiliary definitions used to provide an alternative representation for *rat* and *real*.

fun *nat-gcd* :: *nat* \Rightarrow *nat* \Rightarrow *nat* **where**

nat-gcd $x y = (\text{if } y = 0 \text{ then } x \text{ else } \text{nat-gcd } y (x \bmod y))$

declare *nat-gcd.simps* [*simp del*]

definition *nat-lcm* :: *nat* \Rightarrow *nat* \Rightarrow *nat* **where**

nat-lcm $x y = x * y \text{ div } (\text{nat-gcd } x y)$

lemma *gcd-eq-nitpick-gcd* [*nitpick-unfold*]:

gcd $x y = \text{Nitpick.nat-gcd } x y$
 ⟨proof⟩

lemma *lcm-eq-nitpick-lcm* [*nitpick-unfold*]:

lcm $x y = \text{Nitpick.nat-lcm } x y$
 ⟨proof⟩

definition *Frac* :: *int* \times *int* \Rightarrow *bool* **where**

Frac $\equiv \lambda(a, b). b > 0 \wedge \text{coprime } a b$

consts

Abs-Frac :: *int* \times *int* \Rightarrow *'a*

Rep-Frac :: *'a* \Rightarrow *int* \times *int*

definition *zero-frac* :: *'a* **where**

zero-frac $\equiv \text{Abs-Frac } (0, 1)$

definition *one-frac* :: *'a* **where**

one-frac $\equiv \text{Abs-Frac } (1, 1)$

definition *num* :: *'a* \Rightarrow *int* **where**

num $\equiv \text{fst} \circ \text{Rep-Frac}$

definition *denom* :: *'a* \Rightarrow *int* **where**

denom $\equiv \text{snd} \circ \text{Rep-Frac}$

function *norm-frac* :: *int* \Rightarrow *int* \Rightarrow *int* \times *int* **where**

norm-frac $a b =$

(if $b < 0$ then $\text{norm-frac } (- a) (- b)$
 else if $a = 0 \vee b = 0$ then $(0, 1)$
 else let $c = \text{gcd } a \ b$ in $(a \text{ div } c, b \text{ div } c)$)
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$

declare $\text{norm-frac.simps}[\text{simp del}]$

definition $\text{frac} :: \text{int} \Rightarrow \text{int} \Rightarrow 'a \text{ where}$
 $\text{frac } a \ b \equiv \text{Abs-Frac } (\text{norm-frac } a \ b)$

definition $\text{plus-frac} :: 'a \Rightarrow 'a \Rightarrow 'a \text{ where}$
 $[\text{nitpick-simp}]: \text{plus-frac } q \ r = (\text{let } d = \text{lcm } (\text{denom } q) (\text{denom } r) \text{ in}$
 $\text{frac } (\text{num } q * (d \text{ div } \text{denom } q) + \text{num } r * (d \text{ div } \text{denom } r)) \ d)$

definition $\text{times-frac} :: 'a \Rightarrow 'a \Rightarrow 'a \text{ where}$
 $[\text{nitpick-simp}]: \text{times-frac } q \ r = \text{frac } (\text{num } q * \text{num } r) (\text{denom } q * \text{denom } r)$

definition $\text{uminus-frac} :: 'a \Rightarrow 'a \text{ where}$
 $\text{uminus-frac } q \equiv \text{Abs-Frac } (- \text{num } q, \text{denom } q)$

definition $\text{number-of-frac} :: \text{int} \Rightarrow 'a \text{ where}$
 $\text{number-of-frac } n \equiv \text{Abs-Frac } (n, 1)$

definition $\text{inverse-frac} :: 'a \Rightarrow 'a \text{ where}$
 $\text{inverse-frac } q \equiv \text{frac } (\text{denom } q) (\text{num } q)$

definition $\text{less-frac} :: 'a \Rightarrow 'a \Rightarrow \text{bool} \text{ where}$
 $[\text{nitpick-simp}]: \text{less-frac } q \ r \longleftrightarrow \text{num } (\text{plus-frac } q (\text{uminus-frac } r)) < 0$

definition $\text{less-eq-frac} :: 'a \Rightarrow 'a \Rightarrow \text{bool} \text{ where}$
 $[\text{nitpick-simp}]: \text{less-eq-frac } q \ r \longleftrightarrow \text{num } (\text{plus-frac } q (\text{uminus-frac } r)) \leq 0$

definition $\text{of-frac} :: 'a \Rightarrow 'b::\{\text{inverse,ring-1}\} \text{ where}$
 $\text{of-frac } q \equiv \text{of-int } (\text{num } q) / \text{of-int } (\text{denom } q)$

axiomatization $\text{wf-wfrec} :: ('a \times 'a) \text{ set} \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$

definition $\text{wf-wfrec}' :: ('a \times 'a) \text{ set} \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \text{ where}$
 $[\text{nitpick-simp}]: \text{wf-wfrec}' \ R \ F \ x = F (\text{cut } (\text{wf-wfrec } R \ F) \ R \ x) \ x$

definition $\text{wfrec}' :: ('a \times 'a) \text{ set} \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \text{ where}$
 $\text{wfrec}' \ R \ F \ x \equiv \text{if } \text{wf } R \text{ then } \text{wf-wfrec}' \ R \ F \ x \text{ else } \text{THE } y. \text{wfrec-rel } R \ (\lambda f \ x. F (\text{cut } f \ R \ x) \ x) \ x \ y$

$\langle \text{ML} \rangle$

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) Ex1-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

```

```

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>

```

```

hide-const (open) unreachable The-unsafe rmember

```

```

end

```

89 Greatest Fixpoint (Codata-type) 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*: $\llbracket \bigwedge x. s = x \implies P \rrbracket \implies P$
 $\langle proof \rangle$

lemma *obj-sumE*: $\llbracket \forall x. s = Inl\ x \longrightarrow P; \forall x. s = Inr\ x \longrightarrow P \rrbracket \implies P$
 $\langle proof \rangle$

lemma *not-TrueE*: $\neg\ True \implies P$
 $\langle proof \rangle$

lemma *neq-eq-eq-contradict*: $\llbracket t \neq u; s = t; s = u \rrbracket \implies P$
 $\langle proof \rangle$

lemma *converse-Times*: $(A \times B)^{-1} = B \times A$
 $\langle proof \rangle$

lemma *equiv-proj*:
assumes *e*: *equiv* *A R* **and** *m*: $z \in R$
shows $(proj\ R \circ fst)\ z = (proj\ R \circ snd)\ z$
 $\langle proof \rangle$

definition *image2* **where** $image2\ A\ f\ g = \{(f\ a, g\ a) \mid a. a \in A\}$

lemma *Id-on-Gr*: $Id-on\ A = Gr\ A\ id$
 $\langle proof \rangle$

lemma *image2-eqI*: $\llbracket b = f\ x; c = g\ x; x \in A \rrbracket \implies (b, c) \in image2\ A\ f\ g$
 $\langle proof \rangle$

lemma *IdD*: $(a, b) \in Id \implies a = b$
 $\langle proof \rangle$

lemma *image2-Gr*: $image2\ A\ f\ g = (Gr\ A\ f)^{-1} \circ (Gr\ A\ g)$
 $\langle proof \rangle$

lemma *GrD1*: $(x, fx) \in Gr\ A\ f \implies x \in A$
 $\langle proof \rangle$

lemma *GrD2*: $(x, fx) \in Gr\ A\ f \implies f\ x = fx$
 $\langle proof \rangle$

lemma *Gr-incl*: $Gr\ A\ f \subseteq A \times B \longleftrightarrow f\ 'A \subseteq B$
 $\langle proof \rangle$

lemma *subset-Collect-iff*: $B \subseteq A \implies (B \subseteq \{x \in A. P\ x\}) = (\forall x \in B. P\ x)$

$\langle \text{proof} \rangle$

lemma *subset-CollectI*: $B \subseteq A \implies (\bigwedge x. x \in B \implies Q\ x \implies P\ x) \implies (\{x \in B. Q\ x\} \subseteq \{x \in A. P\ x\})$
 $\langle \text{proof} \rangle$

lemma *in-rel-Collect-case-prod-eq*: $\text{in-rel } (\text{Collect } (\text{case-prod } X)) = X$
 $\langle \text{proof} \rangle$

lemma *Collect-case-prod-in-rel-leI*: $X \subseteq Y \implies X \subseteq \text{Collect } (\text{case-prod } (\text{in-rel } Y))$
 $\langle \text{proof} \rangle$

lemma *Collect-case-prod-in-rel-leE*: $X \subseteq \text{Collect } (\text{case-prod } (\text{in-rel } Y)) \implies (X \subseteq Y \implies R) \implies R$
 $\langle \text{proof} \rangle$

lemma *conversep-in-rel*: $(\text{in-rel } R)^{-1-1} = \text{in-rel } (R^{-1})$
 $\langle \text{proof} \rangle$

lemma *relcompp-in-rel*: $\text{in-rel } R \text{ OO } \text{in-rel } S = \text{in-rel } (R \text{ O } S)$
 $\langle \text{proof} \rangle$

lemma *in-rel-Gr*: $\text{in-rel } (Gr\ A\ f) = Grp\ A\ f$
 $\langle \text{proof} \rangle$

definition *relImage where*
 $\text{relImage } R\ f \equiv \{(f\ a1, f\ a2) \mid a1\ a2. (a1, a2) \in R\}$

definition *relInvImage where*
 $\text{relInvImage } A\ R\ f \equiv \{(a1, a2) \mid a1\ a2. a1 \in A \wedge a2 \in A \wedge (f\ a1, f\ a2) \in R\}$

lemma *relImage-Gr*:
 $\llbracket R \subseteq A \times A \rrbracket \implies \text{relImage } R\ f = (Gr\ A\ f)^{-1} \text{ O } R \text{ O } Gr\ A\ f$
 $\langle \text{proof} \rangle$

lemma *relInvImage-Gr*: $\llbracket R \subseteq B \times B \rrbracket \implies \text{relInvImage } A\ R\ f = Gr\ A\ f \text{ O } R \text{ O } (Gr\ A\ f)^{-1}$
 $\langle \text{proof} \rangle$

lemma *relImage-mono*:
 $R1 \subseteq R2 \implies \text{relImage } R1\ f \subseteq \text{relImage } R2\ f$
 $\langle \text{proof} \rangle$

lemma *relInvImage-mono*:
 $R1 \subseteq R2 \implies \text{relInvImage } A\ R1\ f \subseteq \text{relInvImage } A\ R2\ f$
 $\langle \text{proof} \rangle$

lemma *relInvImage-Id-on*:

$(\bigwedge a1\ a2. f\ a1 = f\ a2 \longleftrightarrow a1 = a2) \implies \text{relInvImage}\ A\ (\text{Id-on}\ B)\ f \subseteq \text{Id}$
 $\langle \text{proof} \rangle$

lemma *relInvImage-UNIV-relImage*:
 $R \subseteq \text{relInvImage}\ \text{UNIV}\ (\text{relImage}\ R\ f)\ f$
 $\langle \text{proof} \rangle$

lemma *relImage-proj*:
assumes *equiv* $A\ R$
shows $\text{relImage}\ R\ (\text{proj}\ R) \subseteq \text{Id-on}\ (A//R)$
 $\langle \text{proof} \rangle$

lemma *relImage-relInvImage*:
assumes $R \subseteq f\ ' A \times f\ ' A$
shows $\text{relImage}\ (\text{relInvImage}\ A\ R\ f)\ f = R$
 $\langle \text{proof} \rangle$

lemma *subst-Pair*: $P\ x\ y \implies a = (x, y) \implies P\ (\text{fst}\ a)\ (\text{snd}\ a)$
 $\langle \text{proof} \rangle$

lemma *fst-diag-id*: $(\text{fst} \circ (\lambda x. (x, x)))\ z = \text{id}\ z\ \langle \text{proof} \rangle$

lemma *snd-diag-id*: $(\text{snd} \circ (\lambda x. (x, x)))\ z = \text{id}\ z\ \langle \text{proof} \rangle$

lemma *fst-diag-fst*: $\text{fst} \circ ((\lambda x. (x, x)) \circ \text{fst}) = \text{fst}\ \langle \text{proof} \rangle$

lemma *snd-diag-fst*: $\text{snd} \circ ((\lambda x. (x, x)) \circ \text{fst}) = \text{fst}\ \langle \text{proof} \rangle$

lemma *fst-diag-snd*: $\text{fst} \circ ((\lambda x. (x, x)) \circ \text{snd}) = \text{snd}\ \langle \text{proof} \rangle$

lemma *snd-diag-snd*: $\text{snd} \circ ((\lambda x. (x, x)) \circ \text{snd}) = \text{snd}\ \langle \text{proof} \rangle$

definition *Succ* **where** $\text{Succ}\ Kl\ kl = \{k . kl\ @\ [k] \in Kl\}$

definition *Shift* **where** $\text{Shift}\ Kl\ k = \{kl. k\ \# kl \in Kl\}$

definition *shift* **where** $\text{shift}\ lab\ k = (\lambda kl. lab\ (k\ \# kl))$

lemma *empty-Shift*: $\llbracket \rrbracket \in Kl; k \in \text{Succ}\ Kl\ \llbracket \rrbracket \implies \llbracket \rrbracket \in \text{Shift}\ Kl\ k$
 $\langle \text{proof} \rangle$

lemma *SuccD*: $k \in \text{Succ}\ Kl\ kl \implies kl\ @\ [k] \in Kl$
 $\langle \text{proof} \rangle$

lemmas $\text{SuccE} = \text{SuccD}[\text{elim-format}]$

lemma *SuccI*: $kl\ @\ [k] \in Kl \implies k \in \text{Succ}\ Kl\ kl$
 $\langle \text{proof} \rangle$

lemma *ShiftD*: $kl \in \text{Shift}\ Kl\ k \implies k\ \# kl \in Kl$
 $\langle \text{proof} \rangle$

lemma *Succ-Shift*: $\text{Succ}\ (\text{Shift}\ Kl\ k)\ kl = \text{Succ}\ Kl\ (k\ \# kl)$
 $\langle \text{proof} \rangle$

lemma *length-Cons*: $\text{length } (x \# xs) = \text{Suc } (\text{length } xs)$
 $\langle \text{proof} \rangle$

lemma *length-append-singleton*: $\text{length } (xs @ [x]) = \text{Suc } (\text{length } xs)$
 $\langle \text{proof} \rangle$

definition *toCard-pred* $A \ r \ f \equiv \text{inj-on } f \ A \wedge f' A \subseteq \text{Field } r \wedge \text{Card-order } r$

definition *toCard* $A \ r \equiv \text{SOME } f. \text{toCard-pred } A \ r \ f$

lemma *ex-toCard-pred*:

$\llbracket |A| \leq_o r; \text{Card-order } r \rrbracket \implies \exists f. \text{toCard-pred } A \ r \ f$
 $\langle \text{proof} \rangle$

lemma *toCard-pred-toCard*:

$\llbracket |A| \leq_o r; \text{Card-order } r \rrbracket \implies \text{toCard-pred } A \ r \ (\text{toCard } A \ r)$
 $\langle \text{proof} \rangle$

lemma *toCard-inj*: $\llbracket |A| \leq_o r; \text{Card-order } r; x \in A; y \in A \rrbracket \implies \text{toCard } A \ r \ x = \text{toCard } A \ r \ y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

definition *fromCard* $A \ r \ k \equiv \text{SOME } b. b \in A \wedge \text{toCard } A \ r \ b = k$

lemma *fromCard-toCard*:

$\llbracket |A| \leq_o r; \text{Card-order } r; b \in A \rrbracket \implies \text{fromCard } A \ r \ (\text{toCard } A \ r \ b) = b$
 $\langle \text{proof} \rangle$

lemma *Inl-Field-csum*: $a \in \text{Field } r \implies \text{Inl } a \in \text{Field } (r +_c s)$
 $\langle \text{proof} \rangle$

lemma *Inr-Field-csum*: $a \in \text{Field } s \implies \text{Inr } a \in \text{Field } (r +_c s)$
 $\langle \text{proof} \rangle$

lemma *rec-nat-0-imp*: $f = \text{rec-nat } f1 \ (\lambda n \text{ rec. } f2 \ n \ \text{rec}) \implies f \ 0 = f1$
 $\langle \text{proof} \rangle$

lemma *rec-nat-Suc-imp*: $f = \text{rec-nat } f1 \ (\lambda n \text{ rec. } f2 \ n \ \text{rec}) \implies f \ (\text{Suc } n) = f2 \ n \ (f \ n)$
 $\langle \text{proof} \rangle$

lemma *rec-list-Nil-imp*: $f = \text{rec-list } f1 \ (\lambda x \ xs \ \text{rec. } f2 \ x \ xs \ \text{rec}) \implies f \ [] = f1$
 $\langle \text{proof} \rangle$

lemma *rec-list-Cons-imp*: $f = \text{rec-list } f1 \ (\lambda x \ xs \ \text{rec. } f2 \ x \ xs \ \text{rec}) \implies f \ (x \# xs) = f2 \ x \ xs \ (f \ xs)$
 $\langle \text{proof} \rangle$

lemma *not-arg-cong-Inr*: $x \neq y \implies \text{Inr } x \neq \text{Inr } y$

$\langle \text{proof} \rangle$

definition *image2p* **where**

$$\text{image2p } f \ g \ R = (\lambda x \ y. \exists x' \ y'. R \ x' \ y' \wedge f \ x' = x \wedge g \ y' = y)$$

lemma *image2pI*: $R \ x \ y \implies \text{image2p } f \ g \ R \ (f \ x) \ (g \ y)$

$\langle \text{proof} \rangle$

lemma *image2pE*: $\llbracket \text{image2p } f \ g \ R \ f x \ g y; (\bigwedge x \ y. f x = f x \implies g y = g y \implies R \ x \ y \implies P) \rrbracket \implies P$

$\langle \text{proof} \rangle$

lemma *rel-fun-iff-geq-image2p*: $\text{rel-fun } R \ S \ f \ g = (\text{image2p } f \ g \ R \leq S)$

$\langle \text{proof} \rangle$

lemma *rel-fun-image2p*: $\text{rel-fun } R \ (\text{image2p } f \ g \ R) \ f \ g$

$\langle \text{proof} \rangle$

89.1 Equivalence relations, quotients, and Hilbert’s choice

lemma *equiv-Eps-in*:

$$\llbracket \text{equiv } A \ r; X \in A//r \rrbracket \implies \text{Eps } (\lambda x. x \in X) \in X$$

$\langle \text{proof} \rangle$

lemma *equiv-Eps-preserves*:

assumes *ECH*: $\text{equiv } A \ r$ **and** $X: X \in A//r$

shows $\text{Eps } (\lambda x. x \in X) \in A$

$\langle \text{proof} \rangle$

lemma *proj-Eps*:

assumes $\text{equiv } A \ r$ **and** $X \in A//r$

shows $\text{proj } r \ (\text{Eps } (\lambda x. x \in X)) = X$

$\langle \text{proof} \rangle$

definition *univ* **where** $\text{univ } f \ X == f \ (\text{Eps } (\lambda x. x \in X))$

lemma *univ-commute*:

assumes *ECH*: $\text{equiv } A \ r$ **and** *RES*: f respects r **and** $x: x \in A$

shows $(\text{univ } f) \ (\text{proj } r \ x) = f \ x$

$\langle \text{proof} \rangle$

lemma *univ-preserves*:

assumes *ECH*: $\text{equiv } A \ r$ **and** *RES*: f respects r **and** *PRES*: $\forall x \in A. f \ x \in B$

shows $\forall X \in A//r. \text{univ } f \ X \in B$

$\langle \text{proof} \rangle$

lemma *card-suc-ordLess-imp-ordLeq*:

assumes *ORD*: *Card-order* r *Card-order* r' *card-order* r'

and *LESS*: $r <_o \text{card-suc } r'$

shows $r \leq_o r'$
 $\langle proof \rangle$

lemma *natLeq-ordLess-cinfinite*: $\llbracket Cinfinite\ r; card-order\ r \rrbracket \implies natLeq <_o card-suc\ r$
 $\langle proof \rangle$

corollary *natLeq-ordLess-cinfinite'*: $\llbracket Cinfinite\ r'; card-order\ r'; r \equiv card-suc\ r' \rrbracket \implies natLeq <_o r$
 $\langle proof \rangle$

$\langle ML \rangle$

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 $F :: ('a \Rightarrow bool) \Rightarrow bool$
assumes *True*: $F\ (\lambda x. True)$
assumes *conj*: $F\ (\lambda x. P\ x) \implies F\ (\lambda x. Q\ x) \implies F\ (\lambda x. P\ x \wedge Q\ x)$
assumes *mono*: $\forall x. P\ x \longrightarrow Q\ x \implies F\ (\lambda x. P\ x) \implies F\ (\lambda x. Q\ x)$

typedef $'a\ filter = \{F :: ('a \Rightarrow bool) \Rightarrow bool. is-filter\ F\}$
 $\langle proof \rangle$

lemma *is-filter-Rep-filter*: $is-filter\ (Rep-filter\ F)$
 $\langle proof \rangle$

lemma *Abs-filter-inverse'*:
assumes *is-filter* F **shows** $Rep-filter\ (Abs-filter\ F) = F$
 $\langle proof \rangle$

90.1.1 Eventually

definition *eventually* :: $('a \Rightarrow bool) \Rightarrow 'a\ filter \Rightarrow bool$
where $eventually\ P\ F \longleftrightarrow Rep-filter\ F\ P$

syntax
 $-eventually :: pptrn \Rightarrow 'a\ filter \Rightarrow bool \Rightarrow bool\ (\langle \langle indent=3\ notation=binder\ \forall_F \rangle \rangle \forall_F - in -./ -) \rangle [0, 0, 10]\ 10)$

syntax-consts

-eventually == eventually

translations

$\forall_F x \text{ in } F. P == \text{CONST eventually } (\lambda x. P) F$

lemma eventually-Abs-filter:

assumes *is-filter F* **shows** *eventually P (Abs-filter F) = F P*

<proof>

lemma filter-eq-iff:

shows $F = F' \longleftrightarrow (\forall P. \text{eventually } P F = \text{eventually } P F')$

<proof>

lemma eventually-True [simp]: eventually (λx. True) F

<proof>

lemma always-eventually: $\forall x. P x \implies \text{eventually } P F$

<proof>

lemma eventuallyI: $(\bigwedge x. P x) \implies \text{eventually } P F$

<proof>

lemma filter-eqI: $(\bigwedge P. \text{eventually } P F \longleftrightarrow \text{eventually } P G) \implies F = G$

<proof>

lemma eventually-mono:

$\llbracket \text{eventually } P F; \bigwedge x. P x \implies Q x \rrbracket \implies \text{eventually } Q F$

<proof>

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*

<proof>

lemma eventually-mp:

assumes *eventually (λx. P x \longrightarrow Q x) F*

assumes *eventually (λx. P x) F*

shows *eventually (λx. Q x) F*

<proof>

lemma eventually-rev-mp:

assumes *eventually (λx. P x) F*

assumes *eventually (λx. P x \longrightarrow Q x) F*

shows *eventually (λx. Q x) F*

<proof>

lemma eventually-conj-iff:

eventually (λx. P x ∧ Q x) F \longleftrightarrow eventually P F ∧ eventually Q F

$\langle \text{proof} \rangle$

lemma *eventually-elim2*:

assumes *eventually* $(\lambda i. P\ i)\ F$
assumes *eventually* $(\lambda i. Q\ i)\ F$
assumes $\bigwedge i. P\ i \implies Q\ i \implies R\ i$
shows *eventually* $(\lambda i. R\ i)\ F$
 $\langle \text{proof} \rangle$

lemma *eventually-cong*:

assumes *eventually* $P\ F$ **and** $\bigwedge x. P\ x \implies Q\ x \longleftrightarrow R\ x$
shows *eventually* $Q\ F \longleftrightarrow \text{eventually } R\ F$
 $\langle \text{proof} \rangle$

lemma *eventually-ball-finite-distrib*:

finite $A \implies (\text{eventually } (\lambda x. \forall y \in A. P\ x\ y)\ \text{net}) \longleftrightarrow (\forall y \in A. \text{eventually } (\lambda x. P\ x\ y)\ \text{net})$
 $\langle \text{proof} \rangle$

lemma *eventually-ball-finite*:

finite $A \implies \forall y \in A. \text{eventually } (\lambda x. P\ x\ y)\ \text{net} \implies \text{eventually } (\lambda x. \forall y \in A. P\ x\ y)\ \text{net}$
 $\langle \text{proof} \rangle$

lemma *eventually-all-finite*:

fixes $P :: 'a \Rightarrow 'b::\text{finite} \Rightarrow \text{bool}$
assumes $\bigwedge y. \text{eventually } (\lambda x. P\ x\ y)\ \text{net}$
shows *eventually* $(\lambda x. \forall y. P\ x\ y)\ \text{net}$
 $\langle \text{proof} \rangle$

lemma *eventually-ex*: $(\forall_{F x \text{ in } F}. \exists y. P\ x\ y) \longleftrightarrow (\exists Y. \forall_{F x \text{ in } F}. P\ x\ (Y\ x))$
 $\langle \text{proof} \rangle$

lemma *not-eventually-impI*: *eventually* $P\ F \implies \neg \text{eventually } Q\ F \implies \neg \text{eventually } (\lambda x. P\ x \longrightarrow Q\ x)\ F$
 $\langle \text{proof} \rangle$

lemma *not-eventuallyD*: $\neg \text{eventually } P\ F \implies \exists x. \neg P\ x$
 $\langle \text{proof} \rangle$

lemma *eventually-subst*:

assumes *eventually* $(\lambda n. P\ n = Q\ n)\ F$
shows *eventually* $P\ F = \text{eventually } Q\ F$ (**is** $?L = ?R$)
 $\langle \text{proof} \rangle$

90.2 Frequently as dual to eventually

definition *frequently* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a\ \text{filter} \Rightarrow \text{bool}$
where *frequently* $P\ F \longleftrightarrow \neg \text{eventually } (\lambda x. \neg P\ x)\ F$

syntax

-frequently :: *pttrn* \Rightarrow 'a *filter* \Rightarrow bool \Rightarrow bool ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder} \exists_F \rangle \rangle \exists_F - \text{in } -./ - \rangle [0, 0, 10] 10 \rangle$)

syntax-consts

-frequently == *frequently*

translations

$\exists_F x \text{ in } F. P == \text{CONST frequently } (\lambda x. P) F$

lemma *not-frequently-False* [simp]: $\neg (\exists_F x \text{ in } F. \text{False})$
 $\langle \text{proof} \rangle$

lemma *frequently-ex*: $\exists_F x \text{ in } F. P x \Longrightarrow \exists x. P x$
 $\langle \text{proof} \rangle$

lemma *frequentlyE*: **assumes** *frequently* *P F* **obtains** *x* **where** *P x*
 $\langle \text{proof} \rangle$

lemma *frequently-mp*:

assumes *ev*: $\forall_F x \text{ in } F. P x \longrightarrow Q x$ **and** *P*: $\exists_F x \text{ in } F. P x$ **shows** $\exists_F x \text{ in } F. Q x$
 $\langle \text{proof} \rangle$

lemma *frequently-rev-mp*:

assumes $\exists_F x \text{ in } F. P x$
assumes $\forall_F x \text{ in } F. P x \longrightarrow Q x$
shows $\exists_F x \text{ in } F. Q x$
 $\langle \text{proof} \rangle$

lemma *frequently-mono*: $(\forall x. P x \longrightarrow Q x) \Longrightarrow \text{frequently } P F \Longrightarrow \text{frequently } Q F$
 $\langle \text{proof} \rangle$

lemma *frequently-elim1*: $\exists_F x \text{ in } F. P x \Longrightarrow (\bigwedge i. P i \Longrightarrow Q i) \Longrightarrow \exists_F x \text{ in } F. Q x$
 $\langle \text{proof} \rangle$

lemma *frequently-disj-iff*: $(\exists_F x \text{ in } F. P x \vee Q x) \longleftrightarrow (\exists_F x \text{ in } F. P x) \vee (\exists_F x \text{ in } F. Q x)$
 $\langle \text{proof} \rangle$

lemma *frequently-disj*: $\exists_F x \text{ in } F. P x \Longrightarrow \exists_F x \text{ in } F. Q x \Longrightarrow \exists_F x \text{ in } F. P x \vee Q x$
 $\langle \text{proof} \rangle$

lemma *frequently-bex-finite-distrib*:

assumes *finite* *A* **shows** $(\exists_F x \text{ in } F. \exists y \in A. P x y) \longleftrightarrow (\exists y \in A. \exists_F x \text{ in } F. P x y)$
 $\langle \text{proof} \rangle$

lemma *frequently-bex-finite*: $\text{finite } A \implies \exists_{Fx \text{ in } F}. \exists y \in A. P\ x\ y \implies \exists y \in A. \exists_{Fx \text{ in } F}. P\ x\ y$
 (proof)

lemma *frequently-all*: $(\exists_{Fx \text{ in } F}. \forall y. P\ x\ y) \longleftrightarrow (\forall Y. \exists_{Fx \text{ in } F}. P\ x\ (Y\ x))$
 (proof)

lemma
shows *not-eventually*: $\neg \text{eventually } P\ F \longleftrightarrow (\exists_{Fx \text{ in } F}. \neg P\ x)$
and *not-frequently*: $\neg \text{frequently } P\ F \longleftrightarrow (\forall_{Fx \text{ in } F}. \neg P\ x)$
 (proof)

lemma *frequently-imp-iff*:
 $(\exists_{Fx \text{ in } F}. P\ x \longrightarrow Q\ x) \longleftrightarrow (\text{eventually } P\ F \longrightarrow \text{frequently } Q\ F)$
 (proof)

lemma *frequently-eventually-conj*:
assumes $\exists_{Fx \text{ in } F}. P\ x$
assumes $\forall_{Fx \text{ in } F}. Q\ x$
shows $\exists_{Fx \text{ in } F}. Q\ x \wedge P\ x$
 (proof)

lemma *frequently-cong*:
assumes *ev*: $\text{eventually } P\ F$ **and** *QR*: $\bigwedge x. P\ x \implies Q\ x \longleftrightarrow R\ x$
shows $\text{frequently } Q\ F \longleftrightarrow \text{frequently } R\ F$
 (proof)

lemma *frequently-eventually-frequently*:
 $\text{frequently } P\ F \implies \text{eventually } Q\ F \implies \text{frequently } (\lambda x. P\ x \wedge Q\ x)\ F$
 (proof)

lemma *eventually-frequently-const-simps* [simp]:
 $(\exists_{Fx \text{ in } F}. P\ x \wedge C) \longleftrightarrow (\exists_{Fx \text{ in } F}. P\ x) \wedge C$
 $(\exists_{Fx \text{ in } F}. C \wedge P\ x) \longleftrightarrow C \wedge (\exists_{Fx \text{ in } F}. P\ x)$
 $(\forall_{Fx \text{ in } F}. P\ x \vee C) \longleftrightarrow (\forall_{Fx \text{ in } F}. P\ x) \vee C$
 $(\forall_{Fx \text{ in } F}. C \vee P\ x) \longleftrightarrow C \vee (\forall_{Fx \text{ in } F}. P\ x)$
 $(\forall_{Fx \text{ in } F}. P\ x \longrightarrow C) \longleftrightarrow ((\exists_{Fx \text{ in } F}. P\ x) \longrightarrow C)$
 $(\forall_{Fx \text{ in } F}. C \longrightarrow P\ x) \longleftrightarrow (C \longrightarrow (\forall_{Fx \text{ in } F}. P\ x))$
 (proof)

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

$\langle ML \rangle$

90.2.1 Finer-than relation

$F \leq F'$ means that filter F is finer than filter F' .

instantiation *filter* :: (type) complete-lattice
begin

definition *le-filter-def*:

$$F \leq F' \longleftrightarrow (\forall P. \text{eventually } P \ F' \longrightarrow \text{eventually } P \ F)$$

definition

$$(F :: 'a \text{ filter}) < F' \longleftrightarrow F \leq F' \wedge \neg F' \leq F$$

definition

$$\text{top} = \text{Abs-filter } (\lambda P. \forall x. P \ x)$$

definition

$$\text{bot} = \text{Abs-filter } (\lambda P. \text{True})$$

definition

$$\text{sup } F \ F' = \text{Abs-filter } (\lambda P. \text{eventually } P \ F \wedge \text{eventually } P \ F')$$

definition

$$\begin{aligned} \text{inf } F \ F' &= \text{Abs-filter} \\ &(\lambda P. \exists Q \ R. \text{eventually } Q \ F \wedge \text{eventually } R \ F' \wedge (\forall x. Q \ x \wedge R \ x \longrightarrow P \ x)) \end{aligned}$$

definition

$$\text{Sup } S = \text{Abs-filter } (\lambda P. \forall F \in S. \text{eventually } P \ F)$$

definition

$$\text{Inf } S = \text{Sup } \{F :: 'a \text{ filter}. \forall F' \in S. F \leq F'\}$$

lemma *eventually-top* [simp]: $\text{eventually } P \ \text{top} \longleftrightarrow (\forall x. P \ x)$
 $\langle \text{proof} \rangle$

lemma *eventually-bot* [simp]: $\text{eventually } P \ \text{bot}$
 $\langle \text{proof} \rangle$

lemma *eventually-sup*:

$$\text{eventually } P \ (\text{sup } F \ F') \longleftrightarrow \text{eventually } P \ F \wedge \text{eventually } P \ F'$$

$\langle \text{proof} \rangle$

lemma *eventually-inf*:

$$\text{eventually } P \ (\text{inf } F \ F') \longleftrightarrow$$

$(\exists Q R. \text{eventually } Q F \wedge \text{eventually } R F' \wedge (\forall x. Q x \wedge R x \longrightarrow P x))$
 $\langle \text{proof} \rangle$

lemma *eventually-Sup*:
 $\text{eventually } P (\text{Sup } S) \longleftrightarrow (\forall F \in S. \text{eventually } P F)$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instance *filter* :: (type) distrib-lattice
 $\langle \text{proof} \rangle$

lemma *filter-leD*:
 $F \leq F' \implies \text{eventually } P F' \implies \text{eventually } P F$
 $\langle \text{proof} \rangle$

lemma *filter-leI*:
 $(\bigwedge P. \text{eventually } P F' \implies \text{eventually } P F) \implies F \leq F'$
 $\langle \text{proof} \rangle$

lemma *eventually-False*:
 $\text{eventually } (\lambda x. \text{False}) F \longleftrightarrow F = \text{bot}$
 $\langle \text{proof} \rangle$

lemma *eventually-frequently*: $F \neq \text{bot} \implies \text{eventually } P F \implies \text{frequently } P F$
 $\langle \text{proof} \rangle$

lemma *eventually-frequentlyE*:
assumes *eventually* $P F$
assumes *eventually* $(\lambda x. \neg P x \vee Q x) F$ $F \neq \text{bot}$
shows *frequently* $Q F$
 $\langle \text{proof} \rangle$

lemma *eventually-const-iff*: $\text{eventually } (\lambda x. P) F \longleftrightarrow P \vee F = \text{bot}$
 $\langle \text{proof} \rangle$

lemma *eventually-const[simp]*: $F \neq \text{bot} \implies \text{eventually } (\lambda x. P) F \longleftrightarrow P$
 $\langle \text{proof} \rangle$

lemma *frequently-const-iff*: $\text{frequently } (\lambda x. P) F \longleftrightarrow P \wedge F \neq \text{bot}$
 $\langle \text{proof} \rangle$

lemma *frequently-const[simp]*: $F \neq \text{bot} \implies \text{frequently } (\lambda x. P) F \longleftrightarrow P$
 $\langle \text{proof} \rangle$

lemma *eventually-happens*: $\text{eventually } P \text{ net} \implies \text{net} = \text{bot} \vee (\exists x. P x)$

$\langle proof \rangle$

lemma *eventually-happens'*:

assumes $F \neq bot$ *eventually* $P\ F$

shows $\exists x. P\ x$

$\langle proof \rangle$

abbreviation (*input*) *trivial-limit* :: 'a filter \Rightarrow bool

where *trivial-limit* $F \equiv F = bot$

lemma *trivial-limit-def*: *trivial-limit* $F \longleftrightarrow$ *eventually* $(\lambda x. False)\ F$

$\langle proof \rangle$

lemma *False-imp-not-eventually*: $(\forall x. \neg P\ x) \Longrightarrow \neg$ *trivial-limit net* $\Longrightarrow \neg$ *eventually* $(\lambda x. P\ x)$ *net*

$\langle proof \rangle$

lemma *trivial-limit-eventually*: *trivial-limit net* \Longrightarrow *eventually* P *net*

$\langle proof \rangle$

lemma *trivial-limit-eq*: *trivial-limit net* $\longleftrightarrow (\forall P. \text{eventually } P\ \text{net})$

$\langle proof \rangle$

lemma *eventually-Inf*: *eventually* P $(\text{Inf } B) \longleftrightarrow (\exists X \subseteq B. \text{finite } X \wedge \text{eventually } P\ (\text{Inf } X))$

$\langle proof \rangle$

lemma *eventually-INF*: *eventually* P $(\bigcap b \in B. F\ b) \longleftrightarrow (\exists X \subseteq B. \text{finite } X \wedge \text{eventually } P\ (\bigcap b \in X. F\ b))$

$\langle proof \rangle$

lemma *Inf-filter-not-bot*:

fixes $B :: 'a$ *filter set*

shows $(\bigwedge X. X \subseteq B \Longrightarrow \text{finite } X \Longrightarrow \text{Inf } X \neq bot) \Longrightarrow \text{Inf } B \neq bot$

$\langle proof \rangle$

lemma *INF-filter-not-bot*:

fixes $F :: 'i \Rightarrow 'a$ *filter*

shows $(\bigwedge X. X \subseteq B \Longrightarrow \text{finite } X \Longrightarrow (\bigcap b \in X. F\ b) \neq bot) \Longrightarrow (\bigcap b \in B. F\ b) \neq bot$

$\langle proof \rangle$

lemma *eventually-Inf-base*:

assumes $B \neq \{\}$ **and** *base*: $\bigwedge F\ G. F \in B \Longrightarrow G \in B \Longrightarrow \exists x \in B. x \leq \text{inf } F\ G$

shows *eventually* P $(\text{Inf } B) \longleftrightarrow (\exists b \in B. \text{eventually } P\ b)$

$\langle proof \rangle$

lemma *eventually-INF-base*:

$B \neq \{\} \Longrightarrow (\bigwedge a\ b. a \in B \Longrightarrow b \in B \Longrightarrow \exists x \in B. F\ x \leq \text{inf } (F\ a)\ (F\ b)) \Longrightarrow$

eventually $P (\bigcap b \in B. F b) \longleftrightarrow (\exists b \in B. \text{eventually } P (F b))$
 ⟨proof⟩

lemma *eventually-INF1*: $i \in I \implies \text{eventually } P (F i) \implies \text{eventually } P (\bigcap i \in I. F i)$
 ⟨proof⟩

lemma *eventually-INF-finite*:

assumes *finite* A

shows $\text{eventually } P (\bigcap x \in A. F x) \longleftrightarrow$
 $(\exists Q. (\forall x \in A. \text{eventually } (Q x) (F x)) \wedge (\forall y. (\forall x \in A. Q x y) \longrightarrow P y))$
 ⟨proof⟩

lemma *eventually-le-le*:

fixes $P :: 'a \Rightarrow ('b :: \text{preorder})$

assumes *eventually* $(\lambda x. P x \leq Q x) F$

assumes *eventually* $(\lambda x. Q x \leq R x) F$

shows *eventually* $(\lambda x. P x \leq R x) F$

⟨proof⟩

90.2.2 Map function for filters

definition *filtermap* :: $('a \Rightarrow 'b) \Rightarrow 'a \text{ filter} \Rightarrow 'b \text{ filter}$

where *filtermap* $f F = \text{Abs-filter } (\lambda P. \text{eventually } (\lambda x. P (f x)) F)$

lemma *eventually-filtermap*:

eventually $P (\text{filtermap } f F) = \text{eventually } (\lambda x. P (f x)) F$
 ⟨proof⟩

lemma *eventually-comp-filtermap*:

eventually $(P \circ f) F \longleftrightarrow \text{eventually } P (\text{filtermap } f F)$
 ⟨proof⟩

lemma *filtermap-compose*: *filtermap* $(f \circ g) F = \text{filtermap } f (\text{filtermap } g F)$

⟨proof⟩

lemma *filtermap-ident*: *filtermap* $(\lambda x. x) F = F$

⟨proof⟩

lemma *filtermap-filtermap*:

filtermap $f (\text{filtermap } g F) = \text{filtermap } (\lambda x. f (g x)) F$
 ⟨proof⟩

lemma *filtermap-mono*: $F \leq F' \implies \text{filtermap } f F \leq \text{filtermap } f F'$

⟨proof⟩

lemma *filtermap-bot [simp]*: *filtermap* $f \text{ bot} = \text{bot}$

⟨proof⟩

lemma *filtermap-bot-iff*: $\text{filtermap } f \ F = \text{bot} \longleftrightarrow F = \text{bot}$
 ⟨proof⟩

lemma *filtermap-sup*: $\text{filtermap } f \ (\sup F1 \ F2) = \sup (\text{filtermap } f \ F1) \ (\text{filtermap } f \ F2)$
 ⟨proof⟩

lemma *filtermap-SUP*: $\text{filtermap } f \ (\bigsqcup b \in B. \ F \ b) = (\bigsqcup b \in B. \ \text{filtermap } f \ (F \ b))$
 ⟨proof⟩

lemma *filtermap-inf*: $\text{filtermap } f \ (\inf F1 \ F2) \leq \inf (\text{filtermap } f \ F1) \ (\text{filtermap } f \ F2)$
 ⟨proof⟩

lemma *filtermap-INF*: $\text{filtermap } f \ (\prod b \in B. \ F \ b) \leq (\prod b \in B. \ \text{filtermap } f \ (F \ b))$
 ⟨proof⟩

lemma *frequently-filtermap*:
 $\text{frequently } P \ (\text{filtermap } f \ F) = \text{frequently } (\lambda x. \ P \ (f \ x)) \ F$
 ⟨proof⟩

90.2.3 Contravariant map function for filters

definition *filtercomap* :: $('a \Rightarrow 'b) \Rightarrow 'b \ \text{filter} \Rightarrow 'a \ \text{filter}$ **where**
 $\text{filtercomap } f \ F = \text{Abs-filter } (\lambda P. \ \exists Q. \ \text{eventually } Q \ F \wedge (\forall x. \ Q \ (f \ x) \longrightarrow P \ x))$

lemma *eventually-filtercomap*:
 $\text{eventually } P \ (\text{filtercomap } f \ F) \longleftrightarrow (\exists Q. \ \text{eventually } Q \ F \wedge (\forall x. \ Q \ (f \ x) \longrightarrow P \ x))$
 ⟨proof⟩

lemma *filtercomap-ident*: $\text{filtercomap } (\lambda x. \ x) \ F = F$
 ⟨proof⟩

lemma *filtercomap-filtercomap*: $\text{filtercomap } f \ (\text{filtercomap } g \ F) = \text{filtercomap } (\lambda x. \ g \ (f \ x)) \ F$
 ⟨proof⟩

lemma *filtercomap-mono*: $F \leq F' \Longrightarrow \text{filtercomap } f \ F \leq \text{filtercomap } f \ F'$
 ⟨proof⟩

lemma *filtercomap-bot* [simp]: $\text{filtercomap } f \ \text{bot} = \text{bot}$
 ⟨proof⟩

lemma *filtercomap-top* [simp]: $\text{filtercomap } f \ \text{top} = \text{top}$
 ⟨proof⟩

lemma *filtercomap-inf*: $\text{filtercomap } f \ (\inf F1 \ F2) = \inf (\text{filtercomap } f \ F1) \ (\text{filtercomap } f \ F2)$

<proof>

lemma *filtercomap-sup*: $\text{filtercomap } f (\sup F1 \ F2) \geq \sup (\text{filtercomap } f \ F1) (\text{filtercomap } f \ F2)$
<proof>

lemma *filtercomap-INF*: $\text{filtercomap } f (\prod b \in B. \ F \ b) = (\prod b \in B. \ \text{filtercomap } f \ (F \ b))$
<proof>

lemma *filtercomap-SUP*:
 $\text{filtercomap } f (\bigsqcup b \in B. \ F \ b) \geq (\bigsqcup b \in B. \ \text{filtercomap } f \ (F \ b))$
<proof>

lemma *filtermap-le-iff-le-filtercomap*: $\text{filtermap } f \ F \leq G \longleftrightarrow F \leq \text{filtercomap } f \ G$
<proof>

lemma *filtercomap-neg-bot*:
assumes $\bigwedge P. \text{eventually } P \ F \implies \exists x. \ P \ (f \ x)$
shows $\text{filtercomap } f \ F \neq \text{bot}$
<proof>

lemma *filtercomap-neg-bot-surj*:
assumes $F \neq \text{bot}$ **and** $\text{surj } f$
shows $\text{filtercomap } f \ F \neq \text{bot}$
<proof>

lemma *eventually-filtercomapI* [intro]:
assumes $\text{eventually } P \ F$
shows $\text{eventually } (\lambda x. \ P \ (f \ x)) (\text{filtercomap } f \ F)$
<proof>

lemma *filtermap-filtercomap*: $\text{filtermap } f (\text{filtercomap } f \ F) \leq F$
<proof>

lemma *filtercomap-filtermap*: $\text{filtercomap } f (\text{filtermap } f \ F) \geq F$
<proof>

90.2.4 Standard filters

definition *principal* :: 'a set \Rightarrow 'a filter **where**
 $\text{principal } S = \text{Abs-filter } (\lambda P. \ \forall x \in S. \ P \ x)$

lemma *eventually-principal*: $\text{eventually } P \ (\text{principal } S) \longleftrightarrow (\forall x \in S. \ P \ x)$
<proof>

lemma *eventually-inf-principal*: $\text{eventually } P \ (\inf F \ (\text{principal } s)) \longleftrightarrow \text{eventually } (\lambda x. \ x \in s \longrightarrow P \ x) \ F$
<proof>

lemma *principal-UNIV[simp]*: *principal UNIV = top*
 ⟨*proof*⟩

lemma *principal-empty[simp]*: *principal {} = bot*
 ⟨*proof*⟩

lemma *principal-eq-bot-iff*: *principal X = bot \longleftrightarrow X = {}*
 ⟨*proof*⟩

lemma *principal-le-iff[iff]*: *principal A \leq principal B \longleftrightarrow A \subseteq B*
 ⟨*proof*⟩

lemma *le-principal*: *F \leq principal A \longleftrightarrow eventually ($\lambda x. x \in A$) F*
 ⟨*proof*⟩

lemma *principal-inject[iff]*: *principal A = principal B \longleftrightarrow A = B*
 ⟨*proof*⟩

lemma *sup-principal[simp]*: *sup (principal A) (principal B) = principal (A \cup B)*
 ⟨*proof*⟩

lemma *inf-principal[simp]*: *inf (principal A) (principal B) = principal (A \cap B)*
 ⟨*proof*⟩

lemma *SUP-principal[simp]*: *($\bigsqcup i \in I. \text{principal } (A \ i)$) = principal ($\bigcup i \in I. A \ i$)*
 ⟨*proof*⟩

lemma *INF-principal-finite*: *finite X \implies ($\bigcap x \in X. \text{principal } (f \ x)$) = principal ($\bigcap x \in X. f \ x$)*
 ⟨*proof*⟩

lemma *filtermap-principal[simp]*: *filtermap f (principal A) = principal (f ‘ A)*
 ⟨*proof*⟩

lemma *filtercomap-principal[simp]*: *filtercomap f (principal A) = principal (f -‘ A)*
 ⟨*proof*⟩

90.2.5 Order filters

definition *at-top* :: (*'a::order*) *filter*
 where *at-top* = ($\bigcap k. \text{principal } \{k \ ..\}$)

lemma *at-top-sub*: *at-top = ($\bigcap k \in \{c :: 'a :: \text{linorder}..\}. \text{principal } \{k \ ..\}$)*
 ⟨*proof*⟩

lemma *eventually-at-top-linorder*: *eventually P at-top \longleftrightarrow ($\exists N :: 'a :: \text{linorder}. \forall n \geq N. P \ n$)*

<proof>

lemma *eventually-filtercomap-at-top-linorder*:

eventually P (*filtercomap* f *at-top*) $\longleftrightarrow (\exists N :: 'a :: \text{linorder}. \forall x. f\ x \geq N \longrightarrow P\ x)$

<proof>

lemma *eventually-at-top-linorderI*:

fixes $c :: 'a :: \text{linorder}$

assumes $\bigwedge x. c \leq x \implies P\ x$

shows *eventually* P *at-top*

<proof>

lemma *eventually-ge-at-top [simp]*:

eventually $(\lambda x. (c :: \text{linorder}) \leq x)$ *at-top*

<proof>

lemma *eventually-at-top-dense*: *eventually* P *at-top* $\longleftrightarrow (\exists N :: 'a :: \{\text{no-top}, \text{linorder}\}. \forall n > N. P\ n)$

<proof>

lemma *eventually-filtercomap-at-top-dense*:

eventually P (*filtercomap* f *at-top*) $\longleftrightarrow (\exists N :: 'a :: \{\text{no-top}, \text{linorder}\}. \forall x. f\ x > N \longrightarrow P\ x)$

<proof>

lemma *eventually-at-top-not-equal [simp]*: *eventually* $(\lambda x :: 'a :: \{\text{no-top}, \text{linorder}\}. x \neq c)$ *at-top*

<proof>

lemma *eventually-gt-at-top [simp]*: *eventually* $(\lambda x. (c :: \{\text{no-top}, \text{linorder}\}) < x)$ *at-top*

<proof>

lemma *eventually-all-ge-at-top*:

assumes *eventually* P (*at-top* :: $('a :: \text{linorder})$ *filter*)

shows *eventually* $(\lambda x. \forall y \geq x. P\ y)$ *at-top*

<proof>

definition *at-bot* :: $('a :: \text{order})$ *filter*

where *at-bot* = $(\bigcap k. \text{principal } \{.. k\})$

lemma *at-bot-sub*: *at-bot* = $(\bigcap k \in \{.. c :: 'a :: \text{linorder}\}. \text{principal } \{.. k\})$

<proof>

lemma *eventually-at-bot-linorder*:

fixes $P :: 'a :: \text{linorder} \Rightarrow \text{bool}$ **shows** *eventually* P *at-bot* $\longleftrightarrow (\exists N. \forall n \leq N. P\ n)$

<proof>

lemma *eventually-at-bot-linorderI*:

fixes $c :: 'a :: \text{linorder}$
assumes $\bigwedge x. x \leq c \implies P\ x$
shows *eventually* P *at-bot*
 $\langle \text{proof} \rangle$

lemma *eventually-filtercomap-at-bot-linorder*:
 $\text{eventually } P \text{ (filtercomap } f \text{ at-bot)} \longleftrightarrow (\exists N :: 'a :: \text{linorder}. \forall x. f\ x \leq N \longrightarrow P\ x)$
 $\langle \text{proof} \rangle$

lemma *eventually-le-at-bot [simp]*:
 $\text{eventually } (\lambda x. x \leq (c :: \text{linorder})) \text{ at-bot}$
 $\langle \text{proof} \rangle$

lemma *eventually-at-bot-dense*: $\text{eventually } P \text{ at-bot} \longleftrightarrow (\exists N :: 'a :: \{\text{no-bot}, \text{linorder}\}. \forall n < N. P\ n)$
 $\langle \text{proof} \rangle$

lemma *eventually-filtercomap-at-bot-dense*:
 $\text{eventually } P \text{ (filtercomap } f \text{ at-bot)} \longleftrightarrow (\exists N :: 'a :: \{\text{no-bot}, \text{linorder}\}. \forall x. f\ x < N \longrightarrow P\ x)$
 $\langle \text{proof} \rangle$

lemma *eventually-at-bot-not-equal [simp]*: $\text{eventually } (\lambda x :: 'a :: \{\text{no-bot}, \text{linorder}\}. x \neq c) \text{ at-bot}$
 $\langle \text{proof} \rangle$

lemma *eventually-gt-at-bot [simp]*:
 $\text{eventually } (\lambda x. x < (c :: \text{unbounded-dense-linorder})) \text{ at-bot}$
 $\langle \text{proof} \rangle$

lemma *trivial-limit-at-bot-linorder [simp]*: $\neg \text{trivial-limit } (\text{at-bot} :: ('a :: \text{linorder}) \text{ filter})$
 $\langle \text{proof} \rangle$

lemma *trivial-limit-at-top-linorder [simp]*: $\neg \text{trivial-limit } (\text{at-top} :: ('a :: \text{linorder}) \text{ filter})$
 $\langle \text{proof} \rangle$

90.3 Sequentially

abbreviation *sequentially* :: *nat filter*
where *sequentially* \equiv *at-top*

lemma *eventually-sequentially*:
 $\text{eventually } P \text{ sequentially} \longleftrightarrow (\exists N. \forall n \geq N. P\ n)$
 $\langle \text{proof} \rangle$

lemma *frequently-sequentially*:
 $\text{frequently } P \text{ sequentially} \longleftrightarrow (\forall N. \exists n \geq N. P\ n)$

$\langle \text{proof} \rangle$

lemma *sequentially-bot* [*simp*, *intro*]: *sequentially* \neq *bot*
 $\langle \text{proof} \rangle$

lemmas *trivial-limit-sequentially* = *sequentially-bot*

lemma *eventually-False-sequentially* [*simp*]:
 $\neg \text{eventually } (\lambda n. \text{False}) \text{ sequentially}$
 $\langle \text{proof} \rangle$

lemma *le-sequentially*:
 $F \leq \text{sequentially} \iff (\forall N. \text{eventually } (\lambda n. N \leq n) F)$
 $\langle \text{proof} \rangle$

lemma *eventually-sequentiallyI* [*intro?*]:
assumes $\bigwedge x. c \leq x \implies P x$
shows *eventually* P *sequentially*
 $\langle \text{proof} \rangle$

lemma *eventually-sequentially-Suc* [*simp*]: *eventually* $(\lambda i. P (\text{Suc } i))$ *sequentially*
 $\iff \text{eventually } P \text{ sequentially}$
 $\langle \text{proof} \rangle$

lemma *eventually-sequentially-seg* [*simp*]: *eventually* $(\lambda n. P (n + k))$ *sequentially*
 $\iff \text{eventually } P \text{ sequentially}$
 $\langle \text{proof} \rangle$

lemma *filtermap-sequentially-ne-bot*: *filtermap* f *sequentially* \neq *bot*
 $\langle \text{proof} \rangle$

90.4 Increasing finite subsets

definition *finite-subsets-at-top* **where**
finite-subsets-at-top $A = (\bigcap X \in \{X. \text{finite } X \wedge X \subseteq A\}. \text{principal } \{Y. \text{finite } Y \wedge X \subseteq Y \wedge Y \subseteq A\})$

abbreviation *finite-sets-at-top* \equiv *finite-subsets-at-top* *UNIV*

lemma *eventually-finite-subsets-at-top*:
 $\text{eventually } P (\text{finite-subsets-at-top } A) \iff$
 $(\exists X. \text{finite } X \wedge X \subseteq A \wedge (\forall Y. \text{finite } Y \wedge X \subseteq Y \wedge Y \subseteq A \longrightarrow P Y))$
 $\langle \text{proof} \rangle$

lemma *eventually-finite-subsets-at-top-weakI* [*intro*]:
assumes $\bigwedge X. \text{finite } X \implies X \subseteq A \implies P X$
shows *eventually* $P (\text{finite-subsets-at-top } A)$
 $\langle \text{proof} \rangle$

lemma *finite-subsets-at-top-neq-bot* [simp]: *finite-subsets-at-top* $A \neq \text{bot}$
 ⟨proof⟩

lemma *filtermap-image-finite-subsets-at-top*:
 assumes *inj-on* f A
 shows *filtermap* $((\cdot) f)$ (*finite-subsets-at-top* A) = *finite-subsets-at-top* $(f \cdot A)$
 ⟨proof⟩

lemma *eventually-finite-subsets-at-top-finite*:
 assumes *finite* A
 shows *eventually* P (*finite-subsets-at-top* A) $\longleftrightarrow P$ A
 ⟨proof⟩

lemma *finite-subsets-at-top-finite*: *finite* $A \implies \text{finite-subsets-at-top } A = \text{principal } \{A\}$
 ⟨proof⟩

90.5 The cofinite filter

definition *cofinite* = *Abs-filter* $(\lambda P. \text{finite } \{x. \neg P x\})$

abbreviation *Inf-many* :: $('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ (**binder** $\langle \exists_\infty \rangle$ 10)
 where *Inf-many* $P \equiv \text{frequently } P$ *cofinite*

abbreviation *Alm-all* :: $('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ (**binder** $\langle \forall_\infty \rangle$ 10)
 where *Alm-all* $P \equiv \text{eventually } P$ *cofinite*

notation (*ASCII*)
Inf-many (**binder** $\langle \text{INFM} \rangle$ 10) and
Alm-all (**binder** $\langle \text{MOST} \rangle$ 10)

lemma *eventually-cofinite*: *eventually* P *cofinite* $\longleftrightarrow \text{finite } \{x. \neg P x\}$
 ⟨proof⟩

lemma *frequently-cofinite*: *frequently* P *cofinite* $\longleftrightarrow \neg \text{finite } \{x. P x\}$
 ⟨proof⟩

lemma *cofinite-bot*[simp]: *cofinite* = (*bot*:: $'a$ filter) $\longleftrightarrow \text{finite } (\text{UNIV} :: 'a \text{ set})$
 ⟨proof⟩

lemma *cofinite-eq-sequentially*: *cofinite* = *sequentially*
 ⟨proof⟩

90.5.1 Product of filters

definition *prod-filter* :: $'a \text{ filter} \Rightarrow 'b \text{ filter} \Rightarrow ('a \times 'b) \text{ filter}$ (**infixr** $\langle \times_F \rangle$ 80)
 where

prod-filter F G =
 $(\bigcap (P, Q) \in \{ (P, Q). \text{eventually } P F \wedge \text{eventually } Q G \}. \text{principal } \{(x, y). P x \wedge Q y\})$

lemma *eventually-prod-filter*: $\text{eventually } P \ (F \times_F G) \longleftrightarrow$
 $(\exists Pf \ Pg. \text{eventually } Pf \ F \wedge \text{eventually } Pg \ G \wedge (\forall x \ y. Pf \ x \longrightarrow Pg \ y \longrightarrow P \ (x, y)))$
 $\langle \text{proof} \rangle$

lemma *eventually-prod1*:
assumes $B \neq \text{bot}$
shows $(\forall_F \ (x, y) \text{ in } A \times_F B. P \ x) \longleftrightarrow (\forall_F \ x \text{ in } A. P \ x)$
 $\langle \text{proof} \rangle$

lemma *eventually-prod2*:
assumes $A \neq \text{bot}$
shows $(\forall_F \ (x, y) \text{ in } A \times_F B. P \ y) \longleftrightarrow (\forall_F \ y \text{ in } B. P \ y)$
 $\langle \text{proof} \rangle$

lemma *eventually-eventually-prod-filter1*:
assumes $\text{eventually } P \ (F \times_F G)$
shows $\text{eventually } (\lambda x. \text{eventually } (\lambda y. P \ (x, y)) \ G) \ F$
 $\langle \text{proof} \rangle$

lemma *eventually-eventually-prod-filter2*:
assumes $\text{eventually } P \ (F \times_F G)$
shows $\text{eventually } (\lambda y. \text{eventually } (\lambda x. P \ (x, y)) \ F) \ G$
 $\langle \text{proof} \rangle$

lemma *INF-filter-bot-base*:
fixes $F :: 'a \Rightarrow 'b \text{ filter}$
assumes $*$: $\bigwedge i \ j. i \in I \implies j \in I \implies \exists k \in I. F \ k \leq F \ i \sqcap F \ j$
shows $(\bigcap_{i \in I} F \ i) = \text{bot} \longleftrightarrow (\exists i \in I. F \ i = \text{bot})$
 $\langle \text{proof} \rangle$

lemma *Collect-empty-eq-bot*: $\text{Collect } P = \{\} \longleftrightarrow P = \perp$
 $\langle \text{proof} \rangle$

lemma *prod-filter-eq-bot*: $A \times_F B = \text{bot} \longleftrightarrow A = \text{bot} \vee B = \text{bot}$
 $\langle \text{proof} \rangle$

lemma *prod-filter-mono*: $F \leq F' \implies G \leq G' \implies F \times_F G \leq F' \times_F G'$
 $\langle \text{proof} \rangle$

lemma *prod-filter-mono-iff*:
assumes nAB : $A \neq \text{bot} \ B \neq \text{bot}$
shows $A \times_F B \leq C \times_F D \longleftrightarrow A \leq C \wedge B \leq D$
 $\langle \text{proof} \rangle$

lemma *eventually-prod-same*: $\text{eventually } P \ (F \times_F F) \longleftrightarrow$
 $(\exists Q. \text{eventually } Q \ F \wedge (\forall x \ y. Q \ x \longrightarrow Q \ y \longrightarrow P \ (x, y)))$
 $\langle \text{proof} \rangle$

lemma *eventually-prod-sequentially*:

eventually P (*sequentially* \times_F *sequentially*) $\longleftrightarrow (\exists N. \forall m \geq N. \forall n \geq N. P (n, m))$
 ⟨proof⟩

lemma *principal-prod-principal*: *principal* $A \times_F$ *principal* $B =$ *principal* $(A \times B)$
 ⟨proof⟩

lemma *le-prod-filterI*:

filtermap *fst* $F \leq A \implies$ *filtermap* *snd* $F \leq B \implies F \leq A \times_F B$
 ⟨proof⟩

lemma *filtermap-fst-prod-filter*: *filtermap* *fst* $(A \times_F B) \leq A$
 ⟨proof⟩

lemma *filtermap-snd-prod-filter*: *filtermap* *snd* $(A \times_F B) \leq B$
 ⟨proof⟩

lemma *prod-filter-INF*:

assumes $I \neq \{\}$ **and** $J \neq \{\}$
shows $(\prod_{i \in I}. A \ i) \times_F (\prod_{j \in J}. B \ j) = (\prod_{i \in I}. \prod_{j \in J}. A \ i \times_F B \ j)$
 ⟨proof⟩

lemma *filtermap-Pair*: *filtermap* $(\lambda x. (f \ x, g \ x)) \ F \leq$ *filtermap* $f \ F \times_F$ *filtermap* $g \ F$
 ⟨proof⟩

lemma *eventually-prodI*: *eventually* $P \ F \implies$ *eventually* $Q \ G \implies$ *eventually* $(\lambda x. P \ (fst \ x) \wedge Q \ (snd \ x)) \ (F \times_F G)$
 ⟨proof⟩

lemma *prod-filter-INF1*: $I \neq \{\} \implies (\prod_{i \in I}. A \ i) \times_F B = (\prod_{i \in I}. A \ i \times_F B)$
 ⟨proof⟩

lemma *prod-filter-INF2*: $J \neq \{\} \implies A \times_F (\prod_{i \in J}. B \ i) = (\prod_{i \in J}. A \times_F B \ i)$
 ⟨proof⟩

lemma *prod-filtermap1*: *prod-filter* $(\text{filtermap } f \ F) \ G =$ *filtermap* $(apfst \ f) \ (\text{prod-filter } F \ G)$
 ⟨proof⟩

lemma *prod-filtermap2*: *prod-filter* $F \ (\text{filtermap } g \ G) =$ *filtermap* $(apsnd \ g) \ (\text{prod-filter } F \ G)$
 ⟨proof⟩

lemma *prod-filter-assoc*:

prod-filter $(\text{prod-filter } F \ G) \ H =$ *filtermap* $(\lambda(x, y, z). ((x, y), z)) \ (\text{prod-filter } F \ (\text{prod-filter } G \ H))$

<proof>

lemma *prod-filter-principal-singleton*: *prod-filter (principal {x}) F = filtermap (Pair x) F*
<proof>

lemma *prod-filter-principal-singleton2*: *prod-filter F (principal {x}) = filtermap (λa. (a, x)) F*
<proof>

lemma *prod-filter-commute*: *prod-filter F G = filtermap prod.swap (prod-filter G F)*
<proof>

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* (⟨⟨*indent=3 notation=*⟨*binder LIM*⟩*LIM*
 (-)/ (-)./ (-) :> (-)⟩ [1000, 10, 0, 10] 10)

syntax-consts

-*LIM* == *filterlim*

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)*
<proof>

lemma *filterlim-top [simp]*: *filterlim f top F*
<proof>

lemma *filterlim-iff*:

(*LIM x F1. f x :> F2*) ⇔ (∀ *P*. *eventually P F2* ⇒ *eventually (λx. P (f x)) F1*)
<proof>

lemma *filterlim-compose*:

filterlim g F3 F2 ⇒ filterlim f F2 F1 ⇒ filterlim (λx. g (f x)) F3 F1
<proof>

lemma *filterlim-mono*:

filterlim f F2 F1 ⇒ F2 ≤ F2' ⇒ F1' ≤ F1 ⇒ filterlim f F2' F1'
<proof>

lemma *filterlim-ident*: *LIM x F. x :> F*
<proof>

lemma *filterlim-cong*:

$F1 = F1' \implies F2 = F2' \implies \text{eventually } (\lambda x. f\ x = g\ x)\ F2 \implies \text{filterlim } f\ F1\ F2$
 $= \text{filterlim } g\ F1'\ F2'$
 ⟨proof⟩

lemma *filterlim-mono-eventually*:

assumes *filterlim* $f\ F\ G$ **and** *ord*: $F \leq F'\ G' \leq G$
assumes *eq*: *eventually* $(\lambda x. f\ x = f'\ x)\ G'$
shows *filterlim* $f'\ F'\ G'$
 ⟨proof⟩

lemma *filtermap-mono-strong*: $\text{inj } f \implies \text{filtermap } f\ F \leq \text{filtermap } f\ G \longleftrightarrow F \leq G$
 ⟨proof⟩

lemma *eventually-compose-filterlim*:

assumes *eventually* $P\ F\ \text{filterlim } f\ F\ G$
shows *eventually* $(\lambda x. P\ (f\ x))\ G$
 ⟨proof⟩

lemma *filtermap-eq-strong*: $\text{inj } f \implies \text{filtermap } f\ F = \text{filtermap } f\ G \longleftrightarrow F = G$
 ⟨proof⟩

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⟩

lemma *filterlim-principal*:

$(\text{LIM } x\ F. f\ x :> \text{principal } S) \longleftrightarrow (\text{eventually } (\lambda x. f\ x \in S)\ F)$
 ⟨proof⟩

lemma *filterlim-filtercomap [intro]*: *filterlim* $f\ F\ (\text{filtercomap } f\ F)$
 ⟨proof⟩

lemma *filterlim-inf*:

$(\text{LIM } x\ F1. f\ x :> \inf F2\ F3) \longleftrightarrow ((\text{LIM } x\ F1. f\ x :> F2) \wedge (\text{LIM } x\ F1. f\ x :> F3))$
 ⟨proof⟩

lemma *filterlim-INF*:

$(\text{LIM } x\ F. f\ x :> (\bigcap b \in B. G\ b)) \longleftrightarrow (\forall b \in B. \text{LIM } x\ F. f\ x :> G\ b)$
 ⟨proof⟩

lemma *filterlim-INF-INF*:

$(\bigwedge m. m \in J \implies \exists i \in I. \text{filtermap } f\ (F\ i) \leq G\ m) \implies \text{LIM } x\ (\bigcap i \in I. F\ i). f\ x$

$:> (\prod_{j \in J}. G\ j)$
 $\langle \text{proof} \rangle$

lemma *filterlim-INF'*: $x \in A \implies \text{filterlim } f\ F\ (G\ x) \implies \text{filterlim } f\ F\ (\prod_{x \in A}. G\ x)$
 $\langle \text{proof} \rangle$

lemma *filterlim-filtercomap-iff*: $\text{filterlim } f\ (\text{filtercomap } g\ G)\ F \longleftrightarrow \text{filterlim } (g \circ f)\ G\ F$
 $\langle \text{proof} \rangle$

lemma *filterlim-iff-le-filtercomap*: $\text{filterlim } f\ F\ G \longleftrightarrow G \leq \text{filtercomap } f\ F$
 $\langle \text{proof} \rangle$

lemma *filterlim-base*:
 $(\bigwedge m\ x. m \in J \implies i\ m \in I) \implies (\bigwedge m\ x. m \in J \implies x \in F\ (i\ m) \implies f\ x \in G\ m) \implies$
 $LIM\ x\ (\prod_{i \in I}. \text{principal } (F\ i)).\ f\ x :> (\prod_{j \in J}. \text{principal } (G\ j))$
 $\langle \text{proof} \rangle$

lemma *filterlim-base-iff*:
assumes $I \neq \{\}$ **and** *chain*: $\bigwedge i\ j. i \in I \implies j \in I \implies F\ i \subseteq F\ j \vee F\ j \subseteq F\ i$
shows $(LIM\ x\ (\prod_{i \in I}. \text{principal } (F\ i)).\ f\ x :> \prod_{j \in J}. \text{principal } (G\ j)) \longleftrightarrow$
 $(\forall j \in J. \exists i \in I. \forall x \in F\ i. f\ x \in G\ j)$
 $\langle \text{proof} \rangle$

lemma *filterlim-filtermap*: $\text{filterlim } f\ F1\ (\text{filtermap } g\ F2) = \text{filterlim } (\lambda x. f\ (g\ x))\ F1\ F2$
 $\langle \text{proof} \rangle$

lemma *filterlim-sup*:
 $\text{filterlim } f\ F\ F1 \implies \text{filterlim } f\ F\ F2 \implies \text{filterlim } f\ F\ (\text{sup } F1\ F2)$
 $\langle \text{proof} \rangle$

lemma *filterlim-sequentially-Suc*:
 $(LIM\ x\ \text{sequentially}. f\ (\text{Suc } x) :> F) \longleftrightarrow (LIM\ x\ \text{sequentially}. f\ x :> F)$
 $\langle \text{proof} \rangle$

lemma *filterlim-Suc*: *filterlim Suc sequentially sequentially*
 $\langle \text{proof} \rangle$

lemma *filterlim-If*:
 $LIM\ x\ \text{inf } F\ (\text{principal } \{x. P\ x\}).\ f\ x :> G \implies$
 $LIM\ x\ \text{inf } F\ (\text{principal } \{x. \neg P\ x\}).\ g\ x :> G \implies$
 $LIM\ x\ F.\ \text{if } P\ x\ \text{then } f\ x\ \text{else } g\ x :> G$
 $\langle \text{proof} \rangle$

lemma *filterlim-Pair*:
 $LIM\ x\ F.\ f\ x :> G \implies LIM\ x\ F.\ g\ x :> H \implies LIM\ x\ F.\ (f\ x, g\ x) :> G \times_F H$

$\langle \text{proof} \rangle$

90.7 Limits to *at-top* and *at-bot*

lemma *filterlim-at-top*:

fixes $f :: 'a \Rightarrow ('b::\text{linorder})$

shows $(\text{LIM } x \ F. f \ x :> \text{at-top}) \longleftrightarrow (\forall Z. \text{eventually } (\lambda x. Z \leq f \ x) \ F)$

$\langle \text{proof} \rangle$

lemma *filterlim-at-top-mono*:

$\text{LIM } x \ F. f \ x :> \text{at-top} \implies \text{eventually } (\lambda x. f \ x \leq (g \ x::'a::\text{linorder})) \ F \implies$

$\text{LIM } x \ F. g \ x :> \text{at-top}$

$\langle \text{proof} \rangle$

lemma *filterlim-at-top-dense*:

fixes $f :: 'a \Rightarrow ('b::\text{unbounded-dense-linorder})$

shows $(\text{LIM } x \ F. f \ x :> \text{at-top}) \longleftrightarrow (\forall Z. \text{eventually } (\lambda x. Z < f \ x) \ F)$

$\langle \text{proof} \rangle$

lemma *filterlim-at-top-ge*:

fixes $f :: 'a \Rightarrow ('b::\text{linorder})$ **and** $c :: 'b$

shows $(\text{LIM } x \ F. f \ x :> \text{at-top}) \longleftrightarrow (\forall Z \geq c. \text{eventually } (\lambda x. Z \leq f \ x) \ F)$

$\langle \text{proof} \rangle$

lemma *filterlim-at-top-at-top*:

fixes $f :: 'a::\text{linorder} \Rightarrow 'b::\text{linorder}$

assumes *mono*: $\bigwedge x \ y. Q \ x \implies Q \ y \implies x \leq y \implies f \ x \leq f \ y$

assumes *bij*: $\bigwedge x. P \ x \implies f \ (g \ x) = x \ \bigwedge x. P \ x \implies Q \ (g \ x)$

assumes *Q*: *eventually Q at-top*

assumes *P*: *eventually P at-top*

shows *filterlim f at-top at-top*

$\langle \text{proof} \rangle$

lemma *filterlim-at-top-gt*:

fixes $f :: 'a \Rightarrow ('b::\text{unbounded-dense-linorder})$ **and** $c :: 'b$

shows $(\text{LIM } x \ F. f \ x :> \text{at-top}) \longleftrightarrow (\forall Z > c. \text{eventually } (\lambda x. Z \leq f \ x) \ F)$

$\langle \text{proof} \rangle$

lemma *filterlim-at-bot*:

fixes $f :: 'a \Rightarrow ('b::\text{linorder})$

shows $(\text{LIM } x \ F. f \ x :> \text{at-bot}) \longleftrightarrow (\forall Z. \text{eventually } (\lambda x. f \ x \leq Z) \ F)$

$\langle \text{proof} \rangle$

lemma *filterlim-at-bot-dense*:

fixes $f :: 'a \Rightarrow ('b::\{\text{dense-linorder}, \text{no-bot}\})$

shows $(\text{LIM } x \ F. f \ x :> \text{at-bot}) \longleftrightarrow (\forall Z. \text{eventually } (\lambda x. f \ x < Z) \ F)$

$\langle \text{proof} \rangle$

lemma *filterlim-at-bot-le*:

fixes $f :: 'a \Rightarrow ('b::linorder)$ **and** $c :: 'b$
shows $(LIM\ x\ F.\ f\ x\ :>\ at_bot) \longleftrightarrow (\forall\ Z \leq c.\ eventually\ (\lambda x.\ Z \geq f\ x)\ F)$
 $\langle proof \rangle$

lemma *filterlim-at-bot-lt*:

fixes $f :: 'a \Rightarrow ('b::unbounded-dense-linorder)$ **and** $c :: 'b$
shows $(LIM\ x\ F.\ f\ x\ :>\ at_bot) \longleftrightarrow (\forall\ Z < c.\ eventually\ (\lambda x.\ Z \geq f\ x)\ F)$
 $\langle proof \rangle$

lemma *filterlim-at-top-div-const-nat*:

assumes $c > 0$
shows $filterlim\ (\lambda x::nat.\ x\ div\ c)\ at_top\ at_top$
 $\langle proof \rangle$

lemma *filterlim-finite-subsets-at-top*:

$filterlim\ f\ (finite_subsets_at_top\ A)\ F \longleftrightarrow$
 $(\forall\ X.\ finite\ X \wedge X \subseteq A \longrightarrow eventually\ (\lambda y.\ finite\ (f\ y) \wedge X \subseteq f\ y \wedge f\ y \subseteq A)$
 $F)$
(is ?lhs = ?rhs)
 $\langle proof \rangle$

lemma *filterlim-atMost-at-top*:

$filterlim\ (\lambda n.\ \{..n\})\ (finite_subsets_at_top\ (UNIV :: nat\ set))\ at_top$
 $\langle proof \rangle$

lemma *filterlim-lessThan-at-top*:

$filterlim\ (\lambda n.\ \{.. $n\})\ (finite_subsets_at_top\ (UNIV :: nat\ set))\ at_top$
 $\langle proof \rangle$$

lemma *filterlim-minus-const-nat-at-top*:

$filterlim\ (\lambda n.\ n - c)\ sequentially\ sequentially$
 $\langle proof \rangle$

lemma *filterlim-add-const-nat-at-top*:

$filterlim\ (\lambda n.\ n + c)\ sequentially\ sequentially$
 $\langle proof \rangle$

90.8 Setup 'a filter for lifting and transfer

lemma *filtermap-id [simp, id-simps]*: $filtermap\ id = id$
 $\langle proof \rangle$

lemma *filtermap-id' [simp]*: $filtermap\ (\lambda x.\ x) = (\lambda F.\ F)$
 $\langle proof \rangle$

context *includes lifting-syntax*

begin

definition *map-filter-on* :: $'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ filter \Rightarrow 'b\ filter$ **where**

$\text{map-filter-on } X f F = \text{Abs-filter } (\lambda P. \text{eventually } (\lambda x. P (f x) \wedge x \in X) F)$

lemma *is-filter-map-filter-on*:

$\text{is-filter } (\lambda P. \forall_F x \text{ in } F. P (f x) \wedge x \in X) \longleftrightarrow \text{eventually } (\lambda x. x \in X) F$
 $\langle \text{proof} \rangle$

lemma *eventually-map-filter-on*: $\text{eventually } P (\text{map-filter-on } X f F) = (\forall_F x \text{ in } F. P (f x) \wedge x \in X)$

if $\text{eventually } (\lambda x. x \in X) F$

$\langle \text{proof} \rangle$

lemma *map-filter-on-UNIV*: $\text{map-filter-on } \text{UNIV} = \text{filtermap}$

$\langle \text{proof} \rangle$

lemma *map-filter-on-comp*: $\text{map-filter-on } X f (\text{map-filter-on } Y g F) = \text{map-filter-on } Y (f \circ g) F$

if $g \text{ ‘ } Y \subseteq X$ **and** $\text{eventually } (\lambda x. x \in Y) F$

$\langle \text{proof} \rangle$

inductive *rel-filter* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{ filter} \Rightarrow 'b \text{ filter} \Rightarrow \text{bool}$ **for** $R F G$
where

$\text{rel-filter } R F G$ **if** $\text{eventually } (\text{case-prod } R) Z \text{ map-filter-on } \{(x, y). R x y\} \text{ fst } Z$
 $= F \text{ map-filter-on } \{(x, y). R x y\} \text{ snd } Z = G$

lemma *rel-filter-eq* [*relator-eq*]: $\text{rel-filter } (=) = (=)$

$\langle \text{proof} \rangle$

lemma *rel-filter-mono* [*relator-mono*]: $\text{rel-filter } A \leq \text{rel-filter } B$ **if** $\text{le}: A \leq B$

$\langle \text{proof} \rangle$

lemma *rel-filter-conversep*: $\text{rel-filter } A^{-1-1} = (\text{rel-filter } A)^{-1-1}$

$\langle \text{proof} \rangle$

lemma *rel-filter-distr* [*relator-distr*]:

$\text{rel-filter } A \text{ OO rel-filter } B = \text{rel-filter } (A \text{ OO } B)$

$\langle \text{proof} \rangle$

lemma *filtermap-parametric*: $((A ==> B) ==> \text{rel-filter } A ==> \text{rel-filter } B) \text{ filtermap filtermap}$

$\langle \text{proof} \rangle$

lemma *rel-filter-Grp*: $\text{rel-filter } (\text{Grp } \text{UNIV } f) = \text{Grp } \text{UNIV } (\text{filtermap } f)$

$\langle \text{proof} \rangle$

lemma *Quotient-filter* [*quot-map*]:

$\text{Quotient } R \text{ Abs Rep } T \Longrightarrow \text{Quotient } (\text{rel-filter } R) (\text{filtermap } \text{Abs}) (\text{filtermap } \text{Rep})$
 $(\text{rel-filter } T)$

$\langle \text{proof} \rangle$

lemma *left-total-rel-filter* [transfer-rule]: *left-total* $A \implies \text{left-total } (\text{rel-filter } A)$
 ⟨proof⟩

lemma *right-total-rel-filter* [transfer-rule]: *right-total* $A \implies \text{right-total } (\text{rel-filter } A)$
 ⟨proof⟩

lemma *bi-total-rel-filter* [transfer-rule]: *bi-total* $A \implies \text{bi-total } (\text{rel-filter } A)$
 ⟨proof⟩

lemma *left-unique-rel-filter* [transfer-rule]: *left-unique* $A \implies \text{left-unique } (\text{rel-filter } A)$
 ⟨proof⟩

lemma *right-unique-rel-filter* [transfer-rule]:
right-unique $A \implies \text{right-unique } (\text{rel-filter } A)$
 ⟨proof⟩

lemma *bi-unique-rel-filter* [transfer-rule]: *bi-unique* $A \implies \text{bi-unique } (\text{rel-filter } A)$
 ⟨proof⟩

lemma *eventually-parametric* [transfer-rule]:
 $((A \implies (=)) \implies \text{rel-filter } A \implies (=)) \text{ eventually eventually}$
 ⟨proof⟩

lemma *frequently-parametric* [transfer-rule]: $((A \implies (=)) \implies \text{rel-filter } A \implies (=)) \text{ frequently frequently}$
 ⟨proof⟩

lemma *is-filter-parametric* [transfer-rule]:
assumes [transfer-rule]: *bi-total* A
assumes [transfer-rule]: *bi-unique* A
shows $((A \implies (=)) \implies (=)) \implies (=) \text{ is-filter is-filter}$
 ⟨proof⟩

lemma *top-filter-parametric* [transfer-rule]: *rel-filter* $A \text{ top top}$ **if** *bi-total* A
 ⟨proof⟩

lemma *bot-filter-parametric* [transfer-rule]: *rel-filter* $A \text{ bot bot}$
 ⟨proof⟩

lemma *principal-parametric* [transfer-rule]: $(\text{rel-set } A \implies \text{rel-filter } A) \text{ principal principal}$
 ⟨proof⟩

lemma *sup-filter-parametric* [transfer-rule]:
 $(\text{rel-filter } A \implies \text{rel-filter } A \implies \text{rel-filter } A) \text{ sup sup}$
 ⟨proof⟩

lemma *Sup-filter-parametric* [transfer-rule]: $(\text{rel-set } (\text{rel-filter } A) ==> \text{rel-filter } A) \text{ Sup Sup}$
 $\langle \text{proof} \rangle$

context

fixes $A :: 'a \Rightarrow 'b \Rightarrow \text{bool}$

assumes [transfer-rule]: *bi-unique* A

begin

lemma *le-filter-parametric* [transfer-rule]:
 $(\text{rel-filter } A ==> \text{rel-filter } A ==> (=)) \ (\le) \ (\le)$
 $\langle \text{proof} \rangle$

lemma *less-filter-parametric* [transfer-rule]:
 $(\text{rel-filter } A ==> \text{rel-filter } A ==> (=)) \ (<) \ (<)$
 $\langle \text{proof} \rangle$

context

assumes [transfer-rule]: *bi-total* A

begin

lemma *Inf-filter-parametric* [transfer-rule]:
 $(\text{rel-set } (\text{rel-filter } A) ==> \text{rel-filter } A) \text{ Inf Inf}$
 $\langle \text{proof} \rangle$

lemma *inf-filter-parametric* [transfer-rule]:
 $(\text{rel-filter } A ==> \text{rel-filter } A ==> \text{rel-filter } A) \text{ inf inf}$
 $\langle \text{proof} \rangle$

end

end

end

context

includes *lifting-syntax*

begin

lemma *prod-filter-parametric* [transfer-rule]:
 $(\text{rel-filter } R ==> \text{rel-filter } S ==> \text{rel-filter } (\text{rel-prod } R \ S)) \text{ prod-filter prod-filter}$
 $\langle \text{proof} \rangle$

end

Code generation for filters

definition *abstract-filter* :: $(\text{unit} \Rightarrow 'a \text{ filter}) \Rightarrow 'a \text{ filter}$
where [simp]: *abstract-filter* $f = f \ ()$

```

code-datatype principal abstract-filter

hide-const (open) abstract-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  $\longleftrightarrow$  eventually P F
  <proof>

lemma bot-eq-principal-empty [code]:
  bot = principal {}
  <proof>

lemma top-eq-principal-UNIV [code]:
  top = principal UNIV
  <proof>

instantiation filter :: (equal) equal
begin

definition equal-filter :: 'a filter  $\Rightarrow$  'a filter  $\Rightarrow$  bool
  where equal-filter F F'  $\longleftrightarrow$  F = F'

lemma equal-filter [code]:
  HOL.equal (principal A) (principal B)  $\longleftrightarrow$  A = B
  <proof>

instance
  <proof>

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* :: $\langle 'a \text{ set} \Rightarrow \text{bool} \rangle$
where *unfold*: $\langle \text{bdd } A \longleftrightarrow (\exists M. \forall x \in A. x \leq M) \rangle$

lemma *empty* [*simp*, *intro*]:
 $\langle \text{bdd } \{\} \rangle$
 $\langle \text{proof} \rangle$

lemma *I* [*intro*]:
 $\langle \text{bdd } A \rangle$ **if** $\langle \bigwedge x. x \in A \Longrightarrow x \leq M \rangle$
 $\langle \text{proof} \rangle$

lemma *E*:
assumes $\langle \text{bdd } A \rangle$
obtains *M* **where** $\langle \bigwedge x. x \in A \Longrightarrow x \leq M \rangle$
 $\langle \text{proof} \rangle$

lemma *I2*:
 $\langle \text{bdd } (f \text{ ` } A) \rangle$ **if** $\langle \bigwedge x. x \in A \Longrightarrow f \ x \leq M \rangle$
 $\langle \text{proof} \rangle$

lemma *mono*:
 $\langle \text{bdd } A \rangle$ **if** $\langle \text{bdd } B \rangle$ $\langle A \subseteq B \rangle$
 $\langle \text{proof} \rangle$

lemma *Int1* [*simp*]:
 $\langle \text{bdd } (A \cap B) \rangle$ **if** $\langle \text{bdd } A \rangle$
 $\langle \text{proof} \rangle$

lemma *Int2* [*simp*]:
 $\langle \text{bdd } (A \cap B) \rangle$ **if** $\langle \text{bdd } B \rangle$
 $\langle \text{proof} \rangle$

end

91.1 Preorders

context *preorder*

begin

sublocale *bdd-above*: *preordering-bdd* $\langle (\leq) \rangle$ $\langle (<) \rangle$
defines *bdd-above-primitive-def*: *bdd-above* = *bdd-above.bdd* $\langle \text{proof} \rangle$

sublocale *bdd-below*: *preordering-bdd* $\langle (\geq) \rangle$ $\langle (>) \rangle$
defines *bdd-below-primitive-def*: *bdd-below* = *bdd-below.bdd* $\langle \text{proof} \rangle$

lemma *bdd-above-def*: $\langle \text{bdd-above } A \longleftrightarrow (\exists M. \forall x \in A. x \leq M) \rangle$
 $\langle \text{proof} \rangle$

lemma *bdd-below-def*: $\langle bdd\text{-}below\ A \longleftrightarrow (\exists M. \forall x \in A. M \leq x) \rangle$
 $\langle proof \rangle$

lemma *bdd-aboveI*: $(\bigwedge x. x \in A \implies x \leq M) \implies bdd\text{-}above\ A$
 $\langle proof \rangle$

lemma *bdd-belowI*: $(\bigwedge x. x \in A \implies m \leq x) \implies bdd\text{-}below\ A$
 $\langle proof \rangle$

lemma *bdd-aboveI2*: $(\bigwedge x. x \in A \implies f\ x \leq M) \implies bdd\text{-}above\ (f'A)$
 $\langle proof \rangle$

lemma *bdd-belowI2*: $(\bigwedge x. x \in A \implies m \leq f\ x) \implies bdd\text{-}below\ (f'A)$
 $\langle proof \rangle$

lemma *bdd-above-empty*: $bdd\text{-}above\ \{\}$
 $\langle proof \rangle$

lemma *bdd-below-empty*: $bdd\text{-}below\ \{\}$
 $\langle proof \rangle$

lemma *bdd-above-mono*: $bdd\text{-}above\ B \implies A \subseteq B \implies bdd\text{-}above\ A$
 $\langle proof \rangle$

lemma *bdd-below-mono*: $bdd\text{-}below\ B \implies A \subseteq B \implies bdd\text{-}below\ A$
 $\langle proof \rangle$

lemma *bdd-above-Int1*: $bdd\text{-}above\ A \implies bdd\text{-}above\ (A \cap B)$
 $\langle proof \rangle$

lemma *bdd-above-Int2*: $bdd\text{-}above\ B \implies bdd\text{-}above\ (A \cap B)$
 $\langle proof \rangle$

lemma *bdd-below-Int1*: $bdd\text{-}below\ A \implies bdd\text{-}below\ (A \cap B)$
 $\langle proof \rangle$

lemma *bdd-below-Int2*: $bdd\text{-}below\ B \implies bdd\text{-}below\ (A \cap B)$
 $\langle proof \rangle$

lemma *bdd-above-Ioo* [*simp*, *intro*]: $bdd\text{-}above\ \{a <..< b\}$
 $\langle proof \rangle$

lemma *bdd-above-Ico* [*simp*, *intro*]: $bdd\text{-}above\ \{a ..< b\}$
 $\langle proof \rangle$

lemma *bdd-above-Iio* [*simp*, *intro*]: $bdd\text{-}above\ \{..< b\}$
 $\langle proof \rangle$

lemma *bdd-above-Ioc* [*simp*, *intro*]: *bdd-above* {*a* <..*b*}
 ⟨*proof*⟩

lemma *bdd-above-Icc* [*simp*, *intro*]: *bdd-above* {*a* .. *b*}
 ⟨*proof*⟩

lemma *bdd-above-Iic* [*simp*, *intro*]: *bdd-above* {..*b*}
 ⟨*proof*⟩

lemma *bdd-below-Ioo* [*simp*, *intro*]: *bdd-below* {*a* <..*b*}
 ⟨*proof*⟩

lemma *bdd-below-Ioc* [*simp*, *intro*]: *bdd-below* {*a* <..*b*}
 ⟨*proof*⟩

lemma *bdd-below-Ioi* [*simp*, *intro*]: *bdd-below* {*a* <..*b*}
 ⟨*proof*⟩

lemma *bdd-below-Ico* [*simp*, *intro*]: *bdd-below* {*a* ..< *b*}
 ⟨*proof*⟩

lemma *bdd-below-Icc* [*simp*, *intro*]: *bdd-below* {*a* .. *b*}
 ⟨*proof*⟩

lemma *bdd-below-Ici* [*simp*, *intro*]: *bdd-below* {*a* ..} *b*
 ⟨*proof*⟩

end

context *order-top*
begin

lemma *bdd-above-top* [*simp*, *intro*!]: *bdd-above* *A*
 ⟨*proof*⟩

end

context *order-bot*
begin

lemma *bdd-below-bot* [*simp*, *intro*!]: *bdd-below* *A*
 ⟨*proof*⟩

end

lemma *bdd-above-image-mono*: *mono* *f* \implies *bdd-above* *A* \implies *bdd-above* (*f*‘*A*)
 ⟨*proof*⟩

lemma *bdd-below-image-mono*: *mono* *f* \implies *bdd-below* *A* \implies *bdd-below* (*f*‘*A*)

<proof>

lemma *bdd-above-image-antimono*: *antimono* $f \implies \text{bdd-below } A \implies \text{bdd-above } (f'A)$
<proof>

lemma *bdd-below-image-antimono*: *antimono* $f \implies \text{bdd-above } A \implies \text{bdd-below } (f'A)$
<proof>

lemma

fixes $X :: 'a::\text{ordered-ab-group-add set}$

shows *bdd-above-uminus*[*simp*]: *bdd-above* (*uminus* ' X) \longleftrightarrow *bdd-below* X

and *bdd-below-uminus*[*simp*]: *bdd-below* (*uminus* ' X) \longleftrightarrow *bdd-above* X

<proof>

91.2 Lattices

context *lattice*

begin

lemma *bdd-above-insert* [*simp*]: *bdd-above* (*insert* a A) = *bdd-above* A
<proof>

lemma *bdd-below-insert* [*simp*]: *bdd-below* (*insert* a A) = *bdd-below* A
<proof>

lemma *bdd-finite* [*simp*]:

assumes *finite* A **shows** *bdd-above-finite*: *bdd-above* A **and** *bdd-below-finite*:
bdd-below A

<proof>

lemma *bdd-above-Un* [*simp*]: *bdd-above* ($A \cup B$) = (*bdd-above* $A \wedge$ *bdd-above* B)
<proof>

lemma *bdd-below-Un* [*simp*]: *bdd-below* ($A \cup B$) = (*bdd-below* $A \wedge$ *bdd-below* B)
<proof>

lemma *bdd-above-image-sup*[*simp*]:

bdd-above ($((\lambda x. \text{sup } (f x) (g x)) ' A)$ \longleftrightarrow *bdd-above* ($f'A$) \wedge *bdd-above* ($g'A$)
<proof>

lemma *bdd-below-image-inf*[*simp*]:

bdd-below ($((\lambda x. \text{inf } (f x) (g x)) ' A)$ \longleftrightarrow *bdd-below* ($f'A$) \wedge *bdd-below* ($g'A$)
<proof>

lemma *bdd-below-UN*[*simp*]: *finite* $I \implies \text{bdd-below } (\bigcup_{i \in I}. A i) = (\forall i \in I. \text{bdd-below } (A i))$
<proof>

lemma *bdd-above-UN[simp]*: *finite I* \implies *bdd-above* $(\bigcup_{i \in I}. A\ i) = (\forall i \in I. \text{bdd-above } (A\ i))$
 $\langle \text{proof} \rangle$

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*: $x \in X \implies \text{bdd-below } X \implies \text{Inf } X \leq x$
and *cInf-greatest*: $X \neq \{\} \implies (\bigwedge x. x \in X \implies z \leq x) \implies z \leq \text{Inf } X$
assumes *cSup-upper*: $x \in X \implies \text{bdd-above } X \implies x \leq \text{Sup } X$
and *cSup-least*: $X \neq \{\} \implies (\bigwedge x. x \in X \implies x \leq z) \implies \text{Sup } X \leq z$
begin

lemma *cSup-upper2*: $x \in X \implies y \leq x \implies \text{bdd-above } X \implies y \leq \text{Sup } X$
 $\langle \text{proof} \rangle$

lemma *cInf-lower2*: $x \in X \implies x \leq y \implies \text{bdd-below } X \implies \text{Inf } X \leq y$
 $\langle \text{proof} \rangle$

lemma *cSup-mono*: $B \neq \{\} \implies \text{bdd-above } A \implies (\bigwedge b. b \in B \implies \exists a \in A. b \leq a) \implies \text{Sup } B \leq \text{Sup } A$
 $\langle \text{proof} \rangle$

lemma *cInf-mono*: $B \neq \{\} \implies \text{bdd-below } A \implies (\bigwedge b. b \in B \implies \exists a \in A. a \leq b) \implies \text{Inf } A \leq \text{Inf } B$
 $\langle \text{proof} \rangle$

lemma *cSup-subset-mono*: $A \neq \{\} \implies \text{bdd-above } B \implies A \subseteq B \implies \text{Sup } A \leq \text{Sup } B$
 $\langle \text{proof} \rangle$

lemma *cInf-superset-mono*: $A \neq \{\} \implies \text{bdd-below } B \implies A \subseteq B \implies \text{Inf } B \leq \text{Inf } A$
 $\langle \text{proof} \rangle$

lemma *cSup-eq-maximum*: $z \in X \implies (\bigwedge x. x \in X \implies x \leq z) \implies \text{Sup } X = z$
 $\langle \text{proof} \rangle$

lemma *cInf-eq-minimum*: $z \in X \implies (\bigwedge x. x \in X \implies z \leq x) \implies \text{Inf } X = z$
 $\langle \text{proof} \rangle$

lemma *cSup-le-iff*: $S \neq \{\} \implies \text{bdd-above } S \implies \text{Sup } S \leq a \longleftrightarrow (\forall x \in S. x \leq a)$
 $\langle \text{proof} \rangle$

lemma *le-cInf-iff*: $S \neq \{\}$ \implies *bdd-below* $S \implies a \leq \text{Inf } S \longleftrightarrow (\forall x \in S. a \leq x)$
 $\langle \text{proof} \rangle$

lemma *cSup-eq-non-empty*:
assumes 1: $X \neq \{\}$
assumes 2: $\bigwedge x. x \in X \implies x \leq a$
assumes 3: $\bigwedge y. (\bigwedge x. x \in X \implies x \leq y) \implies a \leq y$
shows $\text{Sup } X = a$
 $\langle \text{proof} \rangle$

lemma *cInf-eq-non-empty*:
assumes 1: $X \neq \{\}$
assumes 2: $\bigwedge x. x \in X \implies a \leq x$
assumes 3: $\bigwedge y. (\bigwedge x. x \in X \implies y \leq x) \implies y \leq a$
shows $\text{Inf } X = a$
 $\langle \text{proof} \rangle$

lemma *cInf-cSup*: $S \neq \{\}$ \implies *bdd-below* $S \implies \text{Inf } S = \text{Sup } \{x. \forall s \in S. x \leq s\}$
 $\langle \text{proof} \rangle$

lemma *cSup-cInf*: $S \neq \{\}$ \implies *bdd-above* $S \implies \text{Sup } S = \text{Inf } \{x. \forall s \in S. s \leq x\}$
 $\langle \text{proof} \rangle$

lemma *cSup-insert*: $X \neq \{\}$ \implies *bdd-above* $X \implies \text{Sup } (\text{insert } a \ X) = \text{sup } a \ (\text{Sup } X)$
 $\langle \text{proof} \rangle$

lemma *cInf-insert*: $X \neq \{\}$ \implies *bdd-below* $X \implies \text{Inf } (\text{insert } a \ X) = \text{inf } a \ (\text{Inf } X)$
 $\langle \text{proof} \rangle$

lemma *cSup-singleton [simp]*: $\text{Sup } \{x\} = x$
 $\langle \text{proof} \rangle$

lemma *cInf-singleton [simp]*: $\text{Inf } \{x\} = x$
 $\langle \text{proof} \rangle$

lemma *cSup-insert-If*: *bdd-above* $X \implies \text{Sup } (\text{insert } a \ X) = (\text{if } X = \{\} \text{ then } a \text{ else } \text{sup } a \ (\text{Sup } X))$
 $\langle \text{proof} \rangle$

lemma *cInf-insert-If*: *bdd-below* $X \implies \text{Inf } (\text{insert } a \ X) = (\text{if } X = \{\} \text{ then } a \text{ else } \text{inf } a \ (\text{Inf } X))$
 $\langle \text{proof} \rangle$

lemma *le-cSup-finite*: *finite* $X \implies x \in X \implies x \leq \text{Sup } X$
 $\langle \text{proof} \rangle$

lemma *cInf-le-finite*: *finite* $X \implies x \in X \implies \text{Inf } X \leq x$

$\langle proof \rangle$

lemma *cSup-eq-Sup-fin*: $finite\ X \implies X \neq \{\} \implies Sup\ X = Sup\text{-}fin\ X$
 $\langle proof \rangle$

lemma *cInf-eq-Inf-fin*: $finite\ X \implies X \neq \{\} \implies Inf\ X = Inf\text{-}fin\ X$
 $\langle proof \rangle$

lemma *cSup-atMost[simp]*: $Sup\ \{..x\} = x$
 $\langle proof \rangle$

lemma *cSup-greaterThanAtMost[simp]*: $y < x \implies Sup\ \{y<..x\} = x$
 $\langle proof \rangle$

lemma *cSup-atLeastAtMost[simp]*: $y \leq x \implies Sup\ \{y..x\} = x$
 $\langle proof \rangle$

lemma *cInf-atLeast[simp]*: $Inf\ \{x.. \} = x$
 $\langle proof \rangle$

lemma *cInf-atLeastLessThan[simp]*: $y < x \implies Inf\ \{y.. $x\} = y$$
 $\langle proof \rangle$

lemma *cInf-atLeastAtMost[simp]*: $y \leq x \implies Inf\ \{y.. $x\} = y$$
 $\langle proof \rangle$

lemma *cINF-lower*: $bdd\text{-}below\ (f\ '\ A) \implies x \in A \implies \bigcap (f\ '\ A) \leq f\ x$
 $\langle proof \rangle$

lemma *cINF-greatest*: $A \neq \{\} \implies (\bigwedge x. x \in A \implies m \leq f\ x) \implies m \leq \bigcap (f\ '\ A)$
 $\langle proof \rangle$

lemma *cSUP-upper*: $x \in A \implies bdd\text{-}above\ (f\ '\ A) \implies f\ x \leq \bigcup (f\ '\ A)$
 $\langle proof \rangle$

lemma *cSUP-least*: $A \neq \{\} \implies (\bigwedge x. x \in A \implies f\ x \leq M) \implies \bigcup (f\ '\ A) \leq M$
 $\langle proof \rangle$

lemma *cINF-lower2*: $bdd\text{-}below\ (f\ '\ A) \implies x \in A \implies f\ x \leq u \implies \bigcap (f\ '\ A) \leq u$
 $\langle proof \rangle$

lemma *cSUP-upper2*: $bdd\text{-}above\ (f\ '\ A) \implies x \in A \implies u \leq f\ x \implies u \leq \bigcup (f\ '\ A)$
 $\langle proof \rangle$

lemma *cSUP-const [simp]*: $A \neq \{\} \implies (\bigcup x \in A. c) = c$
 $\langle proof \rangle$

lemma *cINF-const [simp]*: $A \neq \{\} \implies (\bigcap x \in A. c) = c$
 $\langle proof \rangle$

lemma *le-cINF-iff*: $A \neq \{\}$ \implies $\text{bdd-below } (f \text{ ' } A) \implies u \leq \prod (f \text{ ' } A) \longleftrightarrow (\forall x \in A. u \leq f x)$
 ⟨proof⟩

lemma *cSUP-le-iff*: $A \neq \{\}$ \implies $\text{bdd-above } (f \text{ ' } A) \implies \bigsqcup (f \text{ ' } A) \leq u \longleftrightarrow (\forall x \in A. f x \leq u)$
 ⟨proof⟩

lemma *less-cINF-D*: $\text{bdd-below } (f \text{ ' } A) \implies y < (\prod i \in A. f i) \implies i \in A \implies y < f i$
 ⟨proof⟩

lemma *cSUP-lessD*: $\text{bdd-above } (f \text{ ' } A) \implies (\bigsqcup i \in A. f i) < y \implies i \in A \implies f i < y$
 ⟨proof⟩

lemma *cINF-insert*: $A \neq \{\}$ \implies $\text{bdd-below } (f \text{ ' } A) \implies \prod (f \text{ ' } \text{insert } a \text{ } A) = \inf (f a) (\prod (f \text{ ' } A))$
 ⟨proof⟩

lemma *cSUP-insert*: $A \neq \{\}$ \implies $\text{bdd-above } (f \text{ ' } A) \implies \bigsqcup (f \text{ ' } \text{insert } a \text{ } A) = \sup (f a) (\bigsqcup (f \text{ ' } A))$
 ⟨proof⟩

lemma *cINF-mono*: $B \neq \{\}$ \implies $\text{bdd-below } (f \text{ ' } A) \implies (\bigwedge m. m \in B \implies \exists n \in A. f n \leq g m) \implies \prod (f \text{ ' } A) \leq \prod (g \text{ ' } B)$
 ⟨proof⟩

lemma *cSUP-mono*: $A \neq \{\}$ \implies $\text{bdd-above } (g \text{ ' } B) \implies (\bigwedge n. n \in A \implies \exists m \in B. f n \leq g m) \implies \bigsqcup (f \text{ ' } A) \leq \bigsqcup (g \text{ ' } B)$
 ⟨proof⟩

lemma *cINF-superset-mono*: $A \neq \{\}$ \implies $\text{bdd-below } (g \text{ ' } B) \implies A \subseteq B \implies (\bigwedge x. x \in B \implies g x \leq f x) \implies \prod (g \text{ ' } B) \leq \prod (f \text{ ' } A)$
 ⟨proof⟩

lemma *cSUP-subset-mono*:
 $\llbracket A \neq \{\}; \text{bdd-above } (g \text{ ' } B); A \subseteq B; \bigwedge x. x \in A \implies f x \leq g x \rrbracket \implies \bigsqcup (f \text{ ' } A) \leq \bigsqcup (g \text{ ' } B)$
 ⟨proof⟩

lemma *less-eq-cInf-inter*: $\text{bdd-below } A \implies \text{bdd-below } B \implies A \cap B \neq \{\} \implies \inf (\inf A) (\inf B) \leq \inf (A \cap B)$
 ⟨proof⟩

lemma *cSup-inter-less-eq*: $\text{bdd-above } A \implies \text{bdd-above } B \implies A \cap B \neq \{\} \implies \sup (A \cap B) \leq \sup (\sup A) (\sup B)$
 ⟨proof⟩

lemma *cInf-union-distrib*: $A \neq \{\} \implies \text{bdd-below } A \implies B \neq \{\} \implies \text{bdd-below } B$

$\implies \text{Inf } (A \cup B) = \text{inf } (\text{Inf } A) (\text{Inf } B)$
 $\langle \text{proof} \rangle$

lemma *cINF-union*: $A \neq \{\}$ $\implies \text{bdd-below } (f \text{ ‘ } A) \implies B \neq \{\} \implies \text{bdd-below } (f \text{ ‘ } B) \implies \bigcap (f \text{ ‘ } (A \cup B)) = \bigcap (f \text{ ‘ } A) \cap \bigcap (f \text{ ‘ } B)$
 $\langle \text{proof} \rangle$

lemma *cSup-union-distrib*: $A \neq \{\} \implies \text{bdd-above } A \implies B \neq \{\} \implies \text{bdd-above } B \implies \text{Sup } (A \cup B) = \text{sup } (\text{Sup } A) (\text{Sup } B)$
 $\langle \text{proof} \rangle$

lemma *cSUP-union*: $A \neq \{\} \implies \text{bdd-above } (f \text{ ‘ } A) \implies B \neq \{\} \implies \text{bdd-above } (f \text{ ‘ } B) \implies \bigsqcup (f \text{ ‘ } (A \cup B)) = \bigsqcup (f \text{ ‘ } A) \sqcup \bigsqcup (f \text{ ‘ } B)$
 $\langle \text{proof} \rangle$

lemma *cINF-inf-distrib*: $A \neq \{\} \implies \text{bdd-below } (f \text{ ‘ } A) \implies \text{bdd-below } (g \text{ ‘ } A) \implies \bigcap (f \text{ ‘ } A) \cap \bigcap (g \text{ ‘ } A) = (\bigcap_{a \in A. \text{inf } (f \text{ ‘ } a) (g \text{ ‘ } a)})$
 $\langle \text{proof} \rangle$

lemma *SUP-sup-distrib*: $A \neq \{\} \implies \text{bdd-above } (f \text{ ‘ } A) \implies \text{bdd-above } (g \text{ ‘ } A) \implies \bigsqcup (f \text{ ‘ } A) \sqcup \bigsqcup (g \text{ ‘ } A) = (\bigsqcup_{a \in A. \text{sup } (f \text{ ‘ } a) (g \text{ ‘ } a)})$
 $\langle \text{proof} \rangle$

lemma *cInf-le-cSup*:

$A \neq \{\} \implies \text{bdd-above } A \implies \text{bdd-below } A \implies \text{Inf } A \leq \text{Sup } A$
 $\langle \text{proof} \rangle$

context

fixes $f :: 'a \Rightarrow 'b :: \text{conditionally-complete-lattice}$

assumes *mono f*

begin

lemma *mono-cInf*: $\llbracket \text{bdd-below } A; A \neq \{\} \rrbracket \implies f (\text{Inf } A) \leq (\text{INF } x \in A. f \text{ ‘ } x)$
 $\langle \text{proof} \rangle$

lemma *mono-cSup*: $\llbracket \text{bdd-above } A; A \neq \{\} \rrbracket \implies (\text{SUP } x \in A. f \text{ ‘ } x) \leq f (\text{Sup } A)$
 $\langle \text{proof} \rangle$

lemma *mono-cINF*: $\llbracket \text{bdd-below } (A \text{ ‘ } I); I \neq \{\} \rrbracket \implies f (\text{INF } i \in I. A \text{ ‘ } i) \leq (\text{INF } x \in I. f \text{ ‘ } (A \text{ ‘ } x))$
 $\langle \text{proof} \rangle$

lemma *mono-cSUP*: $\llbracket \text{bdd-above } (A \text{ ‘ } I); I \neq \{\} \rrbracket \implies (\text{SUP } x \in I. f \text{ ‘ } (A \text{ ‘ } x)) \leq f (\text{SUP } i \in I. A \text{ ‘ } i)$
 $\langle \text{proof} \rangle$

end

end

The special case of well-orderings

lemma *wellorder-Inf*:

fixes $k :: 'a :: \{\text{wellorder}, \text{conditionally-complete-lattice}\}$
assumes $k \in A$ **shows** $\text{Inf } A \in A$
 $\langle \text{proof} \rangle$

lemma *wellorder-Inf-le1*:

fixes $k :: 'a :: \{\text{wellorder}, \text{conditionally-complete-lattice}\}$
assumes $k \in A$ **shows** $\text{Inf } A \leq k$
 $\langle \text{proof} \rangle$

91.4 Complete lattices

instance *complete-lattice* \subseteq *conditionally-complete-lattice*
 $\langle \text{proof} \rangle$

lemma *cSup-eq*:

fixes $a :: 'a :: \{\text{conditionally-complete-lattice}, \text{no-bot}\}$
assumes *upper*: $\bigwedge x. x \in X \implies x \leq a$
assumes *least*: $\bigwedge y. (\bigwedge x. x \in X \implies x \leq y) \implies a \leq y$
shows $\text{Sup } X = a$
 $\langle \text{proof} \rangle$

lemma *cSup-unique*:

fixes $b :: 'a :: \{\text{conditionally-complete-lattice}, \text{no-bot}\}$
assumes $\bigwedge c. (\forall x \in s. x \leq c) \longleftrightarrow b \leq c$
shows $\text{Sup } s = b$
 $\langle \text{proof} \rangle$

lemma *cInf-eq*:

fixes $a :: 'a :: \{\text{conditionally-complete-lattice}, \text{no-top}\}$
assumes *upper*: $\bigwedge x. x \in X \implies a \leq x$
assumes *least*: $\bigwedge y. (\bigwedge x. x \in X \implies y \leq x) \implies y \leq a$
shows $\text{Inf } X = a$
 $\langle \text{proof} \rangle$

lemma *cInf-unique*:

fixes $b :: 'a :: \{\text{conditionally-complete-lattice}, \text{no-top}\}$
assumes $\bigwedge c. (\forall x \in s. x \geq c) \longleftrightarrow b \geq c$
shows $\text{Inf } s = b$
 $\langle \text{proof} \rangle$

class *conditionally-complete-linorder* = *conditionally-complete-lattice* + *linorder*
begin

lemma *less-cSup-iff*:

$X \neq \{\}$ $\implies \text{bdd-above } X \implies y < \text{Sup } X \longleftrightarrow (\exists x \in X. y < x)$
 $\langle \text{proof} \rangle$

lemma *cInf-less-iff*: $X \neq \{\} \implies \text{bdd-below } X \implies \text{Inf } X < y \longleftrightarrow (\exists x \in X. x < y)$
 ⟨proof⟩

lemma *cINF-less-iff*: $A \neq \{\} \implies \text{bdd-below } (f'A) \implies (\bigcap i \in A. f\ i) < a \longleftrightarrow (\exists x \in A. f\ x < a)$
 ⟨proof⟩

lemma *less-cSUP-iff*: $A \neq \{\} \implies \text{bdd-above } (f'A) \implies a < (\bigcup i \in A. f\ i) \longleftrightarrow (\exists x \in A. a < f\ x)$
 ⟨proof⟩

lemma *less-cSupE*:
 assumes $y < \text{Sup } X$ $X \neq \{\}$ obtains x where $x \in X$ $y < x$
 ⟨proof⟩

lemma *less-cSupD*:
 $X \neq \{\} \implies z < \text{Sup } X \implies \exists x \in X. z < x$
 ⟨proof⟩

lemma *cInf-lessD*:
 $X \neq \{\} \implies \text{Inf } X < z \implies \exists x \in X. x < z$
 ⟨proof⟩

lemma *complete-interval*:
 assumes $a < b$ and $P\ a$ and $\neg P\ b$
 shows $\exists c. a \leq c \wedge c \leq b \wedge (\forall x. a \leq x \wedge x < c \longrightarrow P\ x) \wedge$
 $(\forall d. (\forall x. a \leq x \wedge x < d \longrightarrow P\ x) \longrightarrow d \leq c)$
 ⟨proof⟩

end

91.5 Instances

instance *complete-linorder* < *conditionally-complete-linorder*
 ⟨proof⟩

lemma *cSup-eq-Max*: *finite* $(X::'a::\text{conditionally-complete-linorder set}) \implies X \neq \{\} \implies \text{Sup } X = \text{Max } X$
 ⟨proof⟩

lemma *cInf-eq-Min*: *finite* $(X::'a::\text{conditionally-complete-linorder set}) \implies X \neq \{\} \implies \text{Inf } X = \text{Min } X$
 ⟨proof⟩

lemma *cSup-lessThan[simp]*: $\text{Sup } \{.. $x::'a::\{\text{conditionally-complete-linorder, no-bot, dense-linorder}\}$ \} = x$
 ⟨proof⟩

lemma *cSup-greaterThanLessThan[simp]*: $y < x \implies \text{Sup } \{y <.. $x::'a::\{\text{conditionally-complete-linorder, no-bot, dense-linorder}\}$ \} = x$

$\text{dense-linorder}\} = x$
 $\langle \text{proof} \rangle$

lemma $cSup\text{-}atLeastLessThan[simp]$: $y < x \implies Sup \{y..x::'a::\{\text{conditionally-complete-linorder}, \text{dense-linorder}\}\} = x$
 $\langle \text{proof} \rangle$

lemma $cInf\text{-}greaterThan[simp]$: $Inf \{x::'a::\{\text{conditionally-complete-linorder}, \text{no-top}, \text{dense-linorder}\} <.. \} = x$
 $\langle \text{proof} \rangle$

lemma $cInf\text{-}greaterThanAtMost[simp]$: $y < x \implies Inf \{y<..x::'a::\{\text{conditionally-complete-linorder}, \text{dense-linorder}\}\} = y$
 $\langle \text{proof} \rangle$

lemma $cInf\text{-}greaterThanLessThan[simp]$: $y < x \implies Inf \{y<..<x::'a::\{\text{conditionally-complete-linorder}, \text{dense-linorder}\}\} = y$
 $\langle \text{proof} \rangle$

lemma $Sup\text{-}inverse\text{-}eq\text{-}inverse\text{-}Inf$:
fixes $f::'b \Rightarrow 'a::\{\text{conditionally-complete-linorder}, \text{linordered-field}\}$
assumes $bdd\text{-}above \text{ (range } f) \ L > 0$ **and** $geL: \bigwedge x. f\ x \geq L$
shows $(SUP\ x. 1 / f\ x) = 1 / (INF\ x. f\ x)$
 $\langle \text{proof} \rangle$

lemma $Inf\text{-}inverse\text{-}eq\text{-}inverse\text{-}Sup$:
fixes $f::'b \Rightarrow 'a::\{\text{conditionally-complete-linorder}, \text{linordered-field}\}$
assumes $bdd\text{-}above \text{ (range } f) \ L > 0$ **and** $geL: \bigwedge x. f\ x \geq L$
shows $(INF\ x. 1 / f\ x) = 1 / (SUP\ x. f\ x)$
 $\langle \text{proof} \rangle$

lemma $Inf\text{-}insert\text{-}finite$:
fixes $S :: 'a::\text{conditionally-complete-linorder set}$
shows $finite\ S \implies Inf\ (insert\ x\ S) = (\text{if } S = \{\} \text{ then } x \text{ else } min\ x\ (Inf\ S))$
 $\langle \text{proof} \rangle$

lemma $Sup\text{-}insert\text{-}finite$:
fixes $S :: 'a::\text{conditionally-complete-linorder set}$
shows $finite\ S \implies Sup\ (insert\ x\ S) = (\text{if } S = \{\} \text{ then } x \text{ else } max\ x\ (Sup\ S))$
 $\langle \text{proof} \rangle$

lemma $finite\text{-}imp\text{-}less\text{-}Inf$:
fixes $a :: 'a::\text{conditionally-complete-linorder}$
shows $\llbracket finite\ X; x \in X; \bigwedge x. x \in X \implies a < x \rrbracket \implies a < Inf\ X$
 $\langle \text{proof} \rangle$

lemma $finite\text{-}less\text{-}Inf\text{-}iff$:
fixes $a :: 'a :: \text{conditionally-complete-linorder}$
shows $\llbracket finite\ X; X \neq \{\} \rrbracket \implies a < Inf\ X \longleftrightarrow (\forall x \in X. a < x)$

$\langle proof \rangle$

lemma *finite-imp-Sup-less*:

fixes $a :: 'a :: conditionally-complete-linorder$

shows $\llbracket finite\ X; x \in X; \bigwedge x. x \in X \implies a > x \rrbracket \implies a > Sup\ X$

$\langle proof \rangle$

lemma *finite-Sup-less-iff*:

fixes $a :: 'a :: conditionally-complete-linorder$

shows $\llbracket finite\ X; X \neq \{\} \rrbracket \implies a > Sup\ X \longleftrightarrow (\forall x \in X. a > x)$

$\langle proof \rangle$

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*: $\exists b. a < b \vee b < a$

$\langle proof \rangle$

end

context

fixes $f :: 'a \Rightarrow 'b :: \{conditionally-complete-linorder, ordered-ab-group-add\}$

begin

lemma *bdd-above-uminus-image*: $bdd-above\ ((\lambda x. - f\ x)\ 'A) \longleftrightarrow bdd-below\ (f\ 'A)$

$\langle proof \rangle$

lemma *bdd-below-uminus-image*: $bdd-below\ ((\lambda x. - f\ x)\ 'A) \longleftrightarrow bdd-above\ (f\ 'A)$

$\langle proof \rangle$

lemma *uminus-cSUP*:

assumes $bdd-above\ (f\ 'A)\ A \neq \{\}$

shows $-(SUP\ x \in A. f\ x) = (INF\ x \in A. - f\ x)$

$\langle proof \rangle$

end

context

fixes $f :: 'a \Rightarrow 'b :: \{conditionally-complete-linorder, ordered-ab-group-add\}$

begin

lemma *uminus-cINF*:

assumes $bdd-below\ (f\ 'A)\ A \neq \{\}$

shows $-(INF\ x \in A. f\ x) = (SUP\ x \in A. - f\ x)$

$\langle proof \rangle$

lemma *Sup-add-eq*:

assumes *bdd-above* ($f \text{ ‘ } A$) $A \neq \{\}$

shows $(\text{SUP } x \in A. a + f x) = a + (\text{SUP } x \in A. f x)$ (**is** $?L=?R$)
 $\langle \text{proof} \rangle$

lemma *Inf-add-eq*: — you don’t get a shorter proof by duality

assumes *bdd-below* ($f \text{ ‘ } A$) $A \neq \{\}$

shows $(\text{INF } x \in A. a + f x) = a + (\text{INF } x \in A. f x)$ (**is** $?L=?R$)
 $\langle \text{proof} \rangle$

end

instantiation *nat* :: *conditionally-complete-linorder*
begin

definition *Sup* ($X::\text{nat set}$) = (*if* $X=\{\}$ *then* 0 *else* *Max* X)

definition *Inf* ($X::\text{nat set}$) = (*LEAST* $n. n \in X$)

lemma *bdd-above-nat*: *bdd-above* $X \longleftrightarrow \text{finite } (X::\text{nat set})$
 $\langle \text{proof} \rangle$

instance

$\langle \text{proof} \rangle$

end

lemma *Inf-nat-def1*:

fixes $K::\text{nat set}$

assumes $K \neq \{\}$

shows $\text{Inf } K \in K$

$\langle \text{proof} \rangle$

lemma *Sup-nat-empty* [*simp*]: $\text{Sup } \{\} = (0::\text{nat})$
 $\langle \text{proof} \rangle$

instantiation *int* :: *conditionally-complete-linorder*
begin

definition *Sup* ($X::\text{int set}$) = (*THE* $x. x \in X \wedge (\forall y \in X. y \leq x)$)

definition *Inf* ($X::\text{int set}$) = $- (\text{Sup } (\text{uminus ‘ } X))$

instance

$\langle \text{proof} \rangle$

end

lemma *interval-cases*:

fixes $S :: 'a :: \text{conditionally-complete-linorder set}$
assumes $ivl: \bigwedge a\ b\ x. a \in S \implies b \in S \implies a \leq x \implies x \leq b \implies x \in S$
shows $\exists a\ b. S = \{\} \vee$
 $S = UNIV \vee$
 $S = \{..<b\} \vee$
 $S = \{..b\} \vee$
 $S = \{a<..\} \vee$
 $S = \{a..\} \vee$
 $S = \{a<..**b\} \vee**$
 $S = \{a<..b\} \vee$
 $S = \{a..**b\} \vee**$
 $S = \{a..b\}$
 $\langle \text{proof} \rangle$

lemma $cSUP\text{-}eq\text{-}cINF\text{-}D$:
fixes $f :: - \Rightarrow 'b :: \text{conditionally-complete-lattice}$
assumes $eq: (\bigsqcup x \in A. f\ x) = (\bigcap x \in A. f\ x)$
and $bdd: bdd\text{-}above\ (f\ 'A)\ bdd\text{-}below\ (f\ 'A)$
and $a: a \in A$
shows $f\ a = (\bigcap x \in A. f\ x)$
 $\langle \text{proof} \rangle$

lemma $cSUP\text{-}UNION$:
fixes $f :: - \Rightarrow 'b :: \text{conditionally-complete-lattice}$
assumes $ne: A \neq \{\} \wedge x. x \in A \implies B(x) \neq \{\}$
and $bdd\text{-}UN: bdd\text{-}above\ (\bigcup x \in A. f\ 'B\ x)$
shows $(\bigsqcup z \in \bigcup x \in A. B\ x. f\ z) = (\bigsqcup x \in A. \bigsqcup z \in B\ x. f\ z)$
 $\langle \text{proof} \rangle$

lemma $cINF\text{-}UNION$:
fixes $f :: - \Rightarrow 'b :: \text{conditionally-complete-lattice}$
assumes $ne: A \neq \{\} \wedge x. x \in A \implies B(x) \neq \{\}$
and $bdd\text{-}UN: bdd\text{-}below\ (\bigcup x \in A. f\ 'B\ x)$
shows $(\bigcap z \in \bigcup x \in A. B\ x. f\ z) = (\bigcap x \in A. \bigcap z \in B\ x. f\ z)$
 $\langle \text{proof} \rangle$

lemma $cSup\text{-}abs\text{-}le$:
fixes $S :: ('a :: \{\text{linordered-idom}, \text{conditionally-complete-linorder}\})\ \text{set}$
shows $S \neq \{\} \implies (\bigwedge x. x \in S \implies |x| \leq a) \implies |Sup\ S| \leq a$
 $\langle \text{proof} \rangle$

end

92 Factorial Function, Rising Factorials

theory *Factorial*
imports *Groups-List*
begin

92.1 Factorial Function

context *semiring-char-0*

begin

definition *fact* :: *nat* \Rightarrow 'a

where *fact-prod*: *fact* *n* = *of-nat* ($\prod \{1..n\}$)

lemma *fact-prod-Suc*: *fact* *n* = *of-nat* (*prod* *Suc* {*0*..*n*})
 ⟨*proof*⟩

lemma *fact-prod-rev*: *fact* *n* = *of-nat* ($\prod i = 0..<n. n - i$)
 ⟨*proof*⟩

lemma *fact-0* [*simp*]: *fact* 0 = 1
 ⟨*proof*⟩

lemma *fact-1* [*simp*]: *fact* 1 = 1
 ⟨*proof*⟩

lemma *fact-Suc-0* [*simp*]: *fact* (*Suc* 0) = 1
 ⟨*proof*⟩

lemma *fact-Suc* [*simp*]: *fact* (*Suc* *n*) = *of-nat* (*Suc* *n*) * *fact* *n*
 ⟨*proof*⟩

lemma *fact-2* [*simp*]: *fact* 2 = 2
 ⟨*proof*⟩

lemma *fact-split*: $k \leq n \implies \text{fact } n = \text{of-nat } (\text{prod } \text{Suc } \{n - k..<n\}) * \text{fact } (n - k)$
 ⟨*proof*⟩

end

lemma *of-nat-fact* [*simp*]: *of-nat* (*fact* *n*) = *fact* *n*
 ⟨*proof*⟩

lemma *of-int-fact* [*simp*]: *of-int* (*fact* *n*) = *fact* *n*
 ⟨*proof*⟩

lemma *fact-reduce*: $n > 0 \implies \text{fact } n = \text{of-nat } n * \text{fact } (n - 1)$
 ⟨*proof*⟩

lemma *fact-nonzero* [*simp*]: *fact* *n* $\neq (0 :: 'a :: \{\text{semiring-char-0}, \text{semiring-no-zero-divisors}\})$
 ⟨*proof*⟩

lemma *fact-mono-nat*: $m \leq n \implies \text{fact } m \leq (\text{fact } n :: \text{nat})$
 ⟨*proof*⟩

lemma *fact-in-Nats*: $\text{fact } n \in \mathbf{N}$
 $\langle \text{proof} \rangle$

lemma *fact-in-Ints*: $\text{fact } n \in \mathbf{Z}$
 $\langle \text{proof} \rangle$

context
assumes *SORT-CONSTRAINT*('a::linordered-semidom)
begin

lemma *fact-mono*: $m \leq n \implies \text{fact } m \leq (\text{fact } n :: 'a)$
 $\langle \text{proof} \rangle$

lemma *fact-ge-1* [simp]: $\text{fact } n \geq (1 :: 'a)$
 $\langle \text{proof} \rangle$

lemma *fact-gt-zero* [simp]: $\text{fact } n > (0 :: 'a)$
 $\langle \text{proof} \rangle$

lemma *fact-ge-zero* [simp]: $\text{fact } n \geq (0 :: 'a)$
 $\langle \text{proof} \rangle$

lemma *fact-not-neg* [simp]: $\neg \text{fact } n < (0 :: 'a)$
 $\langle \text{proof} \rangle$

lemma *fact-le-power*: $\text{fact } n \leq (\text{of-nat } (n \hat{~} n) :: 'a)$
 $\langle \text{proof} \rangle$

end

lemma *fact-less-mono-nat*: $0 < m \implies m < n \implies \text{fact } m < (\text{fact } n :: \text{nat})$
 $\langle \text{proof} \rangle$

lemma *fact-less-mono*: $0 < m \implies m < n \implies \text{fact } m < (\text{fact } n :: 'a::\text{linordered-semidom})$
 $\langle \text{proof} \rangle$

lemma *fact-ge-Suc-0-nat* [simp]: $\text{fact } n \geq \text{Suc } 0$
 $\langle \text{proof} \rangle$

lemma *dvd-fact*: $1 \leq m \implies m \leq n \implies m \text{ dvd } \text{fact } n$
 $\langle \text{proof} \rangle$

lemma *fact-ge-self*: $\text{fact } n \geq n$
 $\langle \text{proof} \rangle$

lemma *fact-dvd*: $n \leq m \implies \text{fact } n \text{ dvd } (\text{fact } m :: 'a::\text{linordered-semidom})$
 $\langle \text{proof} \rangle$

lemma *fact-mod*: $m \leq n \implies \text{fact } n \text{ mod } (\text{fact } m :: 'a::\{\text{semidom-modulo}, \text{linordered-semidom}\})$

$= 0$
 $\langle \text{proof} \rangle$

lemma *fact-eq-fact-times*:
assumes $m \geq n$
shows $\text{fact } m = \text{fact } n * \prod \{\text{Suc } n..m\}$
 $\langle \text{proof} \rangle$

lemma *fact-div-fact*:
assumes $m \geq n$
shows $\text{fact } m \text{ div } \text{fact } n = \prod \{n + 1..m\}$
 $\langle \text{proof} \rangle$

lemma *fact-num-eq-if*: $\text{fact } m = (\text{if } m = 0 \text{ then } 1 \text{ else } \text{of-nat } m * \text{fact } (m - 1))$
 $\langle \text{proof} \rangle$

lemma *fact-div-fact-le-pow*:
assumes $r \leq n$
shows $\text{fact } n \text{ div } \text{fact } (n - r) \leq n \wedge r$
 $\langle \text{proof} \rangle$

lemma *prod-Suc-fact*: $\text{prod } \text{Suc } \{0..<n\} = \text{fact } n$
 $\langle \text{proof} \rangle$

lemma *prod-Suc-Suc-fact*: $\text{prod } \text{Suc } \{\text{Suc } 0..<n\} = \text{fact } n$
 $\langle \text{proof} \rangle$

lemma *fact-numeral*: $\text{fact } (\text{numeral } k) = \text{numeral } k * \text{fact } (\text{pred-numeral } k)$
— Evaluation for specific numerals
 $\langle \text{proof} \rangle$

92.2 Pochhammer’s symbol: generalized rising factorial

See https://en.wikipedia.org/wiki/Pochhammer_symbol.

context *comm-semiring-1*
begin

definition *pochhammer* :: $'a \Rightarrow \text{nat} \Rightarrow 'a$
where *pochhammer-prod*: $\text{pochhammer } a \ n = \text{prod } (\lambda i. a + \text{of-nat } i) \ \{0..<n\}$

lemma *pochhammer-prod-rev*: $\text{pochhammer } a \ n = \text{prod } (\lambda i. a + \text{of-nat } (n - i)) \ \{1..n\}$
 $\langle \text{proof} \rangle$

lemma *pochhammer-Suc-prod*: $\text{pochhammer } a \ (\text{Suc } n) = \text{prod } (\lambda i. a + \text{of-nat } i) \ \{0..n\}$
 $\langle \text{proof} \rangle$

lemma *pochhammer-Suc-prod-rev*: $\text{pochhammer } a \ (\text{Suc } n) = \text{prod } (\lambda i. a + \text{of-nat } (n - i)) \ \{0..n\}$

$(n - i)) \{0..n\}$
 $\langle \text{proof} \rangle$

lemma *pochhammer-0* [*simp*]: *pochhammer* *a* 0 = 1
 $\langle \text{proof} \rangle$

lemma *pochhammer-1* [*simp*]: *pochhammer* *a* 1 = *a*
 $\langle \text{proof} \rangle$

lemma *pochhammer-Suc0* [*simp*]: *pochhammer* *a* (Suc 0) = *a*
 $\langle \text{proof} \rangle$

lemma *pochhammer-Suc*: *pochhammer* *a* (Suc *n*) = *pochhammer* *a* *n* * (*a* + *of-nat* *n*)
 $\langle \text{proof} \rangle$

end

lemma *pochhammer-nonneg*:
fixes *x* :: 'a :: *linordered-semidom*
shows $x > 0 \implies \text{pochhammer } x \ n \geq 0$
 $\langle \text{proof} \rangle$

lemma *pochhammer-pos*:
fixes *x* :: 'a :: *linordered-semidom*
shows $x > 0 \implies \text{pochhammer } x \ n > 0$
 $\langle \text{proof} \rangle$

context *comm-semiring-1*
begin

lemma *pochhammer-of-nat*: *pochhammer* (*of-nat* *x*) *n* = *of-nat* (*pochhammer* *x* *n*)
 $\langle \text{proof} \rangle$

end

context *comm-ring-1*
begin

lemma *pochhammer-of-int*: *pochhammer* (*of-int* *x*) *n* = *of-int* (*pochhammer* *x* *n*)
 $\langle \text{proof} \rangle$

end

lemma *pochhammer-rec*: *pochhammer* *a* (Suc *n*) = *a* * *pochhammer* (*a* + 1) *n*
 $\langle \text{proof} \rangle$

lemma *pochhammer-rec'*: *pochhammer* *z* (Suc *n*) = (*z* + *of-nat* *n*) * *pochhammer* *z* *n*

$\langle \text{proof} \rangle$

lemma *pochhammer-fact*: $\text{fact } n = \text{pochhammer } 1 \ n$
 $\langle \text{proof} \rangle$

lemma *pochhammer-of-nat-eq-0-lemma*: $k > n \implies \text{pochhammer } (- \ (\text{of-nat } n :: 'a::\text{idom})) \ k = 0$
 $\langle \text{proof} \rangle$

lemma *pochhammer-of-nat-eq-0-lemma'*:
assumes $kn: k \leq n$
shows $\text{pochhammer } (- \ (\text{of-nat } n :: 'a::\{\text{idom}, \text{ring-char-0}\})) \ k \neq 0$
 $\langle \text{proof} \rangle$

lemma *pochhammer-of-nat-eq-0-iff*:
 $\text{pochhammer } (- \ (\text{of-nat } n :: 'a::\{\text{idom}, \text{ring-char-0}\})) \ k = 0 \longleftrightarrow k > n$
(is ?l = ?r)
 $\langle \text{proof} \rangle$

lemma *pochhammer-0-left*:
 $\text{pochhammer } 0 \ n = (\text{if } n = 0 \text{ then } 1 \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *pochhammer-eq-0-iff*: $\text{pochhammer } a \ n = (0::'a::\text{field-char-0}) \longleftrightarrow (\exists k < n. a = - \ \text{of-nat } k)$
 $\langle \text{proof} \rangle$

lemma *pochhammer-eq-0-mono*:
 $\text{pochhammer } a \ n = (0::'a::\text{field-char-0}) \implies m \geq n \implies \text{pochhammer } a \ m = 0$
 $\langle \text{proof} \rangle$

lemma *pochhammer-neq-0-mono*:
 $\text{pochhammer } a \ m \neq (0::'a::\text{field-char-0}) \implies m \geq n \implies \text{pochhammer } a \ n \neq 0$
 $\langle \text{proof} \rangle$

lemma *pochhammer-minus*:
 $\text{pochhammer } (- \ b) \ k = ((- \ 1) \wedge k :: 'a::\text{comm-ring-1}) * \text{pochhammer } (b - \ \text{of-nat } k + 1) \ k$
 $\langle \text{proof} \rangle$

lemma *pochhammer-minus'*:
 $\text{pochhammer } (b - \ \text{of-nat } k + 1) \ k = ((- \ 1) \wedge k :: 'a::\text{comm-ring-1}) * \text{pochhammer } (- \ b) \ k$
 $\langle \text{proof} \rangle$

lemma *pochhammer-same*: $\text{pochhammer } (- \ \text{of-nat } n) \ n = ((- \ 1) \wedge n :: 'a::\{\text{semiring-char-0}, \text{comm-ring-1}, \text{semiring-no-zero-divisors}\}) * \text{fact } n$
 $\langle \text{proof} \rangle$

lemma *pochhammer-product'*: $\text{pochhammer } z \ (n + m) = \text{pochhammer } z \ n * \text{pochhammer } (z + \text{of-nat } n) \ m$
 $\langle \text{proof} \rangle$

lemma *pochhammer-product*:
 $m \leq n \implies \text{pochhammer } z \ n = \text{pochhammer } z \ m * \text{pochhammer } (z + \text{of-nat } m) \ (n - m)$
 $\langle \text{proof} \rangle$

lemma *pochhammer-times-pochhammer-half*:
fixes $z :: 'a::\text{field-char-0}$
shows $\text{pochhammer } z \ (\text{Suc } n) * \text{pochhammer } (z + 1/2) \ (\text{Suc } n) = (\prod_{k=0..2*n+1} z + \text{of-nat } k / 2)$
 $\langle \text{proof} \rangle$

lemma *pochhammer-double*:
fixes $z :: 'a::\text{field-char-0}$
shows $\text{pochhammer } (2 * z) \ (2 * n) = \text{of-nat } (2^{(2*n)}) * \text{pochhammer } z \ n * \text{pochhammer } (z + 1/2) \ n$
 $\langle \text{proof} \rangle$

lemma *fact-double*:
 $\text{fact } (2 * n) = (2^{(2 * n)} * \text{pochhammer } (1 / 2) \ n * \text{fact } n :: 'a::\text{field-char-0})$
 $\langle \text{proof} \rangle$

lemma *pochhammer-absorb-comp*: $(r - \text{of-nat } k) * \text{pochhammer } (-r) \ k = r * \text{pochhammer } (-r + 1) \ k$
(is ?lhs = ?rhs)
for $r :: 'a::\text{comm-ring-1}$
 $\langle \text{proof} \rangle$

92.3 Misc

lemma *fact-code* [code]:
 $\text{fact } n = (\text{of-nat } (\text{fold-atLeastAtMost-nat } ((*)) \ 2 \ n \ 1) :: 'a::\text{semiring-char-0})$
 $\langle \text{proof} \rangle$

lemma *pochhammer-code* [code]:
 $\text{pochhammer } a \ n =$
 $(\text{if } n = 0 \text{ then } 1$
 $\text{else } \text{fold-atLeastAtMost-nat } (\lambda n \text{ acc. } (a + \text{of-nat } n) * \text{acc}) \ 0 \ (n - 1) \ 1)$
 $\langle \text{proof} \rangle$

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 Kam-mueller.

Combinatorial definition

```
definition binomial :: nat ⇒ nat ⇒ nat
  where binomial n k = card {K ∈ Pow {0..<n}. card K = k}
```

```
open-bundle binomial-syntax
begin
notation binomial (infix ‹choose› 64)
end
```

```
lemma binomial-right-mono:
  assumes m ≤ n shows m choose k ≤ n choose k
  ‹proof›
```

```
theorem n-subsets:
  assumes finite A
  shows card {B. B ⊆ A ∧ card B = k} = card A choose k
  ‹proof›
```

Recursive characterization

```
lemma binomial-n-0 [simp]: n choose 0 = 1
  ‹proof›
```

```
lemma binomial-0-Suc [simp]: 0 choose Suc k = 0
  ‹proof›
```

```
lemma binomial-Suc-Suc [simp]: Suc n choose Suc k = (n choose k) + (n choose
  Suc k)
  ‹proof›
```

```
lemma binomial-eq-0: n < k ⇒ n choose k = 0
  ‹proof›
```

```
lemma zero-less-binomial: k ≤ n ⇒ n choose k > 0
  ‹proof›
```

```
lemma binomial-eq-0-iff [simp]: n choose k = 0 ⇔ n < k
```

$\langle \text{proof} \rangle$

lemma *zero-less-binomial-iff* [simp]: $n \text{ choose } k > 0 \iff k \leq n$
 $\langle \text{proof} \rangle$

lemma *binomial-n-n* [simp]: $n \text{ choose } n = 1$
 $\langle \text{proof} \rangle$

lemma *binomial-Suc-n* [simp]: $\text{Suc } n \text{ choose } n = \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *binomial-1* [simp]: $n \text{ choose } \text{Suc } 0 = n$
 $\langle \text{proof} \rangle$

lemma *choose-one*: $n \text{ choose } 1 = n$ **for** $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *choose-reduce-nat*:
 $0 < n \implies 0 < k \implies$
 $n \text{ choose } k = ((n - 1) \text{ choose } (k - 1)) + ((n - 1) \text{ choose } k)$
 $\langle \text{proof} \rangle$

lemma *Suc-times-binomial-eq*: $\text{Suc } n * (n \text{ choose } k) = (\text{Suc } n \text{ choose } \text{Suc } k) * \text{Suc } k$
 $\langle \text{proof} \rangle$

lemma *binomial-le-pow2*: $n \text{ choose } k \leq 2^n$
 $\langle \text{proof} \rangle$

The absorption property.

lemma *Suc-times-binomial*: $\text{Suc } k * (\text{Suc } n \text{ choose } \text{Suc } k) = \text{Suc } n * (n \text{ choose } k)$
 $\langle \text{proof} \rangle$

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*: $(\text{Suc } n \text{ choose } \text{Suc } k) = (\text{Suc } n * (n \text{ choose } k)) \text{ div } \text{Suc } k$
 $\langle \text{proof} \rangle$

Another version of absorption, with -1 instead of Suc .

lemma *times-binomial-minus1-eq*: $0 < k \implies k * (n \text{ choose } k) = n * ((n - 1) \text{ choose } (k - 1))$
 $\langle \text{proof} \rangle$

93.2 The binomial theorem (courtesy of Tobias Nipkow):

Avigad’s version, generalized to any commutative ring

theorem (in *comm-semiring-1*) *binomial-ring*:

$(a + b :: 'a)^\wedge n = (\sum k \leq n. (of\text{-}nat\ (n\ choose\ k)) * a^\wedge k * b^\wedge (n-k))$
 $\langle proof \rangle$

Original version for the naturals.

corollary *binomial*: $(a + b :: nat)^\wedge n = (\sum k \leq n. (of\text{-}nat\ (n\ choose\ k)) * a^\wedge k * b^\wedge (n - k))$
 $\langle proof \rangle$

lemma *binomial-fact-lemma*: $k \leq n \implies fact\ k * fact\ (n - k) * (n\ choose\ k) = fact\ n$
 $\langle proof \rangle$

lemma *binomial-fact'*:
assumes $k \leq n$
shows $n\ choose\ k = fact\ n\ div\ (fact\ k * fact\ (n - k))$
 $\langle proof \rangle$

lemma *binomial-fact*:
assumes $kn: k \leq n$
shows $(of\text{-}nat\ (n\ choose\ k) :: 'a::field\text{-}char\ 0) = fact\ n / (fact\ k * fact\ (n - k))$
 $\langle proof \rangle$

lemma *fact-binomial*:
assumes $k \leq n$
shows $fact\ k * of\text{-}nat\ (n\ choose\ k) = (fact\ n / fact\ (n - k) :: 'a::field\text{-}char\ 0)$
 $\langle proof \rangle$

lemma *binomial-fact-pow*: $(n\ choose\ s) * fact\ s \leq n^\wedge s$
 $\langle proof \rangle$

lemma *choose-two*: $n\ choose\ 2 = n * (n - 1) div\ 2$
 $\langle proof \rangle$

lemma *choose-row-sum*: $(\sum k \leq n. n\ choose\ k) = 2^\wedge n$
 $\langle proof \rangle$

lemma *sum-choose-lower*: $(\sum k \leq n. (r+k)\ choose\ k) = Suc\ (r+n)\ choose\ n$
 $\langle proof \rangle$

lemma *sum-choose-upper*: $(\sum k \leq n. k\ choose\ m) = Suc\ n\ choose\ Suc\ m$
 $\langle proof \rangle$

lemma *choose-alternating-sum*:
 $n > 0 \implies (\sum i \leq n. (-1)^\wedge i * of\text{-}nat\ (n\ choose\ i)) = (0 :: 'a::comm\text{-}ring\ 1)$
 $\langle proof \rangle$

lemma *choose-even-sum*:
assumes $n > 0$
shows $2 * (\sum i \leq n. if\ even\ i\ then\ of\text{-}nat\ (n\ choose\ i)\ else\ 0) = (2^\wedge n ::$

'a::comm-ring-1)
 ⟨proof⟩

lemma choose-odd-sum:

assumes $n > 0$
 shows $2 * (\sum_{i \leq n. \text{if odd } i \text{ then of-nat } (n \text{ choose } i) \text{ else } 0}) = (2 \wedge n ::$
 'a::comm-ring-1)
 ⟨proof⟩

NW diagonal sum property

lemma sum-choose-diagonal:

assumes $m \leq n$
 shows $(\sum_{k \leq m. (n - k) \text{ choose } (m - k)}) = \text{Suc } n \text{ choose } m$
 ⟨proof⟩

93.3 Generalized binomial coefficients

definition gbinomial :: 'a::{semidom-divide,semiring-char-0} \Rightarrow nat \Rightarrow 'a (infix
 ⟨gchoose⟩ 64)

where gbinomial-prod-rev: $a \text{ gchoose } k = (\prod_{i=0..<k. a - \text{of-nat } i}) \text{ div fact } k$

lemma gbinomial-0 [simp]:

$a \text{ gchoose } 0 = 1$
 $0 \text{ gchoose } (\text{Suc } k) = 0$
 ⟨proof⟩

lemma gbinomial-Suc: $a \text{ gchoose } (\text{Suc } k) = \text{prod } (\lambda i. a - \text{of-nat } i) \{0..k\} \text{ div fact } (\text{Suc } k)$

⟨proof⟩

lemma gbinomial-1 [simp]: $a \text{ gchoose } 1 = a$

⟨proof⟩

lemma gbinomial-Suc0 [simp]: $a \text{ gchoose } \text{Suc } 0 = a$

⟨proof⟩

lemma gbinomial-0-left: $0 \text{ gchoose } k = (\text{if } k = 0 \text{ then } 1 \text{ else } 0)$

⟨proof⟩

lemma gbinomial-mult-fact: $\text{fact } k * (a \text{ gchoose } k) = (\prod_{i=0..<k. a - \text{of-nat } i)$

for $a :: 'a::\text{field-char-0}$

⟨proof⟩

lemma gbinomial-mult-fact': $(a \text{ gchoose } k) * \text{fact } k = (\prod_{i=0..<k. a - \text{of-nat } i)$

for $a :: 'a::\text{field-char-0}$

⟨proof⟩

lemma gbinomial-pochhammer: $a \text{ gchoose } k = (-1) \wedge k * \text{pochhammer } (-a) k / \text{fact } k$

for $a :: 'a::\text{field-char-0}$
 $\langle \text{proof} \rangle$

lemma *gbinomial-pochhammer'*: $a \text{ gchoose } k = \text{pochhammer } (a - \text{of-nat } k + 1) \ k$
 $/ \text{ fact } k$
for $a :: 'a::\text{field-char-0}$
 $\langle \text{proof} \rangle$

lemma *gbinomial-binomial*: $n \text{ gchoose } k = n \text{ choose } k$
 $\langle \text{proof} \rangle$

lemma *of-nat-gbinomial*: $\text{of-nat } (n \text{ gchoose } k) = (\text{of-nat } n \text{ gchoose } k :: 'a::\text{field-char-0})$
 $\langle \text{proof} \rangle$

lemma *binomial-gbinomial*: $\text{of-nat } (n \text{ choose } k) = (\text{of-nat } n \text{ gchoose } k :: 'a::\text{field-char-0})$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *gbinomial-mult-1*:
fixes $a :: 'a::\text{field-char-0}$
shows $a * (a \text{ gchoose } k) = \text{of-nat } k * (a \text{ gchoose } k) + \text{of-nat } (\text{Suc } k) * (a \text{ gchoose } (\text{Suc } k))$
(is ?l = ?r)
 $\langle \text{proof} \rangle$

lemma *gbinomial-mult-1'*:
 $(a \text{ gchoose } k) * a = \text{of-nat } k * (a \text{ gchoose } k) + \text{of-nat } (\text{Suc } k) * (a \text{ gchoose } (\text{Suc } k))$
for $a :: 'a::\text{field-char-0}$
 $\langle \text{proof} \rangle$

lemma *gbinomial-Suc-Suc*: $(a + 1) \text{ gchoose } (\text{Suc } k) = (a \text{ gchoose } k) + (a \text{ gchoose } (\text{Suc } k))$
for $a :: 'a::\text{field-char-0}$
 $\langle \text{proof} \rangle$

lemma *gbinomial-reduce-nat*: $0 < k \implies a \text{ gchoose } k = (a-1 \text{ gchoose } k-1) + (a-1 \text{ gchoose } k)$
for $a :: 'a::\text{field-char-0}$
 $\langle \text{proof} \rangle$

lemma *gchoose-row-sum-weighted*:
 $(\sum k = 0..m. (r \text{ gchoose } k) * (r/2 - \text{of-nat } k)) = \text{of-nat } (\text{Suc } m) / 2 * (r \text{ gchoose } (\text{Suc } m))$
for $r :: 'a::\text{field-char-0}$
 $\langle \text{proof} \rangle$

lemma *binomial-symmetric*:

assumes $kn: k \leq n$
shows $n \text{ choose } k = n \text{ choose } (n - k)$
 $\langle \text{proof} \rangle$

lemma *choose-rising-sum*:
 $(\sum_{j \leq m}. ((n + j) \text{ choose } n)) = ((n + m + 1) \text{ choose } (n + 1))$
 $(\sum_{j \leq m}. ((n + j) \text{ choose } n)) = ((n + m + 1) \text{ choose } m)$
 $\langle \text{proof} \rangle$

lemma *choose-linear-sum*: $(\sum_{i \leq n}. i * (n \text{ choose } i)) = n * 2^{(n - 1)}$
 $\langle \text{proof} \rangle$

lemma *choose-alternating-linear-sum*:
assumes $n \neq 1$
shows $(\sum_{i \leq n}. (-1)^i * \text{of-nat } i * \text{of-nat } (n \text{ choose } i) :: 'a::\text{comm-ring-1}) = 0$
 $\langle \text{proof} \rangle$

lemma *vandermonde*: $(\sum_{k \leq r}. (m \text{ choose } k) * (n \text{ choose } (r - k))) = (m + n \text{ choose } r)$
 $\langle \text{proof} \rangle$

lemma *choose-square-sum*: $(\sum_{k \leq n}. (n \text{ choose } k)^2) = ((2 * n) \text{ choose } n)$
 $\langle \text{proof} \rangle$

lemma *pochhammer-binomial-sum*:
fixes $a \ b :: 'a::\text{comm-ring-1}$
shows $\text{pochhammer } (a + b) \ n = (\sum_{k \leq n}. \text{of-nat } (n \text{ choose } k) * \text{pochhammer } a \ k * \text{pochhammer } b \ (n - k))$
 $\langle \text{proof} \rangle$

Contributed by Manuel Eberl, generalised by LCP. Alternative definition of the binomial coefficient as $\prod_{i < k}. (n - i) / (k - i)$.

lemma *gbinomial-altdef-of-nat*: $a \text{ gchoose } k = (\prod_{i = 0..<k}. (a - \text{of-nat } i) / \text{of-nat } (k - i) :: 'a)$
for $k :: \text{nat}$ **and** $a :: 'a::\text{field-char-0}$
 $\langle \text{proof} \rangle$

lemma *gbinomial-ge-n-over-k-pow-k*:
fixes $k :: \text{nat}$
and $a :: 'a::\text{linordered-field}$
assumes $\text{of-nat } k \leq a$
shows $(a / \text{of-nat } k :: 'a)^k \leq a \text{ gchoose } k$
 $\langle \text{proof} \rangle$

lemma *gbinomial-negated-upper*: $(a \text{ gchoose } k) = (-1)^k * ((\text{of-nat } k - a - 1) \text{ gchoose } k)$
 $\langle \text{proof} \rangle$

lemma *gbinomial-minus*: $((-a) \text{ gchoose } k) = (-1)^k * ((a + \text{of-nat } k - 1) \text{ gchoose } k)$

gchoose k
 ⟨proof⟩

lemma *Suc-times-gbinomial*: $\text{of-nat } (\text{Suc } k) * ((a + 1) \text{ gchoose } (\text{Suc } k)) = (a + 1) * (a \text{ gchoose } k)$
 ⟨proof⟩

lemma *gbinomial-factors*: $((a + 1) \text{ gchoose } (\text{Suc } k)) = (a + 1) / \text{of-nat } (\text{Suc } k) * (a \text{ gchoose } k)$
 ⟨proof⟩

lemma *gbinomial-rec*: $((a + 1) \text{ gchoose } (\text{Suc } k)) = (a \text{ gchoose } k) * ((a + 1) / \text{of-nat } (\text{Suc } k))$
 ⟨proof⟩

lemma *gbinomial-of-nat-symmetric*: $k \leq n \implies (\text{of-nat } n) \text{ gchoose } k = (\text{of-nat } n) \text{ gchoose } (n - k)$
 ⟨proof⟩

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'*: $k > 0 \implies a \text{ gchoose } k = (a / \text{of-nat } k) * (a - 1 \text{ gchoose } (k - 1))$
 ⟨proof⟩

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]:

$$k \binom{r}{k} = r \binom{r-1}{k-1}, \quad \text{integer } k.$$

lemma *gbinomial-absorption*: $\text{of-nat } (\text{Suc } k) * (a \text{ gchoose } \text{Suc } k) = a * ((a - 1) \text{ gchoose } k)$
 ⟨proof⟩

The absorption identity for natural number binomial coefficients:

lemma *binomial-absorption*: $\text{Suc } k * (n \text{ choose } \text{Suc } k) = n * ((n - 1) \text{ choose } k)$
 ⟨proof⟩

The absorption companion identity for natural number coefficients, following the proof by GKP [3, p. 157]:

lemma *binomial-absorb-comp*: $(n - k) * (n \text{ choose } k) = n * ((n - 1) \text{ choose } k)$
 (is ?lhs = ?rhs)
 ⟨proof⟩

The generalised absorption companion identity:

lemma *gbinomial-absorb-comp*: $(a - \text{of-nat } k) * (a \text{ gchoose } k) = a * ((a - 1) \text{ gchoose } k)$
 $\langle \text{proof} \rangle$

lemma *gbinomial-addition-formula*:
 $a \text{ gchoose } (\text{Suc } k) = ((a - 1) \text{ gchoose } (\text{Suc } k)) + ((a - 1) \text{ gchoose } k)$
 $\langle \text{proof} \rangle$

lemma *binomial-addition-formula*:
 $0 < n \implies n \text{ choose } (\text{Suc } k) = ((n - 1) \text{ choose } (\text{Suc } k)) + ((n - 1) \text{ choose } k)$
 $\langle \text{proof} \rangle$

Equation 5.9 of the reference material [3, p. 159] is a useful summation formula, operating on both indices:

$$\sum_{k \leq n} \binom{r+k}{k} = \binom{r+n+1}{n}, \quad \text{integer } n.$$

lemma *gbinomial-parallel-sum*: $(\sum k \leq n. (a + \text{of-nat } k) \text{ gchoose } k) = (a + \text{of-nat } n + 1) \text{ gchoose } n$
 $\langle \text{proof} \rangle$

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*:

$$\sum_{0 \leq k \leq n} \binom{k}{m} = \binom{n+1}{m+1}, \quad \text{integers } m, n \geq 0.$$

lemma *gbinomial-sum-up-index*:
 $(\sum j = 0..n. (\text{of-nat } j \text{ gchoose } k) :: 'a::\text{field-char-0}) = (\text{of-nat } n + 1) \text{ gchoose } (k + 1)$
 $\langle \text{proof} \rangle$

lemma *gbinomial-index-swap*:
 $((-1) \wedge k) * ((- (\text{of-nat } n) - 1) \text{ gchoose } k) = ((-1) \wedge n) * ((- (\text{of-nat } k) - 1) \text{ gchoose } n)$
 $(\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma *gbinomial-sum-lower-neg*: $(\sum k \leq m. (a \text{ gchoose } k) * (-1) \wedge k) = (-1) \wedge m * (a - 1 \text{ gchoose } m)$
 $(\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma *gbinomial-partial-row-sum*:

$(\sum k \leq m. (a \text{ gchoose } k) * ((a / 2) - \text{of-nat } k)) = ((\text{of-nat } m + 1) / 2) * (a \text{ gchoose } (m + 1))$
 <proof>

lemma *sum-bounds-lt-plus1*: $(\sum k < m. f (\text{Suc } k)) = (\sum k = 1 .. m. f k)$
 <proof>

lemma *gbinomial-partial-sum-poly*:
 $(\sum k \leq m. (\text{of-nat } m + a \text{ gchoose } k) * x^k * y^{(m-k)}) =$
 $(\sum k \leq m. (-a \text{ gchoose } k) * (-x)^k * (x + y)^{(m-k)})$
 (is ?lhs m = ?rhs m)
 <proof>

lemma *gbinomial-partial-sum-poly-xpos*:
 $(\sum k \leq m. (\text{of-nat } m + a \text{ gchoose } k) * x^k * y^{(m-k)}) =$
 $(\sum k \leq m. (\text{of-nat } k + a - 1 \text{ gchoose } k) * x^k * (x + y)^{(m-k)})$ (is ?lhs =
 ?rhs)
 <proof>

lemma *binomial-r-part-sum*: $(\sum k \leq m. (2 * m + 1 \text{ choose } k)) = 2^{\wedge} (2 * m)$
 <proof>

lemma *gbinomial-r-part-sum*: $(\sum k \leq m. (2 * (\text{of-nat } m) + 1 \text{ gchoose } k)) = 2^{\wedge} (2 * m)$
 (is ?lhs = ?rhs)
 <proof>

lemma *gbinomial-sum-nat-pow2*:
 $(\sum k \leq m. (\text{of-nat } (m + k) \text{ gchoose } k :: 'a::\text{field-char-0}) / 2^{\wedge} k) = 2^{\wedge} m$
 (is ?lhs = ?rhs)
 <proof>

lemma *gbinomial-trinomial-revision*:
 assumes $k \leq m$
 shows $(a \text{ gchoose } m) * (\text{of-nat } m \text{ gchoose } k) = (a \text{ gchoose } k) * (a - \text{of-nat } k \text{ gchoose } (m - k))$
 <proof>

Versions of the theorems above for the natural-number version of "choose"

lemma *binomial-altdef-of-nat*:
 $k \leq n \implies \text{of-nat } (n \text{ choose } k) = (\prod i = 0 .. < k. \text{of-nat } (n - i) / \text{of-nat } (k - i))$
 $:: 'a)$
 for $n k :: \text{nat}$ and $x :: 'a::\text{field-char-0}$
 <proof>

lemma *binomial-ge-n-over-k-pow-k*: $k \leq n \implies (\text{of-nat } n / \text{of-nat } k :: 'a) ^{\wedge} k \leq \text{of-nat } (n \text{ choose } k)$
 for $k n :: \text{nat}$ and $x :: 'a::\text{linordered-field}$
 <proof>

lemma *binomial-le-pow*:
assumes $r \leq n$
shows $n \text{ choose } r \leq n \wedge r$
 $\langle \text{proof} \rangle$

lemma *choose-dvd*:
assumes $k \leq n$ **shows** $\text{fact } k * \text{fact } (n - k) \text{ dvd } (\text{fact } n)$
 $\langle \text{proof} \rangle$

lemma *fact-fact-dvd-fact*:
 $\text{fact } k * \text{fact } n \text{ dvd } (\text{fact } (k + n))$
 $\langle \text{proof} \rangle$

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 = -)
 $\langle \text{proof} \rangle$

The "Subset of a Subset" identity.

lemma *choose-mult*:
 $k \leq m \implies m \leq n \implies (n \text{ choose } m) * (m \text{ choose } k) = (n \text{ choose } k) * ((n - k) \text{ choose } (m - k))$
 $\langle \text{proof} \rangle$

lemma *of-nat-binomial-eq-mult-binomial-Suc*:
assumes $k \leq n$
shows $(\text{of-nat} :: (\text{nat} \Rightarrow ('a :: \text{field-char-0}))) (n \text{ choose } k) = \text{of-nat } (n + 1 - k)$
 $/ \text{of-nat } (n + 1) * \text{of-nat } (\text{Suc } n \text{ choose } k)$
 $\langle \text{proof} \rangle$

93.5 More on Binomial Coefficients

The number of nat lists of length m summing to N is $N + m - 1 \text{ choose } N$:

lemma *card-length-sum-list-rec*:
assumes $m \geq 1$
shows $\text{card } \{l :: \text{nat list. length } l = m \wedge \text{sum-list } l = N\} =$
 $\text{card } \{l. \text{length } l = (m - 1) \wedge \text{sum-list } l = N\} +$
 $\text{card } \{l. \text{length } l = m \wedge \text{sum-list } l + 1 = N\}$
(is card ?C = card ?A + card ?B)
 $\langle \text{proof} \rangle$

lemma *card-length-sum-list*: $\text{card } \{l :: \text{nat list. size } l = m \wedge \text{sum-list } l = N\} = (N + m - 1) \text{ choose } N$
— by Holden Lee, tidied by Tobias Nipkow
 $\langle \text{proof} \rangle$

lemma *card-disjoint-shuffles*:

assumes *set xs* \cap *set ys* = {}

shows *card (shuffles xs ys)* = (*length xs* + *length ys*) *choose length xs*

<proof>

lemma *Suc-times-binomial-add*: *Suc a* * (*Suc (a + b)* *choose Suc a*) = *Suc b* * (*Suc (a + b)* *choose a*)

— by Lukas Bulwahn

<proof>

93.6 Inclusion-exclusion principle

Ported from HOL Light by lcp

lemma *Inter-over-Union*:

$\bigcap \{ \bigcup (\mathcal{F} \ x) \mid x. x \in S \} = \bigcup \{ \bigcap (G \ ' S) \mid G. \forall x \in S. G \ x \in \mathcal{F} \ x \}$

<proof>

lemma *subset-insert-lemma*:

$\{ T. T \subseteq (\text{insert } a \ S) \wedge P \ T \} = \{ T. T \subseteq S \wedge P \ T \} \cup \{ \text{insert } a \ T \mid T. T \subseteq S \wedge P(\text{insert } a \ T) \}$ (is ?L=?R)

<proof>

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 \Rightarrow bool **and** *f* :: 'a set \Rightarrow 'b::ring-1

assumes *disj-add*: $\llbracket P \ S; P \ T; \text{disjnt } S \ T \rrbracket \Longrightarrow f(S \cup T) = f \ S + f \ T$

and *empty*: $P \ \{\}$

and *Int*: $\llbracket P \ S; P \ T \rrbracket \Longrightarrow P(S \cap T)$

and *Un*: $\llbracket P \ S; P \ T \rrbracket \Longrightarrow P(S \cup T)$

and *Diff*: $\llbracket P \ S; P \ T \rrbracket \Longrightarrow P(S - T)$

begin

lemma *f-empty [simp]*: $f \ \{\} = 0$

<proof>

lemma *f-Un-Int*: $\llbracket P \ S; P \ T \rrbracket \Longrightarrow f(S \cup T) + f(S \cap T) = f \ S + f \ T$

<proof>

lemma *restricted-indexed*:

assumes *finite A* **and** *X*: $\bigwedge a. a \in A \Longrightarrow P(X \ a)$

shows $f(\bigcup (X \ ' A)) = (\sum B \mid B \subseteq A \wedge B \neq \{\}). (- \ 1) \wedge (\text{card } B + 1) * f(\bigcap (X \ ' B))$

<proof>

lemma *restricted*:

assumes *finite A* $\bigwedge a. a \in A \implies P\ a$

shows $f(\bigcup A) = (\sum B \mid B \subseteq A \wedge B \neq \{\}. (-1)^\wedge(\text{card } B + 1) * f(\bigcap B))$
 $\langle \text{proof} \rangle$

end

93.7 Versions for unrestrictedly additive functions

lemma *Incl-Excl-UN*:

fixes $f :: 'a \text{ set} \Rightarrow 'b::\text{ring-1}$

assumes $\bigwedge S\ T. \text{disjnt } S\ T \implies f(S \cup T) = f\ S + f\ T$ *finite A*

shows $f(\bigcup (G \text{ ' } A)) = (\sum B \mid B \subseteq A \wedge B \neq \{\}. (-1)^\wedge(\text{card } B + 1) * f(\bigcap (G \text{ ' } B)))$
 $\langle \text{proof} \rangle$

lemma *Incl-Excl-Union*:

fixes $f :: 'a \text{ set} \Rightarrow 'b::\text{ring-1}$

assumes $\bigwedge S\ T. \text{disjnt } S\ T \implies f(S \cup T) = f\ S + f\ T$ *finite A*

shows $f(\bigcup A) = (\sum B \mid B \subseteq A \wedge B \neq \{\}. (-1)^\wedge(\text{card } B + 1) * f(\bigcap B))$
 $\langle \text{proof} \rangle$

The famous inclusion-exclusion formula for the cardinality of a union

lemma *int-card-UNION*:

assumes *finite A* $\bigwedge K. K \in A \implies \text{finite } K$

shows $\text{int } (\text{card } (\bigcup A)) = (\sum I \mid I \subseteq A \wedge I \neq \{\}. (-1)^\wedge(\text{card } I + 1) * \text{int } (\text{card } (\bigcap I)))$
 $\langle \text{proof} \rangle$

A more conventional form

lemma *inclusion-exclusion*:

assumes *finite A* $\bigwedge K. K \in A \implies \text{finite } K$

shows $\text{int}(\text{card}(\bigcup A)) =$
 $(\sum n=1..\text{card } A. (-1)^\wedge(\text{Suc } n) * (\sum B \mid B \subseteq A \wedge \text{card } B = n. \text{int } (\text{card } (\bigcap B))))$ (is -=?R)
 $\langle \text{proof} \rangle$

lemma *card-UNION*:

assumes *finite A* **and** $\bigwedge K. K \in A \implies \text{finite } K$

shows $\text{card } (\bigcup A) = \text{nat } (\sum I \mid I \subseteq A \wedge I \neq \{\}. (-1)^\wedge(\text{card } I + 1) * \text{int } (\text{card } (\bigcap I)))$
 $\langle \text{proof} \rangle$

lemma *card-UNION-nonneg*:

assumes *finite A* **and** $\bigwedge K. K \in A \implies \text{finite } K$

shows $(\sum I \mid I \subseteq A \wedge I \neq \{\}. (-1)^\wedge(\text{card } I + 1) * \text{int } (\text{card } (\bigcap I))) \geq 0$
 $\langle \text{proof} \rangle$

93.8 General "Moebius inversion" inclusion-exclusion principle

This "symmetric" form is from Ira Gessel: "Symmetric Inclusion-Exclusion"

lemma *sum-Un-eq*:

$$\begin{aligned} & \llbracket S \cap T = \{\}; S \cup T = U; \text{finite } U \rrbracket \\ & \implies (\text{sum } f \ S + \text{sum } f \ T = \text{sum } f \ U) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *card-adjust-lemma*: $\llbracket \text{inj-on } f \ S; x = y + \text{card } (f \ ' \ S) \rrbracket \implies x = y + \text{card } S$
 $\langle \text{proof} \rangle$

lemma *card-subsets-step*:

$$\begin{aligned} & \text{assumes } \text{finite } S \ x \notin S \ U \subseteq S \\ & \text{shows } \text{card } \{T. T \subseteq (\text{insert } x \ S) \wedge U \subseteq T \wedge \text{odd}(\text{card } T)\} \\ & \quad = \text{card } \{T. T \subseteq S \wedge U \subseteq T \wedge \text{odd}(\text{card } T)\} + \text{card } \{T. T \subseteq S \wedge U \subseteq T \\ & \quad \wedge \text{even}(\text{card } T)\} \wedge \\ & \quad \text{card } \{T. T \subseteq (\text{insert } x \ S) \wedge U \subseteq T \wedge \text{even}(\text{card } T)\} \\ & \quad = \text{card } \{T. T \subseteq S \wedge U \subseteq T \wedge \text{even}(\text{card } T)\} + \text{card } \{T. T \subseteq S \wedge U \subseteq T \\ & \quad \wedge \text{odd}(\text{card } T)\} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *card-subsupersets-even-odd*:

$$\begin{aligned} & \text{assumes } \text{finite } S \ U \subset S \\ & \text{shows } \text{card } \{T. T \subseteq S \wedge U \subseteq T \wedge \text{even}(\text{card } T)\} \\ & \quad = \text{card } \{T. T \subseteq S \wedge U \subseteq T \wedge \text{odd}(\text{card } T)\} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *sum-alternating-cancels*:

$$\begin{aligned} & \text{assumes } \text{finite } S \ \text{card } \{x. x \in S \wedge \text{even}(f \ x)\} = \text{card } \{x. x \in S \wedge \text{odd}(f \ x)\} \\ & \text{shows } (\sum_{x \in S. (-1) \wedge f \ x} = (0::'b::\text{ring-1})) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *inclusion-exclusion-symmetric*:

$$\begin{aligned} & \text{fixes } f :: 'a \ \text{set} \Rightarrow 'b::\text{ring-1} \\ & \text{assumes } \S: \bigwedge S. \text{finite } S \implies g \ S = (\sum T \in \text{Pow } S. (-1) \wedge \text{card } T * f \ T) \\ & \quad \text{and } \text{finite } S \\ & \text{shows } f \ S = (\sum T \in \text{Pow } S. (-1) \wedge \text{card } T * g \ T) \\ & \langle \text{proof} \rangle \end{aligned}$$

The more typical non-symmetric version.

lemma *inclusion-exclusion-mobius*:

$$\begin{aligned} & \text{fixes } f :: 'a \ \text{set} \Rightarrow 'b::\text{ring-1} \\ & \text{assumes } \S: \bigwedge S. \text{finite } S \implies g \ S = \text{sum } f \ (\text{Pow } S) \ \text{and } \text{finite } S \\ & \text{shows } f \ S = (\sum T \in \text{Pow } S. (-1) \wedge (\text{card } S - \text{card } T) * g \ T) \ (\text{is } - = ?\text{rhs}) \\ & \langle \text{proof} \rangle \end{aligned}$$

93.9 Executable code

lemma *gbinomial-code* [code]:

```

  a gchoose k =
    (if k = 0 then 1
     else fold-atLeastAtMost-nat ( $\lambda k$  acc. (a - of-nat k) * acc) 0 (k - 1) 1 / fact
    k)
  <proof>

```

lemma *binomial-code* [code]:

```

  n choose k =
    (if k > n then 0
     else if 2 * k > n then n choose (n - k)
     else (fold-atLeastAtMost-nat (*) (n - k + 1) n 1 div fact k))
  <proof>

```

end

94 Main HOL

Classical Higher-order Logic – only “Main”, excluding real and complex numbers etc.

theory *Main*

imports

```

  Predicate-Compile
  Quickcheck-Narrowing
  Mirabelle
  Extraction
  Nunchaku
  BNF-Greatest-Fixpoint
  Filter
  Conditionally-Complete-Lattices
  Binomial
  GCD

```

begin

94.1 Namespace cleanup

hide-const (open)

```

  czero cfinite cfinite csum cone ctwo Csum cprod cexp image2 image2p vimage2p
  Gr Grp collect
  fst 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*

94.2 Syntax cleanup

no-notation

```
ordLeq2 (infix <=> 50) and
ordLeq3 (infix <≤> 50) and
ordLess2 (infix <<> 50) and
ordIso2 (infix <=> 50) and
card-of (⟨⟨open-block notation=⟨mixfix card-of⟩⟩|-|⟩) and
BNF-Cardinal-Arithmetic.csum (infixr <+c> 65) and
BNF-Cardinal-Arithmetic.cprod (infixr <*c> 80) and
BNF-Cardinal-Arithmetic.cexp (infixr <^c> 90) and
BNF-Def.convolve (⟨⟨indent=1 notation=⟨mixfix convolve⟩⟩⟨-, / -⟩⟩)
```

bundle cardinal-syntax

begin

notation

```
ordLeq2 (infix <=> 50) and
ordLeq3 (infix <≤> 50) and
ordLess2 (infix <<> 50) and
ordIso2 (infix <=> 50) and
card-of (⟨⟨open-block notation=⟨mixfix card-of⟩⟩|-|⟩) and
BNF-Cardinal-Arithmetic.csum (infixr <+c> 65) and
BNF-Cardinal-Arithmetic.cprod (infixr <*c> 80) and
BNF-Cardinal-Arithmetic.cexp (infixr <^c> 90)
```

alias cinfinite = BNF-Cardinal-Arithmetic.cinfinite

alias czero = BNF-Cardinal-Arithmetic.czzero

alias cone = BNF-Cardinal-Arithmetic.cone

alias ctwo = BNF-Cardinal-Arithmetic.ctwo

end

94.3 Lattice syntax

bundle lattice-syntax

begin

notation

```
bot (⟨⊥⟩) and
top (⟨⊤⟩) and
inf (infixl <⊓> 70) and
sup (infixl <⊔> 65) and
Inf (⟨⟨open-block notation=⟨prefix ⊓⟩⟩⊓ -) [900] 900) and
Sup (⟨⟨open-block notation=⟨prefix ⊔⟩⟩⊔ -) [900] 900)
```

syntax

```
-INF1 :: ptnrs ⇒ 'b ⇒ 'b (⟨⟨indent=3 notation=⟨binder ⊓⟩⟩⊓ -./
-)> [0, 10] 10)
-INF :: ptnr ⇒ 'a set ⇒ 'b ⇒ 'b (⟨⟨indent=3 notation=⟨binder ⊓⟩⟩⊓ -∈-./
```

```

-)› [0, 0, 10] 10)
  -SUP1      :: pptrns ⇒ 'b ⇒ 'b      (⟨(⟨indent=3 notation=⟨binder □⟩)□-./
-)› [0, 10] 10)
  -SUP       :: pptrn ⇒ 'a set ⇒ 'b ⇒ 'b (⟨(⟨indent=3 notation=⟨binder □⟩)□-∈-./
-)› [0, 0, 10] 10)

```

end

unbundle *no lattice-syntax*

end

95 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:  $\bigwedge x. x \in S \implies |x| \leq a$ 
  shows  $|Inf\ S| \leq a$ 
⟨proof⟩

```

lemma *cSup-asclose*:

```

  fixes S :: 'a::{linordered-idom,conditionally-complete-linorder} set
  assumes S: S ≠ {}
    and b:  $\forall x \in S. |x - l| \leq e$ 
  shows  $|Sup\ S - l| \leq e$ 
⟨proof⟩

```

lemma *cInf-asclose*:

```

  fixes S :: 'a::{linordered-idom,conditionally-complete-linorder} set
  assumes S: S ≠ {}
    and b:  $\forall x \in S. |x - l| \leq e$ 
  shows  $|Inf\ S - l| \leq e$ 
⟨proof⟩

```

95.1 Class of Archimedean fields

Archimedean fields have no infinite elements.

class *archimedean-field* = *linordered-field* +
assumes *ex-le-of-int*: $\exists z. x \leq of-int\ z$

lemma *ex-less-of-int*: $\exists z. x < of-int\ z$
for $x :: 'a::archimedean-field$

$\langle \text{proof} \rangle$

lemma *ex-of-int-less*: $\exists z. \text{of-int } z < x$
for $x :: 'a::\text{archimedean-field}$
 $\langle \text{proof} \rangle$

lemma *reals-Archimedean2*: $\exists n. x < \text{of-nat } n$
for $x :: 'a::\text{archimedean-field}$
 $\langle \text{proof} \rangle$

lemma *real-arch-simple*: $\exists n. x \leq \text{of-nat } n$
for $x :: 'a::\text{archimedean-field}$
 $\langle \text{proof} \rangle$

Archimedean fields have no infinitesimal elements.

lemma *reals-Archimedean*:
fixes $x :: 'a::\text{archimedean-field}$
assumes $0 < x$
shows $\exists n. \text{inverse } (\text{of-nat } (\text{Suc } n)) < x$
 $\langle \text{proof} \rangle$

lemma *ex-inverse-of-nat-less*:
fixes $x :: 'a::\text{archimedean-field}$
assumes $0 < x$
shows $\exists n > 0. \text{inverse } (\text{of-nat } n) < x$
 $\langle \text{proof} \rangle$

lemma *ex-less-of-nat-mult*:
fixes $x :: 'a::\text{archimedean-field}$
assumes $0 < x$
shows $\exists n. y < \text{of-nat } n * x$
 $\langle \text{proof} \rangle$

95.2 Existence and uniqueness of floor function

lemma *exists-least-lemma*:
assumes $\neg P \ 0$ **and** $\exists n. P \ n$
shows $\exists n. \neg P \ n \wedge P \ (\text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *floor-exists*:
fixes $x :: 'a::\text{archimedean-field}$
shows $\exists z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1)$
 $\langle \text{proof} \rangle$

lemma *floor-exists1*: $\exists! z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1)$
for $x :: 'a::\text{archimedean-field}$
 $\langle \text{proof} \rangle$

95.3 Floor function

class *floor-ceiling* = *archimedean-field* +
fixes *floor* :: 'a \Rightarrow int ($\langle \langle \text{open-block notation} = \langle \text{mixfix floor} \rangle \rangle [-] \rangle$)
assumes *floor-correct*: of-int $\lfloor x \rfloor \leq x \wedge x < \text{of-int } (\lfloor x \rfloor + 1)$

lemma *floor-unique*: of-int $z \leq x \implies x < \text{of-int } z + 1 \implies \lfloor x \rfloor = z$
 $\langle \text{proof} \rangle$

lemma *floor-eq-iff*: $\lfloor x \rfloor = a \longleftrightarrow \text{of-int } a \leq x \wedge x < \text{of-int } a + 1$
 $\langle \text{proof} \rangle$

lemma *of-int-floor-le [simp]*: of-int $\lfloor x \rfloor \leq x$
 $\langle \text{proof} \rangle$

lemma *le-floor-iff*: $z \leq \lfloor x \rfloor \longleftrightarrow \text{of-int } z \leq x$
 $\langle \text{proof} \rangle$

lemma *floor-less-iff*: $\lfloor x \rfloor < z \longleftrightarrow x < \text{of-int } z$
 $\langle \text{proof} \rangle$

lemma *less-floor-iff*: $z < \lfloor x \rfloor \longleftrightarrow \text{of-int } z + 1 \leq x$
 $\langle \text{proof} \rangle$

lemma *floor-le-iff*: $\lfloor x \rfloor \leq z \longleftrightarrow x < \text{of-int } z + 1$
 $\langle \text{proof} \rangle$

lemma *floor-split[linarith-split]*: $P \lfloor t \rfloor \longleftrightarrow (\forall i. \text{of-int } i \leq t \wedge t < \text{of-int } i + 1 \implies P i)$
 $\langle \text{proof} \rangle$

lemma *floor-eq-imp-diff-1*: $\lfloor x \rfloor = \lfloor y \rfloor \implies |x - y| < 1$
 $\langle \text{proof} \rangle$

lemma *floor-mono*:
assumes $x \leq y$
shows $\lfloor x \rfloor \leq \lfloor y \rfloor$
 $\langle \text{proof} \rangle$

lemma *floor-less-cancel*: $\lfloor x \rfloor < \lfloor y \rfloor \implies x < y$
 $\langle \text{proof} \rangle$

lemma *floor-of-int [simp]*: $\lfloor \text{of-int } z \rfloor = z$
 $\langle \text{proof} \rangle$

lemma *floor-of-nat [simp]*: $\lfloor \text{of-nat } n \rfloor = \text{int } n$
 $\langle \text{proof} \rangle$

lemma *le-floor-add*: $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor$
 $\langle \text{proof} \rangle$

Floor with numerals.

lemma *floor-zero* [simp]: $\lfloor 0 \rfloor = 0$
 $\langle \text{proof} \rangle$

lemma *floor-one* [simp]: $\lfloor 1 \rfloor = 1$
 $\langle \text{proof} \rangle$

lemma *floor-numeral* [simp]: $\lfloor \text{numeral } v \rfloor = \text{numeral } v$
 $\langle \text{proof} \rangle$

lemma *floor-neg-numeral* [simp]: $\lfloor - \text{numeral } v \rfloor = - \text{numeral } v$
 $\langle \text{proof} \rangle$

lemma *zero-le-floor* [simp]: $0 \leq \lfloor x \rfloor \longleftrightarrow 0 \leq x$
 $\langle \text{proof} \rangle$

lemma *one-le-floor* [simp]: $1 \leq \lfloor x \rfloor \longleftrightarrow 1 \leq x$
 $\langle \text{proof} \rangle$

lemma *numeral-le-floor* [simp]: $\text{numeral } v \leq \lfloor x \rfloor \longleftrightarrow \text{numeral } v \leq x$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-le-floor* [simp]: $- \text{numeral } v \leq \lfloor x \rfloor \longleftrightarrow - \text{numeral } v \leq x$
 $\langle \text{proof} \rangle$

lemma *zero-less-floor* [simp]: $0 < \lfloor x \rfloor \longleftrightarrow 1 \leq x$
 $\langle \text{proof} \rangle$

lemma *one-less-floor* [simp]: $1 < \lfloor x \rfloor \longleftrightarrow 2 \leq x$
 $\langle \text{proof} \rangle$

lemma *numeral-less-floor* [simp]: $\text{numeral } v < \lfloor x \rfloor \longleftrightarrow \text{numeral } v + 1 \leq x$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-less-floor* [simp]: $- \text{numeral } v < \lfloor x \rfloor \longleftrightarrow - \text{numeral } v + 1 \leq x$
 $\langle \text{proof} \rangle$

lemma *floor-le-zero* [simp]: $\lfloor x \rfloor \leq 0 \longleftrightarrow x < 1$
 $\langle \text{proof} \rangle$

lemma *floor-le-one* [simp]: $\lfloor x \rfloor \leq 1 \longleftrightarrow x < 2$
 $\langle \text{proof} \rangle$

lemma *floor-le-numeral* [simp]: $\lfloor x \rfloor \leq \text{numeral } v \longleftrightarrow x < \text{numeral } v + 1$
 $\langle \text{proof} \rangle$

lemma *floor-le-neg-numeral* [simp]: $\lfloor x \rfloor \leq - \text{numeral } v \longleftrightarrow x < - \text{numeral } v + 1$
 $\langle \text{proof} \rangle$

$\langle \text{proof} \rangle$

lemma *floor-less-zero* [simp]: $\lfloor x \rfloor < 0 \longleftrightarrow x < 0$
 $\langle \text{proof} \rangle$

lemma *floor-less-one* [simp]: $\lfloor x \rfloor < 1 \longleftrightarrow x < 1$
 $\langle \text{proof} \rangle$

lemma *floor-less-numeral* [simp]: $\lfloor x \rfloor < \text{numeral } v \longleftrightarrow x < \text{numeral } v$
 $\langle \text{proof} \rangle$

lemma *floor-less-neg-numeral* [simp]: $\lfloor x \rfloor < - \text{numeral } v \longleftrightarrow x < - \text{numeral } v$
 $\langle \text{proof} \rangle$

lemma *le-mult-floor-Ints*:
 assumes $0 \leq a$ $a \in \text{Ints}$
 shows $\text{of-int } (\lfloor a \rfloor * \lfloor b \rfloor) \leq (\text{of-int } \lfloor a * b \rfloor) :: 'a :: \text{linordered-idom}$
 $\langle \text{proof} \rangle$

Addition and subtraction of integers.

lemma *floor-add-int*: $\lfloor x \rfloor + z = \lfloor x + \text{of-int } z \rfloor$
 $\langle \text{proof} \rangle$

lemma *int-add-floor*: $z + \lfloor x \rfloor = \lfloor \text{of-int } z + x \rfloor$
 $\langle \text{proof} \rangle$

lemma *one-add-floor*: $\lfloor x \rfloor + 1 = \lfloor x + 1 \rfloor$
 $\langle \text{proof} \rangle$

lemma *floor-diff-of-int* [simp]: $\lfloor x - \text{of-int } z \rfloor = \lfloor x \rfloor - z$
 $\langle \text{proof} \rangle$

lemma *floor-uminus-of-int* [simp]: $\lfloor - (\text{of-int } z) \rfloor = - z$
 $\langle \text{proof} \rangle$

lemma *floor-diff-numeral* [simp]: $\lfloor x - \text{numeral } v \rfloor = \lfloor x \rfloor - \text{numeral } v$
 $\langle \text{proof} \rangle$

lemma *floor-diff-one* [simp]: $\lfloor x - 1 \rfloor = \lfloor x \rfloor - 1$
 $\langle \text{proof} \rangle$

lemma *le-mult-floor*:
 assumes $0 \leq a$ and $0 \leq b$
 shows $\lfloor a \rfloor * \lfloor b \rfloor \leq \lfloor a * b \rfloor$
 $\langle \text{proof} \rangle$

lemma *floor-divide-of-int-eq*: $\lfloor \text{of-int } k / \text{of-int } l \rfloor = k \text{ div } l$
 for $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *floor-divide-of-nat-eq*: $\lfloor \text{of-nat } m / \text{of-nat } n \rfloor = \text{of-nat } (m \text{ div } n)$
for $m \ n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *floor-divide-lower*:
fixes $q :: 'a::\text{floor-ceiling}$
shows $q > 0 \implies \text{of-int } \lfloor p / q \rfloor * q \leq p$
 $\langle \text{proof} \rangle$

lemma *floor-divide-upper*:
fixes $q :: 'a::\text{floor-ceiling}$
shows $q > 0 \implies p < (\text{of-int } \lfloor p / q \rfloor + 1) * q$
 $\langle \text{proof} \rangle$

95.4 Ceiling function

definition *ceiling* :: $'a::\text{floor-ceiling} \Rightarrow \text{int}$ ($\langle \langle \text{open-block notation} = \langle \text{mixfix ceiling} \rangle \rangle \lfloor - \rfloor \rangle$)
where $\lceil x \rceil = - \lfloor -x \rfloor$

lemma *ceiling-correct*: $\text{of-int } \lceil x \rceil - 1 < x \wedge x \leq \text{of-int } \lceil x \rceil$
 $\langle \text{proof} \rangle$

lemma *ceiling-unique*: $\text{of-int } z - 1 < x \implies x \leq \text{of-int } z \implies \lceil x \rceil = z$
 $\langle \text{proof} \rangle$

lemma *ceiling-eq-iff*: $\lceil x \rceil = a \longleftrightarrow \text{of-int } a - 1 < x \wedge x \leq \text{of-int } a$
 $\langle \text{proof} \rangle$

lemma *le-of-int-ceiling* [*simp*]: $x \leq \text{of-int } \lceil x \rceil$
 $\langle \text{proof} \rangle$

lemma *ceiling-le-iff*: $\lceil x \rceil \leq z \longleftrightarrow x \leq \text{of-int } z$
 $\langle \text{proof} \rangle$

lemma *less-ceiling-iff*: $z < \lceil x \rceil \longleftrightarrow \text{of-int } z < x$
 $\langle \text{proof} \rangle$

lemma *ceiling-less-iff*: $\lceil x \rceil < z \longleftrightarrow x \leq \text{of-int } z - 1$
 $\langle \text{proof} \rangle$

lemma *le-ceiling-iff*: $z \leq \lceil x \rceil \longleftrightarrow \text{of-int } z - 1 < x$
 $\langle \text{proof} \rangle$

lemma *ceiling-mono*: $x \geq y \implies \lceil x \rceil \geq \lceil y \rceil$
 $\langle \text{proof} \rangle$

lemma *ceiling-less-cancel*: $\lceil x \rceil < \lceil y \rceil \implies x < y$

$\langle \text{proof} \rangle$

lemma *ceiling-of-int* [simp]: $\lceil \text{of-int } z \rceil = z$
 $\langle \text{proof} \rangle$

lemma *ceiling-of-nat* [simp]: $\lceil \text{of-nat } n \rceil = \text{int } n$
 $\langle \text{proof} \rangle$

lemma *ceiling-add-le*: $\lceil x + y \rceil \leq \lceil x \rceil + \lceil y \rceil$
 $\langle \text{proof} \rangle$

lemma *mult-ceiling-le*:
 assumes $0 \leq a$ and $0 \leq b$
 shows $\lceil a * b \rceil \leq \lceil a \rceil * \lceil b \rceil$
 $\langle \text{proof} \rangle$

lemma *mult-ceiling-le-Ints*:
 assumes $0 \leq a$ $a \in \text{Ints}$
 shows $(\text{of-int } \lceil a * b \rceil :: 'a :: \text{linordered-idom}) \leq \text{of-int}(\lceil a \rceil * \lceil b \rceil)$
 $\langle \text{proof} \rangle$

lemma *finite-int-segment*:
 fixes $a :: 'a :: \text{floor-ceiling}$
 shows *finite* $\{x \in \mathbb{Z}. a \leq x \wedge x \leq b\}$
 $\langle \text{proof} \rangle$

corollary *finite-abs-int-segment*:
 fixes $a :: 'a :: \text{floor-ceiling}$
 shows *finite* $\{k \in \mathbb{Z}. |k| \leq a\}$
 $\langle \text{proof} \rangle$

95.4.1 Ceiling with numerals.

lemma *ceiling-zero* [simp]: $\lceil 0 \rceil = 0$
 $\langle \text{proof} \rangle$

lemma *ceiling-one* [simp]: $\lceil 1 \rceil = 1$
 $\langle \text{proof} \rangle$

lemma *ceiling-numeral* [simp]: $\lceil \text{numeral } v \rceil = \text{numeral } v$
 $\langle \text{proof} \rangle$

lemma *ceiling-neg-numeral* [simp]: $\lceil - \text{numeral } v \rceil = - \text{numeral } v$
 $\langle \text{proof} \rangle$

lemma *ceiling-le-zero* [simp]: $\lceil x \rceil \leq 0 \longleftrightarrow x \leq 0$
 $\langle \text{proof} \rangle$

lemma *ceiling-le-one* [simp]: $\lceil x \rceil \leq 1 \longleftrightarrow x \leq 1$

$\langle \text{proof} \rangle$

lemma *ceiling-le-numeral* [simp]: $\lceil x \rceil \leq \text{numeral } v \longleftrightarrow x \leq \text{numeral } v$
 $\langle \text{proof} \rangle$

lemma *ceiling-le-neg-numeral* [simp]: $\lceil x \rceil \leq - \text{numeral } v \longleftrightarrow x \leq - \text{numeral } v$
 $\langle \text{proof} \rangle$

lemma *ceiling-less-zero* [simp]: $\lceil x \rceil < 0 \longleftrightarrow x \leq -1$
 $\langle \text{proof} \rangle$

lemma *ceiling-less-one* [simp]: $\lceil x \rceil < 1 \longleftrightarrow x \leq 0$
 $\langle \text{proof} \rangle$

lemma *ceiling-less-numeral* [simp]: $\lceil x \rceil < \text{numeral } v \longleftrightarrow x \leq \text{numeral } v - 1$
 $\langle \text{proof} \rangle$

lemma *ceiling-less-neg-numeral* [simp]: $\lceil x \rceil < - \text{numeral } v \longleftrightarrow x \leq - \text{numeral } v - 1$
 $\langle \text{proof} \rangle$

lemma *zero-le-ceiling* [simp]: $0 \leq \lceil x \rceil \longleftrightarrow -1 < x$
 $\langle \text{proof} \rangle$

lemma *one-le-ceiling* [simp]: $1 \leq \lceil x \rceil \longleftrightarrow 0 < x$
 $\langle \text{proof} \rangle$

lemma *numeral-le-ceiling* [simp]: $\text{numeral } v \leq \lceil x \rceil \longleftrightarrow \text{numeral } v - 1 < x$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-le-ceiling* [simp]: $- \text{numeral } v \leq \lceil x \rceil \longleftrightarrow - \text{numeral } v - 1 < x$
 $\langle \text{proof} \rangle$

lemma *zero-less-ceiling* [simp]: $0 < \lceil x \rceil \longleftrightarrow 0 < x$
 $\langle \text{proof} \rangle$

lemma *one-less-ceiling* [simp]: $1 < \lceil x \rceil \longleftrightarrow 1 < x$
 $\langle \text{proof} \rangle$

lemma *numeral-less-ceiling* [simp]: $\text{numeral } v < \lceil x \rceil \longleftrightarrow \text{numeral } v < x$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-less-ceiling* [simp]: $- \text{numeral } v < \lceil x \rceil \longleftrightarrow - \text{numeral } v < x$
 $\langle \text{proof} \rangle$

lemma *ceiling-altdef*: $\lceil x \rceil = (\text{if } x = \text{of-int } \lfloor x \rfloor \text{ then } \lfloor x \rfloor \text{ else } \lfloor x \rfloor + 1)$
 $\langle \text{proof} \rangle$

lemma *floor-le-ceiling* [simp]: $\lfloor x \rfloor \leq \lceil x \rceil$
 ⟨proof⟩

95.4.2 Addition and subtraction of integers.

lemma *ceiling-add-of-int* [simp]: $\lceil x + \text{of-int } z \rceil = \lceil x \rceil + z$
 ⟨proof⟩

lemma *ceiling-add-numeral* [simp]: $\lceil x + \text{numeral } v \rceil = \lceil x \rceil + \text{numeral } v$
 ⟨proof⟩

lemma *ceiling-add-one* [simp]: $\lceil x + 1 \rceil = \lceil x \rceil + 1$
 ⟨proof⟩

lemma *ceiling-diff-of-int* [simp]: $\lceil x - \text{of-int } z \rceil = \lceil x \rceil - z$
 ⟨proof⟩

lemma *ceiling-diff-numeral* [simp]: $\lceil x - \text{numeral } v \rceil = \lceil x \rceil - \text{numeral } v$
 ⟨proof⟩

lemma *ceiling-diff-one* [simp]: $\lceil x - 1 \rceil = \lceil x \rceil - 1$
 ⟨proof⟩

lemma *ceiling-split[linarith-split]*: $P \lceil t \rceil \longleftrightarrow (\forall i. \text{of-int } i - 1 < t \wedge t \leq \text{of-int } i \longrightarrow P i)$
 ⟨proof⟩

lemma *ceiling-eq-imp-diff-1*: $\lceil x \rceil = \lceil y \rceil \implies |x - y| < 1$
 ⟨proof⟩

lemma *ceiling-diff-floor-le-1*: $\lceil x \rceil - \lfloor x \rfloor \leq 1$
 ⟨proof⟩

lemma *floor-eq-ceiling-imp-diff-2*: $\lfloor x \rfloor = \lceil y \rceil \implies |x - y| < 2$
 ⟨proof⟩

lemma *nat-approx-posE*:
 fixes $e :: 'a :: \{\text{archimedean-field, floor-ceiling}\}$
 assumes $0 < e$
 obtains $n :: \text{nat}$ where $1 / \text{of-nat}(\text{Suc } n) < e$
 ⟨proof⟩

lemma *ceiling-divide-upper*:
 fixes $q :: 'a :: \text{floor-ceiling}$
 shows $q > 0 \implies p \leq \text{of-int}(\text{ceiling}(p / q)) * q$
 ⟨proof⟩

lemma *ceiling-divide-lower*:
 fixes $q :: 'a :: \text{floor-ceiling}$

shows $q > 0 \implies (of_int \lceil p / q \rceil - 1) * q < p$
 $\langle proof \rangle$

95.5 Negation

lemma *floor-minus*: $\lfloor -x \rfloor = -\lceil x \rceil$
 $\langle proof \rangle$

lemma *ceiling-minus*: $\lceil -x \rceil = -\lfloor x \rfloor$
 $\langle proof \rangle$

95.6 Natural numbers

lemma *of-nat-floor*: $r \geq 0 \implies of_nat (nat \lfloor r \rfloor) \leq r$
 $\langle proof \rangle$

lemma *of-nat-ceiling*: $of_nat (nat \lceil r \rceil) \geq r$
 $\langle proof \rangle$

lemma *of-nat-int-floor [simp]*: $x \geq 0 \implies of_nat (nat \lfloor x \rfloor) = of_int \lfloor x \rfloor$
 $\langle proof \rangle$

lemma *of-nat-int-ceiling [simp]*: $x \geq 0 \implies of_nat (nat \lceil x \rceil) = of_int \lceil x \rceil$
 $\langle proof \rangle$

95.7 Frac Function

definition *frac* :: $'a \Rightarrow 'a::floor_ceiling$
where $frac\ x \equiv x - of_int \lfloor x \rfloor$

lemma *frac-lt-1*: $frac\ x < 1$
 $\langle proof \rangle$

lemma *frac-eq-0-iff [simp]*: $frac\ x = 0 \longleftrightarrow x \in \mathbb{Z}$
 $\langle proof \rangle$

lemma *frac-ge-0 [simp]*: $frac\ x \geq 0$
 $\langle proof \rangle$

lemma *frac-gt-0-iff [simp]*: $frac\ x > 0 \longleftrightarrow x \notin \mathbb{Z}$
 $\langle proof \rangle$

lemma *frac-of-int [simp]*: $frac\ (of_int\ z) = 0$
 $\langle proof \rangle$

lemma *frac-frac [simp]*: $frac\ (frac\ x) = frac\ x$
 $\langle proof \rangle$

lemma *floor-add*: $\lfloor x + y \rfloor = (if\ frac\ x + frac\ y < 1\ then\ \lfloor x \rfloor + \lfloor y \rfloor\ else\ (\lfloor x \rfloor + \lfloor y \rfloor) + 1)$

$\langle proof \rangle$

lemma *floor-add2*[simp]: $x \in \mathbb{Z} \vee y \in \mathbb{Z} \implies \lfloor x + y \rfloor = \lfloor x \rfloor + \lfloor y \rfloor$
 $\langle proof \rangle$

lemma *frac-add*:
 $frac (x + y) = (if\ frac\ x +\ frac\ y < 1\ then\ frac\ x +\ frac\ y\ else\ (frac\ x +\ frac\ y - 1))$
 $\langle proof \rangle$

lemma *frac-unique-iff*: $frac\ x = a \longleftrightarrow x - a \in \mathbb{Z} \wedge 0 \leq a \wedge a < 1$
for $x :: 'a::floor-ceiling$
 $\langle proof \rangle$

lemma *frac-eq*: $frac\ x = x \longleftrightarrow 0 \leq x \wedge x < 1$
 $\langle proof \rangle$

lemma *frac-eq-id* [simp]: $x \in \{0..<1\} \implies frac\ x = x$
 $\langle proof \rangle$

lemma *frac-in-Ints-iff* [simp]: $frac\ x \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$
 $\langle proof \rangle$

lemma *frac-neg*: $frac\ (-x) = (if\ x \in \mathbb{Z}\ then\ 0\ else\ 1 - frac\ x)$
for $x :: 'a::floor-ceiling$
 $\langle proof \rangle$

lemma *frac-1-eq*: $frac\ (x+1) = frac\ x$
 $\langle proof \rangle$

95.8 Fractional part arithmetic

Many thanks to Stepan Holub

lemma *frac-non-zero*: $frac\ x \neq 0 \implies frac\ (-x) = 1 - frac\ x$
 $\langle proof \rangle$

lemma *frac-add-simps* [simp]:
 $frac\ (frac\ a + b) = frac\ (a + b)$
 $frac\ (a + frac\ b) = frac\ (a + b)$
 $\langle proof \rangle$

lemma *frac-neg-frac*: $frac\ (-frac\ x) = frac\ (-x)$
 $\langle proof \rangle$

lemma *frac-diff-simp*: $frac\ (y - frac\ x) = frac\ (y - x)$
 $\langle proof \rangle$

lemma *frac-diff*: $frac\ (a - b) = frac\ (frac\ a + (-frac\ b))$
 $\langle proof \rangle$

lemma *frac-diff-pos*: $\text{frac } x \leq \text{frac } y \implies \text{frac } (y - x) = \text{frac } y - \text{frac } x$
 $\langle \text{proof} \rangle$

lemma *frac-diff-neg*: **assumes** $\text{frac } y < \text{frac } x$
shows $\text{frac } (y - x) = \text{frac } y + 1 - \text{frac } x$
 $\langle \text{proof} \rangle$

lemma *frac-diff-eq*: **assumes** $\text{frac } y = \text{frac } x$
shows $\text{frac } (y - x) = 0$
 $\langle \text{proof} \rangle$

lemma *frac-diff-zero*: **assumes** $\text{frac } (x - y) = 0$
shows $\text{frac } x = \text{frac } y$
 $\langle \text{proof} \rangle$

lemma *frac-neg-eq-iff*: $\text{frac } (-x) = \text{frac } (-y) \longleftrightarrow \text{frac } x = \text{frac } y$
 $\langle \text{proof} \rangle$

lemma *frac-eqE*:
assumes $\text{frac } x = \text{frac } y$
obtains n **where** $x = y + \text{of-int } n$
 $\langle \text{proof} \rangle$

lemma *frac-add-of-int-right [simp]*: $\text{frac } (x + \text{of-int } n) = \text{frac } x$
 $\langle \text{proof} \rangle$

lemma *frac-add-of-int-left [simp]*: $\text{frac } (\text{of-int } n + x) = \text{frac } x$
 $\langle \text{proof} \rangle$

lemma *frac-add-int-right*: $y \in \mathbb{Z} \implies \text{frac } (x + y) = \text{frac } x$
 $\langle \text{proof} \rangle$

lemma *frac-add-int-left*: $x \in \mathbb{Z} \implies \text{frac } (x + y) = \text{frac } y$
 $\langle \text{proof} \rangle$

lemma *bij-betw-frac*: $\text{bij-betw } \text{frac } \{x..<x+1\} \{0..<1\}$
 $\langle \text{proof} \rangle$

95.9 Rounding to the nearest integer

definition *round* :: ‘a::floor-ceiling \Rightarrow int
where $\text{round } x = \lfloor x + 1/2 \rfloor$

lemma *round-eq-imp-diff-1*: $\text{round } x = \text{round } y \implies |x - y| < 1$
 $\langle \text{proof} \rangle$

lemma *of-int-round-ge*: $\text{of-int } (\text{round } x) \geq x - 1/2$
and *of-int-round-le*: $\text{of-int } (\text{round } x) \leq x + 1/2$

and *of-int-round-abs-le*: $|of-int (round\ x) - x| \leq 1/2$
and *of-int-round-gt*: $of-int (round\ x) > x - 1/2$
 $\langle proof \rangle$

lemma *round-of-int [simp]*: $round (of-int\ n) = n$
 $\langle proof \rangle$

lemma *round-0 [simp]*: $round\ 0 = 0$
 $\langle proof \rangle$

lemma *round-1 [simp]*: $round\ 1 = 1$
 $\langle proof \rangle$

lemma *round-numeral [simp]*: $round (numeral\ n) = numeral\ n$
 $\langle proof \rangle$

lemma *round-neg-numeral [simp]*: $round (-numeral\ n) = -numeral\ n$
 $\langle proof \rangle$

lemma *round-of-nat [simp]*: $round (of-nat\ n) = of-nat\ n$
 $\langle proof \rangle$

lemma *round-mono*: $x \leq y \implies round\ x \leq round\ y$
 $\langle proof \rangle$

lemma *round-unique*: $of-int\ y > x - 1/2 \implies of-int\ y \leq x + 1/2 \implies round\ x = y$
 $\langle proof \rangle$

lemma *round-unique'*: $|x - of-int\ n| < 1/2 \implies round\ x = n$
 $\langle proof \rangle$

lemma *round-altdef*: $round\ x = (if\ frac\ x \geq 1/2\ then\ \lceil x \rceil\ else\ \lfloor x \rfloor)$
 $\langle proof \rangle$

lemma *floor-le-round*: $\lfloor x \rfloor \leq round\ x$
 $\langle proof \rangle$

lemma *ceiling-ge-round*: $\lceil x \rceil \geq round\ x$
 $\langle proof \rangle$

lemma *round-diff-minimal*: $|z - of-int (round\ z)| \leq |z - of-int\ m|$
for $z :: 'a::floor-ceiling$
 $\langle proof \rangle$

bundle *floor-ceiling-syntax*

begin

notation *floor* ($\langle (\langle open-block\ notation = \langle mixfix\ floor \rangle \rangle [-]) \rangle$)
and *ceiling* ($\langle (\langle open-block\ notation = \langle mixfix\ ceiling \rangle \rangle [-]) \rangle$)

end

end

96 Rational numbers

theory *Rat*
imports *Archimedean-Field*
begin

96.1 Rational numbers as quotient

96.1.1 Construction of the type of rational numbers

definition *ratrel* :: $(int \times int) \Rightarrow (int \times int) \Rightarrow bool$
where *ratrel* = $(\lambda x\ y. snd\ x \neq 0 \wedge snd\ y \neq 0 \wedge fst\ x * snd\ y = fst\ y * snd\ x)$

lemma *ratrel-iff* [*simp*]: $ratrel\ x\ y \longleftrightarrow snd\ x \neq 0 \wedge snd\ y \neq 0 \wedge fst\ x * snd\ y = fst\ y * snd\ x$
 $\langle proof \rangle$

lemma *exists-ratrel-reft*: $\exists x. ratrel\ x\ x$
 $\langle proof \rangle$

lemma *symp-ratrel*: *symp* *ratrel*
 $\langle proof \rangle$

lemma *transp-ratrel*: *transp* *ratrel*
 $\langle proof \rangle$

lemma *part-equivp-ratrel*: *part-equivp* *ratrel*
 $\langle proof \rangle$

quotient-type *rat* = $int \times int / partial: ratrel$
morphisms *Rep-Rat* *Abs-Rat*
 $\langle proof \rangle$

lemma *Domainp-cr-rat* [*transfer-domain-rule*]: $Domainp\ pcr-rat = (\lambda x. snd\ x \neq 0)$
 $\langle proof \rangle$

96.1.2 Representation and basic operations

lift-definition *Fract* :: $int \Rightarrow int \Rightarrow rat$
is $\lambda a\ b. if\ b = 0\ then\ (0, 1)\ else\ (a, b)$
 $\langle proof \rangle$

lemma *eq-rat*:
 $\bigwedge a\ b\ c\ d. b \neq 0 \implies d \neq 0 \implies Fract\ a\ b = Fract\ c\ d \longleftrightarrow a * d = c * b$

$\bigwedge a. \text{Fract } a \ 0 = \text{Fract } 0 \ 1$
 $\bigwedge a \ c. \text{Fract } 0 \ a = \text{Fract } 0 \ c$
 $\langle \text{proof} \rangle$

lemma *Rat-cases* [*case-names* *Fract*, *cases type*: *rat*]:
assumes *that*: $\bigwedge a \ b. q = \text{Fract } a \ b \implies b > 0 \implies \text{coprime } a \ b \implies C$
shows *C*
 $\langle \text{proof} \rangle$

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 $P \ q$
 $\langle \text{proof} \rangle$

instantiation *rat* :: *field*
begin

lift-definition *zero-rat* :: *rat* **is** $(0, 1)$
 $\langle \text{proof} \rangle$

lift-definition *one-rat* :: *rat* **is** $(1, 1)$
 $\langle \text{proof} \rangle$

lemma *Zero-rat-def*: $0 = \text{Fract } 0 \ 1$
 $\langle \text{proof} \rangle$

lemma *One-rat-def*: $1 = \text{Fract } 1 \ 1$
 $\langle \text{proof} \rangle$

lift-definition *plus-rat* :: *rat* \Rightarrow *rat* \Rightarrow *rat*
is $\lambda x \ y. (\text{fst } x * \text{snd } y + \text{fst } y * \text{snd } x, \text{snd } x * \text{snd } y)$
 $\langle \text{proof} \rangle$

lemma *add-rat* [*simp*]:
assumes $b \neq 0$ **and** $d \neq 0$
shows $\text{Fract } a \ b + \text{Fract } c \ d = \text{Fract } (a * d + c * b) \ (b * d)$
 $\langle \text{proof} \rangle$

lift-definition *uminus-rat* :: *rat* \Rightarrow *rat* **is** $\lambda x. (- \text{fst } x, \text{snd } x)$
 $\langle \text{proof} \rangle$

lemma *minus-rat* [*simp*]: $- \text{Fract } a \ b = \text{Fract } (- a) \ b$
 $\langle \text{proof} \rangle$

lemma *minus-rat-cancel* [*simp*]: $\text{Fract } (- a) \ (- b) = \text{Fract } a \ b$
 $\langle \text{proof} \rangle$

definition *diff-rat-def*: $q - r = q + - r$ **for** $q \ r :: \text{rat}$

lemma *diff-rat* [*simp*]:

$b \neq 0 \implies d \neq 0 \implies \text{Fract } a \ b - \text{Fract } c \ d = \text{Fract } (a * d - c * b) \ (b * d)$

<proof>

lift-definition *times-rat* :: *rat* \Rightarrow *rat* \Rightarrow *rat*

is $\lambda x \ y. (fst \ x * fst \ y, snd \ x * snd \ y)$

<proof>

lemma *mult-rat* [*simp*]: $\text{Fract } a \ b * \text{Fract } c \ d = \text{Fract } (a * c) \ (b * d)$

<proof>

lemma *mult-rat-cancel*: $c \neq 0 \implies \text{Fract } (c * a) \ (c * b) = \text{Fract } a \ b$

<proof>

lift-definition *inverse-rat* :: *rat* \Rightarrow *rat*

is $\lambda x. \text{if } fst \ x = 0 \text{ then } (0, 1) \text{ else } (snd \ x, fst \ x)$

<proof>

lemma *inverse-rat* [*simp*]: $\text{inverse } (\text{Fract } a \ b) = \text{Fract } b \ a$

<proof>

definition *divide-rat-def*: $q \ \text{div} \ r = q * \text{inverse } r$ **for** $q \ r :: \text{rat}$

lemma *divide-rat* [*simp*]: $\text{Fract } a \ b \ \text{div} \ \text{Fract } c \ d = \text{Fract } (a * d) \ (b * c)$

<proof>

instance

<proof>

end

lemma *div-add-self1-no-field* [*simp*]:

assumes *NO-MATCH* $(x :: 'b :: \text{field}) \ b \ (b :: 'a :: \text{euclidean-semiring-cancel}) \neq 0$

shows $(b + a) \ \text{div} \ b = a \ \text{div} \ b + 1$

<proof>

lemma *div-add-self2-no-field* [*simp*]:

assumes *NO-MATCH* $(x :: 'b :: \text{field}) \ b \ (b :: 'a :: \text{euclidean-semiring-cancel}) \neq 0$

shows $(a + b) \ \text{div} \ b = a \ \text{div} \ b + 1$

<proof>

lemma *of-nat-rat*: $\text{of-nat } k = \text{Fract } (\text{of-nat } k) \ 1$

<proof>

lemma *of-int-rat*: $\text{of-int } k = \text{Fract } k \ 1$

<proof>

lemma *Fract-of-nat-eq*: $\text{Fract } (\text{of-nat } k) \ 1 = \text{of-nat } k$
 ⟨proof⟩

lemma *Fract-of-int-eq*: $\text{Fract } k \ 1 = \text{of-int } k$
 ⟨proof⟩

lemma *rat-number-collapse*:
 $\text{Fract } 0 \ k = 0$
 $\text{Fract } 1 \ 1 = 1$
 $\text{Fract } (\text{numeral } w) \ 1 = \text{numeral } w$
 $\text{Fract } (- \text{numeral } w) \ 1 = - \text{numeral } w$
 $\text{Fract } (- 1) \ 1 = - 1$
 $\text{Fract } k \ 0 = 0$
 ⟨proof⟩

lemma *rat-number-expand*:
 $0 = \text{Fract } 0 \ 1$
 $1 = \text{Fract } 1 \ 1$
 $\text{numeral } k = \text{Fract } (\text{numeral } k) \ 1$
 $- 1 = \text{Fract } (- 1) \ 1$
 $- \text{numeral } k = \text{Fract } (- \text{numeral } k) \ 1$
 ⟨proof⟩

lemma *Rat-cases-nonzero* [case-names *Fract 0*]:
 assumes *Fract*: $\bigwedge a \ b. \ q = \text{Fract } a \ b \implies b > 0 \implies a \neq 0 \implies \text{coprime } a \ b \implies$
 C
 and 0 : $q = 0 \implies C$
 shows C
 ⟨proof⟩

96.1.3 Function *normalize*

lemma *Fract-coprime*: $\text{Fract } (a \ \text{div} \ \text{gcd } a \ b) \ (b \ \text{div} \ \text{gcd } a \ b) = \text{Fract } a \ b$
 ⟨proof⟩

definition *normalize* :: $\text{int} \times \text{int} \Rightarrow \text{int} \times \text{int}$
where *normalize* $p =$
 (if $\text{snd } p > 0$ then (let $a = \text{gcd } (\text{fst } p) \ (\text{snd } p)$ in $(\text{fst } p \ \text{div} \ a, \ \text{snd } p \ \text{div} \ a)$)
 else if $\text{snd } p = 0$ then $(0, \ 1)$
 else (let $a = - \text{gcd } (\text{fst } p) \ (\text{snd } p)$ in $(\text{fst } p \ \text{div} \ a, \ \text{snd } p \ \text{div} \ a)$))

lemma *normalize-crossproduct*:
 assumes $q \neq 0 \ s \neq 0$
 assumes *normalize* $(p, \ q) = \text{normalize } (r, \ s)$
 shows $p * s = r * q$
 ⟨proof⟩

lemma *normalize-eq*: $\text{normalize } (a, \ b) = (p, \ q) \implies \text{Fract } p \ q = \text{Fract } a \ b$

$\langle \text{proof} \rangle$

lemma *normalize-denom-pos*: $\text{normalize } r = (p, q) \implies q > 0$
 $\langle \text{proof} \rangle$

lemma *normalize-coprime*: $\text{normalize } r = (p, q) \implies \text{coprime } p \ q$
 $\langle \text{proof} \rangle$

lemma *normalize-stable* [simp]: $q > 0 \implies \text{coprime } p \ q \implies \text{normalize } (p, q) = (p, q)$
 $\langle \text{proof} \rangle$

lemma *normalize-denom-zero* [simp]: $\text{normalize } (p, 0) = (0, 1)$
 $\langle \text{proof} \rangle$

lemma *normalize-negative* [simp]: $q < 0 \implies \text{normalize } (p, q) = \text{normalize } (-p, -q)$
 $\langle \text{proof} \rangle$

Decompose a fraction into normalized, i.e. coprime numerator and denominator:

definition *quotient-of* :: $\text{rat} \Rightarrow \text{int} \times \text{int}$
where *quotient-of* $x =$
 $(\text{THE pair. } x = \text{Fract } (\text{fst pair}) (\text{snd pair}) \wedge \text{snd pair} > 0 \wedge \text{coprime } (\text{fst pair}) (\text{snd pair}))$

lemma *quotient-of-unique*: $\exists! p. r = \text{Fract } (\text{fst } p) (\text{snd } p) \wedge \text{snd } p > 0 \wedge \text{coprime } (\text{fst } p) (\text{snd } p)$
 $\langle \text{proof} \rangle$

lemma *quotient-of-Fract* [code]: $\text{quotient-of } (\text{Fract } a \ b) = \text{normalize } (a, b)$
 $\langle \text{proof} \rangle$

lemma *quotient-of-number* [simp]:
 $\text{quotient-of } 0 = (0, 1)$
 $\text{quotient-of } 1 = (1, 1)$
 $\text{quotient-of } (\text{numeral } k) = (\text{numeral } k, 1)$
 $\text{quotient-of } (-1) = (-1, 1)$
 $\text{quotient-of } (-\text{numeral } k) = (-\text{numeral } k, 1)$
 $\langle \text{proof} \rangle$

lemma *quotient-of-eq*: $\text{quotient-of } (\text{Fract } a \ b) = (p, q) \implies \text{Fract } p \ q = \text{Fract } a \ b$
 $\langle \text{proof} \rangle$

lemma *quotient-of-denom-pos*: $\text{quotient-of } r = (p, q) \implies q > 0$
 $\langle \text{proof} \rangle$

lemma *quotient-of-denom-pos'*: $\text{snd } (\text{quotient-of } r) > 0$
 $\langle \text{proof} \rangle$

lemma *quotient-of-coprime*: $\text{quotient-of } r = (p, q) \implies \text{coprime } p \ q$
 $\langle \text{proof} \rangle$

lemma *quotient-of-inject*:
assumes $\text{quotient-of } a = \text{quotient-of } b$
shows $a = b$
 $\langle \text{proof} \rangle$

lemma *quotient-of-inject-eq*: $\text{quotient-of } a = \text{quotient-of } b \longleftrightarrow a = b$
 $\langle \text{proof} \rangle$

96.1.4 Various

lemma *Fract-of-int-quotient*: $\text{Fract } k \ l = \text{of-int } k \ / \ \text{of-int } l$
 $\langle \text{proof} \rangle$

lemma *Fract-add-one*: $n \neq 0 \implies \text{Fract } (m + n) \ n = \text{Fract } m \ n + 1$
 $\langle \text{proof} \rangle$

lemma *quotient-of-div*:
assumes $r: \text{quotient-of } r = (n, d)$
shows $r = \text{of-int } n \ / \ \text{of-int } d$
 $\langle \text{proof} \rangle$

lemma *Fract-quotient-of [simp]*: $\text{Fract } (\text{fst } (\text{quotient-of } r)) \ (\text{snd } (\text{quotient-of } r))$
 $= r$
 $\langle \text{proof} \rangle$

96.1.5 The ordered field of rational numbers

lift-definition *positive* :: $\text{rat} \Rightarrow \text{bool}$
is $\lambda x. 0 < \text{fst } x * \text{snd } x$
 $\langle \text{proof} \rangle$

lemma *positive-zero*: $\neg \text{positive } 0$
 $\langle \text{proof} \rangle$

lemma *positive-add*: $\text{positive } x \implies \text{positive } y \implies \text{positive } (x + y)$
 $\langle \text{proof} \rangle$

lemma *positive-mult*: $\text{positive } x \implies \text{positive } y \implies \text{positive } (x * y)$
 $\langle \text{proof} \rangle$

lemma *positive-minus*: $\neg \text{positive } x \implies x \neq 0 \implies \text{positive } (-x)$
 $\langle \text{proof} \rangle$

instantiation *rat* :: *linordered-field*
begin

definition $x < y \longleftrightarrow \text{positive } (y - x)$

definition $x \leq y \longleftrightarrow x < y \vee x = y$ **for** $x\ y :: \text{rat}$

definition $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$ **for** $a :: \text{rat}$

definition $\text{sgn } a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } -1)$ **for** $a :: \text{rat}$

instance

$\langle \text{proof} \rangle$

end

instantiation $\text{rat} :: \text{distrib-lattice}$

begin

definition $(\text{inf} :: \text{rat} \Rightarrow \text{rat} \Rightarrow \text{rat}) = \text{min}$

definition $(\text{sup} :: \text{rat} \Rightarrow \text{rat} \Rightarrow \text{rat}) = \text{max}$

instance

$\langle \text{proof} \rangle$

end

lemma *positive-rat*: $\text{positive } (\text{Fract } a\ b) \longleftrightarrow 0 < a * b$

$\langle \text{proof} \rangle$

lemma *less-rat* [*simp*]:

$b \neq 0 \implies d \neq 0 \implies \text{Fract } a\ b < \text{Fract } c\ d \longleftrightarrow (a * d) * (b * d) < (c * b) * (b * d)$

$\langle \text{proof} \rangle$

lemma *le-rat* [*simp*]:

$b \neq 0 \implies d \neq 0 \implies \text{Fract } a\ b \leq \text{Fract } c\ d \longleftrightarrow (a * d) * (b * d) \leq (c * b) * (b * d)$

$\langle \text{proof} \rangle$

lemma *abs-rat* [*simp*, *code*]: $|\text{Fract } a\ b| = \text{Fract } |a|\ |b|$

$\langle \text{proof} \rangle$

lemma *sgn-rat* [*simp*, *code*]: $\text{sgn } (\text{Fract } a\ b) = \text{of-int } (\text{sgn } a * \text{sgn } b)$

$\langle \text{proof} \rangle$

lemma *Rat-induct-pos* [*case-names* *Fract*, *induct type*: *rat*]:

assumes *step*: $\bigwedge a\ b. 0 < b \implies P\ (\text{Fract } a\ b)$

shows $P\ q$

$\langle \text{proof} \rangle$

lemma *zero-less-Fract-iff*: $0 < b \implies 0 < \text{Fract } a \ b \longleftrightarrow 0 < a$
 ⟨proof⟩

lemma *Fract-less-zero-iff*: $0 < b \implies \text{Fract } a \ b < 0 \longleftrightarrow a < 0$
 ⟨proof⟩

lemma *zero-le-Fract-iff*: $0 < b \implies 0 \leq \text{Fract } a \ b \longleftrightarrow 0 \leq a$
 ⟨proof⟩

lemma *Fract-le-zero-iff*: $0 < b \implies \text{Fract } a \ b \leq 0 \longleftrightarrow a \leq 0$
 ⟨proof⟩

lemma *one-less-Fract-iff*: $0 < b \implies 1 < \text{Fract } a \ b \longleftrightarrow b < a$
 ⟨proof⟩

lemma *Fract-less-one-iff*: $0 < b \implies \text{Fract } a \ b < 1 \longleftrightarrow a < b$
 ⟨proof⟩

lemma *one-le-Fract-iff*: $0 < b \implies 1 \leq \text{Fract } a \ b \longleftrightarrow b \leq a$
 ⟨proof⟩

lemma *Fract-le-one-iff*: $0 < b \implies \text{Fract } a \ b \leq 1 \longleftrightarrow a \leq b$
 ⟨proof⟩

96.1.6 Rationals are an Archimedean field

lemma *rat-floor-lemma*: $\text{of-int } (a \text{ div } b) \leq \text{Fract } a \ b \wedge \text{Fract } a \ b < \text{of-int } (a \text{ div } b + 1)$
 ⟨proof⟩

instance *rat* :: *archimedean-field*
 ⟨proof⟩

instantiation *rat* :: *floor-ceiling*
begin

definition *floor-rat* :: *rat* \Rightarrow *int*
where $[x] = (\text{THE } z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1))$ **for** $x :: \text{rat}$

instance
 ⟨proof⟩

end

lemma *floor-Fract [simp]*: $\lfloor \text{Fract } a \ b \rfloor = a \text{ div } b$
 ⟨proof⟩

96.2 Linear arithmetic setup

⟨ML⟩

96.3 Embedding from Rationals to other Fields

context *field-char-0*

begin

lift-definition *of-rat* :: *rat* \Rightarrow 'a
is $\lambda x.$ *of-int* (fst *x*) / *of-int* (snd *x*)
 ⟨*proof*⟩

end

lemma *of-rat-rat*: $b \neq 0 \implies \text{of-rat } (\text{Fract } a \ b) = \text{of-int } a \ / \ \text{of-int } b$
 ⟨*proof*⟩

lemma *of-rat-0* [*simp*]: *of-rat* 0 = 0
 ⟨*proof*⟩

lemma *of-rat-1* [*simp*]: *of-rat* 1 = 1
 ⟨*proof*⟩

lemma *of-rat-add*: *of-rat* (*a* + *b*) = *of-rat* *a* + *of-rat* *b*
 ⟨*proof*⟩

lemma *of-rat-minus*: *of-rat* (− *a*) = − *of-rat* *a*
 ⟨*proof*⟩

lemma *of-rat-neg-one* [*simp*]: *of-rat* (− 1) = − 1
 ⟨*proof*⟩

lemma *of-rat-diff*: *of-rat* (*a* − *b*) = *of-rat* *a* − *of-rat* *b*
 ⟨*proof*⟩

lemma *of-rat-mult*: *of-rat* (*a* * *b*) = *of-rat* *a* * *of-rat* *b*
 ⟨*proof*⟩

lemma *of-rat-sum*: *of-rat* ($\sum a \in A. f \ a$) = ($\sum a \in A. \text{of-rat } (f \ a)$)
 ⟨*proof*⟩

lemma *of-rat-prod*: *of-rat* ($\prod a \in A. f \ a$) = ($\prod a \in A. \text{of-rat } (f \ a)$)
 ⟨*proof*⟩

lemma *nonzero-of-rat-inverse*: $a \neq 0 \implies \text{of-rat } (\text{inverse } a) = \text{inverse } (\text{of-rat } a)$
 ⟨*proof*⟩

lemma *of-rat-inverse*: (*of-rat* (inverse *a*) :: 'a::field-char-0) = inverse (of-rat *a*)
 ⟨*proof*⟩

lemma *nonzero-of-rat-divide*: $b \neq 0 \implies \text{of-rat } (a \ / \ b) = \text{of-rat } a \ / \ \text{of-rat } b$
 ⟨*proof*⟩

lemma *of-rat-divide*: $(\text{of-rat } (a / b) :: 'a::\text{field-char-0}) = \text{of-rat } a / \text{of-rat } b$
 $\langle \text{proof} \rangle$

lemma *of-rat-power*: $(\text{of-rat } (a ^ n) :: 'a::\text{field-char-0}) = \text{of-rat } a ^ n$
 $\langle \text{proof} \rangle$

lemma *of-rat-eq-iff* [simp]: $\text{of-rat } a = \text{of-rat } b \longleftrightarrow a = b$
 $\langle \text{proof} \rangle$

lemma *of-rat-eq-0-iff* [simp]: $\text{of-rat } a = 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *zero-eq-of-rat-iff* [simp]: $0 = \text{of-rat } a \longleftrightarrow 0 = a$
 $\langle \text{proof} \rangle$

lemma *of-rat-eq-1-iff* [simp]: $\text{of-rat } a = 1 \longleftrightarrow a = 1$
 $\langle \text{proof} \rangle$

lemma *one-eq-of-rat-iff* [simp]: $1 = \text{of-rat } a \longleftrightarrow 1 = a$
 $\langle \text{proof} \rangle$

lemma *of-rat-less*: $(\text{of-rat } r :: 'a::\text{linordered-field}) < \text{of-rat } s \longleftrightarrow r < s$
 $\langle \text{proof} \rangle$

lemma *of-rat-less-eq*: $(\text{of-rat } r :: 'a::\text{linordered-field}) \leq \text{of-rat } s \longleftrightarrow r \leq s$
 $\langle \text{proof} \rangle$

lemma *of-rat-le-0-iff* [simp]: $(\text{of-rat } r :: 'a::\text{linordered-field}) \leq 0 \longleftrightarrow r \leq 0$
 $\langle \text{proof} \rangle$

lemma *zero-le-of-rat-iff* [simp]: $0 \leq (\text{of-rat } r :: 'a::\text{linordered-field}) \longleftrightarrow 0 \leq r$
 $\langle \text{proof} \rangle$

lemma *of-rat-le-1-iff* [simp]: $(\text{of-rat } r :: 'a::\text{linordered-field}) \leq 1 \longleftrightarrow r \leq 1$
 $\langle \text{proof} \rangle$

lemma *one-le-of-rat-iff* [simp]: $1 \leq (\text{of-rat } r :: 'a::\text{linordered-field}) \longleftrightarrow 1 \leq r$
 $\langle \text{proof} \rangle$

lemma *of-rat-less-0-iff* [simp]: $(\text{of-rat } r :: 'a::\text{linordered-field}) < 0 \longleftrightarrow r < 0$
 $\langle \text{proof} \rangle$

lemma *zero-less-of-rat-iff* [simp]: $0 < (\text{of-rat } r :: 'a::\text{linordered-field}) \longleftrightarrow 0 < r$
 $\langle \text{proof} \rangle$

lemma *of-rat-less-1-iff* [simp]: $(\text{of-rat } r :: 'a::\text{linordered-field}) < 1 \longleftrightarrow r < 1$
 $\langle \text{proof} \rangle$

lemma *one-less-of-rat-iff* [simp]: $1 < (\text{of-rat } r :: 'a::\text{linordered-field}) \longleftrightarrow 1 < r$

$\langle \text{proof} \rangle$

lemma *of-rat-eq-id* [simp]: $\text{of-rat} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *abs-of-rat* [simp]:
 $|\text{of-rat } r| = (\text{of-rat } |r| :: 'a :: \text{linordered-field})$
 $\langle \text{proof} \rangle$

Collapse nested embeddings.

lemma *of-rat-of-nat-eq* [simp]: $\text{of-rat } (\text{of-nat } n) = \text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma *of-rat-of-int-eq* [simp]: $\text{of-rat } (\text{of-int } z) = \text{of-int } z$
 $\langle \text{proof} \rangle$

lemma *of-rat-numeral-eq* [simp]: $\text{of-rat } (\text{numeral } w) = \text{numeral } w$
 $\langle \text{proof} \rangle$

lemma *of-rat-neg-numeral-eq* [simp]: $\text{of-rat } (- \text{numeral } w) = - \text{numeral } w$
 $\langle \text{proof} \rangle$

lemma *of-rat-floor* [simp]:
 $\lfloor \text{of-rat } r \rfloor = \lfloor r \rfloor$
 $\langle \text{proof} \rangle$

lemma *of-rat-ceiling* [simp]:
 $\lceil \text{of-rat } r \rceil = \lceil r \rceil$
 $\langle \text{proof} \rangle$

lemmas *zero-rat* = *Zero-rat-def*
lemmas *one-rat* = *One-rat-def*

abbreviation *rat-of-nat* :: $\text{nat} \Rightarrow \text{rat}$
where $\text{rat-of-nat} \equiv \text{of-nat}$

abbreviation *rat-of-int* :: $\text{int} \Rightarrow \text{rat}$
where $\text{rat-of-int} \equiv \text{of-int}$

96.4 The Set of Rational Numbers

context *field-char-0*
begin

definition *Rats* :: $'a \text{ set } (\mathbb{Q})$
where $\mathbb{Q} = \text{range of-rat}$

end

lemma *Rats-cases* [*cases set: Rats*]:
assumes $q \in \mathbb{Q}$
obtains (*of-rat*) r **where** $q = \text{of-rat } r$
 $\langle \text{proof} \rangle$

lemma *Rats-cases'*:
assumes $(x :: 'a :: \text{field-char-0}) \in \mathbb{Q}$
obtains $a\ b$ **where** $b > 0$ *coprime* $a\ b$ $x = \text{of-int } a / \text{of-int } b$
 $\langle \text{proof} \rangle$

lemma *Rats-of-rat* [*simp*]: $\text{of-rat } r \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-of-int* [*simp*]: $\text{of-int } z \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Ints-subset-Rats*: $\mathbb{Z} \subseteq \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-of-nat* [*simp*]: $\text{of-nat } n \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Nats-subset-Rats*: $\mathbb{N} \subseteq \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-number-of* [*simp*]: *numeral* $w \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-0* [*simp*]: $0 \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-1* [*simp*]: $1 \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-add* [*simp*]: $a \in \mathbb{Q} \implies b \in \mathbb{Q} \implies a + b \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-minus-iff* [*simp*]: $- a \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-diff* [*simp*]: $a \in \mathbb{Q} \implies b \in \mathbb{Q} \implies a - b \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-mult* [*simp*]: $a \in \mathbb{Q} \implies b \in \mathbb{Q} \implies a * b \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-inverse* [*simp*]: $a \in \mathbb{Q} \implies \text{inverse } a \in \mathbb{Q}$
for $a :: 'a :: \text{field-char-0}$
 $\langle \text{proof} \rangle$

lemma *Rats-divide* [simp]: $a \in \mathbb{Q} \implies b \in \mathbb{Q} \implies a / b \in \mathbb{Q}$
for $a\ b :: 'a::\text{field-char-0}$
 ⟨proof⟩

lemma *Rats-power* [simp]: $a \in \mathbb{Q} \implies a ^ n \in \mathbb{Q}$
for $a :: 'a::\text{field-char-0}$
 ⟨proof⟩

lemma *Rats-sum* [intro]: $(\bigwedge x. x \in A \implies f\ x \in \mathbb{Q}) \implies \text{sum } f\ A \in \mathbb{Q}$
 ⟨proof⟩

lemma *Rats-prod* [intro]: $(\bigwedge x. x \in A \implies f\ x \in \mathbb{Q}) \implies \text{prod } f\ A \in \mathbb{Q}$
 ⟨proof⟩

lemma *Rats-induct* [case-names of-rat, induct set: Rats]: $q \in \mathbb{Q} \implies (\bigwedge r. P\ (\text{of-rat } r)) \implies P\ q$
 ⟨proof⟩

lemma *Rats-infinite*: $\neg \text{finite } \mathbb{Q}$
 ⟨proof⟩

lemma *Rats-add-iff*: $a \in \mathbb{Q} \vee b \in \mathbb{Q} \implies a+b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
 ⟨proof⟩

lemma *Rats-diff-iff*: $a \in \mathbb{Q} \vee b \in \mathbb{Q} \implies a-b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
 ⟨proof⟩

lemma *Rats-mult-iff*: $a \in \mathbb{Q}-\{0\} \vee b \in \mathbb{Q}-\{0\} \implies a*b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
 ⟨proof⟩

lemma *Rats-inverse-iff* [simp]: $\text{inverse } a \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q}$
 ⟨proof⟩

lemma *Rats-divide-iff*: $a \in \mathbb{Q}-\{0\} \vee b \in \mathbb{Q}-\{0\} \implies a/b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
 ⟨proof⟩

96.5 Implementation of rational numbers as pairs of integers

Formal constructor

definition *Frct* :: $\text{int} \times \text{int} \Rightarrow \text{rat}$
where [simp]: $\text{Frct } p = \text{Fract } (\text{fst } p) (\text{snd } p)$

lemma [code abstype]: $\text{Frct } (\text{quotient-of } q) = q$
 ⟨proof⟩

Numerals

declare *quotient-of-Fract* [code abstract]

definition *of-int* :: *int* \Rightarrow *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)
 ⟨*proof*⟩

lemma *quotient-of-rat-of-int* [*simp*]: *quotient-of* (*rat-of-int* *i*) = (*i*, 1)
 ⟨*proof*⟩

lemma *quotient-of-rat-of-nat* [*simp*]: *quotient-of* (*rat-of-nat* *i*) = (*int* *i*, 1)
 ⟨*proof*⟩

lemma [*code-unfold*]: *numeral* *k* = *Rat.of-int* (*numeral* *k*)
 ⟨*proof*⟩

lemma [*code-unfold*]: $-$ *numeral* *k* = *Rat.of-int* ($-$ *numeral* *k*)
 ⟨*proof*⟩

lemma *Frct-code-post* [*code-post*]:
 Frct (0, *a*) = 0
 Frct (*a*, 0) = 0
 Frct (1, 1) = 1
 Frct (*numeral* *k*, 1) = *numeral* *k*
 Frct (1, *numeral* *k*) = 1 / *numeral* *k*
 Frct (*numeral* *k*, *numeral* *l*) = *numeral* *k* / *numeral* *l*
 Frct ($-$ *a*, *b*) = $-$ *Frct* (*a*, *b*)
 Frct (*a*, $-$ *b*) = $-$ *Frct* (*a*, *b*)
 $-$ ($-$ *Frct* *q*) = *Frct* *q*
 ⟨*proof*⟩

Operations

lemma *rat-zero-code* [*code abstract*]: *quotient-of* 0 = (0, 1)
 ⟨*proof*⟩

lemma *rat-one-code* [*code abstract*]: *quotient-of* 1 = (1, 1)
 ⟨*proof*⟩

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*))
 ⟨*proof*⟩

lemma *rat-uminus-code* [*code abstract*]:
 quotient-of ($-$ *p*) = (let (*a*, *b*) = *quotient-of* *p* in $-$ *a*, *b*)
 ⟨*proof*⟩

lemma *rat-minus-code* [*code abstract*]:

quotient-of ($p - q$) =
 (let (a, c) = *quotient-of* p ; (b, d) = *quotient-of* q
 in *normalize* ($a * d - b * c, c * d$))
 ⟨*proof*⟩

lemma *rat-times-code* [*code abstract*]:

quotient-of ($p * q$) =
 (let (a, c) = *quotient-of* p ; (b, d) = *quotient-of* q
 in *normalize* ($a * b, c * d$))
 ⟨*proof*⟩

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*⟩

lemma *rat-divide-code* [*code abstract*]:

quotient-of (p / q) =
 (let (a, c) = *quotient-of* p ; (b, d) = *quotient-of* q
 in *normalize* ($a * d, c * b$))
 ⟨*proof*⟩

lemma *rat-abs-code* [*code abstract*]:

quotient-of $|p|$ = (let (a, b) = *quotient-of* p in ($|a|, b$))
 ⟨*proof*⟩

lemma *rat-sgn-code* [*code abstract*]: *quotient-of* (*sgn* p) = (*sgn* (*fst* (*quotient-of* p)), 1)
 ⟨*proof*⟩

lemma *rat-floor-code* [*code*]: $\lfloor p \rfloor$ = (let (a, b) = *quotient-of* p in $a \text{ div } b$)
 ⟨*proof*⟩

instantiation *rat* :: *equal*

begin

definition [*code*]: *HOL.equal* $a \ b \longleftrightarrow \text{quotient-of } a = \text{quotient-of } b$

instance

⟨*proof*⟩

lemma *rat-eq-refl* [*code nbe*]: *HOL.equal* ($r :: \text{rat}$) $r \longleftrightarrow \text{True}$
 ⟨*proof*⟩

end

lemma *rat-less-eq-code* [*code*]:

$p \leq q \longleftrightarrow (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q \text{ in } a * d \leq c * b)$
 $\langle \text{proof} \rangle$

lemma *rat-less-code* [code]:

$p < q \longleftrightarrow (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q \text{ in } a * d < c * b)$
 $\langle \text{proof} \rangle$

lemma [code]: *of-rat* $p = (\text{let } (a, b) = \text{quotient-of } p \text{ in of-int } a / \text{of-int } b)$
 $\langle \text{proof} \rangle$

Quickcheck

context

includes *term-syntax*

begin

definition

valterm-fract :: $\text{int} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term}) \Rightarrow$
 $\text{int} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term}) \Rightarrow$
 $\text{rat} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term})$
where [code-unfold]: *valterm-fract* $k\ l = \text{Code-Evaluation.valtermify Fract } \{\cdot\} k$
 $\{\cdot\} l$

end

instantiation *rat* :: *random*

begin

context

includes *state-combinator-syntax*

begin

definition

Quickcheck-Random.random $i =$
Quickcheck-Random.random $i \circ \rightarrow (\lambda \text{num}. \text{Random.range } i \circ \rightarrow (\lambda \text{denom}. \text{Pair}$
 $(\text{let } j = \text{int-of-integer } (\text{integer-of-natural } (\text{denom} + 1))$
 $\text{in valterm-fract num } (j, \lambda u. \text{Code-Evaluation.term-of } j))))$

instance $\langle \text{proof} \rangle$

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 ⟨proof⟩

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, λ-. Code-Evaluation.term-of j))) d) d

instance ⟨proof⟩

end

instance rat :: partial-term-of ⟨proof⟩

lemma [code]:
  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.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))
      partial-term-of (ty :: rat itself) (Quickcheck-Narrowing.Narrowing-variable p tt)
    ≡
    Code-Evaluation.Free (STR "-") (Typerep.Typerep (STR "Rat.rat") [])
    ⟨proof⟩

```


instantiation *rat* :: *narrowing*
begin

definition

narrowing =
 Quickcheck-Narrowing.apply
 (*Quickcheck-Narrowing.apply*
 (*Quickcheck-Narrowing.cons* ($\lambda \text{nom denom. Fract nom denom}$)) *narrowing*)
narrowing

instance $\langle \text{proof} \rangle$

end

96.6 Setup for Nitpick

$\langle ML \rangle$

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

96.7 Float syntax

syntax *-Float* :: *float-const* \Rightarrow 'a ($\langle \langle \text{open-block notation} = \langle \text{literal number} \rangle \rangle \rangle$)

$\langle ML \rangle$

Test:

lemma $123.456 = -111.111 + 200 + 30 + 4 + 5/10 + 6/100 + (7/1000::\text{rat})$
 $\langle \text{proof} \rangle$

96.8 Hiding implementation details

hide-const (*open*) *normalize positive*

lifting-update *rat.lifting*

lifting-forget *rat.lifting*

end

97 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.

97.1 Preliminary lemmas

Useful in convergence arguments

lemma *inverse-of-nat-le*:

fixes $n::nat$ **shows** $\llbracket n \leq m; n \neq 0 \rrbracket \implies 1 / \text{of-nat } m \leq (1::'a::\text{linordered-field}) / \text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma *add-diff-add*: $(a + c) - (b + d) = (a - b) + (c - d)$

for $a\ b\ c\ d :: 'a::\text{ab-group-add}$
 $\langle \text{proof} \rangle$

lemma *minus-diff-minus*: $- a - - b = - (a - b)$

for $a\ b :: 'a::\text{ab-group-add}$
 $\langle \text{proof} \rangle$

lemma *mult-diff-mult*: $(x * y - a * b) = x * (y - b) + (x - a) * b$

for $x\ y\ a\ b :: 'a::\text{ring}$
 $\langle \text{proof} \rangle$

lemma *inverse-diff-inverse*:

fixes $a\ b :: 'a::\text{division-ring}$
assumes $a \neq 0$ **and** $b \neq 0$
shows $\text{inverse } a - \text{inverse } b = - (\text{inverse } a * (a - b) * \text{inverse } b)$
 $\langle \text{proof} \rangle$

lemma *obtain-pos-sum*:

fixes $r :: rat$ **assumes** $r: 0 < r$
obtains $s\ t$ **where** $0 < s$ **and** $0 < t$ **and** $r = s + t$
 $\langle \text{proof} \rangle$

97.2 Sequences that converge to zero

definition *vanishes* :: $(nat \Rightarrow rat) \Rightarrow bool$

where $\text{vanishes } X \longleftrightarrow (\forall r > 0. \exists k. \forall n \geq k. |X\ n| < r)$

lemma *vanishesI*: $(\bigwedge r. 0 < r \implies \exists k. \forall n \geq k. |X\ n| < r) \implies \text{vanishes } X$

$\langle \text{proof} \rangle$

lemma *vanishesD*: $\text{vanishes } X \implies 0 < r \implies \exists k. \forall n \geq k. |X\ n| < r$

$\langle \text{proof} \rangle$

lemma *vanishes-const* [*simp*]: *vanishes* $(\lambda n. c) \longleftrightarrow c = 0$
 $\langle proof \rangle$

lemma *vanishes-minus*: *vanishes* $X \implies \text{vanishes } (\lambda n. - X\ n)$
 $\langle proof \rangle$

lemma *vanishes-add*:
assumes X : *vanishes* X
and Y : *vanishes* Y
shows *vanishes* $(\lambda n. X\ n + Y\ n)$
 $\langle proof \rangle$

lemma *vanishes-diff*:
assumes *vanishes* X *vanishes* Y
shows *vanishes* $(\lambda n. X\ n - Y\ n)$
 $\langle proof \rangle$

lemma *vanishes-mult-bounded*:
assumes X : $\exists a > 0. \forall n. |X\ n| < a$
assumes Y : *vanishes* $(\lambda n. Y\ n)$
shows *vanishes* $(\lambda n. X\ n * Y\ n)$
 $\langle proof \rangle$

97.3 Cauchy sequences

definition *cauchy* :: $(nat \Rightarrow rat) \Rightarrow bool$
where *cauchy* $X \longleftrightarrow (\forall r > 0. \exists k. \forall m \geq k. \forall n \geq k. |X\ m - X\ n| < r)$

lemma *cauchyI*: $(\bigwedge r. 0 < r \implies \exists k. \forall m \geq k. \forall n \geq k. |X\ m - X\ n| < r) \implies \text{cauchy } X$
 $\langle proof \rangle$

lemma *cauchyD*: *cauchy* $X \implies 0 < r \implies \exists k. \forall m \geq k. \forall n \geq k. |X\ m - X\ n| < r$
 $\langle proof \rangle$

lemma *cauchy-const* [*simp*]: *cauchy* $(\lambda n. x)$
 $\langle proof \rangle$

lemma *cauchy-add* [*simp*]:
assumes X : *cauchy* X **and** Y : *cauchy* Y
shows *cauchy* $(\lambda n. X\ n + Y\ n)$
 $\langle proof \rangle$

lemma *cauchy-minus* [*simp*]:
assumes X : *cauchy* X
shows *cauchy* $(\lambda n. - X\ n)$
 $\langle proof \rangle$

lemma *cauchy-diff* [*simp*]:

assumes *cauchy* X *cauchy* Y
shows *cauchy* $(\lambda n. X\ n - Y\ n)$
 $\langle \text{proof} \rangle$

lemma *cauchy-imp-bounded*:
assumes *cauchy* X
shows $\exists b > 0. \forall n. |X\ n| < b$
 $\langle \text{proof} \rangle$

lemma *cauchy-mult* [*simp*]:
assumes X : *cauchy* X **and** Y : *cauchy* Y
shows *cauchy* $(\lambda n. X\ n * Y\ n)$
 $\langle \text{proof} \rangle$

lemma *cauchy-not-vanishes-cases*:
assumes X : *cauchy* X
assumes nz : $\neg \text{vanishes } X$
shows $\exists b > 0. \exists k. (\forall n \geq k. b < -X\ n) \vee (\forall n \geq k. b < X\ n)$
 $\langle \text{proof} \rangle$

lemma *cauchy-not-vanishes*:
assumes X : *cauchy* X
and nz : $\neg \text{vanishes } X$
shows $\exists b > 0. \exists k. \forall n \geq k. b < |X\ n|$
 $\langle \text{proof} \rangle$

lemma *cauchy-inverse* [*simp*]:
assumes X : *cauchy* X
and nz : $\neg \text{vanishes } X$
shows *cauchy* $(\lambda n. \text{inverse } (X\ n))$
 $\langle \text{proof} \rangle$

lemma *vanishes-diff-inverse*:
assumes X : *cauchy* X $\neg \text{vanishes } X$
and Y : *cauchy* Y $\neg \text{vanishes } Y$
and XY : *vanishes* $(\lambda n. X\ n - Y\ n)$
shows *vanishes* $(\lambda n. \text{inverse } (X\ n) - \text{inverse } (Y\ n))$
 $\langle \text{proof} \rangle$

97.4 Equivalence relation on Cauchy sequences

definition *realrel* :: $(\text{nat} \Rightarrow \text{rat}) \Rightarrow (\text{nat} \Rightarrow \text{rat}) \Rightarrow \text{bool}$
where *realrel* = $(\lambda X\ Y. \text{cauchy } X \wedge \text{cauchy } Y \wedge \text{vanishes } (\lambda n. X\ n - Y\ n))$

lemma *realrelI* [*intro?*]: *cauchy* $X \Longrightarrow \text{cauchy } Y \Longrightarrow \text{vanishes } (\lambda n. X\ n - Y\ n) \Longrightarrow \text{realrel } X\ Y$
 $\langle \text{proof} \rangle$

lemma *realrel-refl*: *cauchy* $X \Longrightarrow \text{realrel } X\ X$

<proof>

lemma *symp-realrel*: *symp realrel*
<proof>

lemma *transp-realrel*: *transp realrel*
<proof>

lemma *part-equivp-realrel*: *part-equivp realrel*
<proof>

97.5 The field of real numbers

quotient-type *real* = *nat* \Rightarrow *rat* / *partial*: *realrel*
morphisms *rep-real* *Real*
<proof>

lemma *cr-real-eq*: *pcr-real* = ($\lambda x y. \text{cauchy } x \wedge \text{Real } x = y$)
<proof>

lemma *Real-induct* [*induct type*: *real*]:
assumes $\bigwedge X. \text{cauchy } X \Longrightarrow P (\text{Real } X)$
shows $P x$
<proof>

lemma *eq-Real*: $\text{cauchy } X \Longrightarrow \text{cauchy } Y \Longrightarrow \text{Real } X = \text{Real } Y \longleftrightarrow \text{vanishes } (\lambda n. X\ n - Y\ n)$
<proof>

lemma *Domainp-pcr-real* [*transfer-domain-rule*]: *Domainp pcr-real* = *cauchy*
<proof>

instantiation *real* :: *field*
begin

lift-definition *zero-real* :: *real* **is** $\lambda n. 0$
<proof>

lift-definition *one-real* :: *real* **is** $\lambda n. 1$
<proof>

lift-definition *plus-real* :: *real* \Rightarrow *real* \Rightarrow *real* **is** $\lambda X Y n. X\ n + Y\ n$
<proof>

lift-definition *uminus-real* :: *real* \Rightarrow *real* **is** $\lambda X n. - X\ n$
<proof>

lift-definition *times-real* :: *real* \Rightarrow *real* \Rightarrow *real* **is** $\lambda X Y n. X\ n * Y\ n$
<proof>

lift-definition *inverse-real* :: *real* \Rightarrow *real*
 is $\lambda X. \text{if vanishes } X \text{ then } (\lambda n. 0) \text{ else } (\lambda n. \text{inverse } (X \ n))$
 $\langle \text{proof} \rangle$

definition $x - y = x + - \ y$ **for** $x \ y :: \text{real}$

definition $x \text{ div } y = x * \text{inverse } y$ **for** $x \ y :: \text{real}$

lemma *add-Real*: $\text{cauchy } X \Longrightarrow \text{cauchy } Y \Longrightarrow \text{Real } X + \text{Real } Y = \text{Real } (\lambda n. X \ n + Y \ n)$
 $\langle \text{proof} \rangle$

lemma *minus-Real*: $\text{cauchy } X \Longrightarrow - \ \text{Real } X = \text{Real } (\lambda n. - \ X \ n)$
 $\langle \text{proof} \rangle$

lemma *diff-Real*: $\text{cauchy } X \Longrightarrow \text{cauchy } Y \Longrightarrow \text{Real } X - \text{Real } Y = \text{Real } (\lambda n. X \ n - Y \ n)$
 $\langle \text{proof} \rangle$

lemma *mult-Real*: $\text{cauchy } X \Longrightarrow \text{cauchy } Y \Longrightarrow \text{Real } X * \text{Real } Y = \text{Real } (\lambda n. X \ n * Y \ n)$
 $\langle \text{proof} \rangle$

lemma *inverse-Real*:
 $\text{cauchy } X \Longrightarrow \text{inverse } (\text{Real } X) = (\text{if vanishes } X \text{ then } 0 \text{ else } \text{Real } (\lambda n. \text{inverse } (X \ n)))$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

97.6 Positive reals

lift-definition *positive* :: *real* \Rightarrow *bool*
 is $\lambda X. \exists r > 0. \exists k. \forall n \geq k. r < X \ n$
 $\langle \text{proof} \rangle$

lemma *positive-Real*: $\text{cauchy } X \Longrightarrow \text{positive } (\text{Real } X) \longleftrightarrow (\exists r > 0. \exists k. \forall n \geq k. r < X \ n)$
 $\langle \text{proof} \rangle$

lemma *positive-zero*: $\neg \text{positive } 0$
 $\langle \text{proof} \rangle$

lemma *positive-add*:
 assumes *positive* x *positive* y **shows** *positive* $(x + y)$

$\langle proof \rangle$

lemma *positive-mult*:

assumes *positive x positive y* **shows** *positive (x * y)*

$\langle proof \rangle$

lemma *positive-minus*: $\neg \text{positive } x \implies x \neq 0 \implies \text{positive } (-x)$

$\langle proof \rangle$

instantiation *real :: linordered-field*

begin

definition $x < y \longleftrightarrow \text{positive } (y - x)$

definition $x \leq y \longleftrightarrow x < y \vee x = y$ **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

$\langle proof \rangle$

end

instantiation *real :: 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

$\langle proof \rangle$

end

lemma *of-nat-Real*: $\text{of-nat } x = \text{Real } (\lambda n. \text{of-nat } x)$

$\langle proof \rangle$

lemma *of-int-Real*: $\text{of-int } x = \text{Real } (\lambda n. \text{of-int } x)$

$\langle proof \rangle$

lemma *of-rat-Real*: $\text{of-rat } x = \text{Real } (\lambda n. x)$

$\langle proof \rangle$

instance *real :: archimedean-field*

$\langle proof \rangle$

instantiation *real* :: *floor-ceiling*
begin

definition [*code del*]: $\lfloor x :: \text{real} \rfloor = (\text{THE } z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1))$

instance
 $\langle \text{proof} \rangle$

end

97.7 Completeness

lemma *not-positive-Real*:

assumes *cauchy* *X* **shows** $\neg \text{positive } (\text{Real } X) \longleftrightarrow (\forall r > 0. \exists k. \forall n \geq k. X\ n \leq r)$ (**is** *?lhs* = *?rhs*)
 $\langle \text{proof} \rangle$

lemma *le-Real*:

assumes *cauchy* *X* *cauchy* *Y*
shows $\text{Real } X \leq \text{Real } Y = (\forall r > 0. \exists k. \forall n \geq k. X\ n \leq Y\ n + r)$
 $\langle \text{proof} \rangle$

lemma *le-RealI*:

assumes *Y*: *cauchy* *Y*
shows $\forall n. x \leq \text{of-rat } (Y\ n) \implies x \leq \text{Real } Y$
 $\langle \text{proof} \rangle$

lemma *Real-leI*:

assumes *X*: *cauchy* *X*
assumes *le*: $\forall n. \text{of-rat } (X\ n) \leq y$
shows $\text{Real } X \leq y$
 $\langle \text{proof} \rangle$

lemma *less-RealD*:

assumes *cauchy* *Y*
shows $x < \text{Real } Y \implies \exists n. x < \text{of-rat } (Y\ n)$
 $\langle \text{proof} \rangle$

lemma *of-nat-less-two-power* [*simp*]: $\text{of-nat } n < (2 :: 'a :: \text{linordered-idom}) ^ n$
 $\langle \text{proof} \rangle$

lemma *complete-real*:

fixes *S* :: *real set*
assumes $\exists x. x \in S$ **and** $\exists z. \forall x \in S. x \leq z$
shows $\exists y. (\forall x \in S. x \leq y) \wedge (\forall z. (\forall x \in S. x \leq z) \longrightarrow y \leq z)$
 $\langle \text{proof} \rangle$

instantiation *real* :: *linear-continuum*
begin

97.8 Supremum of a set of reals

definition $\text{Sup } X = (\text{LEAST } z :: \text{real}. \forall x \in X. x \leq z)$

definition $\text{Inf } X = - \text{Sup } (\text{uminus } ' X)$ for $X :: \text{real set}$

instance

$\langle \text{proof} \rangle$

end

97.9 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*

97.10 Embedding numbers into the Reals

abbreviation *real-of-nat* :: $\text{nat} \Rightarrow \text{real}$

where *real-of-nat* \equiv *of-nat*

abbreviation *real* :: $\text{nat} \Rightarrow \text{real}$

where *real* \equiv *of-nat*

abbreviation *real-of-int* :: $\text{int} \Rightarrow \text{real}$

where *real-of-int* \equiv *of-int*

abbreviation *real-of-rat* :: $\text{rat} \Rightarrow \text{real}$

where *real-of-rat* \equiv *of-rat*

declare [[*coercion-enabled*]]

declare [[*coercion of-nat* :: $\text{nat} \Rightarrow \text{int}$]]

declare [[*coercion of-nat* :: $\text{nat} \Rightarrow \text{real}$]]

declare [[*coercion of-int* :: $\text{int} \Rightarrow \text{real}$]]

declare [[*coercion-map map*]]

declare [[*coercion-map* $\lambda f g h x. g (h (f x))$]]

declare [[*coercion-map* $\lambda f g (x,y). (f x, g y)$]]

declare *of-int-eq-0-iff* [*algebra, presburger*]

declare *of-int-eq-1-iff* [*algebra, presburger*]

declare *of-int-eq-iff* [*algebra, presburger*]

declare *of-int-less-0-iff* [*algebra*, *presburger*]
declare *of-int-less-1-iff* [*algebra*, *presburger*]
declare *of-int-less-iff* [*algebra*, *presburger*]
declare *of-int-le-0-iff* [*algebra*, *presburger*]
declare *of-int-le-1-iff* [*algebra*, *presburger*]
declare *of-int-le-iff* [*algebra*, *presburger*]
declare *of-int-0-less-iff* [*algebra*, *presburger*]
declare *of-int-0-le-iff* [*algebra*, *presburger*]
declare *of-int-1-less-iff* [*algebra*, *presburger*]
declare *of-int-1-le-iff* [*algebra*, *presburger*]

lemma *int-less-real-le*: $n < m \longleftrightarrow \text{real-of-int } n + 1 \leq \text{real-of-int } m$
 $\langle \text{proof} \rangle$

lemma *int-le-real-less*: $n \leq m \longleftrightarrow \text{real-of-int } n < \text{real-of-int } m + 1$
 $\langle \text{proof} \rangle$

lemma (*in field-char-0*) *of-int-div-aux*:
 $(\text{of-int } x) / (\text{of-int } d) =$
 $\text{of-int } (x \text{ div } d) + (\text{of-int } (x \bmod d)) / (\text{of-int } d)$
 $\langle \text{proof} \rangle$

lemma *real-of-int-div*:
 $d \text{ dvd } n \implies \text{real-of-int } (n \text{ div } d) = \text{real-of-int } n / \text{real-of-int } d$ **for** $d \text{ } n :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *real-of-int-div2*: $0 \leq \text{real-of-int } n / \text{real-of-int } x - \text{real-of-int } (n \text{ div } x)$
 $\langle \text{proof} \rangle$

lemma *real-of-int-div3*: $\text{real-of-int } n / \text{real-of-int } x - \text{real-of-int } (n \text{ div } x) \leq 1$
 $\langle \text{proof} \rangle$

lemma *real-of-int-div4*: $\text{real-of-int } (n \text{ div } x) \leq \text{real-of-int } n / \text{real-of-int } x$
 $\langle \text{proof} \rangle$

97.11 Embedding the Naturals into the Reals

lemma (*in field-char-0*) *of-nat-of-nat-div-aux*:
 $\text{of-nat } x / \text{of-nat } d = \text{of-nat } (x \text{ div } d) + \text{of-nat } (x \bmod d) / \text{of-nat } d$
 $\langle \text{proof} \rangle$

lemma(*in field-char-0*) *of-nat-of-nat-div*: $d \text{ dvd } n \implies \text{of-nat}(n \text{ div } d) = \text{of-nat } n$
 $/ \text{of-nat } d$
 $\langle \text{proof} \rangle$

lemma (*in linordered-field*) *of-nat-div-le-of-nat*: $\text{of-nat } (n \text{ div } x) \leq \text{of-nat } n /$
 $\text{of-nat } x$
 $\langle \text{proof} \rangle$

lemma *real-of-card*: $\text{real } (\text{card } A) = \text{sum } (\lambda x. 1) A$
 ⟨proof⟩

lemma *nat-less-real-le*: $n < m \longleftrightarrow \text{real } n + 1 \leq \text{real } m$
 ⟨proof⟩

lemma *nat-le-real-less*: $n \leq m \longleftrightarrow \text{real } n < \text{real } m + 1$
 ⟨proof⟩

lemma *real-of-nat-div*: $d \text{ dvd } n \implies \text{real}(n \text{ div } d) = \text{real } n / \text{real } d$
 ⟨proof⟩

lemma *real-binomial-eq-mult-binomial-Suc*:
 assumes $k \leq n$
 shows $\text{real}(n \text{ choose } k) = (n + 1 - k) / (n + 1) * (\text{Suc } n \text{ choose } k)$
 ⟨proof⟩

97.12 The Archimedean Property of the Reals

Not actually the reals any more!

lemma *real-arch-inverse*:
 fixes $e::'a::\text{archimedean-field}$
 shows $0 < e \longleftrightarrow (\exists n::\text{nat}. n \neq 0 \wedge 0 < \text{inverse } (\text{real } n) \wedge \text{inverse } (\text{of-nat } n) < e)$
 ⟨proof⟩

lemma *reals-Archimedean3*:
 fixes $x::'a::\text{archimedean-field}$
 shows $0 < x \implies \forall y. \exists n. y < \text{of-nat } n * x$
 ⟨proof⟩

lemma *real-archimedian-rdiv-eq-0*:
 fixes $x::'a::\text{archimedean-field}$
 assumes $x \geq 0$ and $\bigwedge m::\text{nat}. m > 0 \implies \text{of-nat } m * x \leq c$
 shows $x = 0$
 ⟨proof⟩

lemma *inverse-Suc*: $\text{inverse } (\text{of-nat } (\text{Suc } n)) > (0::'a::\text{archimedean-field})$
 ⟨proof⟩

lemma *Archimedean-eventually-inverse*:
 fixes $\varepsilon::'a::\text{archimedean-field}$ shows $(\forall_F n \text{ in sequentially. } \text{inverse } (\text{of-nat } (\text{Suc } n)) < \varepsilon) \longleftrightarrow 0 < \varepsilon$
 (is ?lhs=?rhs)
 ⟨proof⟩

On the relationship between two different ways of converting to 0

lemma *Inter-eq-Inter-inverse-Suc*:
 assumes $\bigwedge r' r. r' < r \implies A \ r' \subseteq A \ r$

shows $\bigcap (A \text{ ‘ } \{0 < ..\}) = (\bigcap n. A(\text{inverse}(\text{Suc } n)))$
 $\langle \text{proof} \rangle$

97.13 Rationals

lemma *Rats-abs-iff[simp]*:
 $|(x :: \text{real})| \in \mathbb{Q} \longleftrightarrow x \in \mathbb{Q}$
 $\langle \text{proof} \rangle$

lemma *Rats-eq-int-div-int*: $\mathbb{Q} = \{\text{real-of-int } i / \text{real-of-int } j \mid i \text{ j. } j \neq 0\}$ (is - = ?S)
 $\langle \text{proof} \rangle$

lemma *Rats-eq-int-div-nat*: $\mathbb{Q} = \{\text{real-of-int } i / \text{real } n \mid i \text{ n. } n \neq 0\}$
 $\langle \text{proof} \rangle$

lemma *Rats-abs-nat-div-natE*:
assumes $x \in \mathbb{Q}$
obtains $m \text{ n} :: \text{nat}$ **where** $n \neq 0$ **and** $|x| = \text{real } m / \text{real } n$ **and** *coprime* $m \text{ n}$
 $\langle \text{proof} \rangle$

97.14 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 $\exists r \in \mathbb{Q}. x < r \wedge r < y$
 $\langle \text{proof} \rangle$

lemma *of-rat-dense*:
fixes $x \text{ y} :: \text{real}$
assumes $x < y$
shows $\exists q :: \text{rat}. x < \text{of-rat } q \wedge \text{of-rat } q < y$
 $\langle \text{proof} \rangle$

97.15 Numerals and Arithmetic

$\langle \text{ML} \rangle$

97.16 Simprules combining $x + y$ and 0

lemma *real-add-minus-iff [simp]*: $x + - a = 0 \longleftrightarrow x = a$
for $x \text{ a} :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *real-add-less-0-iff*: $x + y < 0 \longleftrightarrow y < - x$

for $x\ y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *real-0-less-add-iff*: $0 < x + y \longleftrightarrow -x < y$
for $x\ y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *real-add-le-0-iff*: $x + y \leq 0 \longleftrightarrow y \leq -x$
for $x\ y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *real-0-le-add-iff*: $0 \leq x + y \longleftrightarrow -x \leq y$
for $x\ y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *mult-ge1-I*: $\llbracket x \geq 1; y \geq 1 \rrbracket \implies x * y \geq (1 :: \text{real})$
 $\langle \text{proof} \rangle$

97.17 Lemmas about powers

lemma *two-realpow-ge-one*: $(1 :: \text{real}) \leq 2 \wedge n$
 $\langle \text{proof} \rangle$

declare *sum-squares-eq-zero-iff* [simp] *sum-power2-eq-zero-iff* [simp]

lemma *real-minus-mult-self-le* [simp]: $-(u * u) \leq x * x$
for $u\ x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *realpow-square-minus-le* [simp]: $-u^2 \leq x^2$
for $u\ x :: \text{real}$
 $\langle \text{proof} \rangle$

97.18 Density of the Reals

lemma *field-lbound-gt-zero*: $0 < d1 \implies 0 < d2 \implies \exists e. 0 < e \wedge e < d1 \wedge e < d2$
for $d1\ d2 :: 'a::\text{linordered-field}$
 $\langle \text{proof} \rangle$

lemma *field-less-half-sum*: $x < y \implies x < (x + y) / 2$
for $x\ y :: 'a::\text{linordered-field}$
 $\langle \text{proof} \rangle$

lemma *field-sum-of-halves*: $x / 2 + x / 2 = x$
for $x :: 'a::\text{linordered-field}$
 $\langle \text{proof} \rangle$

97.19 Archimedean properties and useful consequences

Bernoulli’s inequality

proposition *Bernoulli-inequality:*

fixes $x :: 'a :: \text{linordered-field}$
assumes $-1 \leq x$
shows $1 + \text{of-nat } n * x \leq (1 + x) ^ n$
 $\langle \text{proof} \rangle$

corollary *Bernoulli-inequality-even:*

fixes $x :: 'a :: \text{linordered-field}$
assumes *even* n
shows $1 + \text{of-nat } n * x \leq (1 + x) ^ n$
 $\langle \text{proof} \rangle$

corollary *real-arch-pow:*

fixes $x :: \text{real}$
assumes $x: 1 < x$
shows $\exists n. y < x ^ n$
 $\langle \text{proof} \rangle$

corollary *real-arch-pow-inv:*

fixes $x y :: \text{real}$
assumes $y: y > 0$
and $x1: x < 1$
shows $\exists n. x ^ n < y$
 $\langle \text{proof} \rangle$

lemma *forall-pos-mono:*

$(\bigwedge d e::\text{real}. d < e \implies P d \implies P e) \implies$
 $(\bigwedge n::\text{nat}. n \neq 0 \implies P (\text{inverse } (\text{real } n))) \implies (\bigwedge e. 0 < e \implies P e)$
 $\langle \text{proof} \rangle$

lemma *forall-pos-mono-1:*

$(\bigwedge d e::\text{real}. d < e \implies P d \implies P e) \implies$
 $(\bigwedge n. P (\text{inverse } (\text{real } (\text{Suc } n)))) \implies 0 < e \implies P e$
 $\langle \text{proof} \rangle$

lemma *Archimedean-eventually-pow:*

fixes $x::\text{real}$
assumes $1 < x$
shows $\forall_F n \text{ in sequentially. } b < x ^ n$
 $\langle \text{proof} \rangle$

lemma *Archimedean-eventually-pow-inverse:*

fixes $x::\text{real}$
assumes $|x| < 1 \ \varepsilon > 0$
shows $\forall_F n \text{ in sequentially. } |x ^ n| < \varepsilon$
 $\langle \text{proof} \rangle$

97.20 Floor and Ceiling Functions from the Reals to the Integers

lemma *real-of-nat-less-numeral-iff* [simp]: $\text{real } n < \text{numeral } w \longleftrightarrow n < \text{numeral } w$
 for $n :: \text{nat}$
 <proof>

lemma *numeral-less-real-of-nat-iff* [simp]: $\text{numeral } w < \text{real } n \longleftrightarrow \text{numeral } w < n$
 for $n :: \text{nat}$
 <proof>

lemma *numeral-le-real-of-nat-iff* [simp]: $\text{numeral } n \leq \text{real } m \longleftrightarrow \text{numeral } n \leq m$
 for $m :: \text{nat}$
 <proof>

lemma *of-int-floor-cancel* [simp]: $\text{of-int } \lfloor x \rfloor = x \longleftrightarrow (\exists n :: \text{int}. x = \text{of-int } n)$
 <proof>

lemma *of-int-floor* [simp]: $a \in \mathbb{Z} \implies \text{of-int } (\text{floor } a) = a$
 <proof>

lemma *floor-frac* [simp]: $\lfloor \text{frac } r \rfloor = 0$
 <proof>

lemma *frac-1* [simp]: $\text{frac } 1 = 0$
 <proof>

lemma *frac-in-Rats-iff* [simp]:
 fixes $r :: 'a :: \{\text{floor-ceiling, field-char-0}\}$
 shows $\text{frac } r \in \mathbb{Q} \longleftrightarrow r \in \mathbb{Q}$
 <proof>

lemma *floor-eq*: $\text{real-of-int } n < x \implies x < \text{real-of-int } n + 1 \implies \lfloor x \rfloor = n$
 <proof>

lemma *floor-eq2*: $\text{real-of-int } n \leq x \implies x < \text{real-of-int } n + 1 \implies \lfloor x \rfloor = n$
 <proof>

lemma *floor-eq3*: $\text{real } n < x \implies x < \text{real } (\text{Suc } n) \implies \text{nat } \lfloor x \rfloor = n$
 <proof>

lemma *floor-eq4*: $\text{real } n \leq x \implies x < \text{real } (\text{Suc } n) \implies \text{nat } \lfloor x \rfloor = n$
 <proof>

lemma *real-of-int-floor-ge-diff-one* [simp]: $r - 1 \leq \text{real-of-int } \lfloor r \rfloor$
 <proof>

lemma *real-of-int-floor-gt-diff-one* [simp]: $r - 1 < \text{real-of-int } \lfloor r \rfloor$

$\langle \text{proof} \rangle$

lemma *real-of-int-floor-add-one-ge* [simp]: $r \leq \text{real-of-int } \lfloor r \rfloor + 1$
 $\langle \text{proof} \rangle$

lemma *real-of-int-floor-add-one-gt* [simp]: $r < \text{real-of-int } \lfloor r \rfloor + 1$
 $\langle \text{proof} \rangle$

lemma *floor-divide-real-eq-div*:
 assumes $0 \leq b$
 shows $\lfloor a / \text{real-of-int } b \rfloor = \lfloor a \rfloor \text{ div } b$
 $\langle \text{proof} \rangle$

lemma *floor-one-divide-eq-div-numeral* [simp]:
 $\lfloor 1 / \text{numeral } b :: \text{real} \rfloor = 1 \text{ div numeral } b$
 $\langle \text{proof} \rangle$

lemma *floor-minus-one-divide-eq-div-numeral* [simp]:
 $\lfloor - (1 / \text{numeral } b) :: \text{real} \rfloor = - 1 \text{ div numeral } b$
 $\langle \text{proof} \rangle$

lemma *floor-divide-eq-div-numeral* [simp]:
 $\lfloor \text{numeral } a / \text{numeral } b :: \text{real} \rfloor = \text{numeral } a \text{ div numeral } b$
 $\langle \text{proof} \rangle$

lemma *floor-minus-divide-eq-div-numeral* [simp]:
 $\lfloor - (\text{numeral } a / \text{numeral } b) :: \text{real} \rfloor = - \text{numeral } a \text{ div numeral } b$
 $\langle \text{proof} \rangle$

lemma *of-int-ceiling-cancel* [simp]: $\text{of-int } \lceil x \rceil = x \longleftrightarrow (\exists n :: \text{int}. x = \text{of-int } n)$
 $\langle \text{proof} \rangle$

lemma *of-int-ceiling* [simp]: $a \in \mathbb{Z} \implies \text{of-int } (\text{ceiling } a) = a$
 $\langle \text{proof} \rangle$

lemma *ceiling-eq*: $\text{of-int } n < x \implies x \leq \text{of-int } n + 1 \implies \lceil x \rceil = n + 1$
 $\langle \text{proof} \rangle$

lemma *of-int-ceiling-diff-one-le* [simp]: $\text{of-int } \lceil r \rceil - 1 \leq r$
 $\langle \text{proof} \rangle$

lemma *of-int-ceiling-le-add-one* [simp]: $\text{of-int } \lceil r \rceil \leq r + 1$
 $\langle \text{proof} \rangle$

lemma *ceiling-le*: $x \leq \text{of-int } a \implies \lceil x \rceil \leq a$
 $\langle \text{proof} \rangle$

lemma *ceiling-divide-eq-div*: $\lceil \text{of-int } a / \text{of-int } b \rceil = - (- a \text{ div } b)$
 $\langle \text{proof} \rangle$

lemma *ceiling-divide-eq-div-numeral* [simp]:
 $\lceil \text{numeral } a / \text{numeral } b :: \text{real} \rceil = - (- \text{numeral } a \text{ div numeral } b)$
 ⟨proof⟩

lemma *ceiling-minus-divide-eq-div-numeral* [simp]:
 $\lceil - (\text{numeral } a / \text{numeral } b :: \text{real}) \rceil = - (\text{numeral } a \text{ div numeral } b)$
 ⟨proof⟩

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 $x :: \text{real}$
 ⟨proof⟩

lemma *le-nat-floor*: $\text{real } x \leq a \implies x \leq \text{nat } \lfloor a \rfloor$
 ⟨proof⟩

lemma *le-mult-nat-floor*: $\text{nat } \lfloor a \rfloor * \text{nat } \lfloor b \rfloor \leq \text{nat } \lfloor a * b \rfloor$
 ⟨proof⟩

lemma *nat-ceiling-le-eq* [simp]: $\text{nat } \lceil x \rceil \leq a \longleftrightarrow x \leq \text{real } a$
 ⟨proof⟩

lemma *real-nat-ceiling-ge*: $x \leq \text{real } (\text{nat } \lceil x \rceil)$
 ⟨proof⟩

lemma *Rats-no-top-le*: $\exists q \in \mathbb{Q}. x \leq q$
for $x :: \text{real}$
 ⟨proof⟩

lemma *Rats-no-bot-less*: $\exists q \in \mathbb{Q}. q < x$ **for** $x :: \text{real}$
 ⟨proof⟩

lemma *floor-ceiling-diff-le*: $0 \leq r \implies \text{nat } \lfloor \text{real } k - r \rfloor \leq k - \text{nat } \lceil r \rceil$
 ⟨proof⟩

lemma *floor-ceiling-diff-le'*: $\text{nat } \lfloor r - \text{real } k \rfloor \leq \text{nat } \lceil r \rceil - k$
 ⟨proof⟩

lemma *ceiling-floor-diff-ge*: $\text{nat } \lceil r - \text{real } k \rceil \geq \text{nat } \lfloor r \rfloor - k$
 ⟨proof⟩

lemma *ceiling-floor-diff-ge'*: $r \leq k \implies \text{nat } \lceil r - \text{real } k \rceil \leq k - \text{nat } \lfloor r \rfloor$
 ⟨proof⟩

97.21 Exponentiation with floor

lemma *floor-power*:

assumes $x = \text{of-int } \lfloor x \rfloor$
shows $\lfloor x \wedge n \rfloor = \lfloor x \rfloor \wedge n$
 $\langle \text{proof} \rangle$

lemma *floor-numeral-power* [simp]: $\lfloor \text{numeral } x \wedge n \rfloor = \text{numeral } x \wedge n$
 $\langle \text{proof} \rangle$

lemma *ceiling-numeral-power* [simp]: $\lceil \text{numeral } x \wedge n \rceil = \text{numeral } x \wedge n$
 $\langle \text{proof} \rangle$

97.22 Implementation of rational real numbers

Formal constructor

definition *Ratreal* :: $\text{rat} \Rightarrow \text{real}$
where [code-abbrev, simp]: *Ratreal* = *real-of-rat*

code-datatype *Ratreal*

Quasi-Numerals

lemma [code-abbrev]:
 $\text{real-of-rat } (\text{numeral } k) = \text{numeral } k$
 $\text{real-of-rat } (- \text{numeral } k) = - \text{numeral } k$
 $\text{real-of-rat } (\text{rat-of-int } a) = \text{real-of-int } a$
 $\langle \text{proof} \rangle$

lemma [code-post]:
 $\text{real-of-rat } 0 = 0$
 $\text{real-of-rat } 1 = 1$
 $\text{real-of-rat } (- 1) = - 1$
 $\text{real-of-rat } (1 / \text{numeral } k) = 1 / \text{numeral } k$
 $\text{real-of-rat } (\text{numeral } k / \text{numeral } l) = \text{numeral } k / \text{numeral } l$
 $\text{real-of-rat } (- (1 / \text{numeral } k)) = - (1 / \text{numeral } k)$
 $\text{real-of-rat } (- (\text{numeral } k / \text{numeral } l)) = - (\text{numeral } k / \text{numeral } l)$
 $\langle \text{proof} \rangle$

Operations

lemma *zero-real-code* [code]: $0 = \text{Ratreal } 0$
 $\langle \text{proof} \rangle$

lemma *one-real-code* [code]: $1 = \text{Ratreal } 1$
 $\langle \text{proof} \rangle$

instantiation *real* :: *equal*
begin

definition *HOL.equal* $x y \longleftrightarrow x - y = 0$ **for** $x :: \text{real}$

instance $\langle \text{proof} \rangle$

lemma *real-equal-code* [code]: $HOL.equal\ (Ratreal\ x)\ (Ratreal\ y) \longleftrightarrow HOL.equal\ x\ y$
 ⟨proof⟩

lemma [code nbe]: $HOL.equal\ x\ x \longleftrightarrow True$
for $x :: real$
 ⟨proof⟩

end

lemma *real-less-eq-code* [code]: $Ratreal\ x \leq Ratreal\ y \longleftrightarrow x \leq y$
 ⟨proof⟩

lemma *real-less-code* [code]: $Ratreal\ x < Ratreal\ y \longleftrightarrow x < y$
 ⟨proof⟩

lemma *real-plus-code* [code]: $Ratreal\ x + Ratreal\ y = Ratreal\ (x + y)$
 ⟨proof⟩

lemma *real-times-code* [code]: $Ratreal\ x * Ratreal\ y = Ratreal\ (x * y)$
 ⟨proof⟩

lemma *real-uminus-code* [code]: $- Ratreal\ x = Ratreal\ (- x)$
 ⟨proof⟩

lemma *real-minus-code* [code]: $Ratreal\ x - Ratreal\ y = Ratreal\ (x - y)$
 ⟨proof⟩

lemma *real-inverse-code* [code]: $inverse\ (Ratreal\ x) = Ratreal\ (inverse\ x)$
 ⟨proof⟩

lemma *real-divide-code* [code]: $Ratreal\ x / Ratreal\ y = Ratreal\ (x / y)$
 ⟨proof⟩

lemma *real-floor-code* [code]: $\lfloor Ratreal\ x \rfloor = \lfloor x \rfloor$
 ⟨proof⟩

Quickcheck

context
includes *term-syntax*
begin

definition

$valterm-ratreal :: rat \times (unit \Rightarrow Code-Evaluation.term) \Rightarrow real \times (unit \Rightarrow Code-Evaluation.term)$
where [code-unfold]: $valterm-ratreal\ k = Code-Evaluation.valtermify\ Ratreal\ \{\cdot\}$
 k

end

instantiation *real* :: *random*
begin

context
includes *state-combinator-syntax*
begin

definition
 $\text{Quickcheck-Random.random } i = \text{Quickcheck-Random.random } i \circ \rightarrow (\lambda r. \text{Pair } (\text{valterm-ratreal } r))$

instance $\langle \text{proof} \rangle$

end

end

instantiation *real* :: *exhaustive*
begin

definition
 $\text{exhaustive-real } f \, d = \text{Quickcheck-Exhaustive.exhaustive } (\lambda r. f \, (\text{Ratreal } r)) \, d$

instance $\langle \text{proof} \rangle$

end

instantiation *real* :: *full-exhaustive*
begin

definition
 $\text{full-exhaustive-real } f \, d = \text{Quickcheck-Exhaustive.full-exhaustive } (\lambda r. f \, (\text{valterm-ratreal } r)) \, d$

instance $\langle \text{proof} \rangle$

end

instantiation *real* :: *narrowing*
begin

definition
 $\text{narrowing-real} = \text{Quickcheck-Narrowing.apply } (\text{Quickcheck-Narrowing.cons } \text{Ratreal}) \, \text{narrowing}$

instance $\langle \text{proof} \rangle$

end

97.23 Setup for Nitpick

$\langle ML \rangle$

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

97.24 Setup for SMT

$\langle ML \rangle$

lemma [z3-rule]:

$0 + x = x$
 $x + 0 = x$
 $0 * x = 0$
 $1 * x = x$
 $-x = -1 * x$
 $x + y = y + x$
for $x\ y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma [smt-arith-multiplication]:

fixes $A\ B :: \text{real}$ **and** $p\ n :: \text{real}$
assumes $A \leq B\ 0 < n\ p > 0$
shows $(A / n) * p \leq (B / n) * p$
 $\langle \text{proof} \rangle$

lemma [smt-arith-multiplication]:

fixes $A\ B :: \text{real}$ **and** $p\ n :: \text{real}$
assumes $A < B\ 0 < n\ p > 0$
shows $(A / n) * p < (B / n) * p$
 $\langle \text{proof} \rangle$

lemma [smt-arith-multiplication]:

fixes $A\ B :: \text{real}$ **and** $p\ n :: \text{int}$
assumes $A \leq B\ 0 < n\ p > 0$
shows $(A / n) * p \leq (B / n) * p$
 $\langle \text{proof} \rangle$

lemma [smt-arith-multiplication]:

fixes $A\ B :: \text{real}$ **and** $p\ n :: \text{int}$
assumes $A < B\ 0 < n\ p > 0$
shows $(A / n) * p < (B / n) * p$
 $\langle \text{proof} \rangle$

lemmas [smt-arith-multiplication] =

```

    verit-le-mono-div[THEN mult-left-mono, unfolded int-distrib, of - - ⟨nat (floor (-
:: real))⟩ ⟨nat (floor (- :: real))⟩]
    div-le-mono[THEN mult-left-mono, unfolded int-distrib, of - - ⟨nat (floor (- ::
real))⟩ ⟨nat (floor (- :: real))⟩]
    verit-le-mono-div-int[THEN mult-left-mono, unfolded int-distrib, of - - ⟨floor (-
:: real)⟩ ⟨floor (- :: real)⟩]
    zdiv-mono1[THEN mult-left-mono, unfolded int-distrib, of - - ⟨floor (- :: real)⟩
⟨floor (- :: real)⟩]
    arg-cong[of - - ⟨λa :: real. a / real (n::nat) * real (p::nat)⟩ for n p :: nat, THEN
sym]
    arg-cong[of - - ⟨λa :: real. a / real-of-int n * real-of-int p⟩ for n p :: int, THEN
sym]
    arg-cong[of - - ⟨λa :: real. a / n * p⟩ 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

```

97.25 Setup for Argo

⟨ML⟩

end

98 Topological Spaces

```

theory Topological-Spaces
  imports Main
begin

```

named-theorems continuous-intros structural introduction rules for continuity

98.1 Topological space

```

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 {}
  ⟨proof⟩

```

lemma *open-Un* [*continuous-intros, intro*]: $\text{open } S \implies \text{open } T \implies \text{open } (S \cup T)$
 ⟨*proof*⟩

lemma *open-UN* [*continuous-intros, intro*]: $\forall x \in A. \text{open } (B \ x) \implies \text{open } (\bigcup_{x \in A} B \ x)$
 ⟨*proof*⟩

lemma *open-Inter* [*continuous-intros, intro*]: $\text{finite } S \implies \forall T \in S. \text{open } T \implies \text{open } (\bigcap S)$
 ⟨*proof*⟩

lemma *open-INT* [*continuous-intros, intro*]: $\text{finite } A \implies \forall x \in A. \text{open } (B \ x) \implies \text{open } (\bigcap_{x \in A} B \ x)$
 ⟨*proof*⟩

lemma *openI*:
 assumes $\bigwedge x. x \in S \implies \exists T. \text{open } T \wedge x \in T \wedge T \subseteq S$
 shows $\text{open } S$
 ⟨*proof*⟩

lemma *open-subopen*: $\text{open } S \longleftrightarrow (\forall x \in S. \exists T. \text{open } T \wedge x \in T \wedge T \subseteq S)$
 ⟨*proof*⟩

lemma *closed-empty* [*continuous-intros, intro, simp*]: $\text{closed } \{\}$
 ⟨*proof*⟩

lemma *closed-Un* [*continuous-intros, intro*]: $\text{closed } S \implies \text{closed } T \implies \text{closed } (S \cup T)$
 ⟨*proof*⟩

lemma *closed-UNIV* [*continuous-intros, intro, simp*]: $\text{closed } \text{UNIV}$
 ⟨*proof*⟩

lemma *closed-Int* [*continuous-intros, intro*]: $\text{closed } S \implies \text{closed } T \implies \text{closed } (S \cap T)$
 ⟨*proof*⟩

lemma *closed-INT* [*continuous-intros, intro*]: $\forall x \in A. \text{closed } (B \ x) \implies \text{closed } (\bigcap_{x \in A} B \ x)$
 ⟨*proof*⟩

lemma *closed-Inter* [*continuous-intros, intro*]: $\forall S \in K. \text{closed } S \implies \text{closed } (\bigcap K)$
 ⟨*proof*⟩

lemma *closed-Union* [*continuous-intros, intro*]: $\text{finite } S \implies \forall T \in S. \text{closed } T \implies \text{closed } (\bigcup S)$
 ⟨*proof*⟩

lemma *closed-UN* [*continuous-intros, intro*]:
 $\text{finite } A \implies \forall x \in A. \text{closed } (B \ x) \implies \text{closed } (\bigcup_{x \in A} B \ x)$
 ⟨*proof*⟩

lemma *open-closed*: $\text{open } S \longleftrightarrow \text{closed } (- \ S)$
 ⟨*proof*⟩

lemma *closed-open*: $\text{closed } S \longleftrightarrow \text{open } (- \ S)$
 ⟨*proof*⟩

lemma *open-Diff* [*continuous-intros, intro*]: $\text{open } S \implies \text{closed } T \implies \text{open } (S - T)$
 ⟨*proof*⟩

lemma *closed-Diff* [*continuous-intros, intro*]: $\text{closed } S \implies \text{open } T \implies \text{closed } (S - T)$
 ⟨*proof*⟩

lemma *open-Compl* [*continuous-intros, intro*]: $\text{closed } S \implies \text{open } (- \ S)$
 ⟨*proof*⟩

lemma *closed-Compl* [*continuous-intros, intro*]: $\text{open } S \implies \text{closed } (- \ S)$
 ⟨*proof*⟩

lemma *open-Collect-neg*: $\text{closed } \{x. P \ x\} \implies \text{open } \{x. \neg P \ x\}$
 ⟨*proof*⟩

lemma *open-Collect-conj*:
assumes $\text{open } \{x. P \ x\} \text{ open } \{x. Q \ x\}$
shows $\text{open } \{x. P \ x \wedge Q \ x\}$
 ⟨*proof*⟩

lemma *open-Collect-disj*:
assumes $\text{open } \{x. P \ x\} \text{ open } \{x. Q \ x\}$
shows $\text{open } \{x. P \ x \vee Q \ x\}$
 ⟨*proof*⟩

lemma *open-Collect-ex*: $(\bigwedge i. \text{open } \{x. P \ i \ x\}) \implies \text{open } \{x. \exists i. P \ i \ x\}$
 ⟨*proof*⟩

lemma *open-Collect-imp*: $\text{closed } \{x. P \ x\} \implies \text{open } \{x. Q \ x\} \implies \text{open } \{x. P \ x \longrightarrow Q \ x\}$
 ⟨*proof*⟩

lemma *open-Collect-const*: $\text{open } \{x. P\}$
 ⟨*proof*⟩

lemma *closed-Collect-neg*: $\text{open } \{x. P \ x\} \implies \text{closed } \{x. \neg P \ x\}$
 ⟨*proof*⟩

lemma *closed-Collect-conj*:

assumes *closed* $\{x. P\ x\}$ *closed* $\{x. Q\ x\}$
shows *closed* $\{x. P\ x \wedge Q\ x\}$
 $\langle proof \rangle$

lemma *closed-Collect-disj*:

assumes *closed* $\{x. P\ x\}$ *closed* $\{x. Q\ x\}$
shows *closed* $\{x. P\ x \vee Q\ x\}$
 $\langle proof \rangle$

lemma *closed-Collect-all*: $(\bigwedge i. \text{closed } \{x. P\ i\ x\}) \implies \text{closed } \{x. \forall i. P\ i\ x\}$
 $\langle proof \rangle$

lemma *closed-Collect-imp*: $\text{open } \{x. P\ x\} \implies \text{closed } \{x. Q\ x\} \implies \text{closed } \{x. P\ x \longrightarrow Q\ x\}$
 $\langle proof \rangle$

lemma *closed-Collect-const*: *closed* $\{x. P\}$
 $\langle proof \rangle$

end

98.2 Hausdorff and other separation properties

class *t0-space* = *topological-space* +

assumes *t0-space*: $x \neq y \implies \exists U. \text{open } U \wedge \neg (x \in U \longleftrightarrow y \in U)$

class *t1-space* = *topological-space* +

assumes *t1-space*: $x \neq y \implies \exists U. \text{open } U \wedge x \in U \wedge y \notin U$

instance *t1-space* \subseteq *t0-space*
 $\langle proof \rangle$

context *t1-space* **begin**

lemma *separation-t1*: $x \neq y \longleftrightarrow (\exists U. \text{open } U \wedge x \in U \wedge y \notin U)$
 $\langle proof \rangle$

lemma *closed-singleton [iff]*: *closed* $\{a\}$
 $\langle proof \rangle$

lemma *closed-insert [continuous-intros, simp]*:

assumes *closed* S
shows *closed* $(\text{insert } a\ S)$
 $\langle proof \rangle$

lemma *finite-imp-closed*: *finite* $S \implies \text{closed } S$
 $\langle proof \rangle$

end

T2 spaces are also known as Hausdorff spaces.

class *t2-space* = *topological-space* +
assumes *hausdorff*: $x \neq y \implies \exists U V. \text{open } U \wedge \text{open } V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\}$

instance *t2-space* \subseteq *t1-space*
 $\langle \text{proof} \rangle$

lemma (in *t2-space*) *separation-t2*: $x \neq y \longleftrightarrow (\exists U V. \text{open } U \wedge \text{open } V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\})$
 $\langle \text{proof} \rangle$

lemma (in *t0-space*) *separation-t0*: $x \neq y \longleftrightarrow (\exists U. \text{open } U \wedge \neg (x \in U \longleftrightarrow y \in U))$
 $\langle \text{proof} \rangle$

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*: $\text{closed } S \implies y \notin S \implies \exists U V. \text{open } U \wedge \text{open } V \wedge y \in U \wedge S \subseteq V \wedge U \cap V = \{\}$

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*: $\text{closed } S \implies \text{closed } T \implies S \cap T = \{\} \implies \exists U V. \text{open } U \wedge \text{open } V \wedge S \subseteq U \wedge T \subseteq V \wedge U \cap V = \{\}$

T4 is stronger than T3, and weaker than metric.

instance *t4-space* \subseteq *t3-space*
 $\langle \text{proof} \rangle$

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 $x :: 'a$
 $\langle \text{proof} \rangle$

98.3 Generators for topologies

inductive *generate-topology* :: $'a \text{ set set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$ **for** $S :: 'a \text{ set set}$

where

UNIV: generate-topology *S 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** ($\bigwedge k. k \in K \implies \text{generate-topology } S \ k$)
 | *Basis*: generate-topology *S* *s* **if** $s \in S$

hide-fact (**open**) *UNIV Int UN Basis*

lemma generate-topology-Union:

($\bigwedge k. k \in I \implies \text{generate-topology } S \ (K \ k)$) $\implies \text{generate-topology } S \ (\bigcup_{k \in I} K \ k)$
 ⟨proof⟩

lemma topological-space-generate-topology: class.topological-space (generate-topology *S*)

⟨proof⟩

98.4 Order topologies

class order-topology = order + open +

assumes open-generated-order: open = generate-topology (range ($\lambda a. \{..< a\}$) \cup range ($\lambda a. \{a <..\}$))

begin

subclass topological-space

⟨proof⟩

lemma open-greaterThan [continuous-intros, simp]: open $\{a <..\}$

⟨proof⟩

lemma open-lessThan [continuous-intros, simp]: open $\{..< a\}$

⟨proof⟩

lemma open-greaterThanLessThan [continuous-intros, simp]: open $\{a <..< b\}$

⟨proof⟩

end

class linorder-topology = linorder + order-topology

lemma closed-atMost [continuous-intros, simp]: closed $\{..a\}$

for $a :: 'a::\text{linorder-topology}$

⟨proof⟩

lemma closed-atLeast [continuous-intros, simp]: closed $\{a..\}$

for $a :: 'a::\text{linorder-topology}$

⟨proof⟩

lemma closed-atLeastAtMost [continuous-intros, simp]: closed $\{a..b\}$

for $a\ b :: 'a::\text{linorder-topology}$
 $\langle\text{proof}\rangle$

lemma (in *order*) *less-separate*:

assumes $x < y$
shows $\exists a\ b. x \in \{..< a\} \wedge y \in \{b <..\} \wedge \{..< a\} \cap \{b <..\} = \{\}$
 $\langle\text{proof}\rangle$

instance *linorder-topology* \subseteq *t2-space*
 $\langle\text{proof}\rangle$

lemma (in *linorder-topology*) *open-right*:

assumes *open* $S\ x \in S$
and *gt-ex*: $x < y$
shows $\exists b > x. \{x ..< b\} \subseteq S$
 $\langle\text{proof}\rangle$

lemma (in *linorder-topology*) *open-left*:

assumes *open* $S\ x \in S$
and *lt-ex*: $y < x$
shows $\exists b < x. \{b <.. x\} \subseteq S$
 $\langle\text{proof}\rangle$

lemma *filterlim-atLeastAtMost-at-bot-at-top*:

fixes $f\ g :: 'a \Rightarrow 'b :: \text{linorder-topology}$
assumes *filterlim* $f\ \text{at-bot}\ F\ \text{filterlim}\ g\ \text{at-top}\ F$
assumes [*simp*]: $\bigwedge a\ b. \text{finite}\ \{a..b::'b\}$
shows *filterlim* $(\lambda x. \{f\ x..g\ x\})\ \text{finite-sets-at-top}\ F$
 $\langle\text{proof}\rangle$

98.5 Setup some topologies

98.5.1 Boolean is an order topology

class *discrete-topology* = *topological-space* +
assumes *open-discrete*: $\bigwedge A. \text{open}\ A$

instance *discrete-topology* $<$ *t2-space*
 $\langle\text{proof}\rangle$

instantiation *bool* :: *linorder-topology*
begin

definition *open-bool* :: *bool set* \Rightarrow *bool*

where *open-bool* = *generate-topology* $(\text{range}\ (\lambda a. \{..< a\}) \cup \text{range}\ (\lambda a. \{a <..\}))$

instance
 $\langle\text{proof}\rangle$

end

instance *bool* :: *discrete-topology*

⟨*proof*⟩

instantiation *nat* :: *linorder-topology*

begin

definition *open-nat* :: *nat set* ⇒ *bool*

where *open-nat* = *generate-topology* (*range* ($\lambda a. \{..< a\}$) ∪ *range* ($\lambda a. \{a <..\}$))

instance

⟨*proof*⟩

end

instance *nat* :: *discrete-topology*

⟨*proof*⟩

instantiation *int* :: *linorder-topology*

begin

definition *open-int* :: *int set* ⇒ *bool*

where *open-int* = *generate-topology* (*range* ($\lambda a. \{..< a\}$) ∪ *range* ($\lambda a. \{a <..\}$))

instance

⟨*proof*⟩

end

instance *int* :: *discrete-topology*

⟨*proof*⟩

98.5.2 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* <..*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*:

$open = generate_topology\ T \implies nhds\ x = (INF\ S \in \{S \in T. x \in S\}. principal\ S)$
 ⟨proof⟩

lemma (in *topological-space*) *eventually-nhds*:

$eventually\ P\ (nhds\ a) \longleftrightarrow (\exists\ S. open\ S \wedge a \in S \wedge (\forall\ x \in S. P\ x))$
 ⟨proof⟩

lemma *eventually-eventually*:

$eventually\ (\lambda y. eventually\ P\ (nhds\ y))\ (nhds\ x) = eventually\ P\ (nhds\ x)$
 ⟨proof⟩

lemma (in *topological-space*) *eventually-nhds-in-open*:

$open\ s \implies x \in s \implies eventually\ (\lambda y. y \in s)\ (nhds\ x)$
 ⟨proof⟩

lemma (in *topological-space*) *eventually-nhds-x-imp-x*: $eventually\ P\ (nhds\ x) \implies P\ x$
 ⟨proof⟩

lemma (in *topological-space*) *nhds-neq-bot [simp]*: $nhds\ a \neq bot$

⟨proof⟩

lemma (in *t1-space*) *t1-space-nhds*: $x \neq y \implies (\forall_F\ x\ in\ nhds\ x. x \neq y)$

⟨proof⟩

lemma (in *topological-space*) *nhds-discrete-open*: $open\ \{x\} \implies nhds\ x = principal\ \{x\}$

⟨proof⟩

lemma (in *discrete-topology*) *nhds-discrete*: $nhds\ x = principal\ \{x\}$

⟨proof⟩

lemma (in *discrete-topology*) *at-discrete*: $at\ x\ within\ S = bot$

⟨proof⟩

lemma (in *discrete-topology*) *tendsto-discrete*:

$filterlim\ (f :: 'b \Rightarrow 'a)\ (nhds\ y)\ F \longleftrightarrow eventually\ (\lambda x. f\ x = y)\ F$
 ⟨proof⟩

lemma (in *topological-space*) *at-within-eq*:

$at\ x\ within\ s = (INF\ S \in \{S. open\ S \wedge x \in S\}. principal\ (S \cap s - \{x\}))$
 ⟨proof⟩

lemma (in *topological-space*) *eventually-at-filter*:

$eventually\ P\ (at\ a\ within\ s) \longleftrightarrow eventually\ (\lambda x. x \neq a \longrightarrow x \in s \longrightarrow P\ x)\ (nhds\ a)$
 ⟨proof⟩

lemma (in *topological-space*) *at-le*: $s \subseteq t \implies \text{at } x \text{ within } s \leq \text{at } x \text{ within } t$
 ⟨proof⟩

lemma (in *topological-space*) *eventually-at-topological*:
 $\text{eventually } P \text{ (at } a \text{ within } s) \longleftrightarrow (\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in S. x \neq a \longrightarrow x \in s \longrightarrow P \ x))$
 ⟨proof⟩

lemma *eventually-nhds-conv-at*:
 $\text{eventually } P \text{ (nhds } x) \longleftrightarrow \text{eventually } P \text{ (at } x) \wedge P \ x$
 ⟨proof⟩

lemma *eventually-at-in-open*:
 assumes $\text{open } A \ x \in A$
 shows $\text{eventually } (\lambda y. y \in A - \{x\}) \text{ (at } x)$
 ⟨proof⟩

lemma *eventually-at-in-open'*:
 assumes $\text{open } A \ x \in A$
 shows $\text{eventually } (\lambda y. y \in A) \text{ (at } x)$
 ⟨proof⟩

lemma (in *topological-space*) *at-within-open*: $a \in S \implies \text{open } S \implies \text{at } a \text{ within } S = \text{at } a$
 ⟨proof⟩

lemma (in *topological-space*) *at-within-open-NO-MATCH*:
 $a \in s \implies \text{open } s \implies \text{NO-MATCH UNIV } s \implies \text{at } a \text{ within } s = \text{at } a$
 ⟨proof⟩

lemma (in *topological-space*) *at-within-open-subset*:
 $a \in S \implies \text{open } S \implies S \subseteq T \implies \text{at } a \text{ within } T = \text{at } a$
 ⟨proof⟩

lemma (in *topological-space*) *at-within-nhd*:
 assumes $x \in S \text{ open } S \ T \cap S - \{x\} = U \cap S - \{x\}$
 shows $\text{at } x \text{ within } T = \text{at } x \text{ within } U$
 ⟨proof⟩

lemma (in *topological-space*) *at-within-empty* [simp]: $\text{at } a \text{ within } \{\} = \text{bot}$
 ⟨proof⟩

lemma (in *topological-space*) *at-within-union*:
 $\text{at } x \text{ within } (S \cup T) = \sup (\text{at } x \text{ within } S) \text{ (at } x \text{ within } T)$
 ⟨proof⟩

lemma (in *topological-space*) *at-eq-bot-iff*: $\text{at } a = \text{bot} \longleftrightarrow \text{open } \{a\}$
 ⟨proof⟩

lemma (in *t1-space*) *eventually-neq-at-within*:

eventually ($\lambda w. w \neq x$) (at z within A)

$\langle \text{proof} \rangle$

lemma (in *perfect-space*) *at-neq-bot [simp]*: at $a \neq \text{bot}$

$\langle \text{proof} \rangle$

lemma (in *order-topology*) *nhds-order*:

nhds $x = \inf \text{ (INF } a \in \{x <.. \}. \text{ principal } \{.. < a\}) \text{ (INF } a \in \{.. < x\}. \text{ principal } \{a <.. \})$

$\langle \text{proof} \rangle$

lemma (in *topological-space*) *filterlim-at-within-If*:

assumes *filterlim* f G (at x within $(A \cap \{x. P\ x\})$)

and *filterlim* g G (at x within $(A \cap \{x. \neg P\ x\})$)

shows *filterlim* ($\lambda x. \text{ if } P\ x \text{ then } f\ x \text{ else } g\ x$) G (at x within A)

$\langle \text{proof} \rangle$

lemma (in *topological-space*) *filterlim-at-If*:

assumes *filterlim* f G (at x within $\{x. P\ x\}$)

and *filterlim* g G (at x within $\{x. \neg P\ x\}$)

shows *filterlim* ($\lambda x. \text{ if } P\ x \text{ then } f\ x \text{ else } g\ x$) G (at x)

$\langle \text{proof} \rangle$

lemma (in *linorder-topology*) *at-within-order*:

assumes $\text{UNIV} \neq \{x\}$

shows at x within $s =$

$\inf \text{ (INF } a \in \{x <.. \}. \text{ principal } (\{.. < a\} \cap s - \{x\}))$

$\text{ (INF } a \in \{.. < x\}. \text{ principal } (\{a <.. \} \cap s - \{x\}))$

$\langle \text{proof} \rangle$

lemma (in *linorder-topology*) *at-left-eq*:

$y < x \implies \text{at-left } x = \text{ (INF } a \in \{.. < x\}. \text{ principal } \{a <.. < x\})$

$\langle \text{proof} \rangle$

lemma (in *linorder-topology*) *eventually-at-left*:

$y < x \implies \text{eventually } P \text{ (at-left } x) \longleftrightarrow (\exists b < x. \forall y > b. y < x \longrightarrow P\ y)$

$\langle \text{proof} \rangle$

lemma (in *linorder-topology*) *at-right-eq*:

$x < y \implies \text{at-right } x = \text{ (INF } a \in \{x <.. \}. \text{ principal } \{x <.. < a\})$

$\langle \text{proof} \rangle$

lemma (in *linorder-topology*) *eventually-at-right*:

$x < y \implies \text{eventually } P \text{ (at-right } x) \longleftrightarrow (\exists b > x. \forall y > x. y < b \longrightarrow P\ y)$

$\langle \text{proof} \rangle$

lemma *eventually-at-right-less*: $\forall_F y$ in *at-right* ($x :: 'a :: \{\text{linorder-topology, no-top}\}$).

$x < y$

$\langle \text{proof} \rangle$

lemma *trivial-limit-at-right-top*: *at-right* (*top*::::{*order-top*,*linorder-topology*}) = *bot*
 $\langle \text{proof} \rangle$

lemma *trivial-limit-at-left-bot*: *at-left* (*bot*::::{*order-bot*,*linorder-topology*}) = *bot*
 $\langle \text{proof} \rangle$

lemma *trivial-limit-at-left-real* [*simp*]: $\neg \text{trivial-limit } (\text{at-left } x)$
for $x :: 'a :: \{\text{no-bot}, \text{dense-order}, \text{linorder-topology}\}$
 $\langle \text{proof} \rangle$

lemma *trivial-limit-at-right-real* [*simp*]: $\neg \text{trivial-limit } (\text{at-right } x)$
for $x :: 'a :: \{\text{no-top}, \text{dense-order}, \text{linorder-topology}\}$
 $\langle \text{proof} \rangle$

lemma (**in** *linorder-topology*) *at-eq-sup-left-right*: *at* $x = \text{sup } (\text{at-left } x) (\text{at-right } x)$
 $\langle \text{proof} \rangle$

lemma (**in** *linorder-topology*) *eventually-at-split*:
 $\text{eventually } P (\text{at } x) \longleftrightarrow \text{eventually } P (\text{at-left } x) \wedge \text{eventually } P (\text{at-right } x)$
 $\langle \text{proof} \rangle$

lemma (**in** *order-topology*) *eventually-at-leftI*:
assumes $\bigwedge x. x \in \{a <..< b\} \implies P \ x \ a < b$
shows $\text{eventually } P (\text{at-left } b)$
 $\langle \text{proof} \rangle$

lemma (**in** *order-topology*) *eventually-at-rightI*:
assumes $\bigwedge x. x \in \{a <..< b\} \implies P \ x \ a < b$
shows $\text{eventually } P (\text{at-right } a)$
 $\langle \text{proof} \rangle$

lemma *eventually-filtercomap-nhds*:
 $\text{eventually } P (\text{filtercomap } f (\text{nhds } x)) \longleftrightarrow (\exists S. \text{open } S \wedge x \in S \wedge (\forall x. f \ x \in S \longrightarrow P \ x))$
 $\langle \text{proof} \rangle$

lemma *eventually-filtercomap-at-topological*:
 $\text{eventually } P (\text{filtercomap } f (\text{at } A \text{ within } B)) \longleftrightarrow$
 $(\exists S. \text{open } S \wedge A \in S \wedge (\forall x. f \ x \in S \cap B - \{A\} \longrightarrow P \ x))$ (**is** ?*lhs* = ?*rhs*)
 $\langle \text{proof} \rangle$

lemma *eventually-at-right-field*:
 $\text{eventually } P (\text{at-right } x) \longleftrightarrow (\exists b > x. \forall y > x. y < b \longrightarrow P \ y)$
for $x :: 'a :: \{\text{linordered-field}, \text{linorder-topology}\}$
 $\langle \text{proof} \rangle$

lemma *eventually-at-left-field*:

eventually P (*at-left* x) $\longleftrightarrow (\exists b < x. \forall y > b. y < x \longrightarrow P y)$
for $x :: 'a :: \{\text{linordered-field}, \text{linorder-topology}\}$
 $\langle \text{proof} \rangle$

lemma *filtermap-nhds-eq-imp-filtermap-at-eq*:

assumes *filtermap* f (*nhds* z) = *nhds* ($f z$)
assumes *eventually* $(\lambda x. f x = f z \longrightarrow x = z)$ (*at* z)
shows *filtermap* f (*at* z) = *at* ($f z$)
 $\langle \text{proof} \rangle$

98.5.3 Tendsto

abbreviation (*in topological-space*)

tendsto $:: ('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b \text{ filter} \Rightarrow \text{bool}$ (**infixr** $\langle \longrightarrow \rangle$ 55)
where $(f \longrightarrow l) F \equiv \text{filterlim } f \text{ (nhds } l) F$

definition (*in t2-space*) *Lim* $:: 'f \text{ filter} \Rightarrow ('f \Rightarrow 'a) \Rightarrow 'a$

where *Lim* $A f = (\text{THE } l. (f \longrightarrow l) A)$

lemma (*in topological-space*) *tendsto-eq-rhs*: $(f \longrightarrow x) F \Longrightarrow x = y \Longrightarrow (f \longrightarrow y) F$

$\langle \text{proof} \rangle$

named-theorems *tendsto-intros* *introduction rules for tendsto*

$\langle \text{ML} \rangle$

context *topological-space* **begin**

lemma *tendsto-def*:

$(f \longrightarrow l) F \longleftrightarrow (\forall S. \text{open } S \longrightarrow l \in S \longrightarrow \text{eventually } (\lambda x. f x \in S) F)$
 $\langle \text{proof} \rangle$

lemma *tendsto-cong*: $(f \longrightarrow c) F \longleftrightarrow (g \longrightarrow c) F$ **if** *eventually* $(\lambda x. f x = g x) F$

$\langle \text{proof} \rangle$

lemma *tendsto-mono*: $F \leq F' \Longrightarrow (f \longrightarrow l) F' \Longrightarrow (f \longrightarrow l) F$

$\langle \text{proof} \rangle$

lemma *tendsto-ident-at* [*tendsto-intros*, *simp*, *intro*]: $((\lambda x. x) \longrightarrow a)$ (*at* a *within* s)

$\langle \text{proof} \rangle$

lemma *tendsto-const* [*tendsto-intros*, *simp*, *intro*]: $((\lambda x. k) \longrightarrow k) F$

$\langle \text{proof} \rangle$

lemma *filterlim-at*:

$(\text{LIM } x \ F. \ f \ x \mathrel{>} \text{ at } b \text{ within } s) \longleftrightarrow \text{eventually } (\lambda x. \ f \ x \in s \wedge f \ x \neq b) \ F \wedge (f \longrightarrow b) \ F$
 $\langle \text{proof} \rangle$

lemma (in $-$)

assumes $\text{filterlim } f \ (\text{nhds } L) \ F$

shows $\text{tendsto-imp-filterlim-at-right}$:

$\text{eventually } (\lambda x. \ f \ x > L) \ F \implies \text{filterlim } f \ (\text{at-right } L) \ F$

and $\text{tendsto-imp-filterlim-at-left}$:

$\text{eventually } (\lambda x. \ f \ x < L) \ F \implies \text{filterlim } f \ (\text{at-left } L) \ F$

$\langle \text{proof} \rangle$

lemma $\text{filterlim-at-withinI}$:

assumes $\text{filterlim } f \ (\text{nhds } c) \ F$

assumes $\text{eventually } (\lambda x. \ f \ x \in A - \{c\}) \ F$

shows $\text{filterlim } f \ (\text{at } c \text{ within } A) \ F$

$\langle \text{proof} \rangle$

lemma filterlim-atI :

assumes $\text{filterlim } f \ (\text{nhds } c) \ F$

assumes $\text{eventually } (\lambda x. \ f \ x \neq c) \ F$

shows $\text{filterlim } f \ (\text{at } c) \ F$

$\langle \text{proof} \rangle$

lemma $\text{topological-tendstoI}$:

$(\bigwedge S. \ \text{open } S \implies l \in S \implies \text{eventually } (\lambda x. \ f \ x \in S) \ F) \implies (f \longrightarrow l) \ F$

$\langle \text{proof} \rangle$

lemma $\text{topological-tendstoD}$:

$(f \longrightarrow l) \ F \implies \text{open } S \implies l \in S \implies \text{eventually } (\lambda x. \ f \ x \in S) \ F$

$\langle \text{proof} \rangle$

lemma $\text{tendsto-bot [simp]}$: $(f \longrightarrow a) \ \text{bot}$

$\langle \text{proof} \rangle$

lemma $\text{tendsto-eventually}$: $\text{eventually } (\lambda x. \ f \ x = l) \ \text{net} \implies ((\lambda x. \ f \ x) \longrightarrow l) \ \text{net}$

$\langle \text{proof} \rangle$

lemma $\text{tendsto-principal-singleton[simp]}$:

shows $(f \longrightarrow f \ x) \ (\text{principal } \{x\})$

$\langle \text{proof} \rangle$

end

lemma (in topological-space) $\text{filterlim-within-subset}$:

$\text{filterlim } f \ l \ (\text{at } x \text{ within } S) \implies T \subseteq S \implies \text{filterlim } f \ l \ (\text{at } x \text{ within } T)$

$\langle \text{proof} \rangle$

lemmas *tendsto-within-subset* = *filterlim-within-subset*

lemma (in *order-topology*) *order-tendsto-iff*:

$(f \longrightarrow x) F \longleftrightarrow (\forall l < x. \text{eventually } (\lambda x. l < f x) F) \wedge (\forall u > x. \text{eventually } (\lambda x. f x < u) F)$
 ⟨proof⟩

lemma (in *order-topology*) *order-tendstoI*:

$(\bigwedge a. a < y \implies \text{eventually } (\lambda x. a < f x) F) \implies (\bigwedge a. y < a \implies \text{eventually } (\lambda x. f x < a) F) \implies (f \longrightarrow y) F$
 ⟨proof⟩

lemma (in *order-topology*) *order-tendstoD*:

assumes $(f \longrightarrow y) F$
shows $a < y \implies \text{eventually } (\lambda x. a < f x) F$
and $y < a \implies \text{eventually } (\lambda x. f x < a) F$
 ⟨proof⟩

lemma (in *linorder-topology*) *tendsto-max[tendsto-intros]*:

assumes $X: (X \longrightarrow x) \text{ net}$
and $Y: (Y \longrightarrow y) \text{ net}$
shows $((\lambda x. \max (X x) (Y x)) \longrightarrow \max x y) \text{ net}$
 ⟨proof⟩

lemma (in *linorder-topology*) *tendsto-min[tendsto-intros]*:

assumes $X: (X \longrightarrow x) \text{ net}$
and $Y: (Y \longrightarrow y) \text{ net}$
shows $((\lambda x. \min (X x) (Y x)) \longrightarrow \min x y) \text{ net}$
 ⟨proof⟩

lemma (in *order-topology*)

assumes $a < b$
shows *at-within-Icc-at-right*: *at a within* $\{a..b\} = \text{at-right } a$
and *at-within-Icc-at-left*: *at b within* $\{a..b\} = \text{at-left } b$
 ⟨proof⟩

lemma (in *order-topology*)

shows *at-within-Ici-at-right*: *at a within* $\{a.. \} = \text{at-right } a$
and *at-within-Iic-at-left*: *at a within* $\{..a\} = \text{at-left } a$
 ⟨proof⟩

lemma (in *order-topology*) *at-within-Icc-at*: $a < x \implies x < b \implies \text{at } x \text{ within } \{a..b\}$

= *at x*
 ⟨proof⟩

lemma (in *t2-space*) *tendsto-unique*:

assumes $F \neq \text{bot}$
and $(f \longrightarrow a) F$

and $(f \longrightarrow b) F$
 shows $a = b$
 $\langle proof \rangle$

lemma (in *t2-space*) *tendsto-const-iff*:
 fixes $a b :: 'a$
 assumes $\neg \text{trivial-limit } F$
 shows $((\lambda x. a) \longrightarrow b) F \longleftrightarrow a = b$
 $\langle proof \rangle$

lemma (in *t2-space*) *tendsto-unique'*:
 assumes $F \neq \text{bot}$
 shows $\exists_{\leq 1} l. (f \longrightarrow l) F$
 $\langle proof \rangle$

lemma *Lim-in-closed-set*:
 assumes *closed* S *eventually* $(\lambda x. f(x) \in S) F$ $F \neq \text{bot}$ $(f \longrightarrow l) F$
 shows $l \in S$
 $\langle proof \rangle$

lemma (in *t3-space*) *nhds-closed*:
 assumes $x \in A$ and *open* A
 shows $\exists A'. x \in A' \wedge \text{closed } A' \wedge A' \subseteq A \wedge \text{eventually } (\lambda y. y \in A') (\text{nhds } x)$
 $\langle proof \rangle$

lemma (in *order-topology*) *increasing-tendsto*:
 assumes *bdd*: *eventually* $(\lambda n. f\ n \leq l) F$
 and *en*: $\bigwedge x. x < l \implies \text{eventually } (\lambda n. x < f\ n) F$
 shows $(f \longrightarrow l) F$
 $\langle proof \rangle$

lemma (in *order-topology*) *decreasing-tendsto*:
 assumes *bdd*: *eventually* $(\lambda n. l \leq f\ n) F$
 and *en*: $\bigwedge x. l < x \implies \text{eventually } (\lambda n. f\ n < x) F$
 shows $(f \longrightarrow l) F$
 $\langle proof \rangle$

lemma (in *order-topology*) *tendsto-sandwich*:
 assumes *ev*: *eventually* $(\lambda n. f\ n \leq g\ n)$ *net* *eventually* $(\lambda n. g\ n \leq h\ n)$ *net*
 assumes *lim*: $(f \longrightarrow c)$ *net* $(h \longrightarrow c)$ *net*
 shows $(g \longrightarrow c)$ *net*
 $\langle proof \rangle$

lemma (in *t1-space*) *limit-frequently-eq*:
 assumes $F \neq \text{bot}$
 and *frequently* $(\lambda x. f\ x = c) F$
 and $(f \longrightarrow d) F$
 shows $d = c$
 $\langle proof \rangle$

lemma (in *t1-space*) *tendsto-imp-eventually-ne*:

assumes $(f \longrightarrow c) \ F \ c \neq c'$
 shows *eventually* $(\lambda z. f \ z \neq c') \ F$

<proof>

lemma (in *linorder-topology*) *tendsto-le*:

assumes $F: \neg \text{trivial-limit } F$
 and $x: (f \longrightarrow x) \ F$
 and $y: (g \longrightarrow y) \ F$
 and $ev: \text{eventually } (\lambda x. g \ x \leq f \ x) \ F$
 shows $y \leq x$

<proof>

lemma (in *linorder-topology*) *tendsto-lowerbound*:

assumes $x: (f \longrightarrow x) \ F$
 and $ev: \text{eventually } (\lambda i. a \leq f \ i) \ F$
 and $F: \neg \text{trivial-limit } F$
 shows $a \leq x$

<proof>

lemma (in *linorder-topology*) *tendsto-upperbound*:

assumes $x: (f \longrightarrow x) \ F$
 and $ev: \text{eventually } (\lambda i. a \geq f \ i) \ F$
 and $F: \neg \text{trivial-limit } F$
 shows $a \geq x$

<proof>

lemma *filterlim-at-within-not-equal*:

fixes $f::'a \Rightarrow 'b::t2\text{-space}$
 assumes *filterlim* $f \ (\text{at } a \ \text{within } s) \ F$
 shows *eventually* $(\lambda w. f \ w \in s \wedge f \ w \neq b) \ F$

<proof>

98.5.4 Rules about *Lim*

lemma *tendsto-Lim*: $\neg \text{trivial-limit } net \implies (f \longrightarrow l) \ net \implies \text{Lim } net \ f = l$

<proof>

lemma *Lim-ident-at*: $\neg \text{trivial-limit } (\text{at } x \ \text{within } s) \implies \text{Lim } (\text{at } x \ \text{within } s) \ (\lambda x. x) = x$

<proof>

lemma *Lim-cong*:

assumes $\forall_F \ x \ \text{in } F. f \ x = g \ x \ F = G$
 shows $\text{Lim } F \ f = \text{Lim } F \ g$

<proof>

lemma *eventually-Lim-ident-at*:

$(\forall_F y \text{ in } at\ x \text{ within } X. P\ (Lim\ (at\ x \text{ within } X)\ (\lambda x. x))\ y) \longleftrightarrow$
 $(\forall_F y \text{ in } at\ x \text{ within } X. P\ x\ y) \text{ for } x :: 'a :: t2\text{-space}$
 $\langle proof \rangle$

lemma *filterlim-at-bot-at-right*:

fixes $f :: 'a :: linorder\text{-topology} \Rightarrow 'b :: linorder$
assumes $mono: \bigwedge x\ y. Q\ x \Longrightarrow Q\ y \Longrightarrow x \leq y \Longrightarrow f\ x \leq f\ y$
and $bij: \bigwedge x. P\ x \Longrightarrow f\ (g\ x) = x \bigwedge x. P\ x \Longrightarrow Q\ (g\ x)$
and $Q: \text{eventually } Q\ (at\text{-right } a)$
and $bound: \bigwedge b. Q\ b \Longrightarrow a < b$
and $P: \text{eventually } P\ at\text{-bot}$
shows $filterlim\ f\ at\text{-bot}\ (at\text{-right } a)$
 $\langle proof \rangle$

lemma *filterlim-at-top-at-left*:

fixes $f :: 'a :: linorder\text{-topology} \Rightarrow 'b :: linorder$
assumes $mono: \bigwedge x\ y. Q\ x \Longrightarrow Q\ y \Longrightarrow x \leq y \Longrightarrow f\ x \leq f\ y$
and $bij: \bigwedge x. P\ x \Longrightarrow f\ (g\ x) = x \bigwedge x. P\ x \Longrightarrow Q\ (g\ x)$
and $Q: \text{eventually } Q\ (at\text{-left } a)$
and $bound: \bigwedge b. Q\ b \Longrightarrow b < a$
and $P: \text{eventually } P\ at\text{-top}$
shows $filterlim\ f\ at\text{-top}\ (at\text{-left } a)$
 $\langle proof \rangle$

lemma *filterlim-split-at*:

$filterlim\ f\ F\ (at\text{-left } x) \Longrightarrow filterlim\ f\ F\ (at\text{-right } x) \Longrightarrow$
 $filterlim\ f\ F\ (at\ x)$
for $x :: 'a :: linorder\text{-topology}$
 $\langle proof \rangle$

lemma *filterlim-at-split*:

$filterlim\ f\ F\ (at\ x) \longleftrightarrow filterlim\ f\ F\ (at\text{-left } x) \wedge filterlim\ f\ F\ (at\text{-right } x)$
for $x :: 'a :: linorder\text{-topology}$
 $\langle proof \rangle$

lemma *eventually-nhds-top*:

fixes $P :: 'a :: \{order\text{-top}, linorder\text{-topology}\} \Rightarrow bool$
and $b :: 'a$
assumes $b < top$
shows $eventually\ P\ (nhds\ top) \longleftrightarrow (\exists b < top. (\forall z. b < z \longrightarrow P\ z))$
 $\langle proof \rangle$

lemma *tendsto-at-within-iff-tendsto-nhds*:

$(g \longrightarrow g\ l)\ (at\ l\ \text{within } S) \longleftrightarrow (g \longrightarrow g\ l)\ (inf\ (nhds\ l)\ (principal\ S))$
 $\langle proof \rangle$

98.6 Limits on sequences

abbreviation (in *topological-space*)

$LIMSEQ :: [nat \Rightarrow 'a, 'a] \Rightarrow bool \ (\langle \langle notation = \langle infix LIMSEQ \rangle \rangle (-) / \longrightarrow (-) \rangle$
 $[60, 60] \ 60)$

where $X \longrightarrow L \equiv (X \longrightarrow L) \text{ sequentially}$

abbreviation (in *t2-space*) $lim :: (nat \Rightarrow 'a) \Rightarrow 'a$

where $lim \ X \equiv Lim \text{ sequentially } X$

definition (in *topological-space*) $convergent :: (nat \Rightarrow 'a) \Rightarrow bool$

where $convergent \ X = (\exists L. \ X \longrightarrow L)$

lemma *lim-def*: $lim \ X = (THE \ L. \ X \longrightarrow L)$

<proof>

lemma *lim-explicit*:

$f \longrightarrow f0 \longleftrightarrow (\forall S. \ open \ S \longrightarrow f0 \in S \longrightarrow (\exists N. \ \forall n \geq N. \ f \ n \in S))$

<proof>

lemma *closed-sequentially*:

assumes *closed* S **and** $\bigwedge n. \ f \ n \in S$ **and** $f \longrightarrow l$

shows $l \in S$

<proof>

98.7 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* :: $(nat \Rightarrow 'a::order) \Rightarrow bool$

where $monoseq \ X \longleftrightarrow (\forall m. \ \forall n \geq m. \ X \ m \leq X \ n) \vee (\forall m. \ \forall n \geq m. \ X \ n \leq X \ m)$

abbreviation *incseq* :: $(nat \Rightarrow 'a::order) \Rightarrow bool$

where $incseq \ X \equiv mono \ X$

lemma *incseq-def*: $incseq \ X \longleftrightarrow (\forall m. \ \forall n \geq m. \ X \ n \geq X \ m)$

<proof>

abbreviation *decseq* :: $(nat \Rightarrow 'a::order) \Rightarrow bool$

where $decseq \ X \equiv antimono \ X$

lemma *decseq-def*: $decseq \ X \longleftrightarrow (\forall m. \ \forall n \geq m. \ X \ n \leq X \ m)$

<proof>

98.7.1 Definition of subsequence.

lemma *strict-mono-leD*: $strict-mono \ r \Longrightarrow m \leq n \Longrightarrow r \ m \leq r \ n$

<proof>

lemma *strict-mono-id*: $strict-mono \ id$

$\langle proof \rangle$

lemma *incseq-SucI*: $(\bigwedge n. X\ n \leq X\ (Suc\ n)) \implies incseq\ X$
 $\langle proof \rangle$

lemma *incseqD*: $incseq\ f \implies i \leq j \implies f\ i \leq f\ j$
 $\langle proof \rangle$

lemma *incseq-SucD*: $incseq\ A \implies A\ i \leq A\ (Suc\ i)$
 $\langle proof \rangle$

lemma *incseq-Suc-iff*: $incseq\ f \longleftrightarrow (\forall n. f\ n \leq f\ (Suc\ n))$
 $\langle proof \rangle$

lemma *incseq-const*[*simp*, *intro*]: $incseq\ (\lambda x. k)$
 $\langle proof \rangle$

lemma *decseq-SucI*: $(\bigwedge n. X\ (Suc\ n) \leq X\ n) \implies decseq\ X$
 $\langle proof \rangle$

lemma *decseqD*: $decseq\ f \implies i \leq j \implies f\ j \leq f\ i$
 $\langle proof \rangle$

lemma *decseq-SucD*: $decseq\ A \implies A\ (Suc\ i) \leq A\ i$
 $\langle proof \rangle$

lemma *decseq-Suc-iff*: $decseq\ f \longleftrightarrow (\forall n. f\ (Suc\ n) \leq f\ n)$
 $\langle proof \rangle$

lemma *decseq-const*[*simp*, *intro*]: $decseq\ (\lambda x. k)$
 $\langle proof \rangle$

lemma *monoseq-iff*: $monoseq\ X \longleftrightarrow incseq\ X \vee decseq\ X$
 $\langle proof \rangle$

lemma *monoseq-Suc*: $monoseq\ X \longleftrightarrow (\forall n. X\ n \leq X\ (Suc\ n)) \vee (\forall n. X\ (Suc\ n) \leq X\ n)$
 $\langle proof \rangle$

lemma *monoI1*: $\forall m. \forall n \geq m. X\ m \leq X\ n \implies monoseq\ X$
 $\langle proof \rangle$

lemma *monoI2*: $\forall m. \forall n \geq m. X\ n \leq X\ m \implies monoseq\ X$
 $\langle proof \rangle$

lemma *mono-SucI1*: $\forall n. X\ n \leq X\ (Suc\ n) \implies monoseq\ X$
 $\langle proof \rangle$

lemma *mono-SucI2*: $\forall n. X\ (Suc\ n) \leq X\ n \implies monoseq\ X$

$\langle \text{proof} \rangle$

lemma *monoseq-minus*:

fixes $a :: \text{nat} \Rightarrow 'a::\text{ordered-ab-group-add}$

assumes *monoseq a*

shows *monoseq* $(\lambda n. - a\ n)$

$\langle \text{proof} \rangle$

98.7.2 Subsequence (alternative definition, (e.g. Hoskins))

For any sequence, there is a monotonic subsequence.

lemma *seq-monosub*:

fixes $s :: \text{nat} \Rightarrow 'a::\text{linorder}$

shows $\exists f. \text{strict-mono } f \wedge \text{monoseq } (\lambda n. (s\ (f\ n)))$

$\langle \text{proof} \rangle$

lemma *seq-suble*:

assumes *sf: strict-mono* $(f :: \text{nat} \Rightarrow \text{nat})$

shows $n \leq f\ n$

$\langle \text{proof} \rangle$

lemma *eventually-subseq*:

strict-mono r \implies eventually P sequentially \implies eventually $(\lambda n. P\ (r\ n))$ sequentially

$\langle \text{proof} \rangle$

lemma *not-eventually-sequentiallyD*:

assumes $\neg \text{eventually } P \text{ sequentially}$

shows $\exists r::\text{nat} \Rightarrow \text{nat}. \text{strict-mono } r \wedge (\forall n. \neg P\ (r\ n))$

$\langle \text{proof} \rangle$

lemma *sequentially-offset*:

assumes *eventually* $(\lambda i. P\ i) \text{ sequentially}$

shows *eventually* $(\lambda i. P\ (i + k)) \text{ sequentially}$

$\langle \text{proof} \rangle$

lemma *seq-offset-neg*:

$(f \longrightarrow l) \text{ sequentially} \implies ((\lambda i. f(i - k)) \longrightarrow l) \text{ sequentially}$

$\langle \text{proof} \rangle$

lemma *filterlim-subseq*: *strict-mono f \implies filterlim f sequentially sequentially*

$\langle \text{proof} \rangle$

lemma *strict-mono-o*: *strict-mono r \implies strict-mono s \implies strict-mono $(r \circ s)$*

$\langle \text{proof} \rangle$

lemma *strict-mono-compose*: *strict-mono r \implies strict-mono s \implies strict-mono $(\lambda x. r\ (s\ x))$*

$\langle \text{proof} \rangle$

lemma *incseq-imp-monoseq*: $\text{incseq } X \implies \text{monoseq } X$
 $\langle \text{proof} \rangle$

lemma *decseq-imp-monoseq*: $\text{decseq } X \implies \text{monoseq } X$
 $\langle \text{proof} \rangle$

lemma *decseq-eq-incseq*: $\text{decseq } X = \text{incseq } (\lambda n. - X n)$
for $X :: \text{nat} \Rightarrow 'a::\text{ordered-ab-group-add}$
 $\langle \text{proof} \rangle$

lemma *INT-decseq-offset*:
assumes $\text{decseq } F$
shows $(\bigcap i. F i) = (\bigcap i \in \{n..\}. F i)$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-const-iff*: $(\lambda n. k) \longrightarrow l \iff k = l$
for $k l :: 'a::t2\text{-space}$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-SUP*: $\text{incseq } X \implies X \longrightarrow (\text{SUP } i. X i :: 'a::\{\text{complete-linorder}, \text{linorder-topology}\})$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-INF*: $\text{decseq } X \implies X \longrightarrow (\text{INF } i. X i :: 'a::\{\text{complete-linorder}, \text{linorder-topology}\})$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-ignore-initial-segment*: $f \longrightarrow a \implies (\lambda n. f (n + k)) \longrightarrow a$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-offset*: $(\lambda n. f (n + k)) \longrightarrow a \implies f \longrightarrow a$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-Suc*: $f \longrightarrow l \implies (\lambda n. f (\text{Suc } n)) \longrightarrow l$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-imp-Suc*: $(\lambda n. f (\text{Suc } n)) \longrightarrow l \implies f \longrightarrow l$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-lessThan-iff-atMost*:
shows $(\lambda n. f \{.. $n\}) \longrightarrow x \iff (\lambda n. f \{.. $n\}) \longrightarrow x$
 $\langle \text{proof} \rangle$$$

lemma (in *t2-space*) *LIMSEQ-Uniq*: $\exists_{\leq 1} l. X \longrightarrow l$
 $\langle \text{proof} \rangle$

lemma (in *t2-space*) *LIMSEQ-unique*: $X \longrightarrow a \implies X \longrightarrow b \implies a = b$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-le-const*: $X \longrightarrow x \implies \exists N. \forall n \geq N. a \leq X n \implies a \leq x$

for $a\ x :: 'a::\text{linorder-topology}$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-le*: $X \longrightarrow x \implies Y \longrightarrow y \implies \exists N. \forall n \geq N. X\ n \leq Y\ n$
 $\implies x \leq y$
for $x\ y :: 'a::\text{linorder-topology}$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-le-const2*: $X \longrightarrow x \implies \exists N. \forall n \geq N. X\ n \leq a \implies x \leq a$
for $a\ x :: 'a::\text{linorder-topology}$
 $\langle \text{proof} \rangle$

lemma *Lim-bounded*: $f \longrightarrow l \implies \forall n \geq M. f\ n \leq C \implies l \leq C$
for $l :: 'a::\text{linorder-topology}$
 $\langle \text{proof} \rangle$

lemma *Lim-bounded2*:
fixes $f :: \text{nat} \Rightarrow 'a::\text{linorder-topology}$
assumes $\text{lim}: f \longrightarrow l$ **and** $\text{ge}: \forall n \geq N. f\ n \geq C$
shows $l \geq C$
 $\langle \text{proof} \rangle$

lemma *lim-mono*:
fixes $X\ Y :: \text{nat} \Rightarrow 'a::\text{linorder-topology}$
assumes $\bigwedge n. N \leq n \implies X\ n \leq Y\ n$
and $X \longrightarrow x$
and $Y \longrightarrow y$
shows $x \leq y$
 $\langle \text{proof} \rangle$

lemma *Sup-lim*:
fixes $a :: 'a::\{\text{complete-linorder}, \text{linorder-topology}\}$
assumes $\bigwedge n. b\ n \in s$
and $b \longrightarrow a$
shows $a \leq \text{Sup}\ s$
 $\langle \text{proof} \rangle$

lemma *Inf-lim*:
fixes $a :: 'a::\{\text{complete-linorder}, \text{linorder-topology}\}$
assumes $\bigwedge n. b\ n \in s$
and $b \longrightarrow a$
shows $\text{Inf}\ s \leq a$
 $\langle \text{proof} \rangle$

lemma *SUP-Lim*:
fixes $X :: \text{nat} \Rightarrow 'a::\{\text{complete-linorder}, \text{linorder-topology}\}$
assumes $\text{inc}: \text{incseq}\ X$
and $l: X \longrightarrow l$
shows $(\text{SUP}\ n. X\ n) = l$

$\langle \text{proof} \rangle$

lemma *INF-Lim*:

fixes $X :: \text{nat} \Rightarrow 'a::\{\text{complete-linorder}, \text{linorder-topology}\}$

assumes $\text{dec}: \text{decseq } X$

and $l: X \longrightarrow l$

shows $(\text{INF } n. X \ n) = l$

$\langle \text{proof} \rangle$

lemma *convergentD*: $\text{convergent } X \implies \exists L. X \longrightarrow L$

$\langle \text{proof} \rangle$

lemma *convergentI*: $X \longrightarrow L \implies \text{convergent } X$

$\langle \text{proof} \rangle$

lemma *convergent-LIMSEQ-iff*: $\text{convergent } X \longleftrightarrow X \longrightarrow \lim X$

$\langle \text{proof} \rangle$

lemma *convergent-const*: $\text{convergent } (\lambda n. c)$

$\langle \text{proof} \rangle$

lemma *monoseq-le*:

$\text{monoseq } a \implies a \longrightarrow x \implies$

$(\forall n. a \ n \leq x) \wedge (\forall m. \forall n \geq m. a \ m \leq a \ n) \vee$

$(\forall n. x \leq a \ n) \wedge (\forall m. \forall n \geq m. a \ n \leq a \ m)$

for $x :: 'a::\text{linorder-topology}$

$\langle \text{proof} \rangle$

lemma *LIMSEQ-subseq-LIMSEQ*: $X \longrightarrow L \implies \text{strict-mono } f \implies (X \circ f) \longrightarrow L$

$\langle \text{proof} \rangle$

lemma *convergent-subseq-convergent*: $\text{convergent } X \implies \text{strict-mono } f \implies \text{convergent } (X \circ f)$

$\langle \text{proof} \rangle$

lemma *limI*: $X \longrightarrow L \implies \lim X = L$

$\langle \text{proof} \rangle$

lemma *lim-le*: $\text{convergent } f \implies (\bigwedge n. f \ n \leq x) \implies \lim f \leq x$

for $x :: 'a::\text{linorder-topology}$

$\langle \text{proof} \rangle$

lemma *lim-const* [*simp*]: $\lim (\lambda m. a) = a$

$\langle \text{proof} \rangle$

98.7.3 Increasing and Decreasing Series

lemma *incseq-le*: $\text{incseq } X \implies X \longrightarrow L \implies X \ n \leq L$

for $L :: 'a::linorder-topology$
 $\langle proof \rangle$

lemma *decseq-ge*: $decseq\ X \implies X \longrightarrow L \implies L \leq X\ n$
for $L :: 'a::linorder-topology$
 $\langle proof \rangle$

98.8 First countable topologies

class *first-countable-topology* = *topological-space* +
assumes *first-countable-basis*:
 $\exists A::nat \Rightarrow 'a\ set. (\forall i. x \in A\ i \wedge open\ (A\ i)) \wedge (\forall S. open\ S \wedge x \in S \longrightarrow (\exists i. A\ i \subseteq S))$

lemma (*in first-countable-topology*) *countable-basis-at-decseq*:
obtains $A :: nat \Rightarrow 'a\ set$ **where**
 $\bigwedge i. open\ (A\ i) \bigwedge i. x \in (A\ i)$
 $\bigwedge S. open\ S \implies x \in S \implies eventually\ (\lambda i. A\ i \subseteq S)\ sequentially$
 $\langle proof \rangle$

lemma (*in first-countable-topology*) *nhds-countable*:
obtains $X :: nat \Rightarrow 'a\ set$
where $decseq\ X \bigwedge n. open\ (X\ n) \bigwedge n. x \in X\ n\ nhds\ x = (INF\ n. principal\ (X\ n))$
 $\langle proof \rangle$

lemma (*in first-countable-topology*) *countable-basis*:
obtains $A :: nat \Rightarrow 'a\ set$ **where**
 $\bigwedge i. open\ (A\ i) \bigwedge i. x \in A\ i$
 $\bigwedge F. (\forall n. F\ n \in A\ n) \implies F \longrightarrow x$
 $\langle proof \rangle$

lemma (*in first-countable-topology*) *sequentially-imp-eventually-nhds-within*:
assumes $\forall f. (\forall n. f\ n \in s) \wedge f \longrightarrow a \longrightarrow eventually\ (\lambda n. P\ (f\ n))\ sequentially$
shows $eventually\ P\ (inf\ (nhds\ a)\ (principal\ s))$
 $\langle proof \rangle$

lemma (*in first-countable-topology*) *eventually-nhds-within-iff-sequentially*:
 $eventually\ P\ (inf\ (nhds\ a)\ (principal\ s)) \longleftrightarrow$
 $(\forall f. (\forall n. f\ n \in s) \wedge f \longrightarrow a \longrightarrow eventually\ (\lambda n. P\ (f\ n))\ sequentially)$
 $\langle proof \rangle$

lemma (*in first-countable-topology*) *eventually-nhds-iff-sequentially*:
 $eventually\ P\ (nhds\ a) \longleftrightarrow (\forall f. f \longrightarrow a \longrightarrow eventually\ (\lambda n. P\ (f\ n))\ sequentially)$
 $\langle proof \rangle$

lemma *Inf-as-limit*:

fixes $A::'a::\{\text{linorder-topology, first-countable-topology, complete-linorder}\}$ *set*
assumes $A \neq \{\}$
shows $\exists u. (\forall n. u\ n \in A) \wedge u \longrightarrow \text{Inf } A$
 $\langle \text{proof} \rangle$

lemma *tendsto-at-iff-sequentially*:

$(f \longrightarrow a) \text{ (at } x \text{ within } s) \longleftrightarrow (\forall X. (\forall i. X\ i \in s - \{x\}) \longrightarrow X \longrightarrow x \longrightarrow$
 $((f \circ X) \longrightarrow a))$
for $f :: 'a::\text{first-countable-topology} \Rightarrow -$
 $\langle \text{proof} \rangle$

lemma *approx-from-above-dense-linorder*:

fixes $x::'a::\{\text{dense-linorder, linorder-topology, first-countable-topology}\}$
assumes $x < y$
shows $\exists u. (\forall n. u\ n > x) \wedge (u \longrightarrow x)$
 $\langle \text{proof} \rangle$

lemma *approx-from-below-dense-linorder*:

fixes $x::'a::\{\text{dense-linorder, linorder-topology, first-countable-topology}\}$
assumes $x > y$
shows $\exists u. (\forall n. u\ n < x) \wedge (u \longrightarrow x)$
 $\langle \text{proof} \rangle$

98.9 Function limit at a point

abbreviation $LIM :: ('a::\text{topological-space} \Rightarrow 'b::\text{topological-space}) \Rightarrow 'a \Rightarrow 'b \Rightarrow$
 bool
 $(\langle \langle \text{notation} = \langle \text{infix } LIM \rangle \rangle (-) / -(-) / \rightarrow (-) \rangle [60, 0, 60] 60)$
where $f -a \rightarrow L \equiv (f \longrightarrow L) \text{ (at } a)$

lemma *tendsto-within-open*: $a \in S \implies \text{open } S \implies (f \longrightarrow l) \text{ (at } a \text{ within } S)$
 $\longleftrightarrow (f -a \rightarrow l)$
 $\langle \text{proof} \rangle$

lemma *tendsto-within-open-NO-MATCH*:

$a \in S \implies \text{NO-MATCH UNIV } S \implies \text{open } S \implies (f \longrightarrow l) \text{ (at } a \text{ within } S) \longleftrightarrow$
 $(f \longrightarrow l) \text{ (at } a)$
for $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{topological-space}$
 $\langle \text{proof} \rangle$

lemma *LIM-const-not-eq[tendsto-intros]*: $k \neq L \implies \neg (\lambda x. k) -a \rightarrow L$
for $a :: 'a::\text{perfect-space}$ **and** $k\ L :: 'b::\text{t2-space}$
 $\langle \text{proof} \rangle$

lemmas $LIM\text{-not-zero} = LIM\text{-const-not-eq}$ [where $L = 0$]

lemma *LIM-const-eq*: $(\lambda x. k) -a \rightarrow L \implies k = L$
for $a :: 'a::\text{perfect-space}$ **and** $k\ L :: 'b::\text{t2-space}$
 $\langle \text{proof} \rangle$

lemma *LIM-unique*: $f -a\rightarrow L \implies f -a\rightarrow M \implies L = M$
for $a :: 'a::\text{perfect-space}$ **and** $L\ M :: 'b::t2\text{-space}$
 $\langle \text{proof} \rangle$

lemma *LIM-Uniq*: $\exists_{\leq 1} L :: 'b::t2\text{-space}. f -a\rightarrow L$
for $a :: 'a::\text{perfect-space}$
 $\langle \text{proof} \rangle$

Limits are equal for functions equal except at limit point.

lemma *LIM-equal*: $\forall x. x \neq a \longrightarrow f\ x = g\ x \implies (f -a\rightarrow l) \longleftrightarrow (g -a\rightarrow l)$
 $\langle \text{proof} \rangle$

lemma *LIM-cong*: $a = b \implies (\bigwedge x. x \neq b \implies f\ x = g\ x) \implies l = m \implies (f -a\rightarrow l) \longleftrightarrow (g -b\rightarrow m)$
 $\langle \text{proof} \rangle$

lemma *tendsto-cong-limit*: $(f \longrightarrow l)\ F \implies k = l \implies (f \longrightarrow k)\ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-at-iff-tendsto-nhds*: $g -l\rightarrow g\ l \longleftrightarrow (g \longrightarrow g\ l)\ (\text{nhds } l)$
 $\langle \text{proof} \rangle$

lemma *tendsto-compose*: $g -l\rightarrow g\ l \implies (f \longrightarrow l)\ F \implies ((\lambda x. g\ (f\ x)) \longrightarrow g\ l)\ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-compose-eventually*:
 $g -l\rightarrow m \implies (f \longrightarrow l)\ F \implies \text{eventually } (\lambda x. f\ x \neq l)\ F \implies ((\lambda x. g\ (f\ x)) \longrightarrow m)\ F$
 $\langle \text{proof} \rangle$

lemma *LIM-compose-eventually*:
assumes $f -a\rightarrow b$
and $g -b\rightarrow c$
and $\text{eventually } (\lambda x. f\ x \neq b)\ (\text{at } a)$
shows $(\lambda x. g\ (f\ x)) -a\rightarrow c$
 $\langle \text{proof} \rangle$

lemma *tendsto-compose-filtermap*: $((g \circ f) \longrightarrow T)\ F \longleftrightarrow (g \longrightarrow T)\ (\text{filtermap } f\ F)$
 $\langle \text{proof} \rangle$

lemma *tendsto-compose-at*:
assumes $f: (f \longrightarrow y)\ F$ **and** $g: (g \longrightarrow z)\ (\text{at } y)$ **and** $fg: \text{eventually } (\lambda w. f\ w = y \longrightarrow g\ w = z)\ F$
shows $((g \circ f) \longrightarrow z)\ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-nhds-iff*: $(f \longrightarrow (c :: 'a :: t1\text{-space})) (nhds\ x) \longleftrightarrow f -x\rightarrow c \wedge f\ x = c$
 $\langle proof \rangle$

98.9.1 Relation of LIM and LIMSEQ

lemma (in *first-countable-topology*) *sequentially-imp-eventually-within*:
 $(\forall f. (\forall n. f\ n \in s \wedge f\ n \neq a) \wedge f \longrightarrow a \longrightarrow \text{eventually } (\lambda n. P\ (f\ n)) \text{ sequentially}) \implies$
 $\text{eventually } P\ (\text{at } a\ \text{within } s)$
 $\langle proof \rangle$

lemma (in *first-countable-topology*) *sequentially-imp-eventually-at*:
 $(\forall f. (\forall n. f\ n \neq a) \wedge f \longrightarrow a \longrightarrow \text{eventually } (\lambda n. P\ (f\ n)) \text{ sequentially}) \implies$
 $\text{eventually } P\ (\text{at } a)$
 $\langle proof \rangle$

lemma *LIMSEQ-SEQ-conv*:
 $(\forall S. (\forall n. S\ n \neq a) \wedge S \longrightarrow a \longrightarrow (\lambda n. X\ (S\ n)) \longrightarrow L) \longleftrightarrow X -a\rightarrow L$
 (is ?lhs=?rhs)
 for $a :: 'a :: \text{first-countable-topology}$ and $L :: 'b :: \text{topological-space}$
 $\langle proof \rangle$

lemma *sequentially-imp-eventually-at-left*:
fixes $a :: 'a :: \{\text{linorder-topology}, \text{first-countable-topology}\}$
assumes $b[simp]: b < a$
and $*$: $\bigwedge f. (\bigwedge n. b < f\ n) \implies (\bigwedge n. f\ n < a) \implies \text{incseq } f \implies f \longrightarrow a \implies$
 $\text{eventually } (\lambda n. P\ (f\ n)) \text{ sequentially}$
shows $\text{eventually } P\ (\text{at-left } a)$
 $\langle proof \rangle$

lemma *tendsto-at-left-sequentially*:
fixes $a\ b :: 'b :: \{\text{linorder-topology}, \text{first-countable-topology}\}$
assumes $b < a$
assumes $*$: $\bigwedge S. (\bigwedge n. S\ n < a) \implies (\bigwedge n. b < S\ n) \implies \text{incseq } S \implies S \longrightarrow$
 $a \implies$
 $(\lambda n. X\ (S\ n)) \longrightarrow L$
shows $(X \longrightarrow L) (\text{at-left } a)$
 $\langle proof \rangle$

lemma *sequentially-imp-eventually-at-right*:
fixes $a\ b :: 'a :: \{\text{linorder-topology}, \text{first-countable-topology}\}$
assumes $b[simp]: a < b$
assumes $*$: $\bigwedge f. (\bigwedge n. a < f\ n) \implies (\bigwedge n. f\ n < b) \implies \text{decseq } f \implies f \longrightarrow a$
 \implies
 $\text{eventually } (\lambda n. P\ (f\ n)) \text{ sequentially}$
shows $\text{eventually } P\ (\text{at-right } a)$
 $\langle proof \rangle$

lemma *tendsto-at-right-sequentially*:

fixes $a :: - :: \{\text{linorder-topology}, \text{first-countable-topology}\}$

assumes $a < b$

and $*$: $\bigwedge S. (\bigwedge n. a < S\ n) \implies (\bigwedge n. S\ n < b) \implies \text{decseq } S \implies S \longrightarrow a$

$\implies (\lambda n. X\ (S\ n)) \longrightarrow L$

shows $(X \longrightarrow L)\ (\text{at-right } a)$

<proof>

98.10 Continuity

98.10.1 Continuity on a set

definition *continuous-on* :: $'a\ \text{set} \Rightarrow ('a::\text{topological-space} \Rightarrow 'b::\text{topological-space}) \Rightarrow \text{bool}$

where *continuous-on* $s\ f \longleftrightarrow (\forall x \in s. (f \longrightarrow f\ x)\ (\text{at } x\ \text{within } s))$

lemma *continuous-on-cong* [*cong*]:

$s = t \implies (\bigwedge x. x \in t \implies f\ x = g\ x) \implies \text{continuous-on } s\ f \longleftrightarrow \text{continuous-on } t\ g$

<proof>

lemma *continuous-on-cong-simp*:

$s = t \implies (\bigwedge x. x \in t \implies f\ x = g\ x) \implies \text{continuous-on } s\ f \longleftrightarrow \text{continuous-on } t\ g$

<proof>

lemma *continuous-on-topological*:

continuous-on $s\ f \longleftrightarrow$

$(\forall x \in s. \forall B. \text{open } B \longrightarrow f\ x \in B \longrightarrow (\exists A. \text{open } A \wedge x \in A \wedge (\forall y \in s. y \in A \longrightarrow f\ y \in B)))$

<proof>

lemma *continuous-on-open-invariant*:

continuous-on $s\ f \longleftrightarrow (\forall B. \text{open } B \longrightarrow (\exists A. \text{open } A \wedge A \cap s = f^{-1} B \cap s))$

<proof>

lemma *continuous-on-open-vimage*:

open $s \implies \text{continuous-on } s\ f \longleftrightarrow (\forall B. \text{open } B \longrightarrow \text{open } (f^{-1} B \cap s))$

<proof>

corollary *continuous-imp-open-vimage*:

assumes *continuous-on* $s\ f$ *open* s *open* B $f^{-1} B \subseteq s$

shows *open* $(f^{-1} B)$

<proof>

corollary *open-vimage*[*continuous-intros*]:

assumes *open* s

and *continuous-on* $\text{UNIV } f$

shows *open* $(f^{-1} s)$

$\langle \text{proof} \rangle$

lemma *continuous-on-closed-invariant*:

$\text{continuous-on } s \ f \longleftrightarrow (\forall B. \text{closed } B \longrightarrow (\exists A. \text{closed } A \wedge A \cap s = f - ' B \cap s))$
 $\langle \text{proof} \rangle$

lemma *continuous-on-closed-vimage*:

$\text{closed } s \implies \text{continuous-on } s \ f \longleftrightarrow (\forall B. \text{closed } B \longrightarrow \text{closed } (f - ' B \cap s))$
 $\langle \text{proof} \rangle$

corollary *closed-vimage-Int[continuous-intros]*:

assumes $\text{closed } s$
and $\text{continuous-on } t \ f$
and $t: \text{closed } t$
shows $\text{closed } (f - ' s \cap t)$
 $\langle \text{proof} \rangle$

corollary *closed-vimage[continuous-intros]*:

assumes $\text{closed } s$
and $\text{continuous-on UNIV } f$
shows $\text{closed } (f - ' s)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-empty [simp]*: $\text{continuous-on } \{\} \ f$

$\langle \text{proof} \rangle$

lemma *continuous-on-sing [simp]*: $\text{continuous-on } \{x\} \ f$

$\langle \text{proof} \rangle$

lemma *continuous-on-open-Union*:

$(\bigwedge s. s \in S \implies \text{open } s) \implies (\bigwedge s. s \in S \implies \text{continuous-on } s \ f) \implies \text{continuous-on } (\bigcup S) \ f$
 $\langle \text{proof} \rangle$

lemma *continuous-on-open-UN*:

$(\bigwedge s. s \in S \implies \text{open } (A \ s)) \implies (\bigwedge s. s \in S \implies \text{continuous-on } (A \ s) \ f) \implies \text{continuous-on } (\bigcup_{s \in S} A \ s) \ f$
 $\langle \text{proof} \rangle$

lemma *continuous-on-open-Un*:

$\text{open } s \implies \text{open } t \implies \text{continuous-on } s \ f \implies \text{continuous-on } t \ f \implies \text{continuous-on } (s \cup t) \ f$
 $\langle \text{proof} \rangle$

lemma *continuous-on-closed-Un*:

$\text{closed } s \implies \text{closed } t \implies \text{continuous-on } s \ f \implies \text{continuous-on } t \ f \implies \text{continuous-on } (s \cup t) \ f$
 $\langle \text{proof} \rangle$

lemma *continuous-on-closed-Union*:

assumes *finite I*

$\bigwedge i. i \in I \implies \text{closed } (U\ i)$

$\bigwedge i. i \in I \implies \text{continuous-on } (U\ i)\ f$

shows *continuous-on* $(\bigcup i \in I. U\ i)\ f$

<proof>

lemma *continuous-on-If*:

assumes *closed: closed s closed t*

and *cont: continuous-on s f continuous-on t g*

and *P: $\bigwedge x. x \in s \implies \neg P\ x \implies f\ x = g\ x \bigwedge x. x \in t \implies P\ x \implies f\ x = g\ x$*

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>

lemma *continuous-on-cases*:

closed s \implies closed t \implies continuous-on s f \implies continuous-on t g \implies

$\forall x. (x \in s \wedge \neg P\ x) \vee (x \in t \wedge P\ x) \longrightarrow f\ x = g\ x \implies$

continuous-on $(s \cup t)\ (\lambda x. \text{if } P\ x \text{ then } f\ x \text{ else } g\ x)$

<proof>

lemma *continuous-on-id*[*continuous-intros,simp*]: *continuous-on s* $(\lambda x. x)$

<proof>

lemma *continuous-on-id'*[*continuous-intros,simp*]: *continuous-on s id*

<proof>

lemma *continuous-on-const*[*continuous-intros,simp*]: *continuous-on s* $(\lambda x. c)$

<proof>

lemma *continuous-on-subset*: *continuous-on s f $\implies t \subseteq s \implies$ continuous-on t f*

<proof>

lemma *continuous-on-compose*[*continuous-intros*]:

continuous-on s f \implies continuous-on $(f\ ' s)\ g \implies \text{continuous-on } s\ (g \circ f)$

<proof>

lemma *continuous-on-compose2*:

continuous-on t g \implies continuous-on s f $\implies f\ ' s \subseteq t \implies$ continuous-on s $(\lambda x. g\ (f\ x))$

<proof>

lemma *continuous-on-generate-topology*:

assumes **: open = generate-topology X*

and ***:* $\bigwedge B. B \in X \implies \exists C. \text{open } C \wedge C \cap A = f\ -' B \cap A$

shows *continuous-on A f*

<proof>

lemma *continuous-onI-mono*:

fixes $f :: 'a::\text{linorder-topology} \Rightarrow 'b::\{\text{dense-order}, \text{linorder-topology}\}$
assumes $\text{open } (f'A)$
and $\text{mono}: \bigwedge x y. x \in A \Longrightarrow y \in A \Longrightarrow x \leq y \Longrightarrow f x \leq f y$
shows $\text{continuous-on } A f$
 $\langle \text{proof} \rangle$

lemma $\text{continuous-on-IccI}$:
 $\llbracket (f \longrightarrow f a) \text{ (at-right } a);$
 $(f \longrightarrow f b) \text{ (at-left } b);$
 $(\bigwedge x. a < x \Longrightarrow x < b \Longrightarrow f -x \rightarrow f x); a < b \rrbracket \Longrightarrow$
 $\text{continuous-on } \{a .. b\} f$
for $a::'a::\text{linorder-topology}$
 $\langle \text{proof} \rangle$

lemma
fixes $a b::'a::\text{linorder-topology}$
assumes $\text{continuous-on } \{a .. b\} f$ $a < b$
shows $\text{continuous-on-Icc-at-rightD}: (f \longrightarrow f a) \text{ (at-right } a)$
and $\text{continuous-on-Icc-at-leftD}: (f \longrightarrow f b) \text{ (at-left } b)$
 $\langle \text{proof} \rangle$

lemma $\text{continuous-on-discrete [simp]}$:
 $\text{continuous-on } A (f :: 'a :: \text{discrete-topology} \Rightarrow -)$
 $\langle \text{proof} \rangle$

lemma $\text{continuous-on-of-nat [continuous-intros]}$:
assumes $\text{continuous-on } A f$
shows $\text{continuous-on } A (\lambda n. \text{of-nat } (f n))$
 $\langle \text{proof} \rangle$

lemma $\text{continuous-on-of-int [continuous-intros]}$:
assumes $\text{continuous-on } A f$
shows $\text{continuous-on } A (\lambda n. \text{of-int } (f n))$
 $\langle \text{proof} \rangle$

98.10.2 Continuity at a point

definition $\text{continuous} :: 'a::\text{t2-space filter} \Rightarrow ('a \Rightarrow 'b::\text{topological-space}) \Rightarrow \text{bool}$
where $\text{continuous } F f \longleftrightarrow (f \longrightarrow f (\text{Lim } F (\lambda x. x))) F$

lemma $\text{continuous-bot[continuous-intros, simp]}$: $\text{continuous bot } f$
 $\langle \text{proof} \rangle$

lemma $\text{continuous-trivial-limit}$: $\text{trivial-limit net} \Longrightarrow \text{continuous net } f$
 $\langle \text{proof} \rangle$

lemma continuous-within : $\text{continuous (at } x \text{ within } s) f \longleftrightarrow (f \longrightarrow f x) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *continuous-within-topological*:

continuous (at x within s) f \longleftrightarrow
 $(\forall B. \text{open } B \longrightarrow f\ x \in B \longrightarrow (\exists A. \text{open } A \wedge x \in A \wedge (\forall y \in s. y \in A \longrightarrow f\ y \in B)))$
 ⟨proof⟩

lemma *continuous-within-compose[continuous-intros]*:

continuous (at x within s) f \implies *continuous (at (f x) within f ' s) g* \implies
continuous (at x within s) (g o f)
 ⟨proof⟩

lemma *continuous-within-compose2*:

continuous (at x within s) f \implies *continuous (at (f x) within f ' s) g* \implies
continuous (at x within s) ($\lambda x. g\ (f\ x)$)
 ⟨proof⟩

lemma *continuous-at*: *continuous (at x) f* \longleftrightarrow *f -x→ f x*

⟨proof⟩

lemma *continuous-ident[continuous-intros, simp]*: *continuous (at x within S) ($\lambda x. x$)*

⟨proof⟩

lemma *continuous-id[continuous-intros, simp]*: *continuous (at x within S) id*

⟨proof⟩

lemma *continuous-const[continuous-intros, simp]*: *continuous F ($\lambda x. c$)*

⟨proof⟩

lemma *continuous-on-eq-continuous-within*:

continuous-on s f \longleftrightarrow $(\forall x \in s. \text{continuous (at } x \text{ within } s) f)$
 ⟨proof⟩

lemma *continuous-discrete [simp]*:

continuous (at x within A) (f :: 'a :: discrete-topology \Rightarrow -)
 ⟨proof⟩

Continuity in terms of open preimages.

lemma *continuous-at-open*:

continuous (at x) f \longleftrightarrow $(\forall t. \text{open } t \wedge f\ x \in t \longrightarrow (\exists S. \text{open } S \wedge x \in S \wedge (\forall x' \in S. (f\ x') \in t)))$
 ⟨proof⟩

lemma *continuous-imp-tendsto*:

assumes *continuous (at x0) f* **and** *x \longrightarrow x0*

shows *(f o x) \longrightarrow (f x0)*

⟨proof⟩

abbreviation $isCont :: ('a::t2-space \Rightarrow 'b::topological-space) \Rightarrow 'a \Rightarrow bool$
where $isCont\ f\ a \equiv continuous\ (at\ a)\ f$

lemma $isCont-def$: $isCont\ f\ a \longleftrightarrow f\ -a \rightarrow f\ a$
 $\langle proof \rangle$

lemma $isContD$: $isCont\ f\ x \Longrightarrow f\ -x \rightarrow f\ x$
 $\langle proof \rangle$

lemma $isCont-cong$:
assumes $eventually\ (\lambda x. f\ x = g\ x)\ (nhds\ x)$
shows $isCont\ f\ x \longleftrightarrow isCont\ g\ x$
 $\langle proof \rangle$

lemma $continuous-at-imp-continuous-at-within$: $isCont\ f\ x \Longrightarrow continuous\ (at\ x\ within\ s)\ f$
 $\langle proof \rangle$

lemma $continuous-on-eq-continuous-at$: $open\ s \Longrightarrow continuous-on\ s\ f \longleftrightarrow (\forall x \in s. isCont\ f\ x)$
 $\langle proof \rangle$

lemma $continuous-within-open$: $a \in A \Longrightarrow open\ A \Longrightarrow continuous\ (at\ a\ within\ A)\ f \longleftrightarrow isCont\ f\ a$
 $\langle proof \rangle$

lemma $continuous-at-imp-continuous-on$: $\forall x \in s. isCont\ f\ x \Longrightarrow continuous-on\ s\ f$
 $\langle proof \rangle$

lemma $isCont-o2$: $isCont\ f\ a \Longrightarrow isCont\ g\ (f\ a) \Longrightarrow isCont\ (\lambda x. g\ (f\ x))\ a$
 $\langle proof \rangle$

lemma $continuous-at-compose[continuous-intros]$: $isCont\ f\ a \Longrightarrow isCont\ g\ (f\ a) \Longrightarrow isCont\ (g \circ f)\ a$
 $\langle proof \rangle$

lemma $isCont-tendsto-compose$: $isCont\ g\ l \Longrightarrow (f \longrightarrow l)\ F \Longrightarrow ((\lambda x. g\ (f\ x)) \longrightarrow g\ l)\ F$
 $\langle proof \rangle$

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 $((\lambda x. f\ (g\ x)) \longrightarrow f\ l)\ F$
 $\langle proof \rangle$

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))
 \langle *proof* \rangle

lemma *at-within-isCont-imp-nhds*:

fixes $f :: 'a :: \{t2\text{-space}, \text{perfect-space}\} \Rightarrow 'b :: t2\text{-space}$
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
 \langle *proof* \rangle

lemma *filtermap-nhds-open-map'*:

assumes *cont*: *isCont* f a
and *open* A $a \in A$
and *open-map*: $\bigwedge S. \text{open } S \implies S \subseteq A \implies \text{open } (f \text{ ` } S)$
shows *filtermap* f (*nhds* a) = *nhds* (f a)
 \langle *proof* \rangle

lemma *filtermap-nhds-open-map*:

assumes *cont*: *isCont* f a
and *open-map*: $\bigwedge S. \text{open } S \implies \text{open } (f \text{ ` } S)$
shows *filtermap* f (*nhds* a) = *nhds* (f a)
 \langle *proof* \rangle

lemma *continuous-at-split*:

continuous (at x) $f \iff$ *continuous* (at-left x) $f \wedge$ *continuous* (at-right x) f
for $x :: 'a :: \text{linorder-topology}$
 \langle *proof* \rangle

lemma *continuous-on-max* [*continuous-intros*]:

fixes f $g :: 'a :: \text{topological-space} \Rightarrow 'b :: \text{linorder-topology}$
shows *continuous-on* A $f \implies$ *continuous-on* A $g \implies$ *continuous-on* A ($\lambda x. \text{max}$
 (f x) (g x))
 \langle *proof* \rangle

lemma *continuous-on-min* [*continuous-intros*]:

fixes f $g :: 'a :: \text{topological-space} \Rightarrow 'b :: \text{linorder-topology}$
shows *continuous-on* A $f \implies$ *continuous-on* A $g \implies$ *continuous-on* A ($\lambda x. \text{min}$
 (f x) (g x))
 \langle *proof* \rangle

lemma *continuous-max* [*continuous-intros*]:

fixes $f :: 'a :: t2\text{-space} \Rightarrow 'b :: \text{linorder-topology}$
shows $\llbracket \text{continuous } F$ $f; \text{continuous } F$ $g \rrbracket \implies$ *continuous* F ($\lambda x. (\text{max } (f$ $x)$ (g
 x)))
 \langle *proof* \rangle

lemma *continuous-min* [*continuous-intros*]:

fixes $f :: 'a :: t2\text{-space} \Rightarrow 'b :: \text{linorder-topology}$
shows $\llbracket \text{continuous } F$ $f; \text{continuous } F$ $g \rrbracket \implies$ *continuous* F ($\lambda x. (\text{min } (f$ $x)$ (g x)))

<proof>

The following open/closed Collect lemmas are ported from Sébastien Gouëzel’s *Ergodic-Theory*.

lemma *open-Collect-neq*:

fixes $f\ g :: 'a::\text{topological-space} \Rightarrow 'b::t2\text{-space}$

assumes f : *continuous-on UNIV* f **and** g : *continuous-on UNIV* g

shows *open* $\{x. f\ x \neq g\ x\}$

<proof>

lemma *closed-Collect-eq*:

fixes $f\ g :: 'a::\text{topological-space} \Rightarrow 'b::t2\text{-space}$

assumes f : *continuous-on UNIV* f **and** g : *continuous-on UNIV* g

shows *closed* $\{x. f\ x = g\ x\}$

<proof>

lemma *open-Collect-less*:

fixes $f\ g :: 'a::\text{topological-space} \Rightarrow 'b::\text{linorder-topology}$

assumes f : *continuous-on UNIV* f **and** g : *continuous-on UNIV* g

shows *open* $\{x. f\ x < g\ x\}$

<proof>

lemma *closed-Collect-le*:

fixes $f\ g :: 'a::\text{topological-space} \Rightarrow 'b::\text{linorder-topology}$

assumes f : *continuous-on UNIV* f

and g : *continuous-on UNIV* g

shows *closed* $\{x. f\ x \leq g\ x\}$

<proof>

98.10.3 Open-cover compactness

context *topological-space*

begin

definition *compact* :: $'a\ \text{set} \Rightarrow \text{bool}$ **where**

compact-eq-Heine-Borel:

$\text{compact } S \longleftrightarrow (\forall C. (\forall c \in C. \text{open } c) \wedge S \subseteq \bigcup C \longrightarrow (\exists D \subseteq C. \text{finite } D \wedge S \subseteq \bigcup D))$

lemma *compactI*:

assumes $\bigwedge C. \forall t \in C. \text{open } t \Longrightarrow s \subseteq \bigcup C \Longrightarrow \exists C'. C' \subseteq C \wedge \text{finite } C' \wedge s \subseteq \bigcup C'$

shows *compact* s

<proof>

lemma *compact-empty[simp]*: *compact* $\{\}$

<proof>

lemma *compactE*:

assumes *compact* S $S \subseteq \bigcup \mathcal{T} \wedge B. B \in \mathcal{T} \implies \text{open } B$
obtains \mathcal{T}' **where** $\mathcal{T}' \subseteq \mathcal{T}$ *finite* \mathcal{T}' $S \subseteq \bigcup \mathcal{T}'$
 $\langle \text{proof} \rangle$

lemma *compactE-image*:

assumes *compact* S
and *opn*: $\bigwedge T. T \in C \implies \text{open } (f\ T)$
and $S: S \subseteq (\bigcup_{c \in C}. f\ c)$
obtains C' **where** $C' \subseteq C$ **and** *finite* C' **and** $S \subseteq (\bigcup_{c \in C'}. f\ c)$
 $\langle \text{proof} \rangle$

lemma *compact-Int-closed* [intro]:

assumes *compact* S
and *closed* T
shows *compact* $(S \cap T)$
 $\langle \text{proof} \rangle$

lemma *compact-diff*: $\llbracket \text{compact } S; \text{open } T \rrbracket \implies \text{compact}(S - T)$
 $\langle \text{proof} \rangle$

lemma *inj-setminus*: *inj-on* *uminus* $(A::'a \text{ set set})$
 $\langle \text{proof} \rangle$

98.11 Finite intersection property

lemma *compact-fip*:

compact $U \longleftrightarrow$
 $(\forall A. (\forall a \in A. \text{closed } a) \longrightarrow (\forall B \subseteq A. \text{finite } B \longrightarrow U \cap \bigcap B \neq \{\})) \longrightarrow U \cap \bigcap A \neq \{\}$
(is - \longleftrightarrow ?R)
 $\langle \text{proof} \rangle$

lemma *compact-imp-fip*:

assumes *compact* S
and $\bigwedge T. T \in F \implies \text{closed } T$
and $\bigwedge F'. \text{finite } F' \implies F' \subseteq F \implies S \cap (\bigcap F') \neq \{\}$
shows $S \cap (\bigcap F) \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *compact-imp-fip-image*:

assumes *compact* s
and $P: \bigwedge i. i \in I \implies \text{closed } (f\ i)$
and $Q: \bigwedge I'. \text{finite } I' \implies I' \subseteq I \implies (s \cap (\bigcap_{i \in I'}. f\ i) \neq \{\})$
shows $s \cap (\bigcap_{i \in I}. f\ i) \neq \{\}$
 $\langle \text{proof} \rangle$

end

lemma (in *t2-space*) *compact-imp-closed*:

assumes *compact s*
shows *closed s*
 $\langle \text{proof} \rangle$

lemma *compact-continuous-image*:
assumes *f: continuous-on s f*
and *s: compact s*
shows *compact (f ` s)*
 $\langle \text{proof} \rangle$

lemma *continuous-on-inv*:
fixes *f :: 'a::topological-space \Rightarrow 'b::t2-space*
assumes *continuous-on s f*
and *compact s*
and $\forall x \in s. g (f x) = x$
shows *continuous-on (f ` s) g*
 $\langle \text{proof} \rangle$

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) (the-inv-into s f)*
 $\langle \text{proof} \rangle$

lemma (*in linorder-topology*) *compact-attains-sup*:
assumes *compact S S \neq {}*
shows $\exists s \in S. \forall t \in S. t \leq s$
 $\langle \text{proof} \rangle$

lemma (*in linorder-topology*) *compact-attains-inf*:
assumes *compact S S \neq {}*
shows $\exists s \in S. \forall t \in S. s \leq t$
 $\langle \text{proof} \rangle$

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 ($\exists x \in s. \forall y \in s. f y \leq f x$)*
 $\langle \text{proof} \rangle$

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 ($\exists x \in s. \forall y \in s. f x \leq f y$)*
 $\langle \text{proof} \rangle$

98.12 Connectedness

context *topological-space*

begin

definition *connected* $S \longleftrightarrow$

$\neg (\exists A B. \text{open } A \wedge \text{open } B \wedge S \subseteq A \cup B \wedge A \cap B \cap S = \{\} \wedge A \cap S \neq \{\} \wedge B \cap S \neq \{\})$

lemma *connectedI*:

$(\bigwedge A B. \text{open } A \implies \text{open } B \implies A \cap U \neq \{\} \implies B \cap U \neq \{\} \implies A \cap B \cap U = \{\} \implies U \subseteq A \cup B \implies \text{False})$
 $\implies \text{connected } U$
 $\langle \text{proof} \rangle$

lemma *connected-empty* [simp]: *connected* $\{\}$
 $\langle \text{proof} \rangle$

lemma *connected-sing* [simp]: *connected* $\{x\}$
 $\langle \text{proof} \rangle$

lemma *connectedD*:

$\text{connected } A \implies \text{open } U \implies \text{open } V \implies U \cap V \cap A = \{\} \implies A \subseteq U \cup V$
 $\implies U \cap A = \{\} \vee V \cap A = \{\}$
 $\langle \text{proof} \rangle$

end

lemma *connected-closed*:

connected $s \longleftrightarrow$
 $\neg (\exists A B. \text{closed } A \wedge \text{closed } B \wedge s \subseteq A \cup B \wedge A \cap B \cap s = \{\} \wedge A \cap s \neq \{\} \wedge B \cap s \neq \{\})$
 $\langle \text{proof} \rangle$

lemma *connected-closedD*:

$\llbracket \text{connected } s; A \cap B \cap s = \{\}; s \subseteq A \cup B; \text{closed } A; \text{closed } B \rrbracket \implies A \cap s = \{\}$
 $\vee B \cap s = \{\}$
 $\langle \text{proof} \rangle$

lemma *connected-Union*:

assumes *cs*: $\bigwedge s. s \in S \implies \text{connected } s$
and *ne*: $\bigcap S \neq \{\}$
shows *connected* $(\bigcup S)$
 $\langle \text{proof} \rangle$

lemma *connected-Un*: *connected* $s \implies \text{connected } t \implies s \cap t \neq \{\} \implies \text{connected } (s \cup t)$
 $\langle \text{proof} \rangle$

lemma *connected-diff-open-from-closed*:

assumes *st*: $s \subseteq t$
and *tu*: $t \subseteq u$

and s : *open* s
and t : *closed* t
and u : *connected* u
and ts : *connected* $(t - s)$
shows *connected* $(u - s)$
 \langle *proof* \rangle

lemma *connected-iff-const*:
fixes $S :: 'a::\text{topological-space set}$
shows *connected* $S \longleftrightarrow (\forall P::'a \Rightarrow \text{bool. continuous-on } S P \longrightarrow (\exists c. \forall s \in S. P s = c))$
 \langle *proof* \rangle

lemma *connectedD-const*: *connected* $S \Longrightarrow \text{continuous-on } S P \Longrightarrow \exists c. \forall s \in S. P s = c$
for $P :: 'a::\text{topological-space} \Rightarrow \text{bool}$
 \langle *proof* \rangle

lemma *connectedI-const*:
 $(\bigwedge P::'a::\text{topological-space} \Rightarrow \text{bool. continuous-on } S P \Longrightarrow \exists c. \forall s \in S. P s = c) \Longrightarrow \text{connected } S$
 \langle *proof* \rangle

lemma *connected-local-const*:
assumes *connected* A $a \in A$ $b \in A$
and $*$: $\forall a \in A. \text{eventually } (\lambda b. f a = f b) \text{ (at } a \text{ within } A)$
shows $f a = f b$
 \langle *proof* \rangle

lemma (in *linorder-topology*) *connectedD-interval*:
assumes *connected* U
and xy : $x \in U$ $y \in U$
and $x \leq z$ $z \leq y$
shows $z \in U$
 \langle *proof* \rangle

lemma (in *linorder-topology*) *not-in-connected-cases*:
assumes *conn*: *connected* S
assumes *nbdd*: $x \notin S$
assumes *ne*: $S \neq \{\}$
obtains *bdd-above* $S \bigwedge y. y \in S \Longrightarrow x \geq y \mid \text{bdd-below } S \bigwedge y. y \in S \Longrightarrow x \leq y$
 \langle *proof* \rangle

lemma *connected-continuous-image*:
assumes $*$: *continuous-on* s f
and *connected* s
shows *connected* $(f \text{ ` } s)$
 \langle *proof* \rangle

lemma *connected-Un-UN*:
assumes *connected A* \wedge *X. X ∈ B ⇒ connected X* \wedge *X. X ∈ B ⇒ A ∩ X ≠ {}*
shows *connected (A ∪ ⋃ B)*
 ⟨*proof*⟩

99 Linear Continuum Topologies

class *linear-continuum-topology* = *linorder-topology* + *linear-continuum*
begin

lemma *Inf-notin-open*:
assumes *A: open A*
and bnd: $\forall a \in A. x < a$
shows *Inf A ∉ A*
 ⟨*proof*⟩

lemma *Sup-notin-open*:
assumes *A: open A*
and bnd: $\forall a \in A. a < x$
shows *Sup A ∉ A*
 ⟨*proof*⟩

end

instance *linear-continuum-topology* \subseteq *perfect-space*
 ⟨*proof*⟩

lemma *connectedI-interval*:
fixes *U :: 'a :: linear-continuum-topology set*
assumes *: $\bigwedge x y z. x \in U \Rightarrow y \in U \Rightarrow x \leq z \Rightarrow z \leq y \Rightarrow z \in U$
shows *connected U*
 ⟨*proof*⟩

lemma *connected-iff-interval*: *connected U* \longleftrightarrow $(\forall x \in U. \forall y \in U. \forall z. x \leq z \longrightarrow z \leq y \longrightarrow z \in U)$
for *U :: 'a :: linear-continuum-topology set*
 ⟨*proof*⟩

lemma *connected-UNIV[simp]*: *connected (UNIV :: 'a :: linear-continuum-topology set)*
 ⟨*proof*⟩

lemma *connected-Ioi[simp]*: *connected {a <..}*
for *a :: 'a :: linear-continuum-topology*
 ⟨*proof*⟩

lemma *connected-Ici[simp]*: *connected {a..}*
for *a :: 'a :: linear-continuum-topology*
 ⟨*proof*⟩

lemma *connected-Iio*[simp]: *connected* $\{.. a \}$
for $a :: 'a::linear-continuum-topology$
 $\langle proof \rangle$

lemma *connected-Iic*[simp]: *connected* $\{.. a \}$
for $a :: 'a::linear-continuum-topology$
 $\langle proof \rangle$

lemma *connected-Ioo*[simp]: *connected* $\{a<.. b \}$
for $a\ b :: 'a::linear-continuum-topology$
 $\langle proof \rangle$

lemma *connected-Ioc*[simp]: *connected* $\{a<.. b \}$
for $a\ b :: 'a::linear-continuum-topology$
 $\langle proof \rangle$

lemma *connected-Ico*[simp]: *connected* $\{a.. b \}$
for $a\ b :: 'a::linear-continuum-topology$
 $\langle proof \rangle$

lemma *connected-Icc*[simp]: *connected* $\{a.. b \}$
for $a\ b :: 'a::linear-continuum-topology$
 $\langle proof \rangle$

lemma *connected-contains-Ioo*:
fixes $A :: 'a :: linorder-topology\ set$
assumes *connected* $A\ a \in A\ b \in A$ **shows** $\{a <.. b \} \subseteq A$
 $\langle proof \rangle$

lemma *connected-contains-Icc*:
fixes $A :: 'a::linorder-topology\ set$
assumes *connected* $A\ a \in A\ b \in A$
shows $\{a.. b \} \subseteq A$
 $\langle proof \rangle$

99.1 Intermediate Value Theorem

lemma *IVT'*:
fixes $f :: 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
assumes $y: f\ a \leq y\ y \leq f\ b\ a \leq b$
and $*$: *continuous-on* $\{a .. b\}\ f$
shows $\exists x. a \leq x \wedge x \leq b \wedge f\ x = y$
 $\langle proof \rangle$

lemma *IVT2'*:
fixes $f :: 'a :: linear-continuum-topology \Rightarrow 'b :: linorder-topology$
assumes $y: f\ b \leq y\ y \leq f\ a\ a \leq b$
and $*$: *continuous-on* $\{a .. b\}\ f$

shows $\exists x. a \leq x \wedge x \leq b \wedge f x = y$
 $\langle proof \rangle$

lemma *IVT*:

fixes $f :: 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
shows $f a \leq y \Longrightarrow y \leq f b \Longrightarrow a \leq b \Longrightarrow (\forall x. a \leq x \wedge x \leq b \longrightarrow isCont f x)$
 \Longrightarrow
 $\exists x. a \leq x \wedge x \leq b \wedge f x = y$
 $\langle proof \rangle$

lemma *IVT2*:

fixes $f :: 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
shows $f b \leq y \Longrightarrow y \leq f a \Longrightarrow a \leq b \Longrightarrow (\forall x. a \leq x \wedge x \leq b \longrightarrow isCont f x)$
 \Longrightarrow
 $\exists x. a \leq x \wedge x \leq b \wedge f x = y$
 $\langle proof \rangle$

lemma *continuous-inj-imp-mono*:

fixes $f :: 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
assumes $x: a < x \wedge x < b$
and $cont: continuous-on \{a..b\} f$
and $inj: inj-on f \{a..b\}$
shows $(f a < f x \wedge f x < f b) \vee (f b < f x \wedge f x < f a)$
 $\langle proof \rangle$

lemma *continuous-at-Sup-mono*:

fixes $f :: 'a::\{linorder-topology, conditionally-complete-linorder\} \Rightarrow$
 $'b::\{linorder-topology, conditionally-complete-linorder\}$
assumes $mono f$
and $cont: continuous (at-left (Sup S)) f$
and $S: S \neq \{\} \text{ bdd-above } S$
shows $f (Sup S) = (SUP s \in S. f s)$
 $\langle proof \rangle$

lemma *continuous-at-Sup-antimono*:

fixes $f :: 'a::\{linorder-topology, conditionally-complete-linorder\} \Rightarrow$
 $'b::\{linorder-topology, conditionally-complete-linorder\}$
assumes $antimono f$
and $cont: continuous (at-left (Sup S)) f$
and $S: S \neq \{\} \text{ bdd-above } S$
shows $f (Sup S) = (INF s \in S. f s)$
 $\langle proof \rangle$

lemma *continuous-at-Inf-mono*:

fixes $f :: 'a::\{linorder-topology, conditionally-complete-linorder\} \Rightarrow$
 $'b::\{linorder-topology, conditionally-complete-linorder\}$
assumes $mono f$
and $cont: continuous (at-right (Inf S)) f$
and $S: S \neq \{\} \text{ bdd-below } S$

shows $f \text{ (Inf } S) = (\text{INF } s \in S. f \ s)$
 $\langle \text{proof} \rangle$

lemma *continuous-at-Inf-antimono*:

fixes $f :: 'a :: \{\text{linorder-topology, conditionally-complete-linorder}\} \Rightarrow$
 $'b :: \{\text{linorder-topology, conditionally-complete-linorder}\}$
assumes *antimono* f
and *cont*: *continuous* (*at-right* (*Inf* S)) f
and $S: S \neq \{\}$ *bdd-below* S
shows $f \text{ (Inf } S) = (\text{SUP } s \in S. f \ s)$
 $\langle \text{proof} \rangle$

99.2 Uniform spaces

class *uniformity* =
fixes *uniformity* :: $('a \times 'a)$ *filter*
begin

abbreviation *uniformity-on* :: $'a$ *set* $\Rightarrow ('a \times 'a)$ *filter*
where *uniformity-on* $s \equiv \text{inf } \text{uniformity } (\text{principal } (s \times s))$

end

lemma *uniformity-Abort*:

uniformity =
 $\text{Filter.abstract-filter } (\lambda u. \text{Code.abort } (\text{STR } "uniformity \text{ is not executable'") } (\lambda u.$
 $\text{uniformity}))$
 $\langle \text{proof} \rangle$

class *open-uniformity* = *open* + *uniformity* +
assumes *open-uniformity*:
 $\bigwedge U. \text{open } U \longleftrightarrow (\forall x \in U. \text{eventually } (\lambda(x', y). x' = x \longrightarrow y \in U) \text{ uniformity})$
begin

subclass *topological-space*
 $\langle \text{proof} \rangle$

end

class *uniform-space* = *open-uniformity* +
assumes *uniformity-refl*: $\text{eventually } E \text{ uniformity} \Longrightarrow E \ (x, x)$
and *uniformity-sym*: $\text{eventually } E \text{ uniformity} \Longrightarrow \text{eventually } (\lambda(x, y). E \ (y, x))$
uniformity
and *uniformity-trans*:
 $\text{eventually } E \text{ uniformity} \Longrightarrow$
 $\exists D. \text{eventually } D \text{ uniformity} \wedge (\forall x \ y \ z. D \ (x, y) \longrightarrow D \ (y, z) \longrightarrow E \ (x, z))$
begin

lemma *uniformity-bot*: *uniformity* $\neq \text{bot}$

<proof>

lemma *uniformity-trans'*:

eventually E uniformity \implies

eventually $(\lambda((x, y), (y', z)). y = y' \longrightarrow E(x, z)) (uniformity \times_F uniformity)$

<proof>

lemma *uniformity-transE*:

assumes *eventually E uniformity*

obtains *D where eventually D uniformity $\bigwedge x y z. D(x, y) \implies D(y, z) \implies E(x, z)$*

<proof>

lemma *eventually-nhds-uniformity*:

eventually $P(nhds\ x) \longleftrightarrow eventually (\lambda(x', y). x' = x \longrightarrow P\ y)$ uniformity

(is - $\longleftrightarrow ?N\ P\ x$)

<proof>

99.2.1 Totally bounded sets

definition *totally-bounded* :: 'a set \Rightarrow bool

where *totally-bounded* $S \longleftrightarrow$

$(\forall E. eventually\ E\ uniformity \longrightarrow (\exists X. finite\ X \wedge (\forall s \in S. \exists x \in X. E(x, s))))$

lemma *totally-bounded-empty[iff]*: *totally-bounded* {}

<proof>

lemma *totally-bounded-subset*: *totally-bounded* $S \implies T \subseteq S \implies totally-bounded\ T$

<proof>

lemma *totally-bounded-Union[intro]*:

assumes *$M: finite\ M \bigwedge S. S \in M \implies totally-bounded\ S$*

shows *totally-bounded $(\bigcup M)$*

<proof>

99.2.2 Cauchy filter

definition *cauchy-filter* :: 'a filter \Rightarrow bool

where *cauchy-filter* $F \longleftrightarrow F \times_F F \leq uniformity$

definition *Cauchy* :: (nat \Rightarrow 'a) \Rightarrow bool

where *Cauchy-uniform*: *Cauchy* $X = cauchy-filter\ (filtermap\ X\ sequentially)$

lemma *Cauchy-uniform-iff*:

Cauchy $X \longleftrightarrow (\forall P. eventually\ P\ uniformity \longrightarrow (\exists N. \forall n \geq N. \forall m \geq N. P(X\ n, X\ m)))$

<proof>

lemma *nhds-imp-cauchy-filter*:

assumes *: $F \leq nhds\ x$
shows *cauchy-filter* F
 $\langle proof \rangle$

lemma *LIMSEQ-imp-Cauchy*: $X \longrightarrow x \implies Cauchy\ X$
 $\langle proof \rangle$

lemma *Cauchy-subseq-Cauchy*:
assumes *Cauchy* X *strict-mono* f
shows *Cauchy* $(X \circ f)$
 $\langle proof \rangle$

lemma *convergent-Cauchy*: *convergent* $X \implies Cauchy\ X$
 $\langle proof \rangle$

definition *complete* :: 'a set \Rightarrow bool
where *complete-uniform*: *complete* $S \longleftrightarrow$
 $(\forall F \leq principal\ S. F \neq bot \longrightarrow cauchy-filter\ F \longrightarrow (\exists x \in S. F \leq nhds\ x))$

lemma (*in uniform-space*) *cauchy-filter-complete-converges*:
assumes *cauchy-filter* F *complete* A $F \leq principal\ A$ $F \neq bot$
shows $\exists c. F \leq nhds\ c$
 $\langle proof \rangle$

end

99.2.3 Uniformly continuous functions

definition *uniformly-continuous-on* :: 'a set \Rightarrow ('a::uniform-space \Rightarrow 'b::uniform-space)
 \Rightarrow bool
where *uniformly-continuous-on-uniformity*: *uniformly-continuous-on* $s\ f \longleftrightarrow$
 $(LIM\ (x, y)\ (uniformity-on\ s). (f\ x, f\ y) :> uniformity)$

lemma *uniformly-continuous-onD*:
uniformly-continuous-on $s\ f \implies eventually\ E\ uniformity \implies$
 $eventually\ (\lambda(x, y). x \in s \longrightarrow y \in s \longrightarrow E\ (f\ x, f\ y))\ uniformity$
 $\langle proof \rangle$

lemma *uniformly-continuous-on-const*[*continuous-intros*]: *uniformly-continuous-on*
 $s\ (\lambda x. c)$
 $\langle proof \rangle$

lemma *uniformly-continuous-on-id*[*continuous-intros*]: *uniformly-continuous-on* s
 $(\lambda x. x)$
 $\langle proof \rangle$

lemma *uniformly-continuous-on-compose*:
uniformly-continuous-on $s\ g \implies uniformly-continuous-on\ (g's)\ f \implies$
uniformly-continuous-on $s\ (\lambda x. f\ (g\ x))$

<proof>

lemma *uniformly-continuous-imp-continuous:*

assumes *f: uniformly-continuous-on s f*

shows *continuous-on s f*

<proof>

100 Product Topology

100.1 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) \longleftrightarrow

($\forall x \in S. \exists A B. \text{open } A \wedge \text{open } B \wedge x \in A \times B \wedge A \times B \subseteq S$)

lemma *open-prod-elim:*

assumes *open S and $x \in S$*

obtains *A B where open A and open B and $x \in A \times B$ and $A \times B \subseteq S$*

<proof>

lemma *open-prod-intro:*

assumes $\bigwedge x. x \in S \implies \exists A B. \text{open } A \wedge \text{open } B \wedge x \in A \times B \wedge A \times B \subseteq S$

shows *open S*

<proof>

instance

<proof>

end

declare *[[code abort: open :: ('a::topological-space × 'b::topological-space) set \Rightarrow bool]]*

lemma *open-Times: open S \implies open T \implies open (S × T)*

<proof>

lemma *fst-vimage-eq-Times: fst $-^{\cdot}$ S = S × UNIV*

<proof>

lemma *snd-vimage-eq-Times: snd $-^{\cdot}$ S = UNIV × S*

<proof>

lemma *open-vimage-fst: open S \implies open (fst $-^{\cdot}$ S)*

<proof>

lemma *open-vimage-snd: open S \implies open (snd $-^{\cdot}$ S)*

$\langle \text{proof} \rangle$

lemma *closed-vimage-fst*: $\text{closed } S \implies \text{closed } (\text{fst} - ' S)$
 $\langle \text{proof} \rangle$

lemma *closed-vimage-snd*: $\text{closed } S \implies \text{closed } (\text{snd} - ' S)$
 $\langle \text{proof} \rangle$

lemma *closed-Times*: $\text{closed } S \implies \text{closed } T \implies \text{closed } (S \times T)$
 $\langle \text{proof} \rangle$

lemma *subset-fst-imageI*: $A \times B \subseteq S \implies y \in B \implies A \subseteq \text{fst} - ' S$
 $\langle \text{proof} \rangle$

lemma *subset-snd-imageI*: $A \times B \subseteq S \implies x \in A \implies B \subseteq \text{snd} - ' S$
 $\langle \text{proof} \rangle$

lemma *open-image-fst*:
 assumes *open* S
 shows *open* $(\text{fst} - ' S)$
 $\langle \text{proof} \rangle$

lemma *open-image-snd*:
 assumes *open* S
 shows *open* $(\text{snd} - ' S)$
 $\langle \text{proof} \rangle$

lemma *nhds-prod*: $\text{nhds } (a, b) = \text{nhds } a \times_F \text{nhds } b$
 $\langle \text{proof} \rangle$

100.1.1 Continuity of operations

lemma *tendsto-fst* [*tendsto-intros*]:
 assumes $(f \longrightarrow a) F$
 shows $((\lambda x. \text{fst } (f x)) \longrightarrow \text{fst } a) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-snd* [*tendsto-intros*]:
 assumes $(f \longrightarrow a) F$
 shows $((\lambda x. \text{snd } (f x)) \longrightarrow \text{snd } a) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-Pair* [*tendsto-intros*]:
 assumes $(f \longrightarrow a) F$ and $(g \longrightarrow b) F$
 shows $((\lambda x. (f x, g x)) \longrightarrow (a, b)) F$
 $\langle \text{proof} \rangle$

lemma *continuous-fst* [*continuous-intros*]: $\text{continuous } F f \implies \text{continuous } F (\lambda x. \text{fst } (f x))$

$\langle \text{proof} \rangle$

lemma *continuous-snd*[*continuous-intros*]: *continuous* $F\ f \implies \text{continuous } F\ (\lambda x. \text{snd } (f\ x))$
 $\langle \text{proof} \rangle$

lemma *continuous-Pair*[*continuous-intros*]:
 $\text{continuous } F\ f \implies \text{continuous } F\ g \implies \text{continuous } F\ (\lambda x. (f\ x, g\ x))$
 $\langle \text{proof} \rangle$

lemma *continuous-on-fst*[*continuous-intros*]:
 $\text{continuous-on } s\ f \implies \text{continuous-on } s\ (\lambda x. \text{fst } (f\ x))$
 $\langle \text{proof} \rangle$

lemma *continuous-on-snd*[*continuous-intros*]:
 $\text{continuous-on } s\ f \implies \text{continuous-on } s\ (\lambda x. \text{snd } (f\ x))$
 $\langle \text{proof} \rangle$

lemma *continuous-on-Pair*[*continuous-intros*]:
 $\text{continuous-on } s\ f \implies \text{continuous-on } s\ g \implies \text{continuous-on } s\ (\lambda x. (f\ x, g\ x))$
 $\langle \text{proof} \rangle$

lemma *continuous-on-swap*[*continuous-intros*]: *continuous-on* $A\ \text{prod.swap}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-swap-args*:
assumes *continuous-on* $(A \times B)\ (\lambda(x,y). d\ x\ y)$
shows *continuous-on* $(B \times A)\ (\lambda(x,y). d\ y\ x)$
 $\langle \text{proof} \rangle$

lemma *isCont-fst* [*simp*]: *isCont* $f\ a \implies \text{isCont } (\lambda x. \text{fst } (f\ x))\ a$
 $\langle \text{proof} \rangle$

lemma *isCont-snd* [*simp*]: *isCont* $f\ a \implies \text{isCont } (\lambda x. \text{snd } (f\ x))\ a$
 $\langle \text{proof} \rangle$

lemma *isCont-Pair* [*simp*]: $\llbracket \text{isCont } f\ a; \text{isCont } g\ a \rrbracket \implies \text{isCont } (\lambda x. (f\ x, g\ x))\ a$
 $\langle \text{proof} \rangle$

lemma *continuous-on-compose-Pair*:
assumes $f: \text{continuous-on } (\text{Sigma } A\ B)\ (\lambda(a, b). f\ a\ b)$
assumes $g: \text{continuous-on } C\ g$
assumes $h: \text{continuous-on } C\ h$
assumes *subset*: $\bigwedge c. c \in C \implies g\ c \in A \bigwedge c. c \in C \implies h\ c \in B\ (g\ c)$
shows *continuous-on* $C\ (\lambda c. f\ (g\ c)\ (h\ c))$
 $\langle \text{proof} \rangle$

100.1.2 Connectedness of products**proposition** *connected-Times:*assumes S : *connected* S and T : *connected* T shows *connected* $(S \times T)$ $\langle proof \rangle$ **corollary** *connected-Times-eq [simp]:* $connected (S \times T) \longleftrightarrow S = \{\} \vee T = \{\} \vee connected S \wedge connected T$ (is
?lhs = ?rhs) $\langle proof \rangle$ **100.1.3 Separation axioms****instance** *prod* :: $(t0\text{-space}, t0\text{-space})$ $t0\text{-space}$ $\langle proof \rangle$ **instance** *prod* :: $(t1\text{-space}, t1\text{-space})$ $t1\text{-space}$ $\langle proof \rangle$ **instance** *prod* :: $(t2\text{-space}, t2\text{-space})$ $t2\text{-space}$ $\langle proof \rangle$ **lemma** *isCont-swap[continuous-intros]: isCont prod.swap a* $\langle proof \rangle$ **lemma** *open-diagonal-complement:* $open \{(x,y) \mid x \neq (y::('a::t2\text{-space}))\}$ $\langle proof \rangle$ **lemma** *closed-diagonal:* $closed \{y. \exists x::('a::t2\text{-space}). y = (x,x)\}$ $\langle proof \rangle$ **lemma** *open-superdiagonal:* $open \{(x,y) \mid x > (y::'a::\{linorder\text{-topology}\})\}$ $\langle proof \rangle$ **lemma** *closed-subdiagonal:* $closed \{(x,y) \mid x \leq (y::'a::\{linorder\text{-topology}\})\}$ $\langle proof \rangle$ **lemma** *open-subdiagonal:* $open \{(x,y) \mid x < (y::'a::\{linorder\text{-topology}\})\}$ $\langle proof \rangle$ **lemma** *closed-superdiagonal:* $closed \{(x,y) \mid x \geq (y::('a::\{linorder\text{-topology}\}))\}$ $\langle proof \rangle$

end

theory *Hull*
imports *Main*
begin

100.2 A generic notion of the convex, affine, conic hull, or closed "hull".

definition *hull* :: ('a set \Rightarrow bool) \Rightarrow 'a set \Rightarrow 'a set (**infixl** <i>hull> 75)
where $S \text{ hull } s = \bigcap \{t. S \ t \wedge s \subseteq t\}$

lemma *hull-same*: $S \ s \Longrightarrow S \text{ hull } s = s$
 <proof>

lemma *hull-in*: $(\bigwedge T. \text{Ball } T \ S \Longrightarrow S (\bigcap T)) \Longrightarrow S (S \text{ hull } s)$
 <proof>

lemma *hull-eq*: $(\bigwedge T. \text{Ball } T \ S \Longrightarrow S (\bigcap T)) \Longrightarrow (S \text{ hull } s) = s \longleftrightarrow S \ s$
 <proof>

lemma *hull-hull* [*simp*]: $S \text{ hull } (S \text{ hull } s) = S \text{ hull } s$
 <proof>

lemma *hull-subset*[*intro*]: $s \subseteq (S \text{ hull } s)$
 <proof>

lemma *hull-mono*: $s \subseteq t \Longrightarrow (S \text{ hull } s) \subseteq (S \text{ hull } t)$
 <proof>

lemma *hull-antimono*: $\forall x. S \ x \longrightarrow T \ x \Longrightarrow (T \text{ hull } s) \subseteq (S \text{ hull } s)$
 <proof>

lemma *hull-minimal*: $s \subseteq t \Longrightarrow S \ t \Longrightarrow (S \text{ hull } s) \subseteq t$
 <proof>

lemma *subset-hull*: $S \ t \Longrightarrow S \text{ hull } s \subseteq t \longleftrightarrow s \subseteq t$
 <proof>

lemma *hull-UNIV* [*simp*]: $S \text{ hull } UNIV = UNIV$
 <proof>

lemma *hull-unique*: $s \subseteq t \Longrightarrow S \ t \Longrightarrow (\bigwedge t'. s \subseteq t' \Longrightarrow S \ t' \Longrightarrow t \subseteq t') \Longrightarrow (S \text{ hull } s = t)$
 <proof>

lemma *hull-induct*: $\llbracket a \in Q \text{ hull } S; \bigwedge x. x \in S \Longrightarrow P \ x; Q \ \{x. P \ x\} \rrbracket \Longrightarrow P \ a$
 <proof>

lemma *hull-inc*: $x \in S \implies x \in P \text{ hull } S$

<proof>

lemma *hull-Un-subset*: $(S \text{ hull } s) \cup (S \text{ hull } t) \subseteq (S \text{ hull } (s \cup t))$

<proof>

lemma *hull-Un*:

assumes $T: \bigwedge T. \text{Ball } T \ S \implies S \ (\bigcap T)$

shows $S \text{ hull } (s \cup t) = S \text{ hull } (S \text{ hull } s \cup S \text{ hull } t)$

<proof>

lemma *hull-Un-left*: $P \text{ hull } (S \cup T) = P \text{ hull } (P \text{ hull } S \cup T)$

<proof>

lemma *hull-Un-right*: $P \text{ hull } (S \cup T) = P \text{ hull } (S \cup P \text{ hull } T)$

<proof>

lemma *hull-insert*:

$P \text{ hull } (\text{insert } a \ S) = P \text{ hull } (\text{insert } a \ (P \text{ hull } S))$

<proof>

lemma *hull-redundant-eq*: $a \in (S \text{ hull } s) \longleftrightarrow S \text{ hull } (\text{insert } a \ s) = S \text{ hull } s$

<proof>

lemma *hull-redundant*: $a \in (S \text{ hull } s) \implies S \text{ hull } (\text{insert } a \ s) = S \text{ hull } s$

<proof>

end

101 Modules

Bases of a linear algebra based on modules (i.e. vector spaces of rings).

theory *Modules*

imports *Hull*

begin

101.1 Locale for additive functions

locale *additive* =

fixes $f :: 'a::\text{ab-group-add} \Rightarrow 'b::\text{ab-group-add}$

assumes $\text{add}: f \ (x + y) = f \ x + f \ y$

begin

lemma *zero*: $f \ 0 = 0$

<proof>

lemma *minus*: $f \ (- \ x) = - \ f \ x$

⟨proof⟩

lemma *diff*: $f (x - y) = f x - f y$
 ⟨proof⟩

lemma *sum*: $f (\text{sum } g A) = (\sum x \in A. f (g x))$
 ⟨proof⟩

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 \Rightarrow 'b::ab-group-add \Rightarrow 'b (**infixr** '⟨*s⟩' 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*: $a * s (b * s x) = b * s (a * s x)$
 ⟨proof⟩

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*: $(\text{sum } f A) * s x = (\sum a \in A. (f a) * s x)$
 ⟨proof⟩

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*: $a * s (\text{sum } f A) = (\sum x \in A. a * s (f x))$
 ⟨proof⟩

lemma *sum-constant-scale*: $(\sum x \in A. y) = \text{scale } (\text{of-nat } (\text{card } A)) y$
 ⟨proof⟩

end

⟨ML⟩

context *module*
begin

lemma *[field-simps, field-split-simps]*:

shows *scale-left-distrib-NO-MATCH*: *NO-MATCH* $(x \text{ div } y) \ c \implies (a + b) *s \ x$
 $= a *s \ x + b *s \ x$
and *scale-right-distrib-NO-MATCH*: *NO-MATCH* $(x \text{ div } y) \ a \implies a *s \ (x +$
 $y) = a *s \ x + a *s \ y$
and *scale-left-diff-distrib-NO-MATCH*: *NO-MATCH* $(x \text{ div } y) \ c \implies (a - b)$
 $*s \ x = a *s \ x - b *s \ x$
and *scale-right-diff-distrib-NO-MATCH*: *NO-MATCH* $(x \text{ div } y) \ a \implies a *s \ (x$
 $- y) = a *s \ x - a *s \ y$
 $\langle \text{proof} \rangle$

end

$\langle ML \rangle$

102 Subspace

context *module*
begin

definition *subspace* :: 'b set \implies bool

where *subspace* $S \longleftrightarrow 0 \in S \wedge (\forall x \in S. \forall y \in S. x + y \in S) \wedge (\forall c. \forall x \in S. c *s \ x \in S)$

lemma *subspaceI*:

$0 \in S \implies (\bigwedge y. x \in S \implies y \in S \implies x + y \in S) \implies (\bigwedge c \ x. x \in S \implies c *s \ x \in S) \implies \text{subspace } S$
 $\langle \text{proof} \rangle$

lemma *subspace-UNIV[simp]*: *subspace* UNIV

$\langle \text{proof} \rangle$

lemma *subspace-single-0[simp]*: *subspace* $\{0\}$

$\langle \text{proof} \rangle$

lemma *subspace-0*: *subspace* $S \implies 0 \in S$

$\langle \text{proof} \rangle$

lemma *subspace-add*: *subspace* $S \implies x \in S \implies y \in S \implies x + y \in S$

$\langle \text{proof} \rangle$

lemma *subspace-scale*: *subspace* $S \implies x \in S \implies c *s \ x \in S$

$\langle \text{proof} \rangle$

lemma *subspace-neg*: *subspace* $S \implies x \in S \implies - \ x \in S$

$\langle \text{proof} \rangle$

lemma *subspace-diff*: *subspace* $S \implies x \in S \implies y \in S \implies x - y \in S$

$\langle \text{proof} \rangle$

lemma *subspace-sum*: $\text{subspace } A \implies (\bigwedge x. x \in B \implies f\ x \in A) \implies \text{sum } f\ B \in A$
 $\langle \text{proof} \rangle$

lemma *subspace-Int*: $(\bigwedge i. i \in I \implies \text{subspace } (s\ i)) \implies \text{subspace } (\bigcap_{i \in I}. s\ i)$
 $\langle \text{proof} \rangle$

lemma *subspace-Inter*: $\forall s \in f. \text{subspace } s \implies \text{subspace } (\bigcap f)$
 $\langle \text{proof} \rangle$

lemma *subspace-inter*: $\text{subspace } A \implies \text{subspace } B \implies \text{subspace } (A \cap B)$
 $\langle \text{proof} \rangle$

103 Span: subspace generated by a set

definition *span* :: 'b set \Rightarrow 'b set

where *span-explicit*: $\text{span } b = \{(\sum_{a \in t}. r\ a * s\ a) \mid t\ r. \text{finite } t \wedge t \subseteq b\}$

lemma *span-explicit'*:

$\text{span } b = \{(\sum v \mid f\ v \neq 0. f\ v * s\ v) \mid f. \text{finite } \{v. f\ v \neq 0\} \wedge (\forall v. f\ v \neq 0 \longrightarrow v \in b)\}$
 $\langle \text{proof} \rangle$

lemma *span-alt*:

$\text{span } B = \{(\sum x \mid f\ x \neq 0. f\ x * s\ x) \mid f. \{x. f\ x \neq 0\} \subseteq B \wedge \text{finite } \{x. f\ x \neq 0\}\}$
 $\langle \text{proof} \rangle$

lemma *span-finite*:

assumes *fS*: *finite S*

shows $\text{span } S = \text{range } (\lambda u. \sum_{v \in S}. u\ v * s\ v)$

$\langle \text{proof} \rangle$

lemma *span-induct-alt* [*consumes 1, case-names base step, induct set: span*]:

assumes *x*: $x \in \text{span } S$

assumes *h0*: $h\ 0$ **and** *hS*: $\bigwedge c\ x\ y. x \in S \implies h\ y \implies h\ (c * s\ x + y)$

shows $h\ x$

$\langle \text{proof} \rangle$

lemma *span-mono*: $A \subseteq B \implies \text{span } A \subseteq \text{span } B$

$\langle \text{proof} \rangle$

lemma *span-base*: $a \in S \implies a \in \text{span } S$

$\langle \text{proof} \rangle$

lemma *span-superset*: $S \subseteq \text{span } S$

$\langle \text{proof} \rangle$

lemma *span-zero*: $0 \in \text{span } S$

$\langle \text{proof} \rangle$

lemma *span-UNIV[simp]*: $\text{span UNIV} = \text{UNIV}$
 ⟨proof⟩

lemma *span-add*: $x \in \text{span } S \implies y \in \text{span } S \implies x + y \in \text{span } S$
 ⟨proof⟩

lemma *span-scale*: $x \in \text{span } S \implies c * x \in \text{span } S$
 ⟨proof⟩

lemma *subspace-span [iff]*: $\text{subspace } (\text{span } S)$
 ⟨proof⟩

lemma *span-neg*: $x \in \text{span } S \implies -x \in \text{span } S$
 ⟨proof⟩

lemma *span-diff*: $x \in \text{span } S \implies y \in \text{span } S \implies x - y \in \text{span } S$
 ⟨proof⟩

lemma *span-sum*: $(\bigwedge x. x \in A \implies f x \in \text{span } S) \implies \text{sum } f A \in \text{span } S$
 ⟨proof⟩

lemma *span-minimal*: $S \subseteq T \implies \text{subspace } T \implies \text{span } S \subseteq T$
 ⟨proof⟩

lemma *span-def*: $\text{span } S = \text{subspace hull } S$
 ⟨proof⟩

lemma *span-unique*:
 $S \subseteq T \implies \text{subspace } T \implies (\bigwedge T'. S \subseteq T' \implies \text{subspace } T' \implies T \subseteq T') \implies \text{span } S = T$
 ⟨proof⟩

lemma *span-subspace-induct[consumes 2]*:
 assumes $x: x \in \text{span } S$
 and $P: \text{subspace } P$
 and $SP: \bigwedge x. x \in S \implies x \in P$
 shows $x \in P$
 ⟨proof⟩

lemma (*in module*) *span-induct[consumes 1, case-names base step, induct set: span]*:
 assumes $x: x \in \text{span } S$
 and $P: \text{subspace } (\text{Collect } P)$
 and $SP: \bigwedge x. x \in S \implies P x$
 shows $P x$
 ⟨proof⟩

lemma *span-empty[simp]*: $\text{span } \{\} = \{0\}$

<proof>

lemma *span-subspace*: $A \subseteq B \implies B \subseteq \text{span } A \implies \text{subspace } B \implies \text{span } A = B$
<proof>

lemma *span-span*: $\text{span } (\text{span } A) = \text{span } A$
<proof>

lemma *span-add-eq*: **assumes** $x: x \in \text{span } S$ **shows** $x + y \in \text{span } S \longleftrightarrow y \in \text{span } S$
<proof>

lemma *span-add-eq2*: **assumes** $y: y \in \text{span } S$ **shows** $x + y \in \text{span } S \longleftrightarrow x \in \text{span } S$
<proof>

lemma *span-singleton*: $\text{span } \{x\} = \text{range } (\lambda k. k *s x)$
<proof>

lemma *span-Un*: $\text{span } (S \cup T) = \{x + y \mid x \in \text{span } S \wedge y \in \text{span } T\}$
<proof>

lemma *span-insert*: $\text{span } (\text{insert } a \ S) = \{x. \exists k. (x - k *s a) \in \text{span } S\}$
<proof>

lemma *span-breakdown*:
assumes $bS: b \in S$
and $aS: a \in \text{span } S$
shows $\exists k. a - k *s b \in \text{span } (S - \{b\})$
<proof>

lemma *span-breakdown-eq*: $x \in \text{span } (\text{insert } a \ S) \longleftrightarrow (\exists k. x - k *s a \in \text{span } S)$
<proof>

lemmas *span-clauses* = *span-base span-zero span-add span-scale*

lemma *span-eq-iff[simp]*: $\text{span } s = s \longleftrightarrow \text{subspace } s$
<proof>

lemma *span-eq*: $\text{span } S = \text{span } T \longleftrightarrow S \subseteq \text{span } T \wedge T \subseteq \text{span } S$
<proof>

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>

104 Dependent and independent sets

definition *dependent* :: 'b set \Rightarrow bool

where *dependent-explicit*: $\text{dependent } s \longleftrightarrow (\exists t \ u. \text{finite } t \wedge t \subseteq s \wedge (\sum_{v \in t. u \ v * s \ v} = 0 \wedge (\exists v \in t. u \ v \neq 0)))$

abbreviation *independent* $s \equiv \neg \text{dependent } s$

lemma *dependent-mono*: $\text{dependent } B \Longrightarrow B \subseteq A \Longrightarrow \text{dependent } A$
 $\langle \text{proof} \rangle$

lemma *independent-mono*: $\text{independent } A \Longrightarrow B \subseteq A \Longrightarrow \text{independent } B$
 $\langle \text{proof} \rangle$

lemma *dependent-zero*: $0 \in A \Longrightarrow \text{dependent } A$
 $\langle \text{proof} \rangle$

lemma *independent-empty[intro]*: $\text{independent } \{\}$
 $\langle \text{proof} \rangle$

lemma *independent-explicit-module*:

$\text{independent } s \longleftrightarrow (\forall t \ u \ v. \text{finite } t \longrightarrow t \subseteq s \longrightarrow (\sum_{v \in t. u \ v * s \ v} = 0 \longrightarrow v \in t \longrightarrow u \ v = 0))$
 $\langle \text{proof} \rangle$

lemma *independentD*: $\text{independent } s \Longrightarrow \text{finite } t \Longrightarrow t \subseteq s \Longrightarrow (\sum_{v \in t. u \ v * s \ v} = 0 \Longrightarrow v \in t \Longrightarrow u \ v = 0)$
 $\langle \text{proof} \rangle$

lemma *independent-Union-directed*:

assumes *directed*: $\bigwedge c \ d. c \in C \Longrightarrow d \in C \Longrightarrow c \subseteq d \vee d \subseteq c$

assumes *indep*: $\bigwedge c. c \in C \Longrightarrow \text{independent } c$

shows $\text{independent } (\bigcup C)$

$\langle \text{proof} \rangle$

lemma *dependent-finite*:

assumes *finite* S

shows $\text{dependent } S \longleftrightarrow (\exists u. (\exists v \in S. u \ v \neq 0) \wedge (\sum_{v \in S. u \ v * s \ v} = 0))$

(**is** ?lhs = ?rhs)

$\langle \text{proof} \rangle$

lemma *dependent-alt*:

$\text{dependent } B \longleftrightarrow$

$(\exists X. \text{finite } \{x. X \ x \neq 0\} \wedge \{x. X \ x \neq 0\} \subseteq B \wedge (\sum x | X \ x \neq 0. X \ x * s \ x) = 0 \wedge (\exists x. X \ x \neq 0))$

$\langle \text{proof} \rangle$

lemma *independent-alt*:

$\text{independent } B \longleftrightarrow$

$(\forall X. \text{finite } \{x. X x \neq 0\} \longrightarrow \{x. X x \neq 0\} \subseteq B \longrightarrow (\sum x | X x \neq 0. X x * s x) = 0 \longrightarrow (\forall x. X x = 0))$
 $\langle \text{proof} \rangle$

lemma *independentD-alt:*

$\text{independent } B \Longrightarrow \text{finite } \{x. X x \neq 0\} \Longrightarrow \{x. X x \neq 0\} \subseteq B \Longrightarrow (\sum x | X x \neq 0. X x * s x) = 0 \Longrightarrow X x = 0$
 $\langle \text{proof} \rangle$

lemma *independentD-unique:*

assumes B : *independent* B
and X : *finite* $\{x. X x \neq 0\}$ $\{x. X x \neq 0\} \subseteq B$
and Y : *finite* $\{x. Y x \neq 0\}$ $\{x. Y x \neq 0\} \subseteq B$
and $(\sum x | X x \neq 0. X x * s x) = (\sum x | Y x \neq 0. Y x * s x)$
shows $X = Y$
 $\langle \text{proof} \rangle$

105 Representation of a vector on a specific basis

definition *representation* :: $'b \text{ set} \Rightarrow 'b \Rightarrow 'b \Rightarrow 'a$

where *representation basis* $v =$
 $(\text{if } \text{independent basis} \wedge v \in \text{span basis} \text{ then}$
 $\text{SOME } f. (\forall v. f v \neq 0 \longrightarrow v \in \text{basis}) \wedge \text{finite } \{v. f v \neq 0\} \wedge (\sum v \in \{v. f v \neq 0\}. f v * s v) = v$
 $\text{else } (\lambda b. 0))$

lemma *unique-representation:*

assumes *basis: independent basis*
and *in-basis:* $\bigwedge v. f v \neq 0 \Longrightarrow v \in \text{basis} \bigwedge v. g v \neq 0 \Longrightarrow v \in \text{basis}$
and *[simp]:* $\text{finite } \{v. f v \neq 0\} \text{ finite } \{v. g v \neq 0\}$
and *eq:* $(\sum v \in \{v. f v \neq 0\}. f v * s v) = (\sum v \in \{v. g v \neq 0\}. g v * s v)$
shows $f = g$
 $\langle \text{proof} \rangle$

lemma

shows *representation-ne-zero:* $\bigwedge b. \text{representation basis } v b \neq 0 \Longrightarrow b \in \text{basis}$
and *finite-representation:* $\text{finite } \{b. \text{representation basis } v b \neq 0\}$
and *sum-nonzero-representation-eq:*
 $\text{independent basis} \Longrightarrow v \in \text{span basis} \Longrightarrow (\sum b | \text{representation basis } v b \neq 0. \text{representation basis } v b * s b) = v$
 $\langle \text{proof} \rangle$

lemma *sum-representation-eq:*

$(\sum b \in B. \text{representation basis } v b * s b) = v$
if *independent basis* $v \in \text{span basis}$ *finite* B *basis* $\subseteq B$
 $\langle \text{proof} \rangle$

lemma *representation-eqI:*

assumes *basis: independent basis* **and** $b: v \in \text{span basis}$

and *ne-zero*: $\bigwedge b. f\ b \neq 0 \implies b \in \text{basis}$
and *finite*: $\text{finite } \{b. f\ b \neq 0\}$
and *eq*: $(\sum b \mid f\ b \neq 0. f\ b *s\ b) = v$
shows *representation basis* $v = f$
 ⟨*proof*⟩

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*⟩

lemma *representation-zero*: *representation basis* $0 = (\lambda b. 0)$
 ⟨*proof*⟩

lemma *representation-diff*:
assumes *basis*: *independent basis* **and** $v: v \in \text{span basis}$ **and** $u: u \in \text{span basis}$
shows *representation basis* $(u - v) = (\lambda b. \text{representation basis } u\ b - \text{representation basis } v\ b)$
 ⟨*proof*⟩

lemma *representation-neg*:
independent basis $\implies v \in \text{span basis} \implies \text{representation basis } (-\ v) = (\lambda b. - \text{representation basis } v\ b)$
 ⟨*proof*⟩

lemma *representation-add*:
independent basis $\implies v \in \text{span basis} \implies u \in \text{span basis} \implies$
representation basis $(u + v) = (\lambda b. \text{representation basis } u\ b + \text{representation basis } v\ b)$
 ⟨*proof*⟩

lemma *representation-sum*:
independent basis $\implies (\bigwedge i. i \in I \implies v\ i \in \text{span basis}) \implies$
representation basis $(\text{sum } v\ I) = (\lambda b. \sum i \in I. \text{representation basis } (v\ i)\ b)$
 ⟨*proof*⟩

lemma *representation-scale*:
assumes *basis*: *independent basis* **and** $v: v \in \text{span basis}$
shows *representation basis* $(r *s\ v) = (\lambda b. r * \text{representation basis } v\ b)$
 ⟨*proof*⟩

lemma *representation-extend*:
assumes *basis*: *independent basis* **and** $v: v \in \text{span basis'}$ **and** *basis'*: $\text{basis'} \subseteq \text{basis}$
shows *representation basis* $v = \text{representation basis' } v$
 ⟨*proof*⟩

The set B is the maximal independent set for *span* B , or A is the minimal spanning set

lemma *spanning-subset-independent*:

assumes $BA: B \subseteq A$
and $iA: \text{independent } A$
and $AsB: A \subseteq \text{span } B$
shows $A = B$

$\langle \text{proof} \rangle$

end

A linear function is a mapping between two modules over the same ring.

locale *module-hom* = $m1: \text{module } s1 + m2: \text{module } s2$

for $s1 :: 'a::\text{comm-ring-1} \Rightarrow 'b::\text{ab-group-add} \Rightarrow 'b$ (**infixr** $\langle *a \rangle$ 75)
and $s2 :: 'a::\text{comm-ring-1} \Rightarrow 'c::\text{ab-group-add} \Rightarrow 'c$ (**infixr** $\langle *b \rangle$ 75) +
fixes $f :: 'b \Rightarrow 'c$
assumes $\text{add}: f (b1 + b2) = f b1 + f b2$
and $\text{scale}: f (r *a b) = r *b f b$

begin

lemma *zero[simp]*: $f 0 = 0$

$\langle \text{proof} \rangle$

lemma *neg*: $f (- x) = - f x$

$\langle \text{proof} \rangle$

lemma *diff*: $f (x - y) = f x - f y$

$\langle \text{proof} \rangle$

lemma *sum*: $f (\text{sum } g S) = (\sum a \in S. f (g a))$

$\langle \text{proof} \rangle$

lemma *inj-on-iff-eq-0*:

assumes $s: m1.\text{subspace } s$
shows $\text{inj-on } f s \longleftrightarrow (\forall x \in s. f x = 0 \longrightarrow x = 0)$

$\langle \text{proof} \rangle$

lemma *inj-iff-eq-0*: $\text{inj } f = (\forall x. f x = 0 \longrightarrow x = 0)$

$\langle \text{proof} \rangle$

lemma *subspace-image*: **assumes** $S: m1.\text{subspace } S$ **shows** $m2.\text{subspace } (f \text{ ` } S)$

$\langle \text{proof} \rangle$

lemma *subspace-vimage*: $m2.\text{subspace } S \Longrightarrow m1.\text{subspace } (f \text{ - ` } S)$

$\langle \text{proof} \rangle$

lemma *subspace-kernel*: $m1.\text{subspace } \{x. f x = 0\}$

$\langle \text{proof} \rangle$

lemma *span-image*: $m2.\text{span } (f \text{ ` } S) = f \text{ ` } (m1.\text{span } S)$

$\langle \text{proof} \rangle$

lemma *dependent-inj-imageD*:

assumes $d: m2.dependent (f \text{ ‘ } s)$ **and** $i: inj-on f (m1.span s)$
shows $m1.dependent s$

$\langle proof \rangle$

lemma *eq-0-on-span*:

assumes $f0: \bigwedge x. x \in b \implies f x = 0$ **and** $x: x \in m1.span b$ **shows** $f x = 0$

$\langle proof \rangle$

lemma *independent-injective-image*: $m1.independent s \implies inj-on f (m1.span s) \implies m2.independent (f \text{ ‘ } s)$

$\langle proof \rangle$

lemma *inj-on-span-independent-image*:

assumes $ifB: m2.independent (f \text{ ‘ } B)$ **and** $f: inj-on f B$ **shows** $inj-on f (m1.span B)$

$\langle proof \rangle$

lemma *inj-on-span-iff-independent-image*: $m2.independent (f \text{ ‘ } B) \implies inj-on f (m1.span B) \longleftrightarrow inj-on f B$

$\langle proof \rangle$

lemma *subspace-linear-preimage*: $m2.subspace S \implies m1.subspace \{x. f x \in S\}$

$\langle proof \rangle$

lemma *spans-image*: $V \subseteq m1.span B \implies f \text{ ‘ } V \subseteq m2.span (f \text{ ‘ } B)$

$\langle proof \rangle$

Relation between bases and injectivity/surjectivity of map.

lemma *spanning-surjective-image*:

assumes $us: UNIV \subseteq m1.span S$

and $sf: surj f$

shows $UNIV \subseteq m2.span (f \text{ ‘ } S)$

$\langle proof \rangle$

lemmas *independent-inj-on-image = independent-injective-image*

lemma *independent-inj-image*:

$m1.independent S \implies inj f \implies m2.independent (f \text{ ‘ } S)$

$\langle proof \rangle$

end

lemma *module-hom-iff*:

$module-hom s1 s2 f \longleftrightarrow$

$module s1 \wedge module s2 \wedge$

$(\forall x y. f (x + y) = f x + f y) \wedge (\forall c x. f (s1 c x) = s2 c (f x))$

$\langle proof \rangle$

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* ($\lambda x. 0$)
<proof>

lemma *module-hom-add*: *module-hom s1 s2 f* \Rightarrow *module-hom s1 s2 g* \Rightarrow *module-hom s1 s2* ($\lambda x. f\ x + g\ x$)
<proof>

lemma *module-hom-sub*: *module-hom s1 s2 f* \Rightarrow *module-hom s1 s2 g* \Rightarrow *module-hom s1 s2* ($\lambda x. f\ x - g\ x$)
<proof>

lemma *module-hom-neg*: *module-hom s1 s2 f* \Rightarrow *module-hom s1 s2* ($\lambda x. - f\ x$)
<proof>

lemma *module-hom-scale*: *module-hom s1 s2 f* \Rightarrow *module-hom s1 s2* ($\lambda x. s2\ c\ (f\ x)$)
<proof>

lemma *module-hom-compose-scale*:
module-hom s1 s2 ($\lambda x. s2\ (f\ x)\ (c)$)
if *module-hom s1* $(*)\ f$
<proof>

lemma *bij-module-hom-imp-inv-module-hom*: *module-hom scale1 scale2 f* \Rightarrow *bij f*
 \Rightarrow
module-hom scale2 scale1 (*inv f*)
<proof>

lemma *module-hom-sum*: ($\bigwedge i. i \in I \Rightarrow$ *module-hom s1 s2* (*f i*)) \Rightarrow (*I* = {} \Rightarrow *module s1* \wedge *module s2*) \Rightarrow *module-hom s1 s2* ($\lambda x. \sum_{i \in I}. f\ i\ x$)
<proof>

lemma *module-hom-eq-on-span*: *f x* = *g x*
if *module-hom s1 s2 f* *module-hom s1 s2 g*
and ($\bigwedge x. x \in B \Rightarrow f\ x = g\ x$) *x* \in *m1.span B*
<proof>

end

context *module* **begin**

lemma *module-hom-scale-self*[*simp*]:
module-hom scale scale ($\lambda x. scale\ c\ x$)

<proof>

lemma *module-hom-scale-left[simp]:*
module-hom (*) *scale* ($\lambda r.$ *scale* *r* *x*)
<proof>

lemma *module-hom-id: module-hom scale scale id*
<proof>

lemma *module-hom-ident: module-hom scale scale* ($\lambda x.$ *x*)
<proof>

lemma *module-hom-uminus: module-hom scale scale uminus*
<proof>

end

lemma *module-hom-compose: module-hom s1 s2 f \implies module-hom s2 s3 g \implies*
module-hom s1 s3 (g o f)
<proof>

end

106 Vector Spaces

theory *Vector-Spaces*
imports *Modules*
begin

lemma *isomorphism-expand:*
 $f \circ g = id \wedge g \circ f = id \iff (\forall x. f (g x) = x) \wedge (\forall x. g (f x) = x)$
<proof>

lemma *left-right-inverse-eq:*
assumes *fg: f o g = id*
and *gh: g o h = id*
shows *f = h*
<proof>

lemma *ordLeq3-finite-infinite:*
assumes *A: finite A and B: infinite B* **shows** *ordLeq3 (card-of A) (card-of B)*
<proof>

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.
 $a * s (x + y) = a * s x + a * s y$
 $(a + b) * s x = a * s x + b * s x$

$$a *s (b *s x) = (a * b) *s x$$

$$1 *s x = x$$

lemma *module-iff-vector-space*: $module\ s \longleftrightarrow vector\ space\ s$
 ⟨proof⟩

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*: $module\ hom\ s1\ s2\ f \longleftrightarrow linear\ s1\ s2\ f$
 ⟨proof⟩

lemmas *module-hom-eq-linear* = *module-hom-iff-linear*[*abs-def*, *THEN meta-eq-to-obj-eq*]

lemmas *linear-iff-module-hom* = *module-hom-iff-linear*[*symmetric*]

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*
 ⟨proof⟩

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-representation*:
assumes *independent B span B = UNIV*
shows $linear\ scale\ (*)\ (\lambda v. representation\ B\ v\ b)$
 ⟨proof⟩

lemma *linear-imp-scale*:
fixes *D*::'*a* \Rightarrow '*b*
assumes *linear* (*) *scale D*
obtains *d* **where** $D = (\lambda x. scale\ x\ d)$
 ⟨proof⟩

lemma *scale-eq-0-iff* [*simp*]: $scale\ a\ x = 0 \longleftrightarrow a = 0 \vee x = 0$
 ⟨proof⟩

lemma *scale-left-imp-eq*:
assumes *nonzero*: $a \neq 0$
and *scale*: $scale\ a\ x = scale\ a\ y$
shows $x = y$
 ⟨proof⟩

lemma *scale-right-imp-eq*:
assumes *nonzero*: $x \neq 0$
and *scale*: $\text{scale } a \ x = \text{scale } b \ x$
shows $a = b$
 $\langle \text{proof} \rangle$

lemma *scale-cancel-left [simp]*: $\text{scale } a \ x = \text{scale } a \ y \longleftrightarrow x = y \vee a = 0$
 $\langle \text{proof} \rangle$

lemma *scale-cancel-right [simp]*: $\text{scale } a \ x = \text{scale } b \ x \longleftrightarrow a = b \vee x = 0$
 $\langle \text{proof} \rangle$

lemma *injective-scale*: $c \neq 0 \implies \text{inj } (\lambda x. \text{scale } c \ x)$
 $\langle \text{proof} \rangle$

lemma *dependent-def*: $\text{dependent } P \longleftrightarrow (\exists a \in P. a \in \text{span } (P - \{a\}))$
 $\langle \text{proof} \rangle$

lemma *dependent-single[simp]*: $\text{dependent } \{x\} \longleftrightarrow x = 0$
 $\langle \text{proof} \rangle$

lemma *in-span-insert*:
assumes *a*: $a \in \text{span } (\text{insert } b \ S)$
and *na*: $a \notin \text{span } S$
shows $b \in \text{span } (\text{insert } a \ S)$
 $\langle \text{proof} \rangle$

lemma *dependent-insertD*: **assumes** *a*: $a \notin \text{span } S$ **and** *S*: $\text{dependent } (\text{insert } a \ S)$
shows $\text{dependent } S$
 $\langle \text{proof} \rangle$

lemma *independent-insertI*: $a \notin \text{span } S \implies \text{independent } S \implies \text{independent } (\text{insert } a \ S)$
 $\langle \text{proof} \rangle$

lemma *independent-insert*:
 $\text{independent } (\text{insert } a \ S) \longleftrightarrow (\text{if } a \in S \text{ then } \text{independent } S \text{ else } \text{independent } S \wedge a \notin \text{span } S)$
 $\langle \text{proof} \rangle$

lemma *maximal-independent-subset-extend*:
assumes $S \subseteq V$ *independent* S
obtains B **where** $S \subseteq B$ $B \subseteq V$ *independent* B $V \subseteq \text{span } B$
 $\langle \text{proof} \rangle$

lemma *maximal-independent-subset*:
obtains B **where** $B \subseteq V$ *independent* B $V \subseteq \text{span } B$
 $\langle \text{proof} \rangle$

Extends a basis from B to a basis of the entire space.

definition *extend-basis* :: 'b set \Rightarrow 'b set

where *extend-basis* $B = (\text{SOME } B'. B \subseteq B' \wedge \text{independent } B' \wedge \text{span } B' = \text{UNIV})$

lemma

assumes B : *independent* B

shows *extend-basis-superset*: $B \subseteq \text{extend-basis } B$

and *independent-extend-basis*: *independent* (*extend-basis* B)

and *span-extend-basis[simp]*: $\text{span } (\text{extend-basis } B) = \text{UNIV}$

<proof>

lemma *in-span-delete*:

assumes a : $a \in \text{span } S$ **and** na : $a \notin \text{span } (S - \{b\})$

shows $b \in \text{span } (\text{insert } a (S - \{b\}))$

<proof>

lemma *span-redundant*: $x \in \text{span } S \Longrightarrow \text{span } (\text{insert } x S) = \text{span } S$

<proof>

lemma *span-trans*: $x \in \text{span } S \Longrightarrow y \in \text{span } (\text{insert } x S) \Longrightarrow y \in \text{span } S$

<proof>

lemma *span-insert-0[simp]*: $\text{span } (\text{insert } 0 S) = \text{span } S$

<proof>

lemma *span-delete-0 [simp]*: $\text{span}(S - \{0\}) = \text{span } S$

<proof>

lemma *span-image-scale*:

assumes *finite* S **and** nz : $\bigwedge x. x \in S \Longrightarrow c \ x \neq 0$

shows $\text{span } ((\lambda x. c \ x * s \ x) ` S) = \text{span } S$

<proof>

lemma *exchange-lemma*:

assumes f : *finite* T

and i : *independent* S

and sp : $S \subseteq \text{span } T$

shows $\exists t'. \text{card } t' = \text{card } T \wedge \text{finite } t' \wedge S \subseteq t' \wedge t' \subseteq S \cup T \wedge S \subseteq \text{span } t'$

<proof>

lemma *independent-span-bound*:

assumes f : *finite* T

and i : *independent* S

and sp : $S \subseteq \text{span } T$

shows $\text{finite } S \wedge \text{card } S \leq \text{card } T$

<proof>

lemma *independent-explicit-finite-subsets*:

independent $A \longleftrightarrow (\forall S \subseteq A. \text{finite } S \longrightarrow (\forall u. (\sum_{v \in S} u \cdot v = 0 \longrightarrow (\forall v \in S. u \cdot v = 0)))$
 ⟨proof⟩

lemma *independent-if-scalars-zero*:

assumes *fin-A*: *finite* A

and *sum*: $\bigwedge x. (\sum_{x \in A} f \cdot x = 0 \implies x \in A \implies f \cdot x = 0$

shows *independent* A

⟨proof⟩

lemma *bij-if-span-eq-span-bases*:

assumes B : *independent* B **and** C : *independent* C

and *eq*: $\text{span } B = \text{span } C$

shows $\exists f. \text{bij-betw } f \ B \ C$

⟨proof⟩

definition *dim* :: $'b \text{ set} \Rightarrow \text{nat}$

where *dim* $V = (\text{if } \exists b. \text{independent } b \wedge \text{span } b = \text{span } V \text{ then}$
 $\text{card } (\text{SOME } b. \text{independent } b \wedge \text{span } b = \text{span } V) \text{ else } 0)$

lemma *dim-eq-card*:

assumes BV : $\text{span } B = \text{span } V$ **and** B : *independent* B

shows $\text{dim } V = \text{card } B$

⟨proof⟩

lemma *basis-card-eq-dim*: $B \subseteq V \implies V \subseteq \text{span } B \implies \text{independent } B \implies \text{card } B = \text{dim } V$

⟨proof⟩

lemma *basis-exists*:

obtains B **where** $B \subseteq V$ *independent* B $V \subseteq \text{span } B$ $\text{card } B = \text{dim } V$

⟨proof⟩

lemma *dim-eq-card-independent*: $\text{independent } B \implies \text{dim } B = \text{card } B$

⟨proof⟩

lemma *dim-span[simp]*: $\text{dim } (\text{span } S) = \text{dim } S$

⟨proof⟩

lemma *dim-span-eq-card-independent*: $\text{independent } B \implies \text{dim } (\text{span } B) = \text{card } B$

⟨proof⟩

lemma *dim-le-card*: **assumes** $V \subseteq \text{span } W$ *finite* W **shows** $\text{dim } V \leq \text{card } W$

⟨proof⟩

lemma *span-eq-dim*: $\text{span } S = \text{span } T \implies \text{dim } S = \text{dim } T$

⟨proof⟩

corollary *dim-le-card'*:

$finite\ s \implies dim\ s \leq card\ s$
 $\langle proof \rangle$

lemma *span-card-ge-dim*:

$B \subseteq V \implies V \subseteq span\ B \implies finite\ B \implies dim\ V \leq card\ B$
 $\langle proof \rangle$

lemma *dim-unique*:

$B \subseteq V \implies V \subseteq span\ B \implies independent\ B \implies card\ B = n \implies dim\ V = n$
 $\langle proof \rangle$

lemma *subspace-sums*: $\llbracket subspace\ S; subspace\ T \rrbracket \implies subspace\ \{x + y \mid x \in S \wedge y \in T\}$
 $\langle proof \rangle$

end

lemma *linear-iff*: $linear\ s1\ s2\ f \longleftrightarrow$

$(vector-space\ s1 \wedge vector-space\ s2 \wedge (\forall x\ y. f\ (x + y) = f\ x + f\ y) \wedge (\forall c\ x. f\ (s1\ c\ x) = s2\ c\ (f\ x)))$
 $\langle proof \rangle$

context begin

qualified lemma *linear-compose*: $linear\ s1\ s2\ f \implies linear\ s2\ s3\ g \implies linear\ s1\ s3\ (g \circ f)$
 $\langle proof \rangle$

end

locale *vector-space-pair* = $vs1: vector-space\ s1 + vs2: vector-space\ s2$

for $s1 :: 'a::field \Rightarrow 'b::ab-group-add \Rightarrow 'b$ (**infixr** $\langle *a \rangle$ 75)
and $s2 :: 'a::field \Rightarrow 'c::ab-group-add \Rightarrow 'c$ (**infixr** $\langle *b \rangle$ 75)

begin

context fixes f **assumes** $linear\ s1\ s2\ f$ **begin**

interpretation $linear\ s1\ s2\ f$ $\langle proof \rangle$

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 $linear-subspace-vimage = subspace-vimage$

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*
 ⟨*proof*⟩

lemmas— from locale *module-pair*
linear-eq-on-span = *module-hom-eq-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 *fx*: *f x = 0*
shows *x = 0*
 ⟨*proof*⟩

lemma *linear-eq-on*:
assumes *l*: *linear s1 s2 f linear s1 s2 g*
assumes *x*: *x ∈ vs1.span B* **and** *eq*: $\bigwedge b. b \in B \implies f\ b = g\ b$
shows *f x = g x*
 ⟨*proof*⟩

definition *construct* :: *'b set* \Rightarrow (*'b* \Rightarrow *'c*) \Rightarrow (*'b* \Rightarrow *'c*)
where *construct B g v* = $(\sum b \mid vs1.representation\ (vs1.extend-basis\ B)\ v\ b \neq 0.$
 $vs1.representation\ (vs1.extend-basis\ B)\ v\ b * b\ (if\ b \in B\ then\ g\ b\ else\ 0))$

lemma *construct-cong*: $(\bigwedge b. b \in B \implies f\ b = g\ b) \implies construct\ B\ f = construct\ B\ g$
 ⟨*proof*⟩

lemma *linear-construct*:
assumes *B[simp]*: *vs1.independent B*

shows *linear* $s1\ s2$ (*construct* $B\ f$)
 ⟨*proof*⟩

lemma *construct-basis*:
assumes $B[simp]$: $vs1.independent\ B$ **and** b : $b \in B$
shows *construct* $B\ f\ b = f\ b$
 ⟨*proof*⟩

lemma *construct-outside*:
assumes B : $vs1.independent\ B$ **and** v : $v \in vs1.span\ (vs1.extend-basis\ B - B)$
shows *construct* $B\ f\ v = 0$
 ⟨*proof*⟩

lemma *construct-add*:
assumes $B[simp]$: $vs1.independent\ B$
shows *construct* $B\ (\lambda x. f\ x + g\ x)\ v = \text{construct } B\ f\ v + \text{construct } B\ g\ v$
 ⟨*proof*⟩

lemma *construct-scale*:
assumes $B[simp]$: $vs1.independent\ B$
shows *construct* $B\ (\lambda x. c * b\ f\ x)\ v = c * b\ \text{construct } B\ f\ v$
 ⟨*proof*⟩

lemma *construct-in-span*:
assumes $B[simp]$: $vs1.independent\ B$
shows *construct* $B\ f\ v \in vs2.span\ (f\ ' B)$
 ⟨*proof*⟩

lemma *linear-compose-sum*:
assumes lS : $\forall a \in S. \text{linear } s1\ s2\ (f\ a)$
shows *linear* $s1\ s2\ (\lambda x. \text{sum } (\lambda a. f\ a\ x)\ S)$
 ⟨*proof*⟩

lemma *in-span-in-range-construct*:
 $x \in \text{range } (\text{construct } B\ f)$ **if** i : $vs1.independent\ B$ **and** x : $x \in vs2.span\ (f\ ' B)$
 ⟨*proof*⟩

lemma *range-construct-eq-span*:
 $\text{range } (\text{construct } B\ f) = vs2.span\ (f\ ' B)$
if $vs1.independent\ B$
 ⟨*proof*⟩

lemma *linear-independent-extend-subspace*:
 — legacy: use *construct* instead
assumes $vs1.independent\ B$
shows $\exists g. \text{linear } s1\ s2\ g \wedge (\forall x \in B. g\ x = f\ x) \wedge \text{range } g = vs2.span\ (f\ ' B)$
 ⟨*proof*⟩

lemma *linear-independent-extend*:

$vs1.independent\ B \implies \exists g. linear\ s1\ s2\ g \wedge (\forall x \in B. g\ x = f\ x)$
 $\langle proof \rangle$

lemma *linear-exists-left-inverse-on:*

assumes $lf: linear\ s1\ s2\ f$

assumes $V: vs1.subspace\ V$ **and** $f: inj-on\ f\ V$

shows $\exists g. g \text{ ‘ } UNIV \subseteq V \wedge linear\ s2\ s1\ g \wedge (\forall v \in V. g\ (f\ v) = v)$

$\langle proof \rangle$

lemma *linear-exists-right-inverse-on:*

assumes $lf: linear\ s1\ s2\ f$

assumes $vs1.subspace\ V$

shows $\exists g. g \text{ ‘ } UNIV \subseteq V \wedge linear\ s2\ s1\ g \wedge (\forall v \in f \text{ ‘ } V. f\ (g\ v) = v)$

$\langle proof \rangle$

lemma *linear-inj-on-left-inverse:*

assumes $lf: linear\ s1\ s2\ f$

assumes $fi: inj-on\ f\ (vs1.span\ S)$

shows $\exists g. range\ g \subseteq vs1.span\ S \wedge linear\ s2\ s1\ g \wedge (\forall x \in vs1.span\ S. g\ (f\ x) = x)$

$\langle proof \rangle$

lemma *linear-injective-left-inverse:* $linear\ s1\ s2\ f \implies inj\ f \implies \exists g. linear\ s2\ s1\ g \wedge g \circ f = id$

$\langle proof \rangle$

lemma *linear-surj-right-inverse:*

assumes $lf: linear\ s1\ s2\ f$

assumes $sf: vs2.span\ T \subseteq f \text{ ‘ } vs1.span\ S$

shows $\exists g. range\ g \subseteq vs1.span\ S \wedge linear\ s2\ s1\ g \wedge (\forall x \in vs2.span\ T. f\ (g\ x) = x)$

$\langle proof \rangle$

lemma *linear-surjective-right-inverse:* $linear\ s1\ s2\ f \implies surj\ f \implies \exists g. linear\ s2\ s1\ g \wedge f \circ g = id$

$\langle proof \rangle$

lemma *finite-basis-to-basis-subspace-isomorphism:*

assumes $s: vs1.subspace\ S$

and $t: vs2.subspace\ T$

and $d: vs1.dim\ S = vs2.dim\ T$

and $fB: finite\ B$

and $B: B \subseteq S\ vs1.independent\ B\ S \subseteq vs1.span\ B\ card\ B = vs1.dim\ S$

and $fC: finite\ C$

and $C: C \subseteq T\ vs2.independent\ C\ T \subseteq vs2.span\ C\ card\ C = vs2.dim\ T$

shows $\exists f. linear\ s1\ s2\ f \wedge f \text{ ‘ } B = C \wedge f \text{ ‘ } S = T \wedge inj-on\ f\ S$

$\langle proof \rangle$

end

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 *finiteI-independent*: *independent B \implies finite B*
<proof>

lemma *dim-empty* [*simp*]: *dim {} = 0*
<proof>

lemma *dim-insert*:
dim (insert x S) = (if x \in span S then dim S else dim S + 1)
<proof>

lemma *dim-singleton* [*simp*]: *dim {x} = (if x = 0 then 0 else 1)*
<proof>

proposition *choose-subspace-of-subspace*:
assumes *n \leq dim S*
obtains *T where subspace T T \subseteq span S dim T = n*
<proof>

lemma *basis-subspace-exists*:
assumes *subspace S*
obtains *B where finite B B \subseteq S independent B span B = S card B = dim S*
<proof>

lemma *dim-mono*: **assumes** *V \subseteq span W* **shows** *dim V \leq dim W*
<proof>

lemma *dim-subset*: *S \subseteq T \implies dim S \leq dim T*
<proof>

lemma *dim-eq-0* [*simp*]:
dim S = 0 \longleftrightarrow S \subseteq {0}
<proof>

lemma *dim-UNIV* [*simp*]: *dim UNIV = card Basis*
<proof>

lemma *independent-card-le-dim*: **assumes** *B \subseteq V* **and** *independent B* **shows** *card B \leq dim V*

$\langle \text{proof} \rangle$

lemma *dim-subset-UNIV*: $\dim S \leq \text{dimension}$
 $\langle \text{proof} \rangle$

lemma *card-ge-dim-independent*:
 assumes $BV: B \subseteq V$
 and $iB: \text{independent } B$
 and $dVB: \dim V \leq \text{card } B$
 shows $V \subseteq \text{span } B$
 $\langle \text{proof} \rangle$

lemma *card-le-dim-spanning*:
 assumes $BV: B \subseteq V$
 and $VB: V \subseteq \text{span } B$
 and $fB: \text{finite } B$
 and $dVB: \dim V \geq \text{card } B$
 shows $\text{independent } B$
 $\langle \text{proof} \rangle$

lemma *card-eq-dim*: $B \subseteq V \implies \text{card } B = \dim V \implies \text{finite } B \implies \text{independent } B$
 $\longleftrightarrow V \subseteq \text{span } B$
 $\langle \text{proof} \rangle$

lemma *subspace-dim-equal*:
 assumes *subspace* S
 and *subspace* T
 and $S \subseteq T$
 and $\dim S \geq \dim T$
 shows $S = T$
 $\langle \text{proof} \rangle$

corollary *dim-eq-span*:
 shows $\llbracket S \subseteq T; \dim T \leq \dim S \rrbracket \implies \text{span } S = \text{span } T$
 $\langle \text{proof} \rangle$

lemma *dim-psubset*:
 $\text{span } S \subset \text{span } T \implies \dim S < \dim T$
 $\langle \text{proof} \rangle$

lemma *dim-eq-full*:
 shows $\dim S = \text{dimension} \longleftrightarrow \text{span } S = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *indep-card-eq-dim-span*:
 assumes *independent* B
 shows $\text{finite } B \wedge \text{card } B = \dim (\text{span } B)$
 $\langle \text{proof} \rangle$

More general size bound lemmas.

lemma *independent-bound-general:*

independent S \implies finite S \wedge card S \leq dim S

<proof>

lemma *independent-explicit:*

shows *independent B \longleftrightarrow finite B \wedge ($\forall c. (\sum_{v \in B. c \ v * s \ v) = 0 \longrightarrow (\forall v \in B. c \ v = 0)$)*

<proof>

proposition *dim-sums-Int:*

assumes *subspace S subspace T*

shows *dim {x + y | x y. x \in S \wedge y \in T} + dim(S \cap T) = dim S + dim T (is dim ?ST + - = -)*

<proof>

lemma *dependent-biggerset-general:*

(finite S \implies card S $>$ dim S) \implies dependent S

<proof>

lemma *subset-le-dim:*

S \subseteq span T \implies dim S \leq dim T

<proof>

lemma *linear-inj-imp-surj:*

assumes *lf: linear scale scale f*

and *fi: inj f*

shows *surj f*

<proof>

end

locale *finite-dimensional-vector-space-pair-1 =*

vs1: finite-dimensional-vector-space s1 B1 + vs2: vector-space s2

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)*

begin

sublocale *vector-space-pair s1 s2 <proof>*

lemma *dim-image-eq:*

assumes *lf: linear s1 s2 f*

and *fi: inj-on f (vs1.span S)*

shows *vs2.dim (f ‘ S) = vs1.dim S*

<proof>

lemma *dim-image-le:*

assumes *lf: linear s1 s2 f*

shows *vs2.dim (f ‘ S) \leq vs1.dim (S)*

$\langle proof \rangle$

end

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** $\langle *a \rangle$ 75)

and *B1* :: '*b* *set*

and *s2* :: '*a*::*field* \Rightarrow '*c*::*ab-group-add* \Rightarrow '*c* (**infixr** $\langle *b \rangle$ 75)

and *B2* :: '*c* *set*

begin

sublocale *finite-dimensional-vector-space-pair-1* $\langle proof \rangle$

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*

$\langle proof \rangle$

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*

$\langle proof \rangle$

lemma *linear-injective-isomorphism*:

assumes *lf*: *linear* *s1* *s2* *f*

and *fi*: *inj* *f*

and *dims*: *vs2.dim UNIV* = *vs1.dim UNIV*

shows $\exists f'. \text{linear } s2 \ s1 \ f' \wedge (\forall x. f' (f x) = x) \wedge (\forall x. f (f' x) = x)$

$\langle proof \rangle$

lemma *linear-surjective-isomorphism*:

assumes *lf*: *linear* *s1* *s2* *f*

and *sf*: *surj* *f*

and *dims*: *vs2.dim UNIV* = *vs1.dim UNIV*

shows $\exists f'. \text{linear } s2 \ s1 \ f' \wedge (\forall x. f' (f x) = x) \wedge (\forall x. f (f' x) = x)$

$\langle proof \rangle$

lemma *basis-to-basis-subspace-isomorphism*:

assumes *s*: *vs1.subspace* *S*

and *t*: *vs2.subspace* *T*

and *d*: *vs1.dim* *S* = *vs2.dim* *T*

and *B*: *B* \subseteq *S* *vs1.independent* *B* *S* \subseteq *vs1.span* *B* *card* *B* = *vs1.dim* *S*

and *C*: *C* \subseteq *T* *vs2.independent* *C* *T* \subseteq *vs2.span* *C* *card* *C* = *vs2.dim* *T*

shows $\exists f. \text{linear } s1 \ s2 \ f \wedge f' B = C \wedge f' S = T \wedge \text{inj-on } f \ S$

$\langle proof \rangle$

lemma *basis-change-exists'*:

assumes *vs1.independent B vs2.independent B'*

assumes *vs1.span B = UNIV vs2.span B' = UNIV vs1.dimension = vs2.dimension*

shows $\exists g. \text{linear } s1 \ s2 \ g \wedge \text{bij } g \wedge \text{bij-betw } g \ B \ B'$

<proof>

lemma *basis-change-exists*:

assumes *vs1.dimension = vs2.dimension*

shows $\exists g. \text{linear } s1 \ s2 \ g \wedge \text{bij } g \wedge \text{bij-betw } g \ B1 \ B2$

<proof>

end

context *finite-dimensional-vector-space* **begin**

lemma *linear-surj-imp-inj*:

assumes *lf: linear scale scale f and sf: surj f*

shows *inj f*

<proof>

lemma *linear-inverse-left*:

assumes *lf: linear scale scale f*

and *lf': linear scale scale f'*

shows $f \circ f' = id \longleftrightarrow f' \circ f = id$

<proof>

lemma *left-inverse-linear*:

assumes *lf: linear scale scale f*

and *gf: g o f = id*

shows *linear scale scale g*

<proof>

lemma *inj-linear-imp-inv-linear*:

assumes *linear scale scale f inj f* **shows** *linear scale scale (inv f)*

<proof>

lemma *right-inverse-linear*:

assumes *lf: linear scale scale f*

and *gf: f o g = id*

shows *linear scale scale g*

<proof>

lemma *linear-independent-extend-inj*:

assumes *independent B independent (f ' B) inj-on f B*

shows $\exists g. \text{linear scale scale } g \wedge \text{inj } g \wedge (\forall x \in B. g \ x = f \ x)$

<proof>

end

context *finite-dimensional-vector-space-pair* **begin**

lemma *subspace-isomorphism*:

assumes *s*: *vs1.subspace S*

and *t*: *vs2.subspace T*

and *d*: *vs1.dim S = vs2.dim T*

shows $\exists f. \text{linear } s1 \ s2 \ f \wedge f \circ S = T \wedge \text{inj-on } f \ S$

<proof>

end

hide-const (**open**) *linear*

end

107 Vector Spaces and Algebras over the Reals

theory *Real-Vector-Spaces*

imports *Real Topological-Spaces Vector-Spaces*

begin

107.1 Real vector spaces

class *scaleR* =

fixes *scaleR* :: *real* \Rightarrow '*a* \Rightarrow '*a* (**infixr** '**_R*' 75)

begin

abbreviation *divideR* :: '*a* \Rightarrow *real* \Rightarrow '*a* (**infixl** '*/_R*' 70)

where *x /_R r* \equiv *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*]: $\text{scaleR } a \ x = a * x$

instance

$\langle \text{proof} \rangle$

end

locale *linear* = *Vector-Spaces.linear* $\text{scaleR} :: - \Rightarrow - \Rightarrow 'a :: \text{real-vector} \ \text{scaleR} :: - \Rightarrow - \Rightarrow 'b :: \text{real-vector}$
begin

lemmas $\text{scaleR} = \text{scale}$

end

global-interpretation *real-vector?*: *vector-space* $\text{scaleR} :: \text{real} \Rightarrow 'a \Rightarrow 'a :: \text{real-vector}$

rewrites *Vector-Spaces.linear* $(*_R) \ (*_R) = \text{linear}$

and *Vector-Spaces.linear* $(*) \ (*_R) = \text{linear}$

defines *dependent-raw-def*: $\text{dependent} = \text{real-vector.dependent}$

and *representation-raw-def*: $\text{representation} = \text{real-vector.representation}$

and *subspace-raw-def*: $\text{subspace} = \text{real-vector.subspace}$

and *span-raw-def*: $\text{span} = \text{real-vector.span}$

and *extend-basis-raw-def*: $\text{extend-basis} = \text{real-vector.extend-basis}$

and *dim-raw-def*: $\text{dim} = \text{real-vector.dim}$

$\langle \text{proof} \rangle$

hide-const (**open**)— locale constants

real-vector.dependent

real-vector.independent

real-vector.representation

real-vector.subspace

real-vector.span

real-vector.extend-basis

real-vector.dim

abbreviation $\text{independent } x \equiv \neg \text{dependent } x$

global-interpretation *real-vector?*: *vector-space-pair* $\text{scaleR} :: - \Rightarrow - \Rightarrow 'a :: \text{real-vector}$

$\text{scaleR} :: - \Rightarrow - \Rightarrow 'b :: \text{real-vector}$

rewrites *Vector-Spaces.linear* $(*_R) \ (*_R) = \text{linear}$

and *Vector-Spaces.linear* $(*) \ (*_R) = \text{linear}$

defines *construct-raw-def*: $\text{construct} = \text{real-vector.construct}$

$\langle \text{proof} \rangle$

hide-const (**open**)— locale constants

real-vector.construct

lemma *linear-compose*: $\text{linear } f \implies \text{linear } g \implies \text{linear } (g \circ f)$

<proof>

Recover original theorem names

lemmas *scaleR-left-commute* = *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*]:

$c \neq 0 \implies a = b \ /_R c \iff c *_R a = b$
 $c \neq 0 \implies b \ /_R c = a \iff b = c *_R a$
 $c \neq 0 \implies a + b \ /_R c = (c *_R a + b) \ /_R c$
 $c \neq 0 \implies a \ /_R c + b = (a + c *_R b) \ /_R c$
 $c \neq 0 \implies a - b \ /_R c = (c *_R a - b) \ /_R c$
 $c \neq 0 \implies a \ /_R c - b = (a - c *_R b) \ /_R c$
 $c \neq 0 \implies -(a \ /_R c) + b = (-a + c *_R b) \ /_R c$
 $c \neq 0 \implies -(a \ /_R c) - b = (-a - c *_R b) \ /_R c$
for $a \ b :: 'a :: \text{real-vector}$
<proof>

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 $x :: 'a :: \text{real-vector}$
<proof>

lemma *scaleR-2*:

fixes $x :: 'a::\text{real-vector}$
shows $\text{scaleR } 2 \ x = x + x$
 $\langle \text{proof} \rangle$

lemma *scaleR-half-double* [*simp*]:

fixes $a :: 'a::\text{real-vector}$
shows $(1 / 2) *_{\text{R}} (a + a) = a$
 $\langle \text{proof} \rangle$

lemma *shift-zero-ident* [*simp*]:

fixes $f :: 'a \Rightarrow 'b::\text{real-vector}$
shows $(+)0 \circ f = f$
 $\langle \text{proof} \rangle$

lemma *linear-scale-real*:

fixes $r::\text{real}$ **shows** $\text{linear } f \implies f (r * b) = r * f b$
 $\langle \text{proof} \rangle$

interpretation *scaleR-left*: *additive* $(\lambda a. \text{scaleR } a \ x :: 'a::\text{real-vector})$

$\langle \text{proof} \rangle$

interpretation *scaleR-right*: *additive* $(\lambda x. \text{scaleR } a \ x :: 'a::\text{real-vector})$

$\langle \text{proof} \rangle$

lemma *nonzero-inverse-scaleR-distrib*:

$a \neq 0 \implies x \neq 0 \implies \text{inverse } (\text{scaleR } a \ x) = \text{scaleR } (\text{inverse } a) (\text{inverse } x)$
for $x :: 'a::\text{real-div-algebra}$
 $\langle \text{proof} \rangle$

lemma *inverse-scaleR-distrib*: $\text{inverse } (\text{scaleR } a \ x) = \text{scaleR } (\text{inverse } a) (\text{inverse } x)$

for $x :: 'a::\{\text{real-div-algebra}, \text{division-ring}\}$
 $\langle \text{proof} \rangle$

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*]:

$v + (b / z) *_{\text{R}} w = (\text{if } z = 0 \text{ then } v \text{ else } (z *_{\text{R}} v + b *_{\text{R}} w) /_{\text{R}} z)$
 $a *_{\text{R}} v + (b / z) *_{\text{R}} w = (\text{if } z = 0 \text{ then } a *_{\text{R}} v \text{ else } ((a * z) *_{\text{R}} v + b *_{\text{R}} w) /_{\text{R}} z)$
 $(a / z) *_{\text{R}} v + w = (\text{if } z = 0 \text{ then } w \text{ else } (a *_{\text{R}} v + z *_{\text{R}} w) /_{\text{R}} z)$
 $(a / z) *_{\text{R}} v + b *_{\text{R}} w = (\text{if } z = 0 \text{ then } b *_{\text{R}} w \text{ else } (a *_{\text{R}} v + (b * z) *_{\text{R}} w) /_{\text{R}} z)$
 $v - (b / z) *_{\text{R}} w = (\text{if } z = 0 \text{ then } v \text{ else } (z *_{\text{R}} v - b *_{\text{R}} w) /_{\text{R}} z)$
 $a *_{\text{R}} v - (b / z) *_{\text{R}} w = (\text{if } z = 0 \text{ then } a *_{\text{R}} v \text{ else } ((a * z) *_{\text{R}} v - b *_{\text{R}} w) /_{\text{R}} z)$

$z)$
 $(a / z) *_{\mathcal{R}} v - w = (\text{if } z = 0 \text{ then } -w \text{ else } (a *_{\mathcal{R}} v - z *_{\mathcal{R}} w) /_{\mathcal{R}} z)$
 $(a / z) *_{\mathcal{R}} v - b *_{\mathcal{R}} w = (\text{if } z = 0 \text{ then } -b *_{\mathcal{R}} w \text{ else } (a *_{\mathcal{R}} v - (b * z) *_{\mathcal{R}} w) /_{\mathcal{R}} z)$
for $v :: 'a :: \text{real-vector}$
 $\langle \text{proof} \rangle$

lemma *eq-vector-fraction-iff* [*vector-add-divide-simps*]:
fixes $x :: 'a :: \text{real-vector}$
shows $(x = (u / v) *_{\mathcal{R}} a) \longleftrightarrow (\text{if } v=0 \text{ then } x = 0 \text{ else } v *_{\mathcal{R}} x = u *_{\mathcal{R}} a)$
 $\langle \text{proof} \rangle$

lemma *vector-fraction-eq-iff* [*vector-add-divide-simps*]:
fixes $x :: 'a :: \text{real-vector}$
shows $((u / v) *_{\mathcal{R}} a = x) \longleftrightarrow (\text{if } v=0 \text{ then } x = 0 \text{ else } u *_{\mathcal{R}} a = v *_{\mathcal{R}} x)$
 $\langle \text{proof} \rangle$

lemma *real-vector-affinity-eq*:
fixes $x :: 'a :: \text{real-vector}$
assumes $m0: m \neq 0$
shows $m *_{\mathcal{R}} x + c = y \longleftrightarrow x = \text{inverse } m *_{\mathcal{R}} y - (\text{inverse } m *_{\mathcal{R}} c)$
 $(\text{is } ?lhs \longleftrightarrow ?rhs)$
 $\langle \text{proof} \rangle$

lemma *real-vector-eq-affinity*: $m \neq 0 \implies y = m *_{\mathcal{R}} x + c \longleftrightarrow \text{inverse } m *_{\mathcal{R}} y - (\text{inverse } m *_{\mathcal{R}} c) = x$
for $x :: 'a :: \text{real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-eq-iff* [*simp*]: $b + u *_{\mathcal{R}} a = a + u *_{\mathcal{R}} b \longleftrightarrow a = b \vee u = 1$
for $a :: 'a :: \text{real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-collapse* [*simp*]: $(1 - u) *_{\mathcal{R}} a + u *_{\mathcal{R}} a = a$
for $a :: 'a :: \text{real-vector}$
 $\langle \text{proof} \rangle$

107.2 Embedding of the Reals into any *real-algebra-1*: *of-real*

definition *of-real* :: $\text{real} \Rightarrow 'a :: \text{real-algebra-1}$
where *of-real* $r = \text{scaleR } r \ 1$

lemma *scaleR-conv-of-real*: $\text{scaleR } r \ x = \text{of-real } r * x$
 $\langle \text{proof} \rangle$

lemma *of-real-0* [*simp*]: $\text{of-real } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *of-real-1* [*simp*]: *of-real* 1 = 1
 ⟨*proof*⟩

lemma *of-real-add* [*simp*]: *of-real* (x + y) = *of-real* x + *of-real* y
 ⟨*proof*⟩

lemma *of-real-minus* [*simp*]: *of-real* (− x) = − *of-real* x
 ⟨*proof*⟩

lemma *of-real-diff* [*simp*]: *of-real* (x − y) = *of-real* x − *of-real* y
 ⟨*proof*⟩

lemma *of-real-mult* [*simp*]: *of-real* (x * y) = *of-real* x * *of-real* y
 ⟨*proof*⟩

lemma *of-real-sum* [*simp*]: *of-real* (sum f s) = (∑ x ∈ s. *of-real* (f x))
 ⟨*proof*⟩

lemma *of-real-prod* [*simp*]: *of-real* (prod f s) = (∏ x ∈ s. *of-real* (f x))
 ⟨*proof*⟩

lemma *sum-list-of-real*: *sum-list* (map *of-real* xs) = *of-real* (*sum-list* xs)
 ⟨*proof*⟩

lemma *nonzero-of-real-inverse*:
 x ≠ 0 ⇒ *of-real* (inverse x) = inverse (*of-real* x :: 'a::real-div-algebra)
 ⟨*proof*⟩

lemma *of-real-inverse* [*simp*]:
of-real (inverse x) = inverse (*of-real* x :: 'a::{real-div-algebra, division-ring})
 ⟨*proof*⟩

lemma *nonzero-of-real-divide*:
 y ≠ 0 ⇒ *of-real* (x / y) = (*of-real* x / *of-real* y :: 'a::real-field)
 ⟨*proof*⟩

lemma *of-real-divide* [*simp*]:
of-real (x / y) = (*of-real* x / *of-real* y :: 'a::real-div-algebra)
 ⟨*proof*⟩

lemma *of-real-power* [*simp*]:
of-real (x ^ n) = (*of-real* x :: 'a::{real-algebra-1}) ^ n
 ⟨*proof*⟩

lemma *of-real-power-int* [*simp*]:
of-real (power-int x n) = power-int (*of-real* x :: 'a :: {real-div-algebra, division-ring})
 n
 ⟨*proof*⟩

lemma *of-real-eq-iff* [simp]: *of-real* $x = \text{of-real } y \longleftrightarrow x = y$
 ⟨proof⟩

lemma *inj-of-real*: *inj of-real*
 ⟨proof⟩

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]: $-\text{of-real } x = \text{of-real } y \longleftrightarrow -x = y$
 ⟨proof⟩

lemma *of-real-eq-minus-of-real-iff* [simp]: $\text{of-real } x = -\text{of-real } y \longleftrightarrow x = -y$
 ⟨proof⟩

lemma *of-real-eq-id* [simp]: *of-real* = (*id* :: *real* \Rightarrow *real*)
 ⟨proof⟩

Collapse nested embeddings.

lemma *of-real-of-nat-eq* [simp]: *of-real* (*of-nat* n) = *of-nat* n
 ⟨proof⟩

lemma *of-real-of-int-eq* [simp]: *of-real* (*of-int* z) = *of-int* z
 ⟨proof⟩

lemma *of-real-numeral* [simp]: *of-real* (*numeral* w) = *numeral* w
 ⟨proof⟩

lemma *of-real-neg-numeral* [simp]: *of-real* ($-\text{numeral } w$) = $-\text{numeral } w$
 ⟨proof⟩

lemma *numeral-power-int-eq-of-real-cancel-iff* [simp]:
 $\text{power-int } (\text{numeral } x) \ n = (\text{of-real } y :: 'a :: \{\text{real-div-algebra, division-ring}\}) \longleftrightarrow$
 $\text{power-int } (\text{numeral } x) \ n = y$
 ⟨proof⟩

lemma *of-real-eq-numeral-power-int-cancel-iff* [simp]:
 $(\text{of-real } y :: 'a :: \{\text{real-div-algebra, division-ring}\}) = \text{power-int } (\text{numeral } x) \ n \longleftrightarrow$
 $y = \text{power-int } (\text{numeral } x) \ n$
 ⟨proof⟩

lemma *of-real-eq-of-real-power-int-cancel-iff* [simp]:
 $\text{power-int } (\text{of-real } b :: 'a :: \{\text{real-div-algebra, division-ring}\}) \ w = \text{of-real } x \longleftrightarrow$
 $\text{power-int } b \ w = x$
 ⟨proof⟩

lemma *of-real-in-Ints-iff* [simp]: *of-real* $x \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$
 ⟨proof⟩

lemma *Ints-of-real* [intro]: $x \in \mathbb{Z} \implies \text{of-real } x \in \mathbb{Z}$
 ⟨proof⟩

Every real algebra has characteristic zero.

instance *real-algebra-1* < *ring-char-0*
 ⟨proof⟩

lemma *fraction-scaleR-times* [simp]:
 fixes $a :: 'a::\text{real-algebra-1}$
 shows $(\text{numeral } u / \text{numeral } v) *_{\mathbb{R}} (\text{numeral } w * a) = (\text{numeral } u * \text{numeral } w / \text{numeral } v) *_{\mathbb{R}} a$
 ⟨proof⟩

lemma *inverse-scaleR-times* [simp]:
 fixes $a :: 'a::\text{real-algebra-1}$
 shows $(1 / \text{numeral } v) *_{\mathbb{R}} (\text{numeral } w * a) = (\text{numeral } w / \text{numeral } v) *_{\mathbb{R}} a$
 ⟨proof⟩

lemma *scaleR-times* [simp]:
 fixes $a :: 'a::\text{real-algebra-1}$
 shows $(\text{numeral } u) *_{\mathbb{R}} (\text{numeral } w * a) = (\text{numeral } u * \text{numeral } w) *_{\mathbb{R}} a$
 ⟨proof⟩

instance *real-field* < *field-char-0* ⟨proof⟩

107.3 The Set of Real Numbers

definition *Reals* :: $'a::\text{real-algebra-1}$ set (\mathbb{R})
 where $\mathbb{R} = \text{range of-real}$

lemma *Reals-of-real* [simp]: $\text{of-real } r \in \mathbb{R}$
 ⟨proof⟩

lemma *Reals-of-int* [simp]: $\text{of-int } z \in \mathbb{R}$
 ⟨proof⟩

lemma *Reals-of-nat* [simp]: $\text{of-nat } n \in \mathbb{R}$
 ⟨proof⟩

lemma *Reals-numeral* [simp]: $\text{numeral } w \in \mathbb{R}$
 ⟨proof⟩

lemma *Reals-0* [simp]: $0 \in \mathbb{R}$ and *Reals-1* [simp]: $1 \in \mathbb{R}$
 ⟨proof⟩

lemma *Reals-add* [simp]: $a \in \mathbb{R} \implies b \in \mathbb{R} \implies a + b \in \mathbb{R}$
 ⟨proof⟩

lemma *Reals-minus* [simp]: $a \in \mathbb{R} \implies -a \in \mathbb{R}$

$\langle \text{proof} \rangle$

lemma *Reals-minus-iff* [simp]: $- a \in \mathbb{R} \longleftrightarrow a \in \mathbb{R}$
 $\langle \text{proof} \rangle$

lemma *Reals-diff* [simp]: $a \in \mathbb{R} \implies b \in \mathbb{R} \implies a - b \in \mathbb{R}$
 $\langle \text{proof} \rangle$

lemma *Reals-mult* [simp]: $a \in \mathbb{R} \implies b \in \mathbb{R} \implies a * b \in \mathbb{R}$
 $\langle \text{proof} \rangle$

lemma *nonzero-Reals-inverse*: $a \in \mathbb{R} \implies a \neq 0 \implies \text{inverse } a \in \mathbb{R}$
for $a :: 'a::\text{real-div-algebra}$
 $\langle \text{proof} \rangle$

lemma *Reals-inverse*: $a \in \mathbb{R} \implies \text{inverse } a \in \mathbb{R}$
for $a :: 'a::\{\text{real-div-algebra}, \text{division-ring}\}$
 $\langle \text{proof} \rangle$

lemma *Reals-inverse-iff* [simp]: $\text{inverse } x \in \mathbb{R} \longleftrightarrow x \in \mathbb{R}$
for $x :: 'a::\{\text{real-div-algebra}, \text{division-ring}\}$
 $\langle \text{proof} \rangle$

lemma *nonzero-Reals-divide*: $a \in \mathbb{R} \implies b \in \mathbb{R} \implies b \neq 0 \implies a / b \in \mathbb{R}$
for $a \ b :: 'a::\text{real-field}$
 $\langle \text{proof} \rangle$

lemma *Reals-divide* [simp]: $a \in \mathbb{R} \implies b \in \mathbb{R} \implies a / b \in \mathbb{R}$
for $a \ b :: 'a::\{\text{real-field}, \text{field}\}$
 $\langle \text{proof} \rangle$

lemma *Reals-power* [simp]: $a \in \mathbb{R} \implies a ^ n \in \mathbb{R}$
for $a :: 'a::\text{real-algebra-1}$
 $\langle \text{proof} \rangle$

lemma *Reals-cases* [cases set: *Reals*]:
assumes $q \in \mathbb{R}$
obtains (*of-real*) r **where** $q = \text{of-real } r$
 $\langle \text{proof} \rangle$

lemma *sum-in-Reals* [intro,simp]: $(\bigwedge i. i \in s \implies f \ i \in \mathbb{R}) \implies \text{sum } f \ s \in \mathbb{R}$
 $\langle \text{proof} \rangle$

lemma *prod-in-Reals* [intro,simp]: $(\bigwedge i. i \in s \implies f \ i \in \mathbb{R}) \implies \text{prod } f \ s \in \mathbb{R}$
 $\langle \text{proof} \rangle$

lemma *Reals-induct* [case-names *of-real*, induct set: *Reals*]:
 $q \in \mathbb{R} \implies (\bigwedge r. P \ (\text{of-real } r)) \implies P \ q$
 $\langle \text{proof} \rangle$

107.4 Ordered real vector spaces

class *ordered-real-vector* = *real-vector* + *ordered-ab-group-add* +
assumes *scaleR-left-mono*: $x \leq y \implies 0 \leq a \implies a *_R x \leq a *_R y$
and *scaleR-right-mono*: $a \leq b \implies 0 \leq x \implies a *_R x \leq b *_R x$
begin

lemma *scaleR-mono*:

$a \leq b \implies x \leq y \implies 0 \leq b \implies 0 \leq x \implies a *_R x \leq b *_R y$
 $\langle \text{proof} \rangle$

lemma *scaleR-mono'*:

$a \leq b \implies c \leq d \implies 0 \leq a \implies 0 \leq c \implies a *_R c \leq b *_R d$
 $\langle \text{proof} \rangle$

lemma *pos-le-divideR-eq* [*field-simps*]:

$a \leq b \ /_R c \longleftrightarrow c *_R a \leq b$ (**is** $?P \longleftrightarrow ?Q$) **if** $0 < c$
 $\langle \text{proof} \rangle$

lemma *pos-less-divideR-eq* [*field-simps*]:

$a < b \ /_R c \longleftrightarrow c *_R a < b$ **if** $c > 0$
 $\langle \text{proof} \rangle$

lemma *pos-divideR-le-eq* [*field-simps*]:

$b \ /_R c \leq a \longleftrightarrow b \leq c *_R a$ **if** $c > 0$
 $\langle \text{proof} \rangle$

lemma *pos-divideR-less-eq* [*field-simps*]:

$b \ /_R c < a \longleftrightarrow b < c *_R a$ **if** $c > 0$
 $\langle \text{proof} \rangle$

lemma *pos-le-minus-divideR-eq* [*field-simps*]:

$a \leq - (b \ /_R c) \longleftrightarrow c *_R a \leq - b$ **if** $c > 0$
 $\langle \text{proof} \rangle$

lemma *pos-less-minus-divideR-eq* [*field-simps*]:

$a < - (b \ /_R c) \longleftrightarrow c *_R a < - b$ **if** $c > 0$
 $\langle \text{proof} \rangle$

lemma *pos-minus-divideR-le-eq* [*field-simps*]:

$- (b \ /_R c) \leq a \longleftrightarrow - b \leq c *_R a$ **if** $c > 0$
 $\langle \text{proof} \rangle$

lemma *pos-minus-divideR-less-eq* [*field-simps*]:

$- (b \ /_R c) < a \longleftrightarrow - b < c *_R a$ **if** $c > 0$
 $\langle \text{proof} \rangle$

lemma *scaleR-image-atLeastAtMost*: $c > 0 \implies \text{scaleR } c \text{ ` } \{x..y\} = \{c *_R x..c *_R y\}$

$\langle \text{proof} \rangle$

end

lemma *neg-le-divideR-eq* [*field-simps*]:
 $a \leq b \ /_R c \longleftrightarrow b \leq c *_R a$ (**is** $?P \longleftrightarrow ?Q$) **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *neg-less-divideR-eq* [*field-simps*]:
 $a < b \ /_R c \longleftrightarrow b < c *_R a$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *neg-divideR-le-eq* [*field-simps*]:
 $b \ /_R c \leq a \longleftrightarrow c *_R a \leq b$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *neg-divideR-less-eq* [*field-simps*]:
 $b \ /_R c < a \longleftrightarrow c *_R a < b$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *neg-le-minus-divideR-eq* [*field-simps*]:
 $a \leq - (b \ /_R c) \longleftrightarrow - b \leq c *_R a$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *neg-less-minus-divideR-eq* [*field-simps*]:
 $a < - (b \ /_R c) \longleftrightarrow - b < c *_R a$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *neg-minus-divideR-le-eq* [*field-simps*]:
 $- (b \ /_R c) \leq a \longleftrightarrow c *_R a \leq - b$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *neg-minus-divideR-less-eq* [*field-simps*]:
 $- (b \ /_R c) < a \longleftrightarrow c *_R a < - b$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma [*field-split-simps*]:
 $a = b \ /_R c \longleftrightarrow (\text{if } c = 0 \text{ then } a = 0 \text{ else } c *_R a = b)$
 $b \ /_R c = a \longleftrightarrow (\text{if } c = 0 \text{ then } a = 0 \text{ else } b = c *_R a)$
 $a + b \ /_R c = (\text{if } c = 0 \text{ then } a \text{ else } (c *_R a + b) \ /_R c)$
 $a \ /_R c + b = (\text{if } c = 0 \text{ then } b \text{ else } (a + c *_R b) \ /_R c)$
 $a - b \ /_R c = (\text{if } c = 0 \text{ then } a \text{ else } (c *_R a - b) \ /_R c)$

$a /_R c - b = (\text{if } c = 0 \text{ then } -b \text{ else } (a - c *_R b) /_R c)$
 $-(a /_R c) + b = (\text{if } c = 0 \text{ then } b \text{ else } (-a + c *_R b) /_R c)$
 $-(a /_R c) - b = (\text{if } c = 0 \text{ then } -b \text{ else } (-a - c *_R b) /_R c)$
for $a \ b :: 'a :: \text{real-vector}$
 $\langle \text{proof} \rangle$

lemma *[field-split-simps]*:

$0 < c \implies a \leq b /_R c \longleftrightarrow (\text{if } c > 0 \text{ then } c *_R a \leq b \text{ else if } c < 0 \text{ then } b \leq c *_R a \text{ else } a \leq 0)$
 $0 < c \implies a < b /_R c \longleftrightarrow (\text{if } c > 0 \text{ then } c *_R a < b \text{ else if } c < 0 \text{ then } b < c *_R a \text{ else } a < 0)$
 $0 < c \implies b /_R c \leq a \longleftrightarrow (\text{if } c > 0 \text{ then } b \leq c *_R a \text{ else if } c < 0 \text{ then } c *_R a \leq b \text{ else } a \geq 0)$
 $0 < c \implies b /_R c < a \longleftrightarrow (\text{if } c > 0 \text{ then } b < c *_R a \text{ else if } c < 0 \text{ then } c *_R a < b \text{ else } a > 0)$
 $0 < c \implies a \leq -(b /_R c) \longleftrightarrow (\text{if } c > 0 \text{ then } c *_R a \leq -b \text{ else if } c < 0 \text{ then } -b \leq c *_R a \text{ else } a \leq 0)$
 $0 < c \implies a < -(b /_R c) \longleftrightarrow (\text{if } c > 0 \text{ then } c *_R a < -b \text{ else if } c < 0 \text{ then } -b < c *_R a \text{ else } a < 0)$
 $0 < c \implies -(b /_R c) \leq a \longleftrightarrow (\text{if } c > 0 \text{ then } -b \leq c *_R a \text{ else if } c < 0 \text{ then } c *_R a \leq -b \text{ else } a \geq 0)$
 $0 < c \implies -(b /_R c) < a \longleftrightarrow (\text{if } c > 0 \text{ then } -b < c *_R a \text{ else if } c < 0 \text{ then } c *_R a < -b \text{ else } a > 0)$
for $a \ b :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-nonneg-nonneg*: $0 \leq a \implies 0 \leq x \implies 0 \leq a *_R x$
for $x :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-nonneg-nonpos*: $0 \leq a \implies x \leq 0 \implies a *_R x \leq 0$
for $x :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-nonpos-nonneg*: $a \leq 0 \implies 0 \leq x \implies a *_R x \leq 0$
for $x :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *split-scaleR-neg-le*: $(0 \leq a \wedge x \leq 0) \vee (a \leq 0 \wedge 0 \leq x) \implies a *_R x \leq 0$
for $x :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *le-add-iff1*: $a *_R e + c \leq b *_R e + d \longleftrightarrow (a - b) *_R e + c \leq d$
for $c \ d \ e :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *le-add-iff2*: $a *_R e + c \leq b *_R e + d \longleftrightarrow c \leq (b - a) *_R e + d$
for $c \ d \ e :: 'a :: \text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-left-mono-neg*: $b \leq a \implies c \leq 0 \implies c *_{\mathcal{R}} a \leq c *_{\mathcal{R}} b$
for $a \ b :: 'a::\text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-right-mono-neg*: $b \leq a \implies c \leq 0 \implies a *_{\mathcal{R}} c \leq b *_{\mathcal{R}} c$
for $c :: 'a::\text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-nonpos-nonpos*: $a \leq 0 \implies b \leq 0 \implies 0 \leq a *_{\mathcal{R}} b$
for $b :: 'a::\text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *split-scaleR-pos-le*: $(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \implies 0 \leq a *_{\mathcal{R}} b$
for $b :: 'a::\text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *zero-le-scaleR-iff*:
fixes $b :: 'a::\text{ordered-real-vector}$
shows $0 \leq a *_{\mathcal{R}} b \longleftrightarrow 0 < a \wedge 0 \leq b \vee a < 0 \wedge b \leq 0 \vee a = 0$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *scaleR-le-0-iff*: $a *_{\mathcal{R}} b \leq 0 \longleftrightarrow 0 < a \wedge b \leq 0 \vee a < 0 \wedge 0 \leq b \vee a = 0$
for $b :: 'a::\text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-le-cancel-left*: $c *_{\mathcal{R}} a \leq c *_{\mathcal{R}} b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
for $b :: 'a::\text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-le-cancel-left-pos*: $0 < c \implies c *_{\mathcal{R}} a \leq c *_{\mathcal{R}} b \longleftrightarrow a \leq b$
for $b :: 'a::\text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-le-cancel-left-neg*: $c < 0 \implies c *_{\mathcal{R}} a \leq c *_{\mathcal{R}} b \longleftrightarrow b \leq a$
for $b :: 'a::\text{ordered-real-vector}$
 $\langle \text{proof} \rangle$

lemma *scaleR-left-le-one-le*: $0 \leq x \implies a \leq 1 \implies a *_{\mathcal{R}} x \leq x$
for $x :: 'a::\text{ordered-real-vector}$ **and** $a :: \text{real}$
 $\langle \text{proof} \rangle$

107.5 Real normed vector spaces

class *dist* =
fixes $\text{dist} :: 'a \Rightarrow 'a \Rightarrow \text{real}$

```

class norm =
  fixes norm :: 'a  $\Rightarrow$  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 $\in$ {0 <..}. principal {(x, y). dist
x y < e})
begin

lemma eventually-uniformity-metric:
  eventually P uniformity  $\longleftrightarrow$  ( $\exists e > 0. \forall x y. \text{dist } x y < e \longrightarrow P(x, y)$ )
  <proof>

end

class real-normed-vector = real-vector + sgn-div-norm + dist-norm + unifor-
mity-dist + open-uniformity +
  assumes norm-eq-zero [simp]: norm x = 0  $\longleftrightarrow$  x = 0
  and norm-triangle-ineq: norm (x + y)  $\leq$  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>

lemma bdd-below-norm-image: bdd-below (norm ‘ A)
  <proof>

end

class real-normed-algebra = real-algebra + real-normed-vector +
  assumes norm-mult-ineq: norm (x * y)  $\leq$  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)
  <proof>

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 $x\ y :: 'a :: \text{real-normed-vector}$
shows $r \neq 0 \implies y = x /_R r \longleftrightarrow r *_R y = x$
 $\langle \text{proof} \rangle$

class $\text{real-normed-field} = \text{real-field} + \text{real-normed-div-algebra}$

lemma dist-mult-left :
 $\text{dist } (a * b) (a * c :: 'a :: \text{real-normed-field}) = \text{norm } a * \text{dist } b\ c$
 $\langle \text{proof} \rangle$

lemma dist-mult-right :
 $\text{dist } (b * a) (c * a :: 'a :: \text{real-normed-field}) = \text{norm } a * \text{dist } b\ c$
 $\langle \text{proof} \rangle$

instance $\text{real-normed-div-algebra} < \text{real-normed-algebra-1}$
 $\langle \text{proof} \rangle$

context $\text{real-normed-vector}$ **begin**

lemma norm-zero $[\text{simp}]$: $\text{norm } (0 :: 'a) = 0$
 $\langle \text{proof} \rangle$

lemma $\text{zero-less-norm-iff}$ $[\text{simp}]$: $\text{norm } x > 0 \longleftrightarrow x \neq 0$
 $\langle \text{proof} \rangle$

lemma $\text{norm-not-less-zero}$ $[\text{simp}]$: $\neg \text{norm } x < 0$
 $\langle \text{proof} \rangle$

lemma norm-le-zero-iff $[\text{simp}]$: $\text{norm } x \leq 0 \longleftrightarrow x = 0$
 $\langle \text{proof} \rangle$

lemma norm-minus-cancel $[\text{simp}]$: $\text{norm } (-\ x) = \text{norm } x$
 $\langle \text{proof} \rangle$

lemma $\text{norm-minus-commute}$: $\text{norm } (a - b) = \text{norm } (b - a)$
 $\langle \text{proof} \rangle$

lemma dist-add-cancel $[\text{simp}]$: $\text{dist } (a + b) (a + c) = \text{dist } b\ c$
 $\langle \text{proof} \rangle$

lemma dist-add-cancel2 $[\text{simp}]$: $\text{dist } (b + a) (c + a) = \text{dist } b\ c$
 $\langle \text{proof} \rangle$

lemma norm-uminus-minus : $\text{norm } (-\ x - y) = \text{norm } (x + y)$
 $\langle \text{proof} \rangle$

lemma $\text{norm-triangle-ineq2}$: $\text{norm } a - \text{norm } b \leq \text{norm } (a - b)$
 $\langle \text{proof} \rangle$

lemma *norm-triangle-ineq3*: $|norm\ a - norm\ b| \leq norm\ (a - b)$
 $\langle proof \rangle$

lemma *norm-triangle-ineq4*: $norm\ (a - b) \leq norm\ a + norm\ b$
 $\langle proof \rangle$

lemma *norm-triangle-le-diff*: $norm\ x + norm\ y \leq e \implies norm\ (x - y) \leq e$
 $\langle proof \rangle$

lemma *norm-diff-ineq*: $norm\ a - norm\ b \leq norm\ (a + b)$
 $\langle proof \rangle$

lemma *norm-triangle-sub*: $norm\ x \leq norm\ y + norm\ (x - y)$
 $\langle proof \rangle$

lemma *norm-triangle-le*: $norm\ x + norm\ y \leq e \implies norm\ (x + y) \leq e$
 $\langle proof \rangle$

lemma *norm-triangle-lt*: $norm\ x + norm\ y < e \implies norm\ (x + y) < e$
 $\langle proof \rangle$

lemma *norm-add-leD*: $norm\ (a + b) \leq c \implies norm\ b \leq norm\ a + c$
 $\langle proof \rangle$

lemma *norm-diff-triangle-ineq*: $norm\ ((a + b) - (c + d)) \leq norm\ (a - c) + norm\ (b - d)$
 $\langle proof \rangle$

lemma *norm-diff-triangle-le*: $norm\ (x - z) \leq e1 + e2$
if $norm\ (x - y) \leq e1$ $norm\ (y - z) \leq e2$
 $\langle proof \rangle$

lemma *norm-diff-triangle-less*: $norm\ (x - z) < e1 + e2$
if $norm\ (x - y) < e1$ $norm\ (y - z) < e2$
 $\langle proof \rangle$

lemma *norm-triangle-mono*:
 $norm\ a \leq r \implies norm\ b \leq s \implies norm\ (a + b) \leq r + s$
 $\langle proof \rangle$

lemma *norm-sum*: $norm\ (sum\ f\ A) \leq (\sum\ i \in A. norm\ (f\ i))$
for $f::'b \Rightarrow 'a$
 $\langle proof \rangle$

lemma *sum-norm-le*: $norm\ (sum\ f\ S) \leq sum\ g\ S$
if $\bigwedge x. x \in S \implies norm\ (f\ x) \leq g\ x$
for $f::'b \Rightarrow 'a$
 $\langle proof \rangle$

lemma *abs-norm-cancel* [simp]: $|norm\ a| = norm\ a$
 ⟨proof⟩

lemma *sum-norm-bound*:
 $norm\ (sum\ f\ S) \leq of_nat\ (card\ S) * K$
if $\bigwedge x. x \in S \implies norm\ (f\ x) \leq K$
for $f :: 'b \Rightarrow 'a$
 ⟨proof⟩

lemma *norm-add-less*: $norm\ x < r \implies norm\ y < s \implies norm\ (x + y) < r + s$
 ⟨proof⟩

end

lemma *dist-sum-le*:
fixes $f :: 'a \Rightarrow 'b :: real-normed-vector$
shows $dist\ (\sum x \in A. f\ x)\ (\sum x \in A. g\ x) \leq (\sum x \in A. dist\ (f\ x)\ (g\ x))$
 ⟨proof⟩

lemma *dist-scaleR* [simp]: $dist\ (x *_R\ a)\ (y *_R\ a) = |x - y| * norm\ a$
for $a :: 'a :: real-normed-vector$
 ⟨proof⟩

lemma *norm-mult-less*: $norm\ x < r \implies norm\ y < s \implies norm\ (x * y) < r * s$
for $x\ y :: 'a :: real-normed-algebra$
 ⟨proof⟩

lemma *norm-of-real* [simp]: $norm\ (of_real\ r :: 'a :: real-normed-algebra-1) = |r|$
 ⟨proof⟩

lemma *norm-numeral* [simp]: $norm\ (numeral\ w :: 'a :: real-normed-algebra-1) = numeral\ w$
 ⟨proof⟩

lemma *norm-neg-numeral* [simp]: $norm\ (-\ numeral\ w :: 'a :: real-normed-algebra-1) = numeral\ w$
 ⟨proof⟩

lemma *norm-of-real-add1* [simp]: $norm\ (of_real\ x + 1 :: 'a :: real-normed-div-algebra) = |x + 1|$
 ⟨proof⟩

lemma *norm-of-real-addn* [simp]:
 $norm\ (of_real\ x + numeral\ b :: 'a :: real-normed-div-algebra) = |x + numeral\ b|$
 ⟨proof⟩

lemma *norm-of-int* [simp]: $norm\ (of_int\ z :: 'a :: real-normed-algebra-1) = |of_int\ z|$
 ⟨proof⟩

lemma *norm-of-nat* [simp]: $\text{norm } (\text{of-nat } n :: 'a :: \text{real-normed-algebra-1}) = \text{of-nat } n$
 ⟨proof⟩

lemma *nonzero-norm-inverse*: $a \neq 0 \implies \text{norm } (\text{inverse } a) = \text{inverse } (\text{norm } a)$
for $a :: 'a :: \text{real-normed-div-algebra}$
 ⟨proof⟩

lemma *norm-inverse*: $\text{norm } (\text{inverse } a) = \text{inverse } (\text{norm } a)$
for $a :: 'a :: \{\text{real-normed-div-algebra}, \text{division-ring}\}$
 ⟨proof⟩

lemma *nonzero-norm-divide*: $b \neq 0 \implies \text{norm } (a / b) = \text{norm } a / \text{norm } b$
for $a \ b :: 'a :: \text{real-normed-field}$
 ⟨proof⟩

lemma *norm-divide*: $\text{norm } (a / b) = \text{norm } a / \text{norm } b$
for $a \ b :: 'a :: \{\text{real-normed-field}, \text{field}\}$
 ⟨proof⟩

lemma *dist-divide-right*: $\text{dist } (a/c) (b/c) = \text{dist } a \ b / \text{norm } c$ **for** $c :: 'a :: \text{real-normed-field}$
 ⟨proof⟩

lemma *norm-inverse-le-norm*:
fixes $x :: 'a :: \text{real-normed-div-algebra}$
shows $r \leq \text{norm } x \implies 0 < r \implies \text{norm } (\text{inverse } x) \leq \text{inverse } r$
 ⟨proof⟩

lemma *norm-power-ineq*: $\text{norm } (x \wedge n) \leq \text{norm } x \wedge n$
for $x :: 'a :: \text{real-normed-algebra-1}$
 ⟨proof⟩

lemma *norm-power*: $\text{norm } (x \wedge n) = \text{norm } x \wedge n$
for $x :: 'a :: \text{real-normed-div-algebra}$
 ⟨proof⟩

lemma *norm-power-int*: $\text{norm } (\text{power-int } x \ n) = \text{power-int } (\text{norm } x) \ n$
for $x :: 'a :: \text{real-normed-div-algebra}$
 ⟨proof⟩

lemma *power-eq-imp-eq-norm*:
fixes $w :: 'a :: \text{real-normed-div-algebra}$
assumes $\text{eq}: w \wedge n = z \wedge n$ **and** $n > 0$
shows $\text{norm } w = \text{norm } z$
 ⟨proof⟩

lemma *power-eq-1-iff*:
fixes $w :: 'a :: \text{real-normed-div-algebra}$
shows $w \wedge n = 1 \implies \text{norm } w = 1 \vee n = 0$

$\langle \text{proof} \rangle$

lemma *norm-mult-numeral1* [simp]: $\text{norm } (\text{numeral } w * a) = \text{numeral } w * \text{norm } a$
for $a :: 'a :: \{\text{real-normed-field}, \text{field}\}$
 $\langle \text{proof} \rangle$

lemma *norm-mult-numeral2* [simp]: $\text{norm } (a * \text{numeral } w) = \text{norm } a * \text{numeral } w$
for $a :: 'a :: \{\text{real-normed-field}, \text{field}\}$
 $\langle \text{proof} \rangle$

lemma *norm-divide-numeral* [simp]: $\text{norm } (a / \text{numeral } w) = \text{norm } a / \text{numeral } w$
for $a :: 'a :: \{\text{real-normed-field}, \text{field}\}$
 $\langle \text{proof} \rangle$

lemma *norm-of-real-diff* [simp]:
 $\text{norm } (\text{of-real } b - \text{of-real } a :: 'a :: \text{real-normed-algebra-1}) \leq |b - a|$
 $\langle \text{proof} \rangle$

Despite a superficial resemblance, *norm-eq-1* is not relevant.

lemma *square-norm-one*:
fixes $x :: 'a :: \text{real-normed-div-algebra}$
assumes $x^2 = 1$
shows $\text{norm } x = 1$
 $\langle \text{proof} \rangle$

lemma *norm-less-p1*: $\text{norm } x < \text{norm } (\text{of-real } (\text{norm } x) + 1 :: 'a)$
for $x :: 'a :: \text{real-normed-algebra-1}$
 $\langle \text{proof} \rangle$

lemma *prod-norm*: $\text{prod } (\lambda x. \text{norm } (f x)) A = \text{norm } (\text{prod } f A)$
for $f :: 'a \Rightarrow 'b :: \{\text{comm-semiring-1}, \text{real-normed-div-algebra}\}$
 $\langle \text{proof} \rangle$

lemma *norm-prod-le*:
 $\text{norm } (\text{prod } f A) \leq (\prod_{a \in A} \text{norm } (f a :: 'a :: \{\text{real-normed-algebra-1}, \text{comm-monoid-mult}\}))$
 $\langle \text{proof} \rangle$

lemma *norm-prod-diff*:
fixes $z w :: 'i \Rightarrow 'a :: \{\text{real-normed-algebra-1}, \text{comm-monoid-mult}\}$
shows $(\bigwedge i. i \in I \implies \text{norm } (z i) \leq 1) \implies (\bigwedge i. i \in I \implies \text{norm } (w i) \leq 1) \implies$
 $\text{norm } ((\prod_{i \in I} z i) - (\prod_{i \in I} w i)) \leq (\sum_{i \in I} \text{norm } (z i - w i))$
 $\langle \text{proof} \rangle$

lemma *norm-power-diff*:
fixes $z w :: 'a :: \{\text{real-normed-algebra-1}, \text{comm-monoid-mult}\}$
assumes $\text{norm } z \leq 1 \text{ norm } w \leq 1$

shows $\text{norm } (z \hat{m} - w \hat{m}) \leq m * \text{norm } (z - w)$
 $\langle \text{proof} \rangle$

107.6 Metric spaces

class *metric-space* = *uniformity-dist* + *open-uniformity* +
assumes *dist-eq-0-iff* [*simp*]: $\text{dist } x \ y = 0 \longleftrightarrow x = y$
and *dist-triangle2*: $\text{dist } x \ y \leq \text{dist } x \ z + \text{dist } y \ z$
begin

lemma *dist-self* [*simp*]: $\text{dist } x \ x = 0$
 $\langle \text{proof} \rangle$

lemma *zero-le-dist* [*simp*]: $0 \leq \text{dist } x \ y$
 $\langle \text{proof} \rangle$

lemma *zero-less-dist-iff*: $0 < \text{dist } x \ y \longleftrightarrow x \neq y$
 $\langle \text{proof} \rangle$

lemma *dist-not-less-zero* [*simp*]: $\neg \text{dist } x \ y < 0$
 $\langle \text{proof} \rangle$

lemma *dist-le-zero-iff* [*simp*]: $\text{dist } x \ y \leq 0 \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *dist-commute*: $\text{dist } x \ y = \text{dist } y \ x$
 $\langle \text{proof} \rangle$

lemma *dist-commute-lessI*: $\text{dist } y \ x < e \implies \text{dist } x \ y < e$
 $\langle \text{proof} \rangle$

lemma *dist-triangle*: $\text{dist } x \ z \leq \text{dist } x \ y + \text{dist } y \ z$
 $\langle \text{proof} \rangle$

lemma *dist-triangle3*: $\text{dist } x \ y \leq \text{dist } a \ x + \text{dist } a \ y$
 $\langle \text{proof} \rangle$

lemma *abs-dist-diff-le*: $|\text{dist } a \ b - \text{dist } b \ c| \leq \text{dist } a \ c$
 $\langle \text{proof} \rangle$

lemma *dist-pos-lt*: $x \neq y \implies 0 < \text{dist } x \ y$
 $\langle \text{proof} \rangle$

lemma *dist-nz*: $x \neq y \longleftrightarrow 0 < \text{dist } x \ y$
 $\langle \text{proof} \rangle$

declare *dist-nz* [*symmetric*, *simp*]

lemma *dist-triangle-le*: $\text{dist } x \ z + \text{dist } y \ z \leq e \implies \text{dist } x \ y \leq e$

<proof>

lemma *dist-triangle-lt*: $\text{dist } x \ z + \text{dist } y \ z < e \implies \text{dist } x \ y < e$
<proof>

lemma *dist-triangle-less-add*: $\text{dist } x1 \ y < e1 \implies \text{dist } x2 \ y < e2 \implies \text{dist } x1 \ x2 < e1 + e2$
<proof>

lemma *dist-triangle-half-l*: $\text{dist } x1 \ y < e / 2 \implies \text{dist } x2 \ y < e / 2 \implies \text{dist } x1 \ x2 < e$
<proof>

lemma *dist-triangle-half-r*: $\text{dist } y \ x1 < e / 2 \implies \text{dist } y \ x2 < e / 2 \implies \text{dist } x1 \ x2 < e$
<proof>

lemma *dist-triangle-third*:
assumes $\text{dist } x1 \ x2 < e/3 \ \text{dist } x2 \ x3 < e/3 \ \text{dist } x3 \ x4 < e/3$
shows $\text{dist } x1 \ x4 < e$
<proof>

subclass *uniform-space*
<proof>

lemma *open-dist*: $\text{open } S \longleftrightarrow (\forall x \in S. \exists e > 0. \forall y. \text{dist } y \ x < e \longrightarrow y \in S)$
<proof>

lemma *open-ball*: $\text{open } \{y. \text{dist } x \ y < d\}$
<proof>

subclass *first-countable-topology*
<proof>

end

instance *metric-space* \subseteq *t2-space*
<proof>

Every normed vector space is a metric space.

instance *real-normed-vector* $<$ *metric-space*
<proof>

107.7 Class instances for real numbers

instantiation *real* :: *real-normed-field*
begin

definition *dist-real-def*: $\text{dist } x \ y = |x - y|$

definition *uniformity-real-def* [code del]:

(*uniformity* :: (*real* × *real*) *filter*) = (*INF* *e* ∈ {0 <..*e*}. *principal* {(*x*, *y*). *dist* *x* *y* < *e*})

definition *open-real-def* [code del]:

open (*U* :: *real set*) \longleftrightarrow ($\forall x \in U$. *eventually* ($\lambda(x', y)$. $x' = x \longrightarrow y \in U$) *uniformity*)

definition *real-norm-def* [*simp*]: *norm* *r* = |*r*|

instance

⟨*proof*⟩

end

declare *uniformity-Abort*[**where** '*a=real*, *code*]

lemma *dist-of-real* [*simp*]: *dist* (*of-real* *x* :: '*a*) (*of-real* *y*) = *dist* *x* *y*

for *a* :: '*a::real-normed-div-algebra*

⟨*proof*⟩

declare [[*code abort*: *open* :: *real set* \Rightarrow *bool*]]

instance *real* :: *linorder-topology*

⟨*proof*⟩

instance *real* :: *linear-continuum-topology* ⟨*proof*⟩

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*

⟨*proof*⟩

107.8 Extra type constraints

Only allow *open* in class *topological-space*.

⟨*ML*⟩

Only allow *uniformity* in class *uniform-space*.

⟨*ML*⟩

Only allow *dist* in class *metric-space*.

⟨*ML*⟩

Only allow *norm* in class *real-normed-vector*.

$\langle ML \rangle$

107.9 Sign function

lemma *norm-sgn*: $\text{norm } (\text{sgn } x) = (\text{if } x = 0 \text{ then } 0 \text{ else } 1)$
for $x :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *sgn-zero* [simp]: $\text{sgn } (0 :: 'a::\text{real-normed-vector}) = 0$
 $\langle \text{proof} \rangle$

lemma *sgn-zero-iff*: $\text{sgn } x = 0 \longleftrightarrow x = 0$
for $x :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *sgn-minus*: $\text{sgn } (-x) = -\text{sgn } x$
for $x :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *sgn-scaleR*: $\text{sgn } (\text{scaleR } r \ x) = \text{scaleR } (\text{sgn } r) (\text{sgn } x)$
for $x :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *sgn-one* [simp]: $\text{sgn } (1 :: 'a::\text{real-normed-algebra-1}) = 1$
 $\langle \text{proof} \rangle$

lemma *sgn-of-real*: $\text{sgn } (\text{of-real } r :: 'a::\text{real-normed-algebra-1}) = \text{of-real } (\text{sgn } r)$
 $\langle \text{proof} \rangle$

lemma *sgn-mult*: $\text{sgn } (x * y) = \text{sgn } x * \text{sgn } y$
for $x \ y :: 'a::\text{real-normed-div-algebra}$
 $\langle \text{proof} \rangle$

hide-fact (open) *sgn-mult*

lemma *real-sgn-eq*: $\text{sgn } x = x / |x|$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *zero-le-sgn-iff* [simp]: $0 \leq \text{sgn } x \longleftrightarrow 0 \leq x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *sgn-le-0-iff* [simp]: $\text{sgn } x \leq 0 \longleftrightarrow x \leq 0$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *norm-conv-dist*: $\text{norm } x = \text{dist } x \ 0$

$\langle \text{proof} \rangle$

declare *norm-conv-dist* [*symmetric*, *simp*]

lemma *dist-0-norm* [*simp*]: $\text{dist } 0 \ x = \text{norm } x$
for $x :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *dist-diff* [*simp*]: $\text{dist } a \ (a - b) = \text{norm } b \ \text{dist } (a - b) \ a = \text{norm } b$
 $\langle \text{proof} \rangle$

lemma *dist-of-int*: $\text{dist } (\text{of-int } m) \ (\text{of-int } n :: 'a :: \text{real-normed-algebra-1}) = \text{of-int } |m - n|$
 $\langle \text{proof} \rangle$

lemma *dist-of-nat*:
 $\text{dist } (\text{of-nat } m) \ (\text{of-nat } n :: 'a :: \text{real-normed-algebra-1}) = \text{of-int } |\text{int } m - \text{int } n|$
 $\langle \text{proof} \rangle$

107.10 Bounded Linear and Bilinear Operators

lemma *linearI*: *linear* f
if $\bigwedge b1 \ b2. f \ (b1 + b2) = f \ b1 + f \ b2$
 $\bigwedge r \ b. f \ (r *_R b) = r *_R f \ b$
 $\langle \text{proof} \rangle$

lemma *linear-iff*:
 $\text{linear } f \longleftrightarrow (\forall x \ y. f \ (x + y) = f \ x + f \ y) \wedge (\forall c \ x. f \ (c *_R x) = c *_R f \ x)$
(is $\text{linear } f \longleftrightarrow ?rhs$
 $\langle \text{proof} \rangle$

lemma *linear-of-real* [*simp*]: *linear of-real*
 $\langle \text{proof} \rangle$

lemmas *linear-scaleR-left* = *linear-scale-left*
lemmas *linear-imp-scaleR* = *linear-imp-scale*

corollary *real-linearD*:
fixes $f :: \text{real} \Rightarrow \text{real}$
assumes *linear* f **obtains** c **where** $f = (*) \ c$
 $\langle \text{proof} \rangle$

lemma *linear-times-of-real*: *linear* $(\lambda x. a *_R x)$
 $\langle \text{proof} \rangle$

locale *bounded-linear* = *linear* f **for** $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-vector}$
 $+$
assumes *bounded*: $\exists K. \forall x. \text{norm } (f \ x) \leq \text{norm } x * K$
begin

lemma *pos-bounded*: $\exists K > 0. \forall x. \text{norm } (f\ x) \leq \text{norm } x * K$
 $\langle \text{proof} \rangle$

lemma *nonneg-bounded*: $\exists K \geq 0. \forall x. \text{norm } (f\ x) \leq \text{norm } x * K$
 $\langle \text{proof} \rangle$

lemma *linear*: *linear* f
 $\langle \text{proof} \rangle$

end

lemma *bounded-linear-intro*:
 assumes $\bigwedge x\ y. f\ (x + y) = f\ x + f\ y$
 and $\bigwedge r\ x. f\ (\text{scaleR } r\ x) = \text{scaleR } r\ (f\ x)$
 and $\bigwedge x. \text{norm } (f\ x) \leq \text{norm } x * K$
 shows *bounded-linear* f
 $\langle \text{proof} \rangle$

locale *bounded-bilinear* =
 fixes *prod* :: $'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-vector} \Rightarrow 'c::\text{real-normed-vector}$
 (infixl $\langle ** \rangle$ 70)
 assumes *add-left*: $\text{prod } (a + a')\ b = \text{prod } a\ b + \text{prod } a'\ b$
 and *add-right*: $\text{prod } a\ (b + b') = \text{prod } a\ b + \text{prod } a\ b'$
 and *scaleR-left*: $\text{prod } (\text{scaleR } r\ a)\ b = \text{scaleR } r\ (\text{prod } a\ b)$
 and *scaleR-right*: $\text{prod } a\ (\text{scaleR } r\ b) = \text{scaleR } r\ (\text{prod } a\ b)$
 and *bounded*: $\exists K. \forall a\ b. \text{norm } (\text{prod } a\ b) \leq \text{norm } a * \text{norm } b * K$
begin

lemma *pos-bounded*: $\exists K > 0. \forall a\ b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K$
 $\langle \text{proof} \rangle$

lemma *nonneg-bounded*: $\exists K \geq 0. \forall a\ b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K$
 $\langle \text{proof} \rangle$

lemma *additive-right*: *additive* $(\lambda b. \text{prod } a\ b)$
 $\langle \text{proof} \rangle$

lemma *additive-left*: *additive* $(\lambda a. \text{prod } a\ b)$
 $\langle \text{proof} \rangle$

lemma *zero-left*: $\text{prod } 0\ b = 0$
 $\langle \text{proof} \rangle$

lemma *zero-right*: $\text{prod } a\ 0 = 0$
 $\langle \text{proof} \rangle$

lemma *minus-left*: $\text{prod } (-\ a)\ b = -\ \text{prod } a\ b$
 $\langle \text{proof} \rangle$

lemma *minus-right*: $\text{prod } a \ (-\ b) = -\ \text{prod } a \ b$
 $\langle \text{proof} \rangle$

lemma *diff-left*: $\text{prod } (a - a') \ b = \text{prod } a \ b - \text{prod } a' \ b$
 $\langle \text{proof} \rangle$

lemma *diff-right*: $\text{prod } a \ (b - b') = \text{prod } a \ b - \text{prod } a \ b'$
 $\langle \text{proof} \rangle$

lemma *sum-left*: $\text{prod } (\text{sum } g \ S) \ x = \text{sum } ((\lambda i. \text{prod } (g \ i) \ x)) \ S$
 $\langle \text{proof} \rangle$

lemma *sum-right*: $\text{prod } x \ (\text{sum } g \ S) = \text{sum } ((\lambda i. (\text{prod } x \ (g \ i)))) \ S$
 $\langle \text{proof} \rangle$

lemma *bounded-linear-left*: $\text{bounded-linear } (\lambda a. a \ ** \ b)$
 $\langle \text{proof} \rangle$

lemma *bounded-linear-right*: $\text{bounded-linear } (\lambda b. a \ ** \ b)$
 $\langle \text{proof} \rangle$

lemma *prod-diff-prod*: $(x \ ** \ y - a \ ** \ b) = (x - a) \ ** \ (y - b) + (x - a) \ ** \ b + a \ ** \ (y - b)$
 $\langle \text{proof} \rangle$

lemma *flip*: $\text{bounded-bilinear } (\lambda x \ y. y \ ** \ x)$
 $\langle \text{proof} \rangle$

lemma *comp1*:
assumes $\text{bounded-linear } g$
shows $\text{bounded-bilinear } (\lambda x. (**) (g \ x))$
 $\langle \text{proof} \rangle$

lemma *comp*: $\text{bounded-linear } f \implies \text{bounded-linear } g \implies \text{bounded-bilinear } (\lambda x \ y. f \ x \ ** \ g \ y)$
 $\langle \text{proof} \rangle$

end

lemma *bounded-linear-ident[simp]*: $\text{bounded-linear } (\lambda x. x)$
 $\langle \text{proof} \rangle$

lemma *bounded-linear-zero[simp]*: $\text{bounded-linear } (\lambda x. 0)$
 $\langle \text{proof} \rangle$

lemma *bounded-linear-add*:
assumes $\text{bounded-linear } f$

and *bounded-linear* g
shows *bounded-linear* $(\lambda x. f\ x + g\ x)$
 $\langle proof \rangle$

lemma *bounded-linear-minus*:
assumes *bounded-linear* f
shows *bounded-linear* $(\lambda x. - f\ x)$
 $\langle proof \rangle$

lemma *bounded-linear-sub*: *bounded-linear* $f \implies$ *bounded-linear* $g \implies$ *bounded-linear*
 $(\lambda x. f\ x - g\ x)$
 $\langle proof \rangle$

lemma *bounded-linear-sum*:
fixes $f :: 'i \Rightarrow 'a::real-normed-vector \Rightarrow 'b::real-normed-vector$
shows $(\bigwedge i. i \in I \implies \text{bounded-linear } (f\ i)) \implies \text{bounded-linear } (\lambda x. \sum_{i \in I}. f\ i\ x)$
 $\langle proof \rangle$

lemma *bounded-linear-compose*:
assumes *bounded-linear* f
and *bounded-linear* g
shows *bounded-linear* $(\lambda x. f\ (g\ x))$
 $\langle proof \rangle$

lemma *bounded-bilinear-mult*: *bounded-bilinear* $((*) :: 'a \Rightarrow 'a \Rightarrow 'a::real-normed-algebra)$
 $\langle proof \rangle$

lemma *bounded-linear-mult-left*: *bounded-linear* $(\lambda x::'a::real-normed-algebra. x *$
 $y)$
 $\langle proof \rangle$

lemma *bounded-linear-mult-right*: *bounded-linear* $(\lambda y::'a::real-normed-algebra. x *$
 $y)$
 $\langle proof \rangle$

lemmas *bounded-linear-mult-const* =
bounded-linear-mult-left [THEN *bounded-linear-compose*]

lemmas *bounded-linear-const-mult* =
bounded-linear-mult-right [THEN *bounded-linear-compose*]

lemma *bounded-linear-divide*: *bounded-linear* $(\lambda x. x / y)$
for $y :: 'a::real-normed-field$
 $\langle proof \rangle$

lemma *bounded-bilinear-scaleR*: *bounded-bilinear* *scaleR*
 $\langle proof \rangle$

lemma *bounded-linear-scaleR-left*: *bounded-linear* $(\lambda r. \text{scaleR } r \ x)$
 $\langle \text{proof} \rangle$

lemma *bounded-linear-scaleR-right*: *bounded-linear* $(\lambda x. \text{scaleR } r \ x)$
 $\langle \text{proof} \rangle$

lemmas *bounded-linear-scaleR-const* =
bounded-linear-scaleR-left [THEN *bounded-linear-compose*]

lemmas *bounded-linear-const-scaleR* =
bounded-linear-scaleR-right [THEN *bounded-linear-compose*]

lemma *bounded-linear-of-real*: *bounded-linear* $(\lambda r. \text{of-real } r)$
 $\langle \text{proof} \rangle$

lemma *real-bounded-linear*: *bounded-linear* $f \longleftrightarrow (\exists c::\text{real}. f = (\lambda x. x * c))$
for $f :: \text{real} \Rightarrow \text{real}$
 $\langle \text{proof} \rangle$

instance *real-normed-algebra-1* \subseteq *perfect-space*
 $\langle \text{proof} \rangle$

107.11 Filters and Limits on Metric Space

lemma (**in** *metric-space*) *nhds-metric*: *nhds* $x = (\text{INF } e \in \{0 < ..\}. \text{principal } \{y. \text{dist } y \ x < e\})$
 $\langle \text{proof} \rangle$

lemma *tendsto-iff-uniformity*:

— More general analogus of *tendsto-iff* below. Applies to all uniform spaces, not just metric ones.

fixes $l :: \langle 'b :: \text{uniform-space} \rangle$
shows $\langle (f \longrightarrow l) \ F \longleftrightarrow (\forall E. \text{eventually } E \ \text{uniformity} \longrightarrow (\forall_F x \text{ in } F. E \ (f \ x, l))) \rangle$
 $\langle \text{proof} \rangle$

lemma (**in** *metric-space*) *tendsto-iff*: $(f \longrightarrow l) \ F \longleftrightarrow (\forall e > 0. \text{eventually } (\lambda x. \text{dist } (f \ x) \ l < e) \ F)$
 $\langle \text{proof} \rangle$

lemma *tendsto-dist-iff*:

$((f \longrightarrow l) \ F) \longleftrightarrow (((\lambda x. \text{dist } (f \ x) \ l) \longrightarrow 0) \ F)$
 $\langle \text{proof} \rangle$

lemma (**in** *metric-space*) *tendstoI* [intro?]:

$(\bigwedge e. 0 < e \Longrightarrow \text{eventually } (\lambda x. \text{dist } (f \ x) \ l < e) \ F) \Longrightarrow (f \longrightarrow l) \ F$
 $\langle \text{proof} \rangle$

lemma (in *metric-space*) *tendstoD*: $(f \longrightarrow l) F \implies 0 < e \implies \text{eventually } (\lambda x. \text{dist } (f x) l < e) F$
 ⟨proof⟩

lemma (in *metric-space*) *eventually-nhds-metric*:
 $\text{eventually } P \text{ (nhds } a) \longleftrightarrow (\exists d > 0. \forall x. \text{dist } x a < d \longrightarrow P x)$
 ⟨proof⟩

lemma *eventually-at*: $\text{eventually } P \text{ (at } a \text{ within } S) \longleftrightarrow (\exists d > 0. \forall x \in S. x \neq a \wedge \text{dist } x a < d \longrightarrow P x)$
for $a :: 'a :: \text{metric-space}$
 ⟨proof⟩

lemma *frequently-at*: $\text{frequently } P \text{ (at } a \text{ within } S) \longleftrightarrow (\forall d > 0. \exists x \in S. x \neq a \wedge \text{dist } x a < d \wedge P x)$
for $a :: 'a :: \text{metric-space}$
 ⟨proof⟩

lemma *eventually-at-le*: $\text{eventually } P \text{ (at } a \text{ within } S) \longleftrightarrow (\exists d > 0. \forall x \in S. x \neq a \wedge \text{dist } x a \leq d \longrightarrow P x)$
for $a :: 'a :: \text{metric-space}$
 ⟨proof⟩

lemma *eventually-at-left-real*: $a > (b :: \text{real}) \implies \text{eventually } (\lambda x. x \in \{b <..<a\}) \text{ (at-left } a)$
 ⟨proof⟩

lemma *eventually-at-right-real*: $a < (b :: \text{real}) \implies \text{eventually } (\lambda x. x \in \{a <..
 ⟨proof⟩$

lemma *metric-tendsto-imp-tendsto*:
fixes $a :: 'a :: \text{metric-space}$
and $b :: 'b :: \text{metric-space}$
assumes $f: (f \longrightarrow a) F$
and $le: \text{eventually } (\lambda x. \text{dist } (g x) b \leq \text{dist } (f x) a) F$
shows $(g \longrightarrow b) F$
 ⟨proof⟩

lemma *filterlim-real-sequentially*: $\text{LIM } x \text{ sequentially. real } x :> \text{at-top}$
 ⟨proof⟩

lemma *filterlim-nat-sequentially*: $\text{filterlim nat sequentially at-top}$
 ⟨proof⟩

lemma *filterlim-floor-sequentially*: $\text{filterlim floor at-top at-top}$
 ⟨proof⟩

lemma *filterlim-sequentially-iff-filterlim-real*:

filterlim f sequentially F \longleftrightarrow *filterlim* ($\lambda x. \text{real } (f x)$) *at-top F* (**is** ?lhs = ?rhs)
 <proof>

107.11.1 Limits of Sequences

lemma *lim-sequentially*: $X \longrightarrow L \longleftrightarrow (\forall r > 0. \exists no. \forall n \geq no. \text{dist } (X n) L < r)$
for $L :: 'a::\text{metric-space}$
 <proof>

lemmas *LIMSEQ-def* = *lim-sequentially*

lemma *LIMSEQ-iff-nz*: $X \longrightarrow L \longleftrightarrow (\forall r > 0. \exists no > 0. \forall n \geq no. \text{dist } (X n) L < r)$
for $L :: 'a::\text{metric-space}$
 <proof>

lemma *metric-LIMSEQ-I*: $(\bigwedge r. 0 < r \implies \exists no. \forall n \geq no. \text{dist } (X n) L < r) \implies X \longrightarrow L$
for $L :: 'a::\text{metric-space}$
 <proof>

lemma *metric-LIMSEQ-D*: $X \longrightarrow L \implies 0 < r \implies \exists no. \forall n \geq no. \text{dist } (X n) L < r$
for $L :: 'a::\text{metric-space}$
 <proof>

lemma *LIMSEQ-norm-0*:
assumes $\bigwedge n::\text{nat. norm } (f n) < 1 / \text{real } (\text{Suc } n)$
shows $f \longrightarrow 0$
 <proof>

107.11.2 Limits of Functions

lemma *LIM-def*: $f -a \rightarrow L \longleftrightarrow (\forall r > 0. \exists s > 0. \forall x. x \neq a \wedge \text{dist } x a < s \longrightarrow \text{dist } (f x) L < r)$
for $a :: 'a::\text{metric-space}$ **and** $L :: 'b::\text{metric-space}$
 <proof>

lemma *metric-LIM-I*:
 $(\bigwedge r. 0 < r \implies \exists s > 0. \forall x. x \neq a \wedge \text{dist } x a < s \longrightarrow \text{dist } (f x) L < r) \implies f -a \rightarrow L$
for $a :: 'a::\text{metric-space}$ **and** $L :: 'b::\text{metric-space}$
 <proof>

lemma *metric-LIM-D*: $f -a \rightarrow L \implies 0 < r \implies \exists s > 0. \forall x. x \neq a \wedge \text{dist } x a < s \longrightarrow \text{dist } (f x) L < r$
for $a :: 'a::\text{metric-space}$ **and** $L :: 'b::\text{metric-space}$
 <proof>

lemma *metric-LIM-imp-LIM*:

fixes $l :: 'a::\text{metric-space}$
and $m :: 'b::\text{metric-space}$
assumes $f: f -a \rightarrow l$
and $le: \bigwedge x. x \neq a \implies \text{dist } (g \ x) \ m \leq \text{dist } (f \ x) \ l$
shows $g -a \rightarrow m$
 $\langle \text{proof} \rangle$

lemma *metric-LIM-equal2*:

fixes $a :: 'a::\text{metric-space}$
assumes $g -a \rightarrow l \ 0 < R$
and $\bigwedge x. x \neq a \implies \text{dist } x \ a < R \implies f \ x = g \ x$
shows $f -a \rightarrow l$
 $\langle \text{proof} \rangle$

lemma *metric-LIM-compose2*:

fixes $a :: 'a::\text{metric-space}$
assumes $f: f -a \rightarrow b$
and $g: g -b \rightarrow c$
and $\text{inj}: \exists d > 0. \forall x. x \neq a \wedge \text{dist } x \ a < d \longrightarrow f \ x \neq b$
shows $(\lambda x. g \ (f \ x)) -a \rightarrow c$
 $\langle \text{proof} \rangle$

lemma *metric-isCont-LIM-compose2*:

fixes $f :: 'a :: \text{metric-space} \Rightarrow -$
assumes $f \ [\text{unfolded isCont-def}]: \text{isCont } f \ a$
and $g: g -f \ a \rightarrow l$
and $\text{inj}: \exists d > 0. \forall x. x \neq a \wedge \text{dist } x \ a < d \longrightarrow f \ x \neq f \ a$
shows $(\lambda x. g \ (f \ x)) -a \rightarrow l$
 $\langle \text{proof} \rangle$

107.12 Complete metric spaces

107.13 Cauchy sequences

lemma (**in** *metric-space*) *Cauchy-def*: $\text{Cauchy } X = (\forall e > 0. \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X \ m) \ (X \ n) < e)$
 $\langle \text{proof} \rangle$

lemma (**in** *metric-space*) *Cauchy-altdef*: $\text{Cauchy } f \longleftrightarrow (\forall e > 0. \exists M. \forall m \geq M. \forall n > m. \text{dist } (f \ m) \ (f \ n) < e)$
(is ?lhs \longleftrightarrow ?rhs)
 $\langle \text{proof} \rangle$

lemma (**in** *metric-space*) *Cauchy-altdef2*: $\text{Cauchy } s \longleftrightarrow (\forall e > 0. \exists N :: \text{nat}. \forall n \geq N. \text{dist } (s \ n) \ (s \ N) < e)$ **(is ?lhs = ?rhs)**
 $\langle \text{proof} \rangle$

lemma (**in** *metric-space*) *metric-CauchyI*:

$(\bigwedge e. 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X \ m) \ (X \ n) < e) \implies \text{Cauchy } X$

$\langle \text{proof} \rangle$

lemma (in *metric-space*) *CauchyI*:

$(\bigwedge e. 0 < e \implies \exists M. \forall m \geq M. \forall n > m. \text{dist } (X\ m) (X\ n) < e) \implies \text{Cauchy } X$
 $\langle \text{proof} \rangle$

lemma (in *metric-space*) *metric-CauchyD*:

$\text{Cauchy } X \implies 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X\ m) (X\ n) < e$
 $\langle \text{proof} \rangle$

lemma (in *metric-space*) *metric-Cauchy-iff2*:

$\text{Cauchy } X = (\forall j. (\exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X\ m) (X\ n) < \text{inverse}(\text{real } (\text{Suc } j))))$
 $\langle \text{proof} \rangle$

lemma *Cauchy-iff2*: $\text{Cauchy } X \longleftrightarrow (\forall j. (\exists M. \forall m \geq M. \forall n \geq M. |X\ m - X\ n| < \text{inverse}(\text{real } (\text{Suc } j))))$
 $\langle \text{proof} \rangle$

lemma *lim-1-over-n [tendsto-intros]*: $((\lambda n. 1 / \text{of_nat } n) \longrightarrow (0 :: 'a :: \text{real-normed-field}))$
sequentially
 $\langle \text{proof} \rangle$

lemma (in *metric-space*) *complete-def*:

shows $\text{complete } S = (\forall f. (\forall n. f\ n \in S) \wedge \text{Cauchy } f \longrightarrow (\exists l \in S. f \longrightarrow l))$
 $\langle \text{proof} \rangle$

apparently unused

lemma (in *metric-space*) *totally-bounded-metric*:

$\text{totally-bounded } S \longleftrightarrow (\forall e > 0. \exists k. \text{finite } k \wedge S \subseteq (\bigcup x \in k. \{y. \text{dist } x\ y < e\}))$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *cauchy-filter-metric*:

fixes $F :: 'a :: \{\text{uniformity-dist}, \text{uniform-space}\} \text{ filter}$

shows $\text{cauchy-filter } F \longleftrightarrow (\forall e. e > 0 \longrightarrow (\exists P. \text{eventually } P\ F \wedge (\forall x\ y. P\ x \wedge P\ y \longrightarrow \text{dist } x\ y < e)))$
 $\langle \text{proof} \rangle$

lemma *cauchy-filter-metric-filtermap*:

fixes $f :: 'a \Rightarrow 'b :: \{\text{uniformity-dist}, \text{uniform-space}\}$

shows $\text{cauchy-filter } (\text{filtermap } f\ F) \longleftrightarrow (\forall e. e > 0 \longrightarrow (\exists P. \text{eventually } P\ F \wedge (\forall x\ y. P\ x \wedge P\ y \longrightarrow \text{dist } (f\ x) (f\ y) < e)))$
 $\langle \text{proof} \rangle$

⟨ML⟩

107.13.1 Cauchy Sequences are Convergent

```
class complete-space = metric-space +
  assumes Cauchy-convergent: Cauchy X  $\implies$  convergent X
```

```
lemma Cauchy-convergent-iff: Cauchy X  $\longleftrightarrow$  convergent X
  for X :: nat  $\Rightarrow$  'a::complete-space
  ⟨proof⟩
```

To prove that a Cauchy sequence converges, it suffices to show that a subsequence converges.

```
lemma Cauchy-converges-subseq:
  fixes u::nat  $\Rightarrow$  'a::metric-space
  assumes Cauchy u
    strict-mono r
    (u  $\circ$  r)  $\longrightarrow$  l
  shows u  $\longrightarrow$  l
  ⟨proof⟩
```

107.14 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/numseq/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 :: nat  $\Rightarrow$  real
  assumes inc:  $\bigwedge n. f n \leq f (Suc n)$ 
    and bdd:  $\bigwedge n. f n \leq l$ 
    and en:  $\bigwedge e. 0 < e \implies \exists n. l \leq f n + e$ 
  shows f  $\longrightarrow$  l
  ⟨proof⟩
```

```
lemma real-Cauchy-convergent:
  fixes X :: nat  $\Rightarrow$  real
  assumes X: Cauchy X
  shows convergent X
  ⟨proof⟩
```

```
instance real :: complete-space
  ⟨proof⟩
```

```
class banach = real-normed-vector + complete-space
```

```
instance real :: banach ⟨proof⟩
```

lemma *tendsto-at-topI-sequentially*:
fixes $f :: \text{real} \Rightarrow 'b::\text{first-countable-topology}$
assumes $*$: $\bigwedge X. \text{filterlim } X \text{ at-top sequentially} \implies (\lambda n. f (X n)) \longrightarrow y$
shows $(f \longrightarrow y) \text{ at-top}$
 $\langle \text{proof} \rangle$

lemma *tendsto-at-topI-sequentially-real*:
fixes $f :: \text{real} \Rightarrow \text{real}$
assumes *mono*: $\text{mono } f$
and *limseq*: $(\lambda n. f (\text{real } n)) \longrightarrow y$
shows $(f \longrightarrow y) \text{ at-top}$
 $\langle \text{proof} \rangle$

end

108 Limits on Real Vector Spaces

theory *Limits*
imports *Real-Vector-Spaces*
begin

lemma *range-mult [simp]*:
fixes $a::\text{real}$ **shows** $\text{range } ((*) a) = (\text{if } a=0 \text{ then } \{0\} \text{ else } \text{UNIV})$
 $\langle \text{proof} \rangle$

108.1 Filter going to infinity norm

definition *at-infinity* :: $'a::\text{real-normed-vector}$ *filter*
where $\text{at-infinity} = (\text{INF } r. \text{principal } \{x. r \leq \text{norm } x\})$

lemma *eventually-at-infinity*: $\text{eventually } P \text{ at-infinity} \longleftrightarrow (\exists b. \forall x. b \leq \text{norm } x \longrightarrow P x)$
 $\langle \text{proof} \rangle$

lemma *eventually-at-infinityI*:
fixes $P::'a::\text{real-normed-vector} \Rightarrow \text{bool}$
assumes $\bigwedge x. c \leq \text{norm } x \implies P x$
shows $\text{eventually } P \text{ at-infinity}$
 $\langle \text{proof} \rangle$

corollary *eventually-at-infinity-pos*:
 $\text{eventually } p \text{ at-infinity} \longleftrightarrow (\exists b. 0 < b \wedge (\forall x. \text{norm } x \geq b \longrightarrow p x))$
 $\langle \text{proof} \rangle$

lemma *at-infinity-eq-at-top-bot*: $(\text{at-infinity} :: \text{real filter}) = \text{sup at-top at-bot}$
 $\langle \text{proof} \rangle$

lemma *at-top-le-at-infinity*: $\text{at-top} \leq (\text{at-infinity} :: \text{real filter})$

<proof>

lemma *at-bot-le-at-infinity*: $at_bot \leq (at_infinity :: real\ filter)$
<proof>

lemma *filterlim-at-top-imp-at-infinity*: $filterlim\ f\ at_top\ F \implies filterlim\ f\ at_infinity\ F$
for $f :: - \Rightarrow real$
<proof>

lemma *filterlim-real-at-infinity-sequentially*: $filterlim\ real\ at_infinity\ sequentially$
<proof>

lemma *lim-infinity-imp-sequentially*: $(f \longrightarrow l)\ at_infinity \implies ((\lambda n. f(n)) \longrightarrow l)\ sequentially$
<proof>

108.1.1 Boundedness

definition $Bfun :: ('a \Rightarrow 'b::metric-space) \Rightarrow 'a\ filter \Rightarrow bool$
where *Bfun-metric-def*: $Bfun\ f\ F = (\exists y. \exists K>0. eventually\ (\lambda x. dist\ (f\ x)\ y \leq K)\ F)$

abbreviation $Bseq :: (nat \Rightarrow 'a::metric-space) \Rightarrow bool$
where $Bseq\ X \equiv Bfun\ X\ sequentially$

lemma *Bseq-conv-Bfun*: $Bseq\ X \longleftrightarrow Bfun\ X\ sequentially$ *<proof>*

lemma *Bseq-ignore-initial-segment*: $Bseq\ X \implies Bseq\ (\lambda n. X\ (n + k))$
<proof>

lemma *Bseq-offset*: $Bseq\ (\lambda n. X\ (n + k)) \implies Bseq\ X$
<proof>

lemma *Bfun-def*: $Bfun\ f\ F \longleftrightarrow (\exists K>0. eventually\ (\lambda x. norm\ (f\ x) \leq K)\ F)$
<proof>

lemma *BfunI*:
assumes $K: eventually\ (\lambda x. norm\ (f\ x) \leq K)\ F$
shows $Bfun\ f\ F$
<proof>

lemma *BfunE*:
assumes $Bfun\ f\ F$
obtains B **where** $0 < B$ **and** $eventually\ (\lambda x. norm\ (f\ x) \leq B)\ F$
<proof>

lemma *Cauchy-Bseq*:
assumes *Cauchy* X **shows** $Bseq\ X$

$\langle \text{proof} \rangle$

108.1.2 Bounded Sequences

lemma *BseqI'*: $(\bigwedge n. \text{norm } (X\ n) \leq K) \implies \text{Bseq } X$
 $\langle \text{proof} \rangle$

lemma *Bseq-def*: $\text{Bseq } X \longleftrightarrow (\exists K > 0. \forall n. \text{norm } (X\ n) \leq K)$
 $\langle \text{proof} \rangle$

lemma *BseqE*: $\text{Bseq } X \implies (\bigwedge K. 0 < K \implies \forall n. \text{norm } (X\ n) \leq K \implies Q) \implies Q$
 $\langle \text{proof} \rangle$

lemma *BseqD*: $\text{Bseq } X \implies \exists K. 0 < K \wedge (\forall n. \text{norm } (X\ n) \leq K)$
 $\langle \text{proof} \rangle$

lemma *BseqI*: $0 < K \implies \forall n. \text{norm } (X\ n) \leq K \implies \text{Bseq } X$
 $\langle \text{proof} \rangle$

lemma *Bseq-bdd-above*: $\text{Bseq } X \implies \text{bdd-above } (\text{range } X)$
for $X :: \text{nat} \Rightarrow \text{real}$
 $\langle \text{proof} \rangle$

lemma *Bseq-bdd-above'*: $\text{Bseq } X \implies \text{bdd-above } (\text{range } (\lambda n. \text{norm } (X\ n)))$
for $X :: \text{nat} \Rightarrow 'a :: \text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *Bseq-bdd-below*: $\text{Bseq } X \implies \text{bdd-below } (\text{range } X)$
for $X :: \text{nat} \Rightarrow \text{real}$
 $\langle \text{proof} \rangle$

lemma *Bseq-eventually-mono*:
assumes *eventually* $(\lambda n. \text{norm } (f\ n) \leq \text{norm } (g\ n))$ *sequentially* $\text{Bseq } g$
shows $\text{Bseq } f$
 $\langle \text{proof} \rangle$

lemma *lemma-NBseq-def*: $(\exists K > 0. \forall n. \text{norm } (X\ n) \leq K) \longleftrightarrow (\exists N. \forall n. \text{norm } (X\ n) \leq \text{real}(\text{Suc } N))$
 $\langle \text{proof} \rangle$

Alternative definition for *Bseq*.

lemma *Bseq-iff*: $\text{Bseq } X \longleftrightarrow (\exists N. \forall n. \text{norm } (X\ n) \leq \text{real}(\text{Suc } N))$
 $\langle \text{proof} \rangle$

lemma *lemma-NBseq-def2*: $(\exists K > 0. \forall n. \text{norm } (X\ n) \leq K) = (\exists N. \forall n. \text{norm } (X\ n) < \text{real}(\text{Suc } N))$
 $\langle \text{proof} \rangle$

Yet another definition for *Bseq*.

lemma *Bseq-iff1a*: $Bseq\ X \longleftrightarrow (\exists N. \forall n. norm\ (X\ n) < real\ (Suc\ N))$
 $\langle proof \rangle$

108.1.3 A Few More Equivalence Theorems for Boundedness

Alternative formulation for boundedness.

lemma *Bseq-iff2*: $Bseq\ X \longleftrightarrow (\exists k > 0. \exists x. \forall n. norm\ (X\ n + -\ x) \leq k)$
 $\langle proof \rangle$

Alternative formulation for boundedness.

lemma *Bseq-iff3*: $Bseq\ X \longleftrightarrow (\exists k > 0. \exists N. \forall n. norm\ (X\ n + -\ X\ N) \leq k)$
 $(is\ ?P \longleftrightarrow ?Q)$
 $\langle proof \rangle$

108.1.4 Upper Bounds and Lubs of Bounded Sequences

lemma *Bseq-minus-iff*: $Bseq\ (\lambda n. -\ (X\ n) :: 'a::real-normed-vector) \longleftrightarrow Bseq\ X$
 $\langle proof \rangle$

lemma *Bseq-add*:
fixes $f :: nat \Rightarrow 'a::real-normed-vector$
assumes $Bseq\ f$
shows $Bseq\ (\lambda x. f\ x + c)$
 $\langle proof \rangle$

lemma *Bseq-add-iff*: $Bseq\ (\lambda x. f\ x + c) \longleftrightarrow Bseq\ f$
for $f :: nat \Rightarrow 'a::real-normed-vector$
 $\langle proof \rangle$

lemma *Bseq-mult*:
fixes $f\ g :: nat \Rightarrow 'a::real-normed-field$
assumes $Bseq\ f$ **and** $Bseq\ g$
shows $Bseq\ (\lambda x. f\ x * g\ x)$
 $\langle proof \rangle$

lemma *Bfun-const [simp]*: $Bfun\ (\lambda -. c)\ F$
 $\langle proof \rangle$

lemma *Bseq-cmult-iff*:
fixes $c :: 'a::real-normed-field$
assumes $c \neq 0$
shows $Bseq\ (\lambda x. c * f\ x) \longleftrightarrow Bseq\ f$
 $\langle proof \rangle$

lemma *Bseq-subseq*: $Bseq\ f \implies Bseq\ (\lambda x. f\ (g\ x))$
for $f :: nat \Rightarrow 'a::real-normed-vector$
 $\langle proof \rangle$

lemma *Bseq-Suc-iff*: $Bseq\ (\lambda n. f\ (Suc\ n)) \longleftrightarrow Bseq\ f$

for $f :: nat \Rightarrow 'a::real-normed-vector$
 $\langle proof \rangle$

lemma *increasing-Bseq-subseq-iff*:

assumes $\bigwedge x y. x \leq y \implies norm (f x :: 'a::real-normed-vector) \leq norm (f y)$
strict-mono g
shows $Bseq (\lambda x. f (g x)) \longleftrightarrow Bseq f$
 $\langle proof \rangle$

lemma *nonneg-incseq-Bseq-subseq-iff*:

fixes $f :: nat \Rightarrow real$
and $g :: nat \Rightarrow nat$
assumes $\bigwedge x. f x \geq 0$ *incseq f* *strict-mono g*
shows $Bseq (\lambda x. f (g x)) \longleftrightarrow Bseq f$
 $\langle proof \rangle$

lemma *Bseq-eq-bounded*: $range f \subseteq \{a..b\} \implies Bseq f$

for $a b :: real$
 $\langle proof \rangle$

lemma *incseq-bounded*: $incseq X \implies \forall i. X i \leq B \implies Bseq X$

for $B :: real$
 $\langle proof \rangle$

lemma *decseq-bounded*: $decseq X \implies \forall i. B \leq X i \implies Bseq X$

for $B :: real$
 $\langle proof \rangle$

108.1.5 Polynomial function extremal theorem, from HOL Light

lemma *polyfun-extremal-lemma*:

fixes $c :: nat \Rightarrow 'a::real-normed-div-algebra$
assumes $0 < e$
shows $\exists M. \forall z. M \leq norm(z) \longrightarrow norm (\sum_{i \leq n. c(i) * z^i} \leq e * norm(z)$
 $\wedge (Suc n)$
 $\langle proof \rangle$

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* $(\lambda z. norm (\sum_{i \leq n. c(i) * z^i} \geq B)$ *at-infinity*
 $\langle proof \rangle$

108.2 Convergence to Zero

definition $Zfun :: ('a \Rightarrow 'b::real-normed-vector) \Rightarrow 'a \text{ filter} \Rightarrow bool$

where $Zfun f F = (\forall r > 0. \text{eventually } (\lambda x. norm (f x) < r) F)$

lemma *ZfunI*: $(\bigwedge r. 0 < r \implies \text{eventually } (\lambda x. norm (f x) < r) F) \implies Zfun f F$

$\langle proof \rangle$

lemma *ZfunD*: $Zfun\ f\ F \implies 0 < r \implies eventually\ (\lambda x. norm\ (f\ x) < r)\ F$
 ⟨proof⟩

lemma *Zfun-ssubst*: $eventually\ (\lambda x. f\ x = g\ x)\ F \implies Zfun\ g\ F \implies Zfun\ f\ F$
 ⟨proof⟩

lemma *Zfun-zero*: $Zfun\ (\lambda x. 0)\ F$
 ⟨proof⟩

lemma *Zfun-norm-iff*: $Zfun\ (\lambda x. norm\ (f\ x))\ F = Zfun\ (\lambda x. f\ x)\ F$
 ⟨proof⟩

lemma *Zfun-imp-Zfun*:
 assumes $f: Zfun\ f\ F$
 and $g: eventually\ (\lambda x. norm\ (g\ x) \leq norm\ (f\ x) * K)\ F$
 shows $Zfun\ (\lambda x. g\ x)\ F$
 ⟨proof⟩

lemma *Zfun-le*: $Zfun\ g\ F \implies \forall x. norm\ (f\ x) \leq norm\ (g\ x) \implies Zfun\ f\ F$
 ⟨proof⟩

lemma *Zfun-add*:
 assumes $f: Zfun\ f\ F$
 and $g: Zfun\ g\ F$
 shows $Zfun\ (\lambda x. f\ x + g\ x)\ F$
 ⟨proof⟩

lemma *Zfun-minus*: $Zfun\ f\ F \implies Zfun\ (\lambda x. -\ f\ x)\ F$
 ⟨proof⟩

lemma *Zfun-diff*: $Zfun\ f\ F \implies Zfun\ g\ F \implies Zfun\ (\lambda x. f\ x - g\ x)\ F$
 ⟨proof⟩

lemma (in *bounded-linear*) *Zfun*:
 assumes $g: Zfun\ g\ F$
 shows $Zfun\ (\lambda x. f\ (g\ x))\ F$
 ⟨proof⟩

lemma (in *bounded-bilinear*) *Zfun*:
 assumes $f: Zfun\ f\ F$
 and $g: Zfun\ g\ F$
 shows $Zfun\ (\lambda x. f\ x ** g\ x)\ F$
 ⟨proof⟩

lemma (in *bounded-bilinear*) *Zfun-left*: $Zfun\ f\ F \implies Zfun\ (\lambda x. f\ x ** a)\ F$
 ⟨proof⟩

lemma (in *bounded-bilinear*) *Zfun-right*: $Zfun\ f\ F \implies Zfun\ (\lambda x. a ** f\ x)\ F$

$\langle \text{proof} \rangle$

lemmas $Zfun\text{-}mult = bounded\text{-}bilinear.Zfun \ [OF \ bounded\text{-}bilinear\text{-}mult]$

lemmas $Zfun\text{-}mult\text{-}right = bounded\text{-}bilinear.Zfun\text{-}right \ [OF \ bounded\text{-}bilinear\text{-}mult]$

lemmas $Zfun\text{-}mult\text{-}left = bounded\text{-}bilinear.Zfun\text{-}left \ [OF \ bounded\text{-}bilinear\text{-}mult]$

lemma $tendsto\text{-}Zfun\text{-}iff: (f \longrightarrow a) \ F = Zfun \ (\lambda x. f \ x - a) \ F$

$\langle \text{proof} \rangle$

lemma $tendsto\text{-}0\text{-}le:$

$(f \longrightarrow 0) \ F \implies eventually \ (\lambda x. norm \ (g \ x) \leq norm \ (f \ x) * K) \ F \implies (g \longrightarrow 0) \ F$

$\langle \text{proof} \rangle$

108.2.1 Distance and norms

lemma $tendsto\text{-}dist \ [tendsto\text{-}intros]:$

fixes $l \ m :: 'a :: metric\text{-}space$

assumes $f: (f \longrightarrow l) \ F$

and $g: (g \longrightarrow m) \ F$

shows $((\lambda x. dist \ (f \ x) \ (g \ x)) \longrightarrow dist \ l \ m) \ F$

$\langle \text{proof} \rangle$

lemma $continuous\text{-}dist[continuous\text{-}intros]:$

fixes $f \ g :: - \Rightarrow 'a :: metric\text{-}space$

shows $continuous \ F \ f \implies continuous \ F \ g \implies continuous \ F \ (\lambda x. dist \ (f \ x) \ (g \ x))$

$\langle \text{proof} \rangle$

lemma $continuous\text{-}on\text{-}dist[continuous\text{-}intros]:$

fixes $f \ g :: - \Rightarrow 'a :: metric\text{-}space$

shows $continuous\text{-}on \ s \ f \implies continuous\text{-}on \ s \ g \implies continuous\text{-}on \ s \ (\lambda x. dist \ (f \ x) \ (g \ x))$

$\langle \text{proof} \rangle$

lemma $continuous\text{-}at\text{-}dist: isCont \ (dist \ a) \ b$

$\langle \text{proof} \rangle$

lemma $tendsto\text{-}norm \ [tendsto\text{-}intros]: (f \longrightarrow a) \ F \implies ((\lambda x. norm \ (f \ x)) \longrightarrow norm \ a) \ F$

$\langle \text{proof} \rangle$

lemma $continuous\text{-}norm \ [continuous\text{-}intros]: continuous \ F \ f \implies continuous \ F \ (\lambda x. norm \ (f \ x))$

$\langle \text{proof} \rangle$

lemma $continuous\text{-}on\text{-}norm \ [continuous\text{-}intros]:$

$continuous\text{-}on \ s \ f \implies continuous\text{-}on \ s \ (\lambda x. norm \ (f \ x))$

$\langle \text{proof} \rangle$

lemma *continuous-on-norm-id* [*continuous-intros*]: *continuous-on S norm*
 ⟨*proof*⟩

lemma *tendsto-norm-zero*: $(f \longrightarrow 0) F \implies ((\lambda x. \text{norm } (f x)) \longrightarrow 0) F$
 ⟨*proof*⟩

lemma *tendsto-norm-zero-cancel*: $((\lambda x. \text{norm } (f x)) \longrightarrow 0) F \implies (f \longrightarrow 0) F$
 ⟨*proof*⟩

lemma *tendsto-norm-zero-iff*: $((\lambda x. \text{norm } (f x)) \longrightarrow 0) F \longleftrightarrow (f \longrightarrow 0) F$
 ⟨*proof*⟩

lemma *tendsto-rabs* [*tendsto-intros*]: $(f \longrightarrow l) F \implies ((\lambda x. |f x|) \longrightarrow |l|) F$
for $l :: \text{real}$
 ⟨*proof*⟩

lemma *continuous-rabs* [*continuous-intros*]:
continuous F f \implies continuous F ($\lambda x. |f x| :: \text{real}$)
 ⟨*proof*⟩

lemma *continuous-on-rabs* [*continuous-intros*]:
continuous-on s f \implies continuous-on s ($\lambda x. |f x| :: \text{real}$)
 ⟨*proof*⟩

lemma *tendsto-rabs-zero*: $(f \longrightarrow (0 :: \text{real})) F \implies ((\lambda x. |f x|) \longrightarrow 0) F$
 ⟨*proof*⟩

lemma *tendsto-rabs-zero-cancel*: $((\lambda x. |f x|) \longrightarrow (0 :: \text{real})) F \implies (f \longrightarrow 0) F$
 ⟨*proof*⟩

lemma *tendsto-rabs-zero-iff*: $((\lambda x. |f x|) \longrightarrow (0 :: \text{real})) F \longleftrightarrow (f \longrightarrow 0) F$
 ⟨*proof*⟩

108.3 Topological Monoid

class *topological-monoid-add* = *topological-space* + *monoid-add* +
assumes *tendsto-add-Pair*: *LIM x (nhds a \times_F nhds b). fst x + snd x \rightarrow nhds (a + b)*

class *topological-comm-monoid-add* = *topological-monoid-add* + *comm-monoid-add*

lemma *tendsto-add* [*tendsto-intros*]:
fixes $a b :: 'a :: \text{topological-monoid-add}$
shows $(f \longrightarrow a) F \implies (g \longrightarrow b) F \implies ((\lambda x. f x + g x) \longrightarrow a + b) F$
 ⟨*proof*⟩

lemma *continuous-add* [*continuous-intros*]:

fixes $f\ g :: - \Rightarrow 'b::\text{topological-monoid-add}$
shows $\text{continuous } F\ f \Longrightarrow \text{continuous } F\ g \Longrightarrow \text{continuous } F\ (\lambda x. f\ x + g\ x)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-add* [*continuous-intros*]:
fixes $f\ g :: - \Rightarrow 'b::\text{topological-monoid-add}$
shows $\text{continuous-on } s\ f \Longrightarrow \text{continuous-on } s\ g \Longrightarrow \text{continuous-on } s\ (\lambda x. f\ x + g\ x)$
 $\langle \text{proof} \rangle$

lemma *tendsto-add-zero*:
fixes $f\ g :: - \Rightarrow 'b::\text{topological-monoid-add}$
shows $(f \longrightarrow 0)\ F \Longrightarrow (g \longrightarrow 0)\ F \Longrightarrow ((\lambda x. f\ x + g\ x) \longrightarrow 0)\ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-sum* [*tendsto-intros*]:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c::\text{topological-comm-monoid-add}$
shows $(\bigwedge i. i \in I \Longrightarrow (f\ i \longrightarrow a\ i)\ F) \Longrightarrow ((\lambda x. \sum_{i \in I} f\ i\ x) \longrightarrow (\sum_{i \in I} a\ i))\ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-null-sum*:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c::\text{topological-comm-monoid-add}$
assumes $\bigwedge i. i \in I \Longrightarrow ((\lambda x. f\ x\ i) \longrightarrow 0)\ F$
shows $((\lambda i. \text{sum } (f\ i)\ I) \longrightarrow 0)\ F$
 $\langle \text{proof} \rangle$

lemma *continuous-sum* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'b::t2\text{-space} \Rightarrow 'c::\text{topological-comm-monoid-add}$
shows $(\bigwedge i. i \in I \Longrightarrow \text{continuous } F\ (f\ i)) \Longrightarrow \text{continuous } F\ (\lambda x. \sum_{i \in I} f\ i\ x)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-sum* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'b::\text{topological-space} \Rightarrow 'c::\text{topological-comm-monoid-add}$
shows $(\bigwedge i. i \in I \Longrightarrow \text{continuous-on } S\ (f\ i)) \Longrightarrow \text{continuous-on } S\ (\lambda x. \sum_{i \in I} f\ i\ x)$
 $\langle \text{proof} \rangle$

instance *nat* :: *topological-comm-monoid-add*
 $\langle \text{proof} \rangle$

instance *int* :: *topological-comm-monoid-add*
 $\langle \text{proof} \rangle$

108.3.1 Topological group

class *topological-group-add* = *topological-monoid-add* + *group-add* +
assumes *tendsto-uminus-nhds*: $(\text{uminus} \longrightarrow -\ a)\ (\text{nhds } a)$
begin

lemma *tendsto-minus* [*tendsto-intros*]: $(f \longrightarrow a) F \implies ((\lambda x. - f x) \longrightarrow - a) F$
<proof>

end

class *topological-ab-group-add* = *topological-group-add* + *ab-group-add*

instance *topological-ab-group-add* < *topological-comm-monoid-add* *<proof>*

lemma *continuous-minus* [*continuous-intros*]: *continuous* $F f \implies \text{continuous } F (\lambda x. - f x)$
for $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{topological-group-add}$
<proof>

lemma *continuous-on-minus* [*continuous-intros*]: *continuous-on* $s f \implies \text{continuous-on } s (\lambda x. - f x)$
for $f :: - \Rightarrow 'b::\text{topological-group-add}$
<proof>

lemma *tendsto-minus-cancel*: $((\lambda x. - f x) \longrightarrow - a) F \implies (f \longrightarrow a) F$
for $a :: 'a::\text{topological-group-add}$
<proof>

lemma *tendsto-minus-cancel-left*:
 $(f \longrightarrow - (y::\text{topological-group-add})) F \longleftrightarrow ((\lambda x. - f x) \longrightarrow y) F$
<proof>

lemma *tendsto-diff* [*tendsto-intros*]:
fixes $a b :: 'a::\text{topological-group-add}$
shows $(f \longrightarrow a) F \implies (g \longrightarrow b) F \implies ((\lambda x. f x - g x) \longrightarrow a - b) F$
<proof>

lemma *continuous-diff* [*continuous-intros*]:
fixes $f g :: 'a::t2\text{-space} \Rightarrow 'b::\text{topological-group-add}$
shows *continuous* $F f \implies \text{continuous } F g \implies \text{continuous } F (\lambda x. f x - g x)$
<proof>

lemma *continuous-on-diff* [*continuous-intros*]:
fixes $f g :: - \Rightarrow 'b::\text{topological-group-add}$
shows *continuous-on* $s f \implies \text{continuous-on } s g \implies \text{continuous-on } s (\lambda x. f x - g x)$
<proof>

lemma *continuous-on-op-minus*: *continuous-on* $(s::'a::\text{topological-group-add set}) ((-) x)$
<proof>

instance *real-normed-vector* < *topological-ab-group-add*
 ⟨*proof*⟩

lemmas *real-tendsto-sandwich* = *tendsto-sandwich*[**where** 'a=*real*]

108.3.2 Linear operators and multiplication

lemma *linear-times* [*simp*]: *linear* ($\lambda x. c * x$)
for $c :: 'a :: \text{real-algebra}$
 ⟨*proof*⟩

lemma (**in** *bounded-linear*) *tendsto*: $(g \longrightarrow a) F \implies ((\lambda x. f (g x)) \longrightarrow f a) F$
 ⟨*proof*⟩

lemma (**in** *bounded-linear*) *continuous*: *continuous* $F g \implies \text{continuous } F (\lambda x. f (g x))$
 ⟨*proof*⟩

lemma (**in** *bounded-linear*) *continuous-on*: *continuous-on* $s g \implies \text{continuous-on } s (\lambda x. f (g x))$
 ⟨*proof*⟩

lemma (**in** *bounded-linear*) *tendsto-zero*: $(g \longrightarrow 0) F \implies ((\lambda x. f (g x)) \longrightarrow 0) F$
 ⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *tendsto*:
 $(f \longrightarrow a) F \implies (g \longrightarrow b) F \implies ((\lambda x. f x ** g x) \longrightarrow a ** b) F$
 ⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *continuous*:
continuous $F f \implies \text{continuous } F g \implies \text{continuous } F (\lambda x. f x ** g x)$
 ⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *continuous-on*:
continuous-on $s f \implies \text{continuous-on } s g \implies \text{continuous-on } s (\lambda x. f x ** g x)$
 ⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *tendsto-zero*:
assumes $f: (f \longrightarrow 0) F$
and $g: (g \longrightarrow 0) F$
shows $((\lambda x. f x ** g x) \longrightarrow 0) F$
 ⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *tendsto-left-zero*:
 $(f \longrightarrow 0) F \implies ((\lambda x. f x ** c) \longrightarrow 0) F$
 ⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *tendsto-right-zero*:

$(f \longrightarrow 0) F \implies ((\lambda x. c ** f x) \longrightarrow 0) F$
 $\langle \text{proof} \rangle$

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*: *LIM* x (*nhds* $a \times_F \text{nhds } b$). *fst* $x * \text{snd } x > \text{nhds}$
 $(a * b)$

instance *real-normed-algebra* < *topological-semigroup-mult*
 $\langle \text{proof} \rangle$

lemma *tendsto-mult* [*tendsto-intros*]:
fixes $a b :: 'a :: \text{topological-semigroup-mult}$
shows $(f \longrightarrow a) F \implies (g \longrightarrow b) F \implies ((\lambda x. f x * g x) \longrightarrow a * b) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-mult-left*: $(f \longrightarrow l) F \implies ((\lambda x. c * (f x)) \longrightarrow c * l) F$
for $c :: 'a :: \text{topological-semigroup-mult}$
 $\langle \text{proof} \rangle$

lemma *tendsto-mult-right*: $(f \longrightarrow l) F \implies ((\lambda x. (f x) * c) \longrightarrow l * c) F$
for $c :: 'a :: \text{topological-semigroup-mult}$
 $\langle \text{proof} \rangle$

lemma *tendsto-mult-left-iff* [*simp*]:
 $c \neq 0 \implies \text{tendsto}(\lambda x. c * f x) (c * l) F \longleftrightarrow \text{tendsto } f l F$ **for** $c :: 'a :: \{\text{topological-semigroup-mult}, \text{field}\}$
 $\langle \text{proof} \rangle$

lemma *tendsto-mult-right-iff* [*simp*]:
 $c \neq 0 \implies \text{tendsto}(\lambda x. f x * c) (l * c) F \longleftrightarrow \text{tendsto } f l F$ **for** $c :: 'a :: \{\text{topological-semigroup-mult}, \text{field}\}$
 $\langle \text{proof} \rangle$

lemma *tendsto-zero-mult-left-iff* [*simp*]:
fixes $c :: 'a :: \{\text{topological-semigroup-mult}, \text{field}\}$ **assumes** $c \neq 0$ **shows** $(\lambda n. c * a$
 $n) \longrightarrow 0 \longleftrightarrow a \longrightarrow 0$
 $\langle \text{proof} \rangle$

lemma *tendsto-zero-mult-right-iff* [*simp*]:
fixes $c :: 'a :: \{\text{topological-semigroup-mult}, \text{field}\}$ **assumes** $c \neq 0$ **shows** $(\lambda n. a * c)$
 $\longrightarrow 0 \longleftrightarrow a \longrightarrow 0$
 $\langle \text{proof} \rangle$

lemma *tendsto-zero-divide-iff* [*simp*]:

fixes $c :: 'a :: \{\text{topological-semigroup-mult, field}\}$ **assumes** $c \neq 0$ **shows** $(\lambda n. a \ n / c) \longrightarrow 0 \iff a \longrightarrow 0$
 $\langle \text{proof} \rangle$

lemma *lim-const-over-n* [*tendsto-intros*]:

fixes $a :: 'a :: \text{real-normed-field}$
shows $(\lambda n. a \ / \ \text{of-nat } n) \longrightarrow 0$
 $\langle \text{proof} \rangle$

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* [*continuous-intros*] =
bounded-bilinear.continuous-on [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-zero* =
bounded-bilinear.tendsto-zero [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-left-zero* =
bounded-bilinear.tendsto-left-zero [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-right-zero* =
bounded-bilinear.tendsto-right-zero [*OF bounded-bilinear-mult*]

lemma *continuous-mult-left*:

fixes $c :: 'a :: \text{real-normed-algebra}$
shows $\text{continuous } F \ f \implies \text{continuous } F \ (\lambda x. c * f \ x)$
 $\langle \text{proof} \rangle$

lemma *continuous-mult-right*:

fixes $c :: 'a :: \text{real-normed-algebra}$
shows $\text{continuous } F \ f \implies \text{continuous } F \ (\lambda x. f \ x * c)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-mult-left*:

fixes $c :: 'a :: \text{real-normed-algebra}$

shows $\text{continuous-on } s \ f \implies \text{continuous-on } s \ (\lambda x. c * f \ x)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-mult-right*:
fixes $c :: 'a :: \text{real-normed-algebra}$
shows $\text{continuous-on } s \ f \implies \text{continuous-on } s \ (\lambda x. f \ x * c)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-mult-const* [simp]:
fixes $c :: 'a :: \text{real-normed-algebra}$
shows $\text{continuous-on } s \ ((*) \ c)$
 $\langle \text{proof} \rangle$

lemma *tendsto-divide-zero*:
fixes $c :: 'a :: \text{real-normed-field}$
shows $(f \longrightarrow 0) \ F \implies ((\lambda x. f \ x / c) \longrightarrow 0) \ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-power* [tendsto-intros]: $(f \longrightarrow a) \ F \implies ((\lambda x. f \ x ^ n) \longrightarrow a ^ n) \ F$
for $f :: 'a \Rightarrow 'b :: \{\text{power}, \text{real-normed-algebra}\}$
 $\langle \text{proof} \rangle$

lemma *tendsto-null-power*: $\llbracket (f \longrightarrow 0) \ F; 0 < n \rrbracket \implies ((\lambda x. f \ x ^ n) \longrightarrow 0) \ F$
for $f :: 'a \Rightarrow 'b :: \{\text{power}, \text{real-normed-algebra-1}\}$
 $\langle \text{proof} \rangle$

lemma *continuous-power* [continuous-intros]: $\text{continuous } F \ f \implies \text{continuous } F \ (\lambda x. (f \ x) ^ n)$
for $f :: 'a :: \text{t2-space} \Rightarrow 'b :: \{\text{power}, \text{real-normed-algebra}\}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-power* [continuous-intros]:
fixes $f :: - \Rightarrow 'b :: \{\text{power}, \text{real-normed-algebra}\}$
shows $\text{continuous-on } s \ f \implies \text{continuous-on } s \ (\lambda x. (f \ x) ^ n)$
 $\langle \text{proof} \rangle$

lemma *tendsto-prod* [tendsto-intros]:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c :: \{\text{real-normed-algebra}, \text{comm-ring-1}\}$
shows $(\bigwedge i. i \in S \implies (f \ i \longrightarrow L \ i) \ F) \implies ((\lambda x. \prod_{i \in S} f \ i \ x) \longrightarrow (\prod_{i \in S} L \ i)) \ F$
 $\langle \text{proof} \rangle$

lemma *continuous-prod* [continuous-intros]:
fixes $f :: 'a \Rightarrow 'b :: \text{t2-space} \Rightarrow 'c :: \{\text{real-normed-algebra}, \text{comm-ring-1}\}$
shows $(\bigwedge i. i \in S \implies \text{continuous } F \ (f \ i)) \implies \text{continuous } F \ (\lambda x. \prod_{i \in S} f \ i \ x)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-prod* [continuous-intros]:

fixes $f :: 'a \Rightarrow - \Rightarrow 'c :: \{ \text{real-normed-algebra}, \text{comm-ring-1} \}$
shows $(\bigwedge i. i \in S \Rightarrow \text{continuous-on } s \ (f \ i)) \Rightarrow \text{continuous-on } s \ (\lambda x. \prod_{i \in S}. f \ i \ x)$
 $\langle \text{proof} \rangle$

lemma *tendsto-of-real-iff*:
 $((\lambda x. \text{of-real } (f \ x) :: 'a :: \text{real-normed-div-algebra}) \longrightarrow \text{of-real } c) \ F \longleftrightarrow (f \longrightarrow c) \ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-add-const-iff*:
 $((\lambda x. c + f \ x :: 'a :: \text{topological-group-add}) \longrightarrow c + d) \ F \longleftrightarrow (f \longrightarrow d) \ F$
 $\langle \text{proof} \rangle$

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 $(f \longrightarrow a) \ F \ (g \longrightarrow b) \ F$
shows $((\lambda x. f \ x \wedge g \ x) \longrightarrow a \wedge b) \ F$
 $\langle \text{proof} \rangle$

lemma *continuous-mult'* [*continuous-intros*]:
fixes $f \ g :: - \Rightarrow 'b :: \text{topological-semigroup-mult}$
shows $\text{continuous } F \ f \Rightarrow \text{continuous } F \ g \Rightarrow \text{continuous } F \ (\lambda x. f \ x * g \ x)$
 $\langle \text{proof} \rangle$

lemma *continuous-power'* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'b :: \text{topological-monoid-mult}$
shows $\text{continuous } F \ f \Rightarrow \text{continuous } F \ g \Rightarrow \text{continuous } F \ (\lambda x. f \ x \wedge g \ x)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-mult'* [*continuous-intros*]:
fixes $f \ g :: - \Rightarrow 'b :: \text{topological-semigroup-mult}$
shows $\text{continuous-on } A \ f \Rightarrow \text{continuous-on } A \ g \Rightarrow \text{continuous-on } A \ (\lambda x. f \ x * g \ x)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-power'* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'b :: \text{topological-monoid-mult}$
shows $\text{continuous-on } A \ f \Rightarrow \text{continuous-on } A \ g \Rightarrow \text{continuous-on } A \ (\lambda x. f \ x \wedge g \ x)$
 $\langle \text{proof} \rangle$

lemma *tendsto-mult-one*:
fixes $f \ g :: - \Rightarrow 'b :: \text{topological-monoid-mult}$
shows $(f \longrightarrow 1) \ F \Rightarrow (g \longrightarrow 1) \ F \Rightarrow ((\lambda x. f \ x * g \ x) \longrightarrow 1) \ F$

<proof>

lemma *tendsto-prod'* [*tendsto-intros*]:

fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c :: \text{topological-comm-monoid-mult}$
shows $(\bigwedge i. i \in I \implies (f\ i \longrightarrow a\ i)\ F) \implies ((\lambda x. \prod_{i \in I}. f\ i\ x) \longrightarrow (\prod_{i \in I}. a\ i))\ F$
<proof>

lemma *tendsto-one-prod'*:

fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c :: \text{topological-comm-monoid-mult}$
assumes $\bigwedge i. i \in I \implies ((\lambda x. f\ x\ i) \longrightarrow 1)\ F$
shows $((\lambda i. \text{prod}\ (f\ i)\ I) \longrightarrow 1)\ F$
<proof>

lemma *LIMSEQ-prod-0*:

fixes $f :: \text{nat} \Rightarrow 'a :: \{\text{semidom}, \text{topological-space}\}$
assumes $f\ i = 0$
shows $(\lambda n. \text{prod}\ f\ \{..n\}) \longrightarrow 0$
<proof>

lemma *LIMSEQ-prod-nonneg*:

fixes $f :: \text{nat} \Rightarrow 'a :: \{\text{linordered-semidom}, \text{linorder-topology}\}$
assumes $0: \bigwedge n. 0 \leq f\ n$ **and** $a: (\lambda n. \text{prod}\ f\ \{..n\}) \longrightarrow a$
shows $a \geq 0$
<proof>

lemma *continuous-prod'* [*continuous-intros*]:

fixes $f :: 'a \Rightarrow 'b :: t2\text{-space} \Rightarrow 'c :: \text{topological-comm-monoid-mult}$
shows $(\bigwedge i. i \in I \implies \text{continuous}\ F\ (f\ i)) \implies \text{continuous}\ F\ (\lambda x. \prod_{i \in I}. f\ i\ x)$
<proof>

lemma *continuous-on-prod'* [*continuous-intros*]:

fixes $f :: 'a \Rightarrow 'b :: \text{topological-space} \Rightarrow 'c :: \text{topological-comm-monoid-mult}$
shows $(\bigwedge i. i \in I \implies \text{continuous-on}\ S\ (f\ i)) \implies \text{continuous-on}\ S\ (\lambda x. \prod_{i \in I}. f\ i\ x)$
<proof>

instance *nat* :: *topological-comm-monoid-mult*

<proof>

instance *int* :: *topological-comm-monoid-mult*

<proof>

class *comm-real-normed-algebra-1* = *real-normed-algebra-1* + *comm-monoid-mult*

context *real-normed-field*

begin

subclass *comm-real-normed-algebra-1*

$\langle \text{proof} \rangle$

end

108.3.3 Inverse and division

lemma (in *bounded-bilinear*) *Zfun-prod-Bfun*:

assumes $f: \text{Zfun } f \ F$

and $g: \text{Bfun } g \ F$

shows $\text{Zfun } (\lambda x. f \ x \ ** \ g \ x) \ F$

$\langle \text{proof} \rangle$

lemma (in *bounded-bilinear*) *Bfun-prod-Zfun*:

assumes $f: \text{Bfun } f \ F$

and $g: \text{Zfun } g \ F$

shows $\text{Zfun } (\lambda x. f \ x \ ** \ g \ x) \ F$

$\langle \text{proof} \rangle$

lemma *Bfun-inverse*:

fixes $a :: 'a::\text{real-normed-div-algebra}$

assumes $f: (f \longrightarrow a) \ F$

assumes $a: a \neq 0$

shows $\text{Bfun } (\lambda x. \text{inverse } (f \ x)) \ F$

$\langle \text{proof} \rangle$

lemma *tendsto-inverse* [*tendsto-intros*]:

fixes $a :: 'a::\text{real-normed-div-algebra}$

assumes $f: (f \longrightarrow a) \ F$

and $a: a \neq 0$

shows $((\lambda x. \text{inverse } (f \ x)) \longrightarrow \text{inverse } a) \ F$

$\langle \text{proof} \rangle$

lemma *continuous-inverse*:

fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-div-algebra}$

assumes *continuous* $F \ f$

and $f \ (Lim \ F \ (\lambda x. \ x)) \neq 0$

shows *continuous* $F \ (\lambda x. \text{inverse } (f \ x))$

$\langle \text{proof} \rangle$

lemma *continuous-at-within-inverse* [*continuous-intros*]:

fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-div-algebra}$

assumes *continuous* (at a within s) f

and $f \ a \neq 0$

shows *continuous* (at a within s) $(\lambda x. \text{inverse } (f \ x))$

$\langle \text{proof} \rangle$

lemma *continuous-on-inverse* [*continuous-intros*]:

fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-div-algebra}$

assumes *continuous-on* $s \ f$

and $\forall x \in s. f\ x \neq 0$
shows *continuous-on* $s\ (\lambda x. \text{inverse}\ (f\ x))$
 $\langle \text{proof} \rangle$

lemma *tendsto-divide* [*tendsto-intros*]:
fixes $a\ b :: 'a::\text{real-normed-field}$
shows $(f \longrightarrow a)\ F \Longrightarrow (g \longrightarrow b)\ F \Longrightarrow b \neq 0 \Longrightarrow ((\lambda x. f\ x / g\ x) \longrightarrow a / b)\ F$
 $\langle \text{proof} \rangle$

lemma *continuous-divide*:
fixes $f\ g :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-field}$
assumes *continuous* $F\ f$
and *continuous* $F\ g$
and $g\ (\text{Lim}\ F\ (\lambda x. x)) \neq 0$
shows *continuous* $F\ (\lambda x. (f\ x) / (g\ x))$
 $\langle \text{proof} \rangle$

lemma *continuous-at-within-divide*[*continuous-intros*]:
fixes $f\ g :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-field}$
assumes *continuous* $(\text{at}\ a\ \text{within}\ s)\ f$ *continuous* $(\text{at}\ a\ \text{within}\ s)\ g$
and $g\ a \neq 0$
shows *continuous* $(\text{at}\ a\ \text{within}\ s)\ (\lambda x. (f\ x) / (g\ x))$
 $\langle \text{proof} \rangle$

lemma *isCont-divide*[*continuous-intros*, *simp*]:
fixes $f\ g :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-field}$
assumes *isCont* $f\ a$ *isCont* $g\ a$ $g\ a \neq 0$
shows *isCont* $(\lambda x. (f\ x) / g\ x)\ a$
 $\langle \text{proof} \rangle$

lemma *continuous-on-divide*[*continuous-intros*]:
fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-field}$
assumes *continuous-on* $s\ f$ *continuous-on* $s\ g$
and $\forall x \in s. g\ x \neq 0$
shows *continuous-on* $s\ (\lambda x. (f\ x) / (g\ x))$
 $\langle \text{proof} \rangle$

lemma *continuous-cmult-left-iff*:
fixes $c :: 'a::\text{real-normed-field}$
assumes $c \neq 0$
shows *continuous* $F\ (\lambda x. c * f\ x) \longleftrightarrow \text{continuous}\ F\ f$
 $\langle \text{proof} \rangle$

lemma *continuous-cmult-right-iff*:
fixes $c :: 'a::\text{real-normed-field}$
assumes $c \neq 0$
shows *continuous* $F\ (\lambda x. f\ x * c) \longleftrightarrow \text{continuous}\ F\ f$
 $\langle \text{proof} \rangle$

lemma *continuous-cdivide-iff*:

fixes $c :: 'a :: \text{real-normed-field}$

assumes $c \neq 0$

shows $\text{continuous } F (\lambda x. f x / c) \longleftrightarrow \text{continuous } F f$

<proof>

lemma *continuous-cong*:

assumes *eventually* $(\lambda x. f x = g x) F f (\text{Lim } F (\lambda x. x)) = g (\text{Lim } F (\lambda x. x))$

shows $\text{continuous } F f \longleftrightarrow \text{continuous } F g$

<proof>

lemma *continuous-at-within-cong*:

assumes $f x = g x$ *eventually* $(\lambda x. f x = g x) (\text{at } x \text{ within } S)$

shows $\text{continuous } (\text{at } x \text{ within } S) f \longleftrightarrow \text{continuous } (\text{at } x \text{ within } S) g$

<proof>

lemma *tendsto-power-int* [*tendsto-intros*]:

fixes $a :: 'a :: \text{real-normed-div-algebra}$

assumes $f: (f \longrightarrow a) F$

and $a: a \neq 0$

shows $((\lambda x. \text{power-int } (f x) n) \longrightarrow \text{power-int } a n) F$

<proof>

lemma *continuous-power-int*:

fixes $f :: 'a :: t2\text{-space} \Rightarrow 'b :: \text{real-normed-div-algebra}$

assumes $\text{continuous } F f$

and $f (\text{Lim } F (\lambda x. x)) \neq 0$

shows $\text{continuous } F (\lambda x. \text{power-int } (f x) n)$

<proof>

lemma *continuous-at-within-power-int* [*continuous-intros*]:

fixes $f :: 'a :: t2\text{-space} \Rightarrow 'b :: \text{real-normed-div-algebra}$

assumes $\text{continuous } (\text{at } a \text{ within } s) f$

and $f a \neq 0$

shows $\text{continuous } (\text{at } a \text{ within } s) (\lambda x. \text{power-int } (f x) n)$

<proof>

lemma *continuous-on-power-int* [*continuous-intros*]:

fixes $f :: 'a :: \text{topological-space} \Rightarrow 'b :: \text{real-normed-div-algebra}$

assumes $\text{continuous-on } s f$ **and** $n \geq 0 \vee (\forall x \in s. f x \neq 0)$

shows $\text{continuous-on } s (\lambda x. \text{power-int } (f x) n)$

<proof>

lemma *tendsto-power-int'* [*tendsto-intros*]:

fixes $a :: 'a :: \text{real-normed-div-algebra}$

assumes $f: (f \longrightarrow a) F$

and $a: a \neq 0 \vee n \geq 0$

shows $((\lambda x. \text{power-int } (f x) n) \longrightarrow \text{power-int } a n) F$

$\langle \text{proof} \rangle$

lemma *tendsto-sgn* [*tendsto-intros*]: $(f \longrightarrow l) F \implies l \neq 0 \implies ((\lambda x. \text{sgn } (f x)) \longrightarrow \text{sgn } l) F$
for $l :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *continuous-sgn*:
fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-vector}$
assumes *continuous* $F f$
and $f (\text{Lim } F (\lambda x. x)) \neq 0$
shows *continuous* $F (\lambda x. \text{sgn } (f x))$
 $\langle \text{proof} \rangle$

lemma *continuous-at-within-sgn*[*continuous-intros*]:
fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-vector}$
assumes *continuous* (at a within s) f
and $f a \neq 0$
shows *continuous* (at a within s) $(\lambda x. \text{sgn } (f x))$
 $\langle \text{proof} \rangle$

lemma *isCont-sgn*[*continuous-intros*]:
fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-vector}$
assumes *isCont* $f a$
and $f a \neq 0$
shows *isCont* $(\lambda x. \text{sgn } (f x)) a$
 $\langle \text{proof} \rangle$

lemma *continuous-on-sgn*[*continuous-intros*]:
fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$
assumes *continuous-on* $s f$
and $\forall x \in s. f x \neq 0$
shows *continuous-on* $s (\lambda x. \text{sgn } (f x))$
 $\langle \text{proof} \rangle$

lemma *filterlim-at-infinity*:
fixes $f :: - \Rightarrow 'a::\text{real-normed-vector}$
assumes $0 \leq c$
shows $(\text{LIM } x F. f x :> \text{at-infinity}) \longleftrightarrow (\forall r > c. \text{eventually } (\lambda x. r \leq \text{norm } (f x)) F)$
 $\langle \text{proof} \rangle$

lemma *filterlim-at-infinity-imp-norm-at-top*:
fixes F
assumes *filterlim* $f \text{ at-infinity } F$
shows *filterlim* $(\lambda x. \text{norm } (f x)) \text{ at-top } F$
 $\langle \text{proof} \rangle$

lemma *filterlim-norm-at-top-imp-at-infinity*:

fixes F
assumes $\text{filterlim } (\lambda x. \text{norm } (f x)) \text{ at-top } F$
shows $\text{filterlim } f \text{ at-infinity } F$
 $\langle \text{proof} \rangle$

lemma *filterlim-norm-at-top*: $\text{filterlim } \text{norm } \text{at-top } \text{at-infinity}$
 $\langle \text{proof} \rangle$

lemma *filterlim-at-infinity-conv-norm-at-top*:
 $\text{filterlim } f \text{ at-infinity } G \longleftrightarrow \text{filterlim } (\lambda x. \text{norm } (f x)) \text{ at-top } G$
 $\langle \text{proof} \rangle$

lemma *eventually-not-equal-at-infinity*:
 $\text{eventually } (\lambda x. x \neq (a :: 'a :: \{\text{real-normed-vector}\})) \text{ at-infinity}$
 $\langle \text{proof} \rangle$

lemma *filterlim-int-of-nat-at-topD*:
fixes F
assumes $\text{filterlim } (\lambda x. f \text{ (int } x)) F \text{ at-top}$
shows $\text{filterlim } f F \text{ at-top}$
 $\langle \text{proof} \rangle$

lemma *filterlim-int-sequentially* [*tendsto-intros*]:
 $\text{filterlim } \text{int } \text{at-top } \text{sequentially}$
 $\langle \text{proof} \rangle$

lemma *filterlim-real-of-int-at-top* [*tendsto-intros*]:
 $\text{filterlim } \text{real-of-int } \text{at-top } \text{at-top}$
 $\langle \text{proof} \rangle$

lemma *filterlim-abs-real*: $\text{filterlim } (\text{abs} :: \text{real} \Rightarrow \text{real}) \text{ at-top } \text{at-top}$
 $\langle \text{proof} \rangle$

lemma *filterlim-of-real-at-infinity* [*tendsto-intros*]:
 $\text{filterlim } (\text{of-real} :: \text{real} \Rightarrow 'a :: \text{real-normed-algebra-1}) \text{ at-infinity } \text{at-top}$
 $\langle \text{proof} \rangle$

lemma *not-tendsto-and-filterlim-at-infinity*:
fixes $c :: 'a :: \text{real-normed-vector}$
assumes $F \neq \text{bot}$
and $(f \longrightarrow c) F$
and $\text{filterlim } f \text{ at-infinity } F$
shows False
 $\langle \text{proof} \rangle$

lemma *filterlim-at-infinity-imp-not-convergent*:
assumes $\text{filterlim } f \text{ at-infinity } \text{sequentially}$
shows $\neg \text{convergent } f$
 $\langle \text{proof} \rangle$

lemma *filterlim-at-infinity-imp-eventually-ne*:

assumes *filterlim f at-infinity F*

shows *eventually ($\lambda z. f\ z \neq c$) F*

<proof>

lemma *tendsto-of-nat [tendsto-intros]*:

filterlim (of-nat :: nat \Rightarrow 'a::real-normed-algebra-1) at-infinity sequentially

<proof>

108.4 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*.

lemmas *filterlim-split-at-real = filterlim-split-at[where 'a=real]*

lemma *filtermap-nhds-shift*: *filtermap ($\lambda x. x - d$) (nhds a) = nhds (a - d)*

for *a d :: 'a::real-normed-vector*

<proof>

lemma *filtermap-nhds-minus*: *filtermap ($\lambda x. - x$) (nhds a) = nhds (- a)*

for *a :: 'a::real-normed-vector*

<proof>

lemma *filtermap-at-shift*: *filtermap ($\lambda x. x - d$) (at a) = at (a - d)*

for *a d :: 'a::real-normed-vector*

<proof>

lemma *filtermap-at-right-shift*: *filtermap ($\lambda x. x - d$) (at-right a) = at-right (a - d)*

for *a d :: real*

<proof>

lemma *filterlim-shift*:

fixes *d :: 'a::real-normed-vector*

assumes *filterlim f F (at a)*

shows *filterlim (f \circ (+) d) F (at (a - d))*

<proof>

lemma *filterlim-shift-iff*:

fixes *d :: 'a::real-normed-vector*

shows *filterlim (f \circ (+) d) F (at (a - d)) = filterlim f F (at a) (is ?lhs = ?rhs)*

<proof>

lemma *at-right-to-0*: *at-right a = filtermap ($\lambda x. x + a$) (at-right 0)*

for *a :: real*

<proof>

lemma *filterlim-at-right-to-0:*

filterlim f F (at-right a) \longleftrightarrow *filterlim* $(\lambda x. f (x + a))$ F (at-right 0)
for $a :: \text{real}$
 ⟨proof⟩

lemma *eventually-at-right-to-0:*

eventually P (at-right a) \longleftrightarrow *eventually* $(\lambda x. P (x + a))$ (at-right 0)
for $a :: \text{real}$
 ⟨proof⟩

lemma *at-to-0:* $\text{at } a = \text{filtermap } (\lambda x. x + a) \text{ (at } 0)$

for $a :: 'a :: \text{real-normed-vector}$
 ⟨proof⟩

lemma *filterlim-at-to-0:*

filterlim f F (at a) \longleftrightarrow *filterlim* $(\lambda x. f (x + a))$ F (at 0)
for $a :: 'a :: \text{real-normed-vector}$
 ⟨proof⟩

lemma *eventually-at-to-0:*

eventually P (at a) \longleftrightarrow *eventually* $(\lambda x. P (x + a))$ (at 0)
for $a :: 'a :: \text{real-normed-vector}$
 ⟨proof⟩

lemma *filtermap-at-minus:* $\text{filtermap } (\lambda x. - x) \text{ (at } a) = \text{at } (- a)$

for $a :: 'a :: \text{real-normed-vector}$
 ⟨proof⟩

lemma *at-left-minus:* $\text{at-left } a = \text{filtermap } (\lambda x. - x) \text{ (at-right } (- a))$

for $a :: \text{real}$
 ⟨proof⟩

lemma *at-right-minus:* $\text{at-right } a = \text{filtermap } (\lambda x. - x) \text{ (at-left } (- a))$

for $a :: \text{real}$
 ⟨proof⟩

lemma *filtermap-linear-at-within:*

assumes $\text{bij } f$ **and** $\text{cont: isCont } f \ a$ **and** $\text{open-map: } \bigwedge S. \text{ open } S \implies \text{open } (f'S)$
shows $\text{filtermap } f \text{ (at } a \text{ within } S) = \text{at } (f a) \text{ within } f'S$
 ⟨proof⟩

lemma *filterlim-at-left-to-right:*

filterlim f F (at-left a) \longleftrightarrow *filterlim* $(\lambda x. f (- x))$ F (at-right $(-a)$)
for $a :: \text{real}$
 ⟨proof⟩

lemma *eventually-at-left-to-right:*

eventually P (at-left a) \longleftrightarrow *eventually* $(\lambda x. P (- x))$ (at-right $(-a)$)
for $a :: \text{real}$

$\langle \text{proof} \rangle$

lemma *filterlim-uminus-at-top-at-bot*: $LIM\ x\ at\ bot. -\ x :: real :> at\ top$
 $\langle \text{proof} \rangle$

lemma *filterlim-uminus-at-bot-at-top*: $LIM\ x\ at\ top. -\ x :: real :> at\ bot$
 $\langle \text{proof} \rangle$

lemma *at-bot-mirror* :
shows $(at\ bot :: ('a :: \{ordered\text{-}ab\text{-}group\text{-}add, linorder\}\ filter)) = filtermap\ uminus\ at\ top$
 $\langle \text{proof} \rangle$

lemma *at-top-mirror* :
shows $(at\ top :: ('a :: \{ordered\text{-}ab\text{-}group\text{-}add, linorder\}\ filter)) = filtermap\ uminus\ at\ bot$
 $\langle \text{proof} \rangle$

lemma *filterlim-at-top-mirror*: $(LIM\ x\ at\ top. f\ x :> F) \longleftrightarrow (LIM\ x\ at\ bot. f\ (-x :: real) :> F)$
 $\langle \text{proof} \rangle$

lemma *filterlim-at-bot-mirror*: $(LIM\ x\ at\ bot. f\ x :> F) \longleftrightarrow (LIM\ x\ at\ top. f\ (-x :: real) :> F)$
 $\langle \text{proof} \rangle$

lemma *filterlim-uminus-at-top*: $(LIM\ x\ F. f\ x :> at\ top) \longleftrightarrow (LIM\ x\ F. -\ (f\ x) :: real :> at\ bot)$
 $\langle \text{proof} \rangle$

lemma *tendsto-at-botI-sequentially*:
fixes $f :: real \Rightarrow 'b :: first\text{-}countable\text{-}topology$
assumes $*$: $\bigwedge X. filterlim\ X\ at\ bot\ sequentially \implies (\lambda n. f\ (X\ n)) \longrightarrow y$
shows $(f \longrightarrow y)\ at\ bot$
 $\langle \text{proof} \rangle$

lemma *filterlim-at-infinity-imp-filterlim-at-top*:
assumes $filterlim\ (f :: 'a \Rightarrow real)\ at\ infinity\ F$
assumes *eventually* $(\lambda x. f\ x > 0)\ F$
shows $filterlim\ f\ at\ top\ F$
 $\langle \text{proof} \rangle$

lemma *filterlim-at-infinity-imp-filterlim-at-bot*:
assumes $filterlim\ (f :: 'a \Rightarrow real)\ at\ infinity\ F$
assumes *eventually* $(\lambda x. f\ x < 0)\ F$
shows $filterlim\ f\ at\ bot\ F$
 $\langle \text{proof} \rangle$

lemma *filterlim-uminus-at-bot*: $(LIM\ x\ F. f\ x :> at\ bot) \longleftrightarrow (LIM\ x\ F. -\ (f\ x) ::$

real :> *at-top*)
 ⟨*proof*⟩

lemma *filterlim-inverse-at-top-right*: *LIM x at-right (0::real). inverse x :> at-top*
 ⟨*proof*⟩

lemma *tendsto-inverse-0*:
 fixes *x* :: - \Rightarrow '*a*::*real-normed-div-algebra*
 shows (*inverse* \longrightarrow (*0*::'*a*)) *at-infinity*
 ⟨*proof*⟩

lemma *tendsto-add-filterlim-at-infinity*:
 fixes *c* :: '*b*::*real-normed-vector*
 and *F* :: '*a* *filter*
 assumes (*f* \longrightarrow *c*) *F*
 and *filterlim g at-infinity F*
 shows *filterlim* ($\lambda x. f\ x + g\ x$) *at-infinity F*
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *filterlim-inverse-at-right-top*: *LIM x at-top. inverse x :> at-right (0::real)*
 ⟨*proof*⟩

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
 ⟨*proof*⟩

lemma *filterlim-inverse-at-bot-neg*:
LIM x (at-left (0::real)). inverse x :> at-bot
 ⟨*proof*⟩

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
 ⟨*proof*⟩

lemma *at-right-to-top*: (*at-right (0::real)*) = *filtermap inverse at-top*
 ⟨*proof*⟩

lemma *eventually-at-right-to-top*:
eventually P (at-right (0::real)) \longleftrightarrow *eventually* ($\lambda x. P\ (inverse\ x)$) *at-top*

<proof>

lemma *filterlim-at-right-to-top*:

filterlim f F (at-right (0::real)) \longleftrightarrow (LIM x at-top. f (inverse x) :> F)

<proof>

lemma *at-top-to-right*: *at-top = filtermap inverse (at-right (0::real))*

<proof>

lemma *eventually-at-top-to-right*:

eventually P at-top \longleftrightarrow eventually ($\lambda x. P (inverse x)$) (at-right (0::real))

<proof>

lemma *filterlim-at-top-to-right*:

filterlim f F at-top \longleftrightarrow (LIM x (at-right (0::real)). f (inverse x) :> F)

<proof>

lemma *filterlim-inverse-at-infinity*:

fixes *x :: - \Rightarrow 'a::{real-normed-div-algebra, division-ring}*

shows *filterlim inverse at-infinity (at (0::'a))*

<proof>

lemma *filterlim-inverse-at-iff*:

fixes *g :: 'a \Rightarrow 'b::{real-normed-div-algebra, division-ring}*

shows *(LIM x F. inverse (g x) :> at 0) \longleftrightarrow (LIM x F. g x :> at-infinity)*

<proof>

lemma *tendsto-mult-filterlim-at-infinity*:

fixes *c :: 'a::real-normed-field*

assumes *(f \longrightarrow c) F c \neq 0*

assumes *filterlim g at-infinity F*

shows *filterlim ($\lambda x. f x * g x$) at-infinity F*

<proof>

lemma *filterlim-power-int-neg-at-infinity*:

fixes *f :: - \Rightarrow 'a::{real-normed-div-algebra, division-ring}*

assumes *n < 0 and lim: (f \longrightarrow 0) F and ev: eventually ($\lambda x. f x \neq 0$) F*

shows *filterlim ($\lambda x. f x \text{ powi } n$) at-infinity F*

<proof>

lemma *tendsto-inverse-0-at-top*: *LIM x F. f x :> at-top \implies (($\lambda x. inverse (f x) :: real$) \longrightarrow 0) F*

<proof>

lemma *filterlim-inverse-at-top-iff*:

eventually ($\lambda x. 0 < f x$) F \implies (LIM x F. inverse (f x) :> at-top) \longleftrightarrow (f \longrightarrow (0 :: real)) F

<proof>

lemma *filterlim-at-top-iff-inverse-0:*

eventually $(\lambda x. 0 < f x) F \implies (LIM x F. f x :> at-top) \longleftrightarrow ((inverse \circ f) \longrightarrow (0 :: real)) F$
<proof>

lemma *real-tendsto-divide-at-top:*

fixes $c :: real$
assumes $(f \longrightarrow c) F$
assumes *filterlim g at-top F*
shows $((\lambda x. f x / g x) \longrightarrow 0) F$
<proof>

lemma *mult-nat-left-at-top:* $c > 0 \implies filterlim (\lambda x. c * x) at-top sequentially$

for $c :: nat$
<proof>

lemma *mult-nat-right-at-top:* $c > 0 \implies filterlim (\lambda x. x * c) at-top sequentially$

for $c :: nat$
<proof>

lemma *filterlim-times-pos:*

*LIM x F1. c * f x :> at-right l*
if *filterlim f (at-right p) F1* $0 < c \wedge l = c * p$
for $c :: 'a :: \{linordered-field, linorder-topology\}$
<proof>

lemma *filtermap-nhds-times:* $c \neq 0 \implies filtermap (times c) (nhds a) = nhds (c * a)$

for $a c :: 'a :: real-normed-field$
<proof>

lemma *filtermap-times-pos-at-right:*

fixes $c :: 'a :: \{linordered-field, linorder-topology\}$
assumes $c > 0$
shows *filtermap (times c) (at-right p) = at-right (c * p)*
<proof>

lemma *at-to-infinity:* $(at (0 :: 'a :: \{real-normed-field, field\})) = filtermap inverse at-infinity$

<proof>

lemma *lim-at-infinity-0:*

fixes $l :: 'a :: \{real-normed-field, field\}$
shows $(f \longrightarrow l) at-infinity \longleftrightarrow ((f \circ inverse) \longrightarrow l) (at (0 :: 'a))$
<proof>

lemma *lim-zero-infinity:*

fixes $l :: 'a :: \{real-normed-field, field\}$
shows $((\lambda x. f(1 / x)) \longrightarrow l) (at (0 :: 'a)) \implies (f \longrightarrow l) at-infinity$
<proof>

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 *filterlim* $?f \text{ at-top } ?F = (LIM \ x \ ?F. - \ ?f \ x \ :> \text{ at-bot})$.

lemma *filterlim-tendsto-pos-mult-at-top*:
assumes $f: (f \longrightarrow c) \ F$
and $c: 0 < c$
and $g: LIM \ x \ F. \ g \ x \ :> \text{ at-top}$
shows $LIM \ x \ F. (f \ x * g \ x :: real) :> \text{ at-top}$
 $\langle proof \rangle$

lemma *filterlim-at-top-mult-at-top*:
assumes $f: LIM \ x \ F. \ f \ x \ :> \text{ at-top}$
and $g: LIM \ x \ F. \ g \ x \ :> \text{ at-top}$
shows $LIM \ x \ F. (f \ x * g \ x :: real) :> \text{ at-top}$
 $\langle proof \rangle$

lemma *filterlim-at-top-mult-tendsto-pos*:
assumes $f: (f \longrightarrow c) \ F$
and $c: 0 < c$
and $g: LIM \ x \ F. \ g \ x \ :> \text{ at-top}$
shows $LIM \ x \ F. (g \ x * f \ x :: real) :> \text{ at-top}$
 $\langle proof \rangle$

lemma *filterlim-tendsto-pos-mult-at-bot*:
fixes $c :: real$
assumes $(f \longrightarrow c) \ F \ 0 < c \ \text{filterlim} \ g \ \text{at-bot} \ F$
shows $LIM \ x \ F. \ f \ x * g \ x \ :> \text{ at-bot}$
 $\langle proof \rangle$

lemma *filterlim-tendsto-neg-mult-at-bot*:
fixes $c :: real$
assumes $c: (f \longrightarrow c) \ F \ c < 0 \ \text{and} \ g: \text{filterlim} \ g \ \text{at-top} \ F$
shows $LIM \ x \ F. \ f \ x * g \ x \ :> \text{ at-bot}$
 $\langle proof \rangle$

lemma *filterlim-cmult-at-bot-at-top*:
assumes $\text{filterlim} \ (h :: - \Rightarrow real) \ \text{at-top} \ F \ c \neq 0 \ G = (\text{if } c > 0 \text{ then } \text{at-top} \text{ else } \text{at-bot})$
shows $\text{filterlim} \ (\lambda x. \ c * h \ x) \ G \ F$
 $\langle proof \rangle$

lemma *filterlim-pow-at-top*:
fixes $f :: 'a \Rightarrow real$
assumes $0 < n$
and $f: LIM \ x \ F. \ f \ x \ :> \text{ at-top}$
shows $LIM \ x \ F. (f \ x)^n :: real :> \text{ at-top}$
 $\langle proof \rangle$

lemma *filterlim-pow-at-bot-even*:

fixes $f :: \text{real} \Rightarrow \text{real}$

shows $0 < n \implies \text{LIM } x \ F. f \ x :> \text{at-bot} \implies \text{even } n \implies \text{LIM } x \ F. (f \ x)^\wedge n :> \text{at-top}$
 $\langle \text{proof} \rangle$

lemma *filterlim-pow-at-bot-odd*:

fixes $f :: \text{real} \Rightarrow \text{real}$

shows $0 < n \implies \text{LIM } x \ F. f \ x :> \text{at-bot} \implies \text{odd } n \implies \text{LIM } x \ F. (f \ x)^\wedge n :> \text{at-bot}$
 $\langle \text{proof} \rangle$

lemma *filterlim-power-at-infinity [tendsto-intros]*:

fixes F **and** $f :: 'a \Rightarrow 'b :: \text{real-normed-div-algebra}$

assumes *filterlim f at-infinity F n > 0*

shows *filterlim $(\lambda x. f \ x^\wedge n)$ at-infinity F*

$\langle \text{proof} \rangle$

lemma *filterlim-tendsto-add-at-top*:

assumes $f: (f \longrightarrow c) \ F$

and $g: \text{LIM } x \ F. g \ x :> \text{at-top}$

shows $\text{LIM } x \ F. (f \ x + g \ x :: \text{real}) :> \text{at-top}$

$\langle \text{proof} \rangle$

lemma *filterlim-tendsto-add-at-top-iff*:

assumes $f: (f \longrightarrow c) \ F$

shows $(\text{LIM } x \ F. (f \ x + g \ x :: \text{real}) :> \text{at-top}) \longleftrightarrow (\text{LIM } x \ F. g \ x :> \text{at-top})$
 $\langle \text{proof} \rangle$

lemma *filterlim-tendsto-add-at-bot-iff*:

fixes $c :: \text{real}$

assumes $f: (f \longrightarrow c) \ F$

shows $(\text{LIM } x \ F. f \ x + g \ x :> \text{at-bot}) \longleftrightarrow (\text{LIM } x \ F. g \ x :> \text{at-bot})$
 $\langle \text{proof} \rangle$

lemma *LIM-at-top-divide*:

fixes $f \ g :: 'a \Rightarrow \text{real}$

assumes $f: (f \longrightarrow a) \ F \ 0 < a$

and $g: (g \longrightarrow 0) \ F \text{ eventually } (\lambda x. 0 < g \ x) \ F$

shows $\text{LIM } x \ F. f \ x / g \ x :> \text{at-top}$

$\langle \text{proof} \rangle$

lemma *filterlim-at-top-add-at-top*:

assumes $f: \text{LIM } x \ F. f \ x :> \text{at-top}$

and $g: \text{LIM } x \ F. g \ x :> \text{at-top}$

shows $\text{LIM } x \ F. (f \ x + g \ x :: \text{real}) :> \text{at-top}$

$\langle \text{proof} \rangle$

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 $((\lambda x. f\ x / g\ x) \longrightarrow 0) F$
 $\langle \text{proof} \rangle$

lemma *linear-plus-1-le-power*:
fixes $x :: \text{real}$
assumes $x: 0 \leq x$
shows $\text{real } n * x + 1 \leq (x + 1) ^ n$
 $\langle \text{proof} \rangle$

lemma *filterlim-realpow-sequentially-gt1*:
fixes $x :: 'a :: \text{real-normed-div-algebra}$
assumes $x[\text{arith}]: 1 < \text{norm } x$
shows $LIM\ n\ \text{sequentially}. x ^ n :> \text{at-infinity}$
 $\langle \text{proof} \rangle$

lemma *filterlim-divide-at-infinity*:
fixes $f\ g :: 'a \Rightarrow 'a :: \text{real-normed-field}$
assumes $\text{filterlim } f\ (\text{nhds } c) F\ \text{filterlim } g\ (\text{at } 0) F\ c \neq 0$
shows $\text{filterlim } (\lambda x. f\ x / g\ x)\ \text{at-infinity } F$
 $\langle \text{proof} \rangle$

108.5 Floor and Ceiling

lemma *eventually-floor-less*:
fixes $f :: 'a \Rightarrow 'b::\{\text{order-topology}, \text{floor-ceiling}\}$
assumes $f: (f \longrightarrow l) F$
and $l: l \notin \mathbb{Z}$
shows $\forall_F\ x\ \text{in } F. \text{of-int } (\text{floor } l) < f\ x$
 $\langle \text{proof} \rangle$

lemma *eventually-less-ceiling*:
fixes $f :: 'a \Rightarrow 'b::\{\text{order-topology}, \text{floor-ceiling}\}$
assumes $f: (f \longrightarrow l) F$
and $l: l \notin \mathbb{Z}$
shows $\forall_F\ x\ \text{in } F. f\ x < \text{of-int } (\text{ceiling } l)$
 $\langle \text{proof} \rangle$

lemma *eventually-floor-eq*:
fixes $f::'a \Rightarrow 'b::\{\text{order-topology}, \text{floor-ceiling}\}$
assumes $f: (f \longrightarrow l) F$
and $l: l \notin \mathbb{Z}$
shows $\forall_F\ x\ \text{in } F. \text{floor } (f\ x) = \text{floor } l$
 $\langle \text{proof} \rangle$

lemma *eventually-ceiling-eq*:

fixes $f::'a \Rightarrow 'b::\{\text{order-topology, floor-ceiling}\}$
assumes $f: (f \longrightarrow l) F$
and $l: l \notin \mathbb{Z}$
shows $\forall_F x \text{ in } F. \text{ ceiling } (f x) = \text{ ceiling } l$
 $\langle \text{proof} \rangle$

lemma *tendsto-of-int-floor*:
fixes $f::'a \Rightarrow 'b::\{\text{order-topology, floor-ceiling}\}$
assumes $(f \longrightarrow l) F$
and $l \notin \mathbb{Z}$
shows $((\lambda x. \text{ of-int } (\text{floor } (f x)) :: 'c::\{\text{ring-1, topological-space}\}) \longrightarrow \text{ of-int } (\text{floor } l)) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-of-int-ceiling*:
fixes $f::'a \Rightarrow 'b::\{\text{order-topology, floor-ceiling}\}$
assumes $(f \longrightarrow l) F$
and $l \notin \mathbb{Z}$
shows $((\lambda x. \text{ of-int } (\text{ceiling } (f x)) :: 'c::\{\text{ring-1, topological-space}\}) \longrightarrow \text{ of-int } (\text{ceiling } l)) F$
 $\langle \text{proof} \rangle$

lemma *continuous-on-of-int-floor*:
 $\text{continuous-on } (\text{UNIV} - \mathbb{Z}::'a::\{\text{order-topology, floor-ceiling}\} \text{ set})$
 $(\lambda x. \text{ of-int } (\text{floor } x) :: 'b::\{\text{ring-1, topological-space}\})$
 $\langle \text{proof} \rangle$

lemma *continuous-on-of-int-ceiling*:
 $\text{continuous-on } (\text{UNIV} - \mathbb{Z}::'a::\{\text{order-topology, floor-ceiling}\} \text{ set})$
 $(\lambda x. \text{ of-int } (\text{ceiling } x) :: 'b::\{\text{ring-1, topological-space}\})$
 $\langle \text{proof} \rangle$

108.6 Limits of Sequences

lemma *[trans]*: $X = Y \Longrightarrow Y \longrightarrow z \Longrightarrow X \longrightarrow z$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-iff*:
fixes $L :: 'a::\text{real-normed-vector}$
shows $(X \longrightarrow L) = (\forall r > 0. \exists no. \forall n \geq no. \text{ norm } (X n - L) < r)$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-I*: $(\bigwedge r. 0 < r \Longrightarrow \exists no. \forall n \geq no. \text{ norm } (X n - L) < r) \Longrightarrow X \longrightarrow L$
for $L :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-D*: $X \longrightarrow L \Longrightarrow 0 < r \Longrightarrow \exists no. \forall n \geq no. \text{ norm } (X n - L) < r$

for $L :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-linear*: $X \longrightarrow x \implies l > 0 \implies (\lambda n. X (n * l)) \longrightarrow x$
 $\langle \text{proof} \rangle$

Transformation of limit.

lemma *Lim-transform*: $(g \longrightarrow a) F \implies ((\lambda x. f x - g x) \longrightarrow 0) F \implies (f \longrightarrow a) F$
for $a b :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *Lim-transform2*: $(f \longrightarrow a) F \implies ((\lambda x. f x - g x) \longrightarrow 0) F \implies (g \longrightarrow a) F$
for $a b :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

proposition *Lim-transform-eq*: $((\lambda x. f x - g x) \longrightarrow 0) F \implies (f \longrightarrow a) F \iff (g \longrightarrow a) F$
for $a :: 'a::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *Lim-transform-eventually*:
 $\llbracket (f \longrightarrow l) F; \text{eventually } (\lambda x. f x = g x) F \rrbracket \implies (g \longrightarrow l) F$
 $\langle \text{proof} \rangle$

lemma *Lim-transform-within*:
assumes $(f \longrightarrow l) \text{ (at } x \text{ within } S)$
and $0 < d$
and $\bigwedge x'. x' \in S \implies 0 < \text{dist } x' x \implies \text{dist } x' x < d \implies f x' = g x'$
shows $(g \longrightarrow l) \text{ (at } x \text{ within } S)$
 $\langle \text{proof} \rangle$

lemma *filterlim-transform-within*:
assumes $\text{filterlim } g \ G \text{ (at } x \text{ within } S)$
assumes $G \leq F \ 0 < d \ (\bigwedge x'. x' \in S \implies 0 < \text{dist } x' x \implies \text{dist } x' x < d \implies f x' = g x')$
shows $\text{filterlim } f \ F \text{ (at } x \text{ within } S)$
 $\langle \text{proof} \rangle$

Common case assuming being away from some crucial point like 0.

lemma *Lim-transform-away-within*:
fixes $a b :: 'a::\text{t1-space}$
assumes $a \neq b$
and $\forall x \in S. x \neq a \wedge x \neq b \longrightarrow f x = g x$
and $(f \longrightarrow l) \text{ (at } a \text{ within } S)$
shows $(g \longrightarrow l) \text{ (at } a \text{ within } S)$
 $\langle \text{proof} \rangle$

lemma *Lim-transform-away-at:*
fixes $a\ b :: 'a::t1\text{-space}$
assumes $ab: a \neq b$
and $fg: \forall x. x \neq a \wedge x \neq b \longrightarrow f\ x = g\ x$
and $fl: (f \longrightarrow l)\ (at\ a)$
shows $(g \longrightarrow l)\ (at\ a)$
 $\langle proof \rangle$

Alternatively, within an open set.

lemma *Lim-transform-within-open:*
assumes $(f \longrightarrow l)\ (at\ a\ within\ T)$
and $open\ s$ **and** $a \in s$
and $\bigwedge x. x \in s \implies x \neq a \implies f\ x = g\ x$
shows $(g \longrightarrow l)\ (at\ a\ within\ T)$
 $\langle proof \rangle$

A congruence rule allowing us to transform limits assuming not at point.

lemma *Lim-cong-within:*
assumes $a = b$
and $x = y$
and $S = T$
and $\bigwedge x. x \neq b \implies x \in T \implies f\ x = g\ x$
shows $(f \longrightarrow x)\ (at\ a\ within\ S) \longleftrightarrow (g \longrightarrow y)\ (at\ b\ within\ T)$
 $\langle proof \rangle$

An unbounded sequence’s inverse tends to 0.

lemma *LIMSEQ-inverse-zero:*
assumes $\bigwedge r::real. \exists N. \forall n \geq N. r < X\ n$
shows $(\lambda n. inverse\ (X\ n)) \longrightarrow 0$
 $\langle proof \rangle$

The sequence $1 / n$ tends to 0 as n tends to infinity.

lemma *LIMSEQ-inverse-real-of-nat:* $(\lambda n. inverse\ (real\ (Suc\ n))) \longrightarrow 0$
 $\langle proof \rangle$

The sequence $r + 1 / n$ tends to r as n tends to infinity is now easily proved.

lemma *LIMSEQ-inverse-real-of-nat-add:* $(\lambda n. r + inverse\ (real\ (Suc\ n))) \longrightarrow r$
 $\langle proof \rangle$

lemma *LIMSEQ-inverse-real-of-nat-add-minus:* $(\lambda n. r + -inverse\ (real\ (Suc\ n))) \longrightarrow r$
 $\langle proof \rangle$

lemma *LIMSEQ-inverse-real-of-nat-add-minus-mult:* $(\lambda n. r * (1 + -inverse\ (real\ (Suc\ n)))) \longrightarrow r$
 $\langle proof \rangle$

lemma *lim-inverse-n*: $((\lambda n. \text{inverse}(\text{of-nat } n)) \longrightarrow (0 :: 'a :: \text{real-normed-field}))$ *sequentially*
 ⟨proof⟩

lemma *lim-inverse-n'*: $((\lambda n. 1 / n) \longrightarrow 0)$ *sequentially*
 ⟨proof⟩

lemma *LIMSEQ-Suc-n-over-n*: $(\lambda n. \text{of-nat } (\text{Suc } n) / \text{of-nat } n :: 'a :: \text{real-normed-field})$
 $\longrightarrow 1$
 ⟨proof⟩

lemma *LIMSEQ-n-over-Suc-n*: $(\lambda n. \text{of-nat } n / \text{of-nat } (\text{Suc } n) :: 'a :: \text{real-normed-field})$
 $\longrightarrow 1$
 ⟨proof⟩

108.7 Convergence on sequences

lemma *convergent-cong*:
assumes *eventually* $(\lambda x. f\ x = g\ x)$ *sequentially*
shows *convergent* $f \longleftrightarrow \text{convergent } g$
 ⟨proof⟩

lemma *convergent-Suc-iff*: *convergent* $(\lambda n. f\ (\text{Suc } n)) \longleftrightarrow \text{convergent } f$
 ⟨proof⟩

lemma *convergent-ignore-initial-segment*: *convergent* $(\lambda n. f\ (n + m)) = \text{convergent } f$
 ⟨proof⟩

lemma *convergent-add*:
fixes $X\ Y :: \text{nat} \Rightarrow 'a :: \text{topological-monoid-add}$
assumes *convergent* $(\lambda n. X\ n)$
and *convergent* $(\lambda n. Y\ n)$
shows *convergent* $(\lambda n. X\ n + Y\ n)$
 ⟨proof⟩

lemma *convergent-sum*:
fixes $X :: 'a \Rightarrow \text{nat} \Rightarrow 'b :: \text{topological-comm-monoid-add}$
shows $(\bigwedge i. i \in A \implies \text{convergent } (\lambda n. X\ i\ n)) \implies \text{convergent } (\lambda n. \sum_{i \in A} X\ i\ n)$
 ⟨proof⟩

lemma *(in bounded-linear) convergent*:
assumes *convergent* $(\lambda n. X\ n)$
shows *convergent* $(\lambda n. f\ (X\ n))$
 ⟨proof⟩

lemma *(in bounded-bilinear) convergent*:
assumes *convergent* $(\lambda n. X\ n)$

and *convergent* ($\lambda n. Y\ n$)
shows *convergent* ($\lambda n. X\ n ** Y\ n$)
 $\langle \text{proof} \rangle$

lemma *convergent-minus-iff*:
fixes $X :: \text{nat} \Rightarrow 'a::\text{topological-group-add}$
shows *convergent* $X \longleftrightarrow \text{convergent } (\lambda n. -\ X\ n)$
 $\langle \text{proof} \rangle$

lemma *convergent-diff*:
fixes $X\ Y :: \text{nat} \Rightarrow 'a::\text{topological-group-add}$
assumes *convergent* ($\lambda n. X\ n$)
assumes *convergent* ($\lambda n. Y\ n$)
shows *convergent* ($\lambda n. X\ n - Y\ n$)
 $\langle \text{proof} \rangle$

lemma *convergent-norm*:
assumes *convergent* f
shows *convergent* ($\lambda n. \text{norm } (f\ n)$)
 $\langle \text{proof} \rangle$

lemma *convergent-of-real*:
convergent $f \implies \text{convergent } (\lambda n. \text{of-real } (f\ n) :: 'a::\text{real-normed-algebra-1})$
 $\langle \text{proof} \rangle$

lemma *convergent-add-const-iff*:
convergent ($\lambda n. c + f\ n :: 'a::\text{topological-ab-group-add}$) \longleftrightarrow *convergent* f
 $\langle \text{proof} \rangle$

lemma *convergent-add-const-right-iff*:
convergent ($\lambda n. f\ n + c :: 'a::\text{topological-ab-group-add}$) \longleftrightarrow *convergent* f
 $\langle \text{proof} \rangle$

lemma *convergent-diff-const-right-iff*:
convergent ($\lambda n. f\ n - c :: 'a::\text{topological-ab-group-add}$) \longleftrightarrow *convergent* f
 $\langle \text{proof} \rangle$

lemma *convergent-mult*:
fixes $X\ Y :: \text{nat} \Rightarrow 'a::\text{topological-semigroup-mult}$
assumes *convergent* ($\lambda n. X\ n$)
and *convergent* ($\lambda n. Y\ n$)
shows *convergent* ($\lambda n. X\ n * Y\ n$)
 $\langle \text{proof} \rangle$

lemma *convergent-mult-const-iff*:
assumes $c \neq 0$
shows *convergent* ($\lambda n. c * f\ n :: 'a::\{\text{field}, \text{topological-semigroup-mult}\}$) \longleftrightarrow *convergent* f
 $\langle \text{proof} \rangle$

lemma *convergent-mult-const-right-iff*:
fixes $c :: 'a :: \{\text{field}, \text{topological-semigroup-mult}\}$
assumes $c \neq 0$
shows $\text{convergent } (\lambda n. f\ n * c) \longleftrightarrow \text{convergent } f$
 $\langle \text{proof} \rangle$

lemma *convergent-imp-Bseq*: $\text{convergent } f \implies \text{Bseq } f$
 $\langle \text{proof} \rangle$

A monotone sequence converges to its least upper bound.

lemma *LIMSEQ-incseq-SUP*:
fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{conditionally-complete-linorder}, \text{linorder-topology}\}$
assumes $u: \text{bdd-above } (\text{range } X)$
and $X: \text{incseq } X$
shows $X \longrightarrow (\text{SUP } i. X\ i)$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-decseq-INF*:
fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{conditionally-complete-linorder}, \text{linorder-topology}\}$
assumes $u: \text{bdd-below } (\text{range } X)$
and $X: \text{decseq } X$
shows $X \longrightarrow (\text{INF } i. X\ i)$
 $\langle \text{proof} \rangle$

Main monotonicity theorem.

lemma *Bseq-monoseq-convergent*: $\text{Bseq } X \implies \text{monoseq } X \implies \text{convergent } X$
for $X :: \text{nat} \Rightarrow \text{real}$
 $\langle \text{proof} \rangle$

lemma *Bseq-mono-convergent*: $\text{Bseq } X \implies (\forall m\ n. m \leq n \longrightarrow X\ m \leq X\ n) \implies \text{convergent } X$
for $X :: \text{nat} \Rightarrow \text{real}$
 $\langle \text{proof} \rangle$

lemma *monoseq-imp-convergent-iff-Bseq*: $\text{monoseq } f \implies \text{convergent } f \longleftrightarrow \text{Bseq } f$
for $f :: \text{nat} \Rightarrow \text{real}$
 $\langle \text{proof} \rangle$

lemma *Bseq-monoseq-convergent'-inc*:
fixes $f :: \text{nat} \Rightarrow \text{real}$
shows $\text{Bseq } (\lambda n. f\ (n + M)) \implies (\bigwedge m\ n. M \leq m \implies m \leq n \implies f\ m \leq f\ n) \implies \text{convergent } f$
 $\langle \text{proof} \rangle$

lemma *Bseq-monoseq-convergent'-dec*:
fixes $f :: \text{nat} \Rightarrow \text{real}$
shows $\text{Bseq } (\lambda n. f\ (n + M)) \implies (\bigwedge m\ n. M \leq m \implies m \leq n \implies f\ m \geq f\ n) \implies \text{convergent } f$

<proof>

lemma *Cauchy-iff*: $\text{Cauchy } X \longleftrightarrow (\forall e > 0. \exists M. \forall m \geq M. \forall n \geq M. \text{norm } (X\ m - X\ n) < e)$
for $X :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$
<proof>

lemma *CauchyI*: $(\bigwedge e. 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{norm } (X\ m - X\ n) < e) \implies \text{Cauchy } X$
for $X :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$
<proof>

lemma *CauchyD*: $\text{Cauchy } X \implies 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{norm } (X\ m - X\ n) < e$
for $X :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$
<proof>

lemma *incseq-convergent*:
fixes $X :: \text{nat} \Rightarrow \text{real}$
assumes *incseq* X
and $\forall i. X\ i \leq B$
obtains L **where** $X \longrightarrow L \ \forall i. X\ i \leq L$
<proof>

lemma *decseq-convergent*:
fixes $X :: \text{nat} \Rightarrow \text{real}$
assumes *decseq* X
and $\forall i. B \leq X\ i$
obtains L **where** $X \longrightarrow L \ \forall i. L \leq X\ i$
<proof>

lemma *monoseq-convergent*:
fixes $X :: \text{nat} \Rightarrow \text{real}$
assumes X : *monoseq* X **and** B : $\bigwedge i. |X\ i| \leq B$
obtains L **where** $X \longrightarrow L$
<proof>

108.8 More about *filterlim* (thanks to Wenda Li)

lemma *filterlim-at-infinity-times*:
fixes $f :: 'a \Rightarrow 'b::\text{real-normed-field}$
assumes *filterlim* f *at-infinity* F *filterlim* g *at-infinity* F
shows *filterlim* $(\lambda x. f\ x * g\ x)$ *at-infinity* F
<proof>

lemma *filterlim-at-top-at-bot[elim]*:
fixes $f::'a \Rightarrow 'b::\text{unbounded-dense-linorder}$ **and** $F::'a$ *filter*
assumes *top:filterlim* f *at-top* F **and** *bot: filterlim* f *at-bot* F **and** $F \neq \text{bot}$
shows *False*

<proof>

lemma *filterlim-at-top-nhds[elim]*:

fixes $f::'a \Rightarrow 'b::\{\text{unbounded-dense-linorder}, \text{order-topology}\}$ **and** $F::'a \text{ filter}$
assumes *top:filterlim f at-top F* **and** *tendsto: (f \longrightarrow c) F* **and** $F \neq \text{bot}$
shows *False*

<proof>

lemma *filterlim-at-bot-nhds[elim]*:

fixes $f::'a \Rightarrow 'b::\{\text{unbounded-dense-linorder}, \text{order-topology}\}$ **and** $F::'a \text{ filter}$
assumes *top:filterlim f at-bot F* **and** *tendsto: (f \longrightarrow c) F* **and** $F \neq \text{bot}$
shows *False*

<proof>

lemma *eventually-times-inverse-1*:

fixes $f::'a \Rightarrow 'b::\{\text{field}, \text{t2-space}\}$
assumes $(f \longrightarrow c) F$ $c \neq 0$
shows $\forall_F x \text{ in } F. \text{inverse } (f x) * f x = 1$

<proof>

lemma *filterlim-at-infinity-divide-iff*:

fixes $f::'a \Rightarrow 'b::\text{real-normed-field}$
assumes $(f \longrightarrow c) F$ $c \neq 0$
shows $(\text{LIM } x F. f x / g x :> \text{at-infinity}) \longleftrightarrow (\text{LIM } x F. g x :> \text{at } 0)$

<proof>

lemma *filterlim-tendsto-pos-mult-at-top-iff*:

fixes $f::'a \Rightarrow \text{real}$
assumes $(f \longrightarrow c) F$ **and** $0 < c$
shows $(\text{LIM } x F. (f x * g x) :> \text{at-top}) \longleftrightarrow (\text{LIM } x F. g x :> \text{at-top})$

<proof>

lemma *filterlim-tendsto-pos-mult-at-bot-iff*:

fixes $c :: \text{real}$
assumes $(f \longrightarrow c) F$ $0 < c$
shows $(\text{LIM } x F. f x * g x :> \text{at-bot}) \longleftrightarrow \text{filterlim } g \text{ at-bot } F$

<proof>

lemma *filterlim-tendsto-neg-mult-at-top-iff*:

fixes $f::'a \Rightarrow \text{real}$
assumes $(f \longrightarrow c) F$ **and** $c < 0$
shows $(\text{LIM } x F. (f x * g x) :> \text{at-top}) \longleftrightarrow (\text{LIM } x F. g x :> \text{at-bot})$

<proof>

lemma *filterlim-tendsto-neg-mult-at-bot-iff*:

fixes $c :: \text{real}$
assumes $(f \longrightarrow c) F$ $0 > c$
shows $(\text{LIM } x F. f x * g x :> \text{at-bot}) \longleftrightarrow \text{filterlim } g \text{ at-top } F$

<proof>

108.9 Power Sequences

lemma *Bseq-realpow*: $0 \leq x \implies x \leq 1 \implies Bseq (\lambda n. x ^ n)$
for $x :: real$
 $\langle proof \rangle$

lemma *monoseq-realpow*: $0 \leq x \implies x \leq 1 \implies monoseq (\lambda n. x ^ n)$
for $x :: real$
 $\langle proof \rangle$

lemma *convergent-realpow*: $0 \leq x \implies x \leq 1 \implies convergent (\lambda n. x ^ n)$
for $x :: real$
 $\langle proof \rangle$

lemma *LIMSEQ-inverse-realpow-zero*: $1 < x \implies (\lambda n. inverse (x ^ n)) \longrightarrow 0$
for $x :: real$
 $\langle proof \rangle$

lemma *LIMSEQ-realpow-zero*:
fixes $x :: real$
assumes $0 \leq x < 1$
shows $(\lambda n. x ^ n) \longrightarrow 0$
 $\langle proof \rangle$

lemma *LIMSEQ-power-zero* [*tendsto-intros*]: $norm\ x < 1 \implies (\lambda n. x ^ n) \longrightarrow 0$
for $x :: 'a::real-normed-algebra-1$
 $\langle proof \rangle$

lemma *LIMSEQ-divide-realpow-zero*: $1 < x \implies (\lambda n. a / (x ^ n) :: real) \longrightarrow 0$
 $\langle proof \rangle$

lemma
tendsto-power-zero:
fixes $x::'a::real-normed-algebra-1$
assumes *filterlim f at-top F*
assumes $norm\ x < 1$
shows $((\lambda y. x ^ (f\ y)) \longrightarrow 0)\ F$
 $\langle proof \rangle$

Limit of c^n for $|c| < 1$.

lemma *LIMSEQ-abs-realpow-zero*: $|c| < 1 \implies (\lambda n. |c| ^ n :: real) \longrightarrow 0$
 $\langle proof \rangle$

lemma *LIMSEQ-abs-realpow-zero2*: $|c| < 1 \implies (\lambda n. c ^ n :: real) \longrightarrow 0$
 $\langle proof \rangle$

108.10 Limits of Functions

lemma *LIM-eq*: $f -a \rightarrow L = (\forall r > 0. \exists s > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < s \longrightarrow \text{norm } (f x - L) < r)$

for $a :: 'a :: \text{real-normed-vector}$ **and** $L :: 'b :: \text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *LIM-I*:

$(\bigwedge r. 0 < r \Longrightarrow \exists s > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < s \longrightarrow \text{norm } (f x - L) < r) \Longrightarrow f -a \rightarrow L$

for $a :: 'a :: \text{real-normed-vector}$ **and** $L :: 'b :: \text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *LIM-D*: $f -a \rightarrow L \Longrightarrow 0 < r \Longrightarrow \exists s > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < s \longrightarrow \text{norm } (f x - L) < r$

for $a :: 'a :: \text{real-normed-vector}$ **and** $L :: 'b :: \text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *LIM-offset*: $f -a \rightarrow L \Longrightarrow (\lambda x. f (x + k)) - (a - k) \rightarrow L$

for $a :: 'a :: \text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *LIM-offset-zero*: $f -a \rightarrow L \Longrightarrow (\lambda h. f (a + h)) - 0 \rightarrow L$

for $a :: 'a :: \text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *LIM-offset-zero-cancel*: $(\lambda h. f (a + h)) - 0 \rightarrow L \Longrightarrow f -a \rightarrow L$

for $a :: 'a :: \text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *LIM-offset-zero-iff*: $\text{NO-MATCH } 0 \ a \Longrightarrow f -a \rightarrow L \longleftrightarrow (\lambda h. f (a + h)) - 0 \rightarrow L$

for $f :: 'a :: \text{real-normed-vector} \Rightarrow -$
 $\langle \text{proof} \rangle$

lemma *tendsto-offset-zero-iff*:

fixes $f :: 'a :: \text{real-normed-vector} \Rightarrow -$

assumes $\text{NO-MATCH } 0 \ a \ a \in S \text{ open } S$

shows $(f \longrightarrow L) \text{ (at } a \text{ within } S) \longleftrightarrow ((\lambda h. f (a + h)) \longrightarrow L) \text{ (at } 0)$

$\langle \text{proof} \rangle$

lemma *LIM-zero*: $(f \longrightarrow l) \ F \Longrightarrow ((\lambda x. f x - l) \longrightarrow 0) \ F$

for $f :: 'a \Rightarrow 'b :: \text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *LIM-zero-cancel*:

fixes $f :: 'a \Rightarrow 'b :: \text{real-normed-vector}$

shows $((\lambda x. f x - l) \longrightarrow 0) \ F \Longrightarrow (f \longrightarrow l) \ F$

$\langle \text{proof} \rangle$

lemma *LIM-zero-iff*: $((\lambda x. f\ x - l) \longrightarrow 0) \ F = (f \longrightarrow l) \ F$
for $f :: 'a \Rightarrow 'b::\text{real-normed-vector}$
 $\langle \text{proof} \rangle$

lemma *LIM-imp-LIM*:
fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$
fixes $g :: 'a::\text{topological-space} \Rightarrow 'c::\text{real-normed-vector}$
assumes $f: f - a \rightarrow l$
and $le: \bigwedge x. x \neq a \implies \text{norm } (g\ x - m) \leq \text{norm } (f\ x - l)$
shows $g - a \rightarrow m$
 $\langle \text{proof} \rangle$

lemma *LIM-equal2*:
fixes $f\ g :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{topological-space}$
assumes $0 < R$
and $\bigwedge x. x \neq a \implies \text{norm } (x - a) < R \implies f\ x = g\ x$
shows $g - a \rightarrow l \implies f - a \rightarrow l$
 $\langle \text{proof} \rangle$

lemma *LIM-compose2*:
fixes $a :: 'a::\text{real-normed-vector}$
assumes $f: f - a \rightarrow b$
and $g: g - b \rightarrow c$
and $inj: \exists d > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < d \longrightarrow f\ x \neq b$
shows $(\lambda x. g\ (f\ x)) - a \rightarrow c$
 $\langle \text{proof} \rangle$

lemma *real-LIM-sandwich-zero*:
fixes $f\ g :: 'a::\text{topological-space} \Rightarrow \text{real}$
assumes $f: f - a \rightarrow 0$
and $1: \bigwedge x. x \neq a \implies 0 \leq g\ x$
and $2: \bigwedge x. x \neq a \implies g\ x \leq f\ x$
shows $g - a \rightarrow 0$
 $\langle \text{proof} \rangle$

108.11 Continuity

lemma *LIM-isCont-iff*: $(f - a \rightarrow f\ a) = ((\lambda h. f\ (a + h)) - 0 \rightarrow f\ a)$
for $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{topological-space}$
 $\langle \text{proof} \rangle$

lemma *isCont-iff*: $\text{isCont } f\ x = (\lambda h. f\ (x + h)) - 0 \rightarrow f\ x$
for $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{topological-space}$
 $\langle \text{proof} \rangle$

lemma *isCont-LIM-compose2*:
fixes $a :: 'a::\text{real-normed-vector}$
assumes f [unfolded *isCont-def*]: $\text{isCont } f\ a$
and $g: g - f\ a \rightarrow l$

and *inj*: $\exists d > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < d \longrightarrow f x \neq f a$
shows $(\lambda x. g (f x)) -a \rightarrow l$
<proof>

lemma *isCont-norm* [*simp*]: $\text{isCont } f a \Longrightarrow \text{isCont } (\lambda x. \text{norm } (f x)) a$
for $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-vector}$
<proof>

lemma *isCont-rabs* [*simp*]: $\text{isCont } f a \Longrightarrow \text{isCont } (\lambda x. |f x|) a$
for $f :: 'a::t2\text{-space} \Rightarrow \text{real}$
<proof>

lemma *isCont-add* [*simp*]: $\text{isCont } f a \Longrightarrow \text{isCont } g a \Longrightarrow \text{isCont } (\lambda x. f x + g x) a$
for $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{topological-monoid-add}$
<proof>

lemma *isCont-minus* [*simp*]: $\text{isCont } f a \Longrightarrow \text{isCont } (\lambda x. - f x) a$
for $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-vector}$
<proof>

lemma *isCont-diff* [*simp*]: $\text{isCont } f a \Longrightarrow \text{isCont } g a \Longrightarrow \text{isCont } (\lambda x. f x - g x) a$
for $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-vector}$
<proof>

lemma *isCont-mult* [*simp*]: $\text{isCont } f a \Longrightarrow \text{isCont } g a \Longrightarrow \text{isCont } (\lambda x. f x * g x) a$
for $f g :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-algebra}$
<proof>

lemma (**in** *bounded-linear*) *isCont*: $\text{isCont } g a \Longrightarrow \text{isCont } (\lambda x. f (g x)) a$
<proof>

lemma (**in** *bounded-bilinear*) *isCont*: $\text{isCont } f a \Longrightarrow \text{isCont } g a \Longrightarrow \text{isCont } (\lambda x. f x ** g x) a$
<proof>

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*]: $\text{isCont } f a \Longrightarrow \text{isCont } (\lambda x. f x ^ n) a$
for $f :: 'a::t2\text{-space} \Rightarrow 'b::\{\text{power}, \text{real-normed-algebra}\}$
<proof>

lemma *isCont-sum* [*simp*]: $\forall i \in A. \text{isCont } (f i) a \Longrightarrow \text{isCont } (\lambda x. \sum_{i \in A} f i x) a$
for $f :: 'a \Rightarrow 'b::t2\text{-space} \Rightarrow 'c::\text{topological-comm-monoid-add}$
<proof>

108.12 Uniform Continuity

lemma *uniformly-continuous-on-def*:

fixes $f :: 'a::\text{metric-space} \Rightarrow 'b::\text{metric-space}$

shows $\text{uniformly-continuous-on } s \ f \longleftrightarrow$

$(\forall e>0. \exists d>0. \forall x\in s. \forall x'\in s. \text{dist } x' \ x < d \longrightarrow \text{dist } (f \ x') \ (f \ x) < e)$

$\langle \text{proof} \rangle$

abbreviation $\text{isUCont} :: ['a::\text{metric-space} \Rightarrow 'b::\text{metric-space}] \Rightarrow \text{bool}$

where $\text{isUCont } f \equiv \text{uniformly-continuous-on UNIV } f$

lemma *isUCont-def*: $\text{isUCont } f \longleftrightarrow (\forall r>0. \exists s>0. \forall x \ y. \text{dist } x \ y < s \longrightarrow \text{dist } (f \ x) \ (f \ y) < r)$

$\langle \text{proof} \rangle$

lemma *isUCont-isCont*: $\text{isUCont } f \Longrightarrow \text{isCont } f \ x$

$\langle \text{proof} \rangle$

lemma *uniformly-continuous-on-Cauchy*:

fixes $f :: 'a::\text{metric-space} \Rightarrow 'b::\text{metric-space}$

assumes $\text{uniformly-continuous-on } S \ f \ \text{Cauchy } X \ \bigwedge n. X \ n \in S$

shows $\text{Cauchy } (\lambda n. f \ (X \ n))$

$\langle \text{proof} \rangle$

lemma *isUCont-Cauchy*: $\text{isUCont } f \Longrightarrow \text{Cauchy } X \Longrightarrow \text{Cauchy } (\lambda n. f \ (X \ n))$

$\langle \text{proof} \rangle$

lemma (in *bounded-linear*) *isUCont*: $\text{isUCont } f$

$\langle \text{proof} \rangle$

lemma (in *bounded-linear*) *Cauchy*: $\text{Cauchy } X \Longrightarrow \text{Cauchy } (\lambda n. f \ (X \ n))$

$\langle \text{proof} \rangle$

lemma *LIM-less-bound*:

fixes $f :: \text{real} \Rightarrow \text{real}$

assumes $\text{ev}: b < x \ \forall \ x' \in \{ b <..< x \}. \ 0 \leq f \ x' \ \text{and} \ \text{isCont } f \ x$

shows $0 \leq f \ x$

$\langle \text{proof} \rangle$

108.13 Nested Intervals and Bisection – Needed for Compactness

lemma *nested-sequence-unique*:

assumes $\forall n. f \ n \leq f \ (\text{Suc } n) \ \forall n. g \ (\text{Suc } n) \leq g \ n \ \forall n. f \ n \leq g \ n \ (\lambda n. f \ n - g \ n) \longrightarrow 0$

shows $\exists l::\text{real}. ((\forall n. f \ n \leq l) \wedge f \longrightarrow l) \wedge ((\forall n. l \leq g \ n) \wedge g \longrightarrow l)$

$\langle \text{proof} \rangle$

lemma *Bolzano*[consumes 1, case-names trans local]:

fixes $P :: \text{real} \Rightarrow \text{real} \Rightarrow \text{bool}$

assumes $[arith]: a \leq b$
and $trans: \bigwedge a \ b \ c. P \ a \ b \implies P \ b \ c \implies a \leq b \implies b \leq c \implies P \ a \ c$
and $local: \bigwedge x. a \leq x \implies x \leq b \implies \exists d > 0. \forall a \ b. a \leq x \wedge x \leq b \wedge b - a < d$
 $\longrightarrow P \ a \ b$
shows $P \ a \ b$
 $\langle proof \rangle$

lemma *compact-Icc*[simp, intro]: $compact \ \{a \ .. \ b :: real\}$
 $\langle proof \rangle$

lemma *continuous-image-closed-interval*:
fixes $a \ b$ **and** $f :: real \Rightarrow real$
defines $S \equiv \{a..b\}$
assumes $a \leq b$ **and** $f: continuous-on \ S \ f$
shows $\exists c \ d. f \ S = \{c..d\} \wedge c \leq d$
 $\langle proof \rangle$

lemma *open-Collect-positive*:
fixes $f :: 'a :: topological-space \Rightarrow real$
assumes $f: continuous-on \ s \ f$
shows $\exists A. open \ A \wedge A \cap s = \{x \in s. 0 < f \ x\}$
 $\langle proof \rangle$

lemma *open-Collect-less-Int*:
fixes $f \ g :: 'a :: topological-space \Rightarrow real$
assumes $f: continuous-on \ s \ f$
and $g: continuous-on \ s \ g$
shows $\exists A. open \ A \wedge A \cap s = \{x \in s. f \ x < g \ x\}$
 $\langle proof \rangle$

108.14 Boundedness of continuous functions

By bisection, function continuous on closed interval is bounded above

lemma *isCont-eq-Ub*:
fixes $f :: real \Rightarrow 'a :: linorder-topology$
shows $a \leq b \implies \forall x :: real. a \leq x \wedge x \leq b \longrightarrow isCont \ f \ x \implies$
 $\exists M. (\forall x. a \leq x \wedge x \leq b \longrightarrow f \ x \leq M) \wedge (\exists x. a \leq x \wedge x \leq b \wedge f \ x = M)$
 $\langle proof \rangle$

lemma *isCont-eq-Lb*:
fixes $f :: real \Rightarrow 'a :: linorder-topology$
shows $a \leq b \implies \forall x. a \leq x \wedge x \leq b \longrightarrow isCont \ f \ x \implies$
 $\exists M. (\forall x. a \leq x \wedge x \leq b \longrightarrow M \leq f \ x) \wedge (\exists x. a \leq x \wedge x \leq b \wedge f \ x = M)$
 $\langle proof \rangle$

lemma *isCont-bounded*:
fixes $f :: real \Rightarrow 'a :: linorder-topology$
shows $a \leq b \implies \forall x. a \leq x \wedge x \leq b \longrightarrow isCont \ f \ x \implies \exists M. \forall x. a \leq x \wedge x \leq$

$b \longrightarrow f\ x \leq M$
 $\langle proof \rangle$

lemma *isCont-has-Ub*:

fixes $f :: real \Rightarrow 'a::linorder-topology$
shows $a \leq b \implies \forall x. a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x \implies$
 $\exists M. (\forall x. a \leq x \wedge x \leq b \longrightarrow f\ x \leq M) \wedge (\forall N. N < M \longrightarrow (\exists x. a \leq x \wedge x$
 $\leq b \wedge N < f\ x))$
 $\langle proof \rangle$

lemma *isCont-Lb-Ub*:

fixes $f :: real \Rightarrow real$
assumes $a \leq b \ \forall x. a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x$
shows $\exists L\ M. (\forall x. a \leq x \wedge x \leq b \longrightarrow L \leq f\ x \wedge f\ x \leq M) \wedge$
 $(\forall y. L \leq y \wedge y \leq M \longrightarrow (\exists x. a \leq x \wedge x \leq b \wedge (f\ x = y)))$
 $\langle proof \rangle$

Continuity of inverse function.

lemma *isCont-inverse-function*:

fixes $f\ g :: real \Rightarrow real$
assumes $d: 0 < d$
and inj: $\bigwedge z. |z - x| \leq d \implies g\ (f\ z) = z$
and cont: $\bigwedge z. |z - x| \leq d \implies isCont\ f\ z$
shows $isCont\ g\ (f\ x)$
 $\langle proof \rangle$

lemma *isCont-inverse-function2*:

fixes $f\ g :: real \Rightarrow real$
shows
 $\llbracket a < x; x < b; \bigwedge z. \llbracket a \leq z; z \leq b \rrbracket \implies g\ (f\ z) = z; \bigwedge z. \llbracket a \leq z; z \leq b \rrbracket \implies isCont\ f\ z \rrbracket \implies isCont\ g\ (f\ x)$
 $\langle proof \rangle$

Bartle/Sherbert: Introduction to Real Analysis, Theorem 4.2.9, p. 110.

lemma *LIM-fun-gt-zero*: $f -c \rightarrow l \implies 0 < l \implies \exists r. 0 < r \wedge (\forall x. x \neq c \wedge |c - x| < r \longrightarrow 0 < f\ x)$
for $f :: real \Rightarrow real$
 $\langle proof \rangle$

lemma *LIM-fun-less-zero*: $f -c \rightarrow l \implies l < 0 \implies \exists r. 0 < r \wedge (\forall x. x \neq c \wedge |c - x| < r \longrightarrow f\ x < 0)$
for $f :: real \Rightarrow real$
 $\langle proof \rangle$

lemma *LIM-fun-not-zero*: $f -c \rightarrow l \implies l \neq 0 \implies \exists r. 0 < r \wedge (\forall x. x \neq c \wedge |c - x| < r \longrightarrow f\ x \neq 0)$
for $f :: real \Rightarrow real$
 $\langle proof \rangle$

lemma *Lim-topological:*

$(f \longrightarrow l) \text{ net} \longleftrightarrow$
 $\text{trivial-limit net} \vee (\forall S. \text{ open } S \longrightarrow l \in S \longrightarrow \text{eventually } (\lambda x. f x \in S) \text{ net})$
 $\langle \text{proof} \rangle$

lemma *eventually-within-Un:*

$\text{eventually } P \text{ (at } x \text{ within } (s \cup t)) \longleftrightarrow$
 $\text{eventually } P \text{ (at } x \text{ within } s) \wedge \text{eventually } P \text{ (at } x \text{ within } t)$
 $\langle \text{proof} \rangle$

lemma *Lim-within-Un:*

$(f \longrightarrow l) \text{ (at } x \text{ within } (s \cup t)) \longleftrightarrow$
 $(f \longrightarrow l) \text{ (at } x \text{ within } s) \wedge (f \longrightarrow l) \text{ (at } x \text{ within } t)$
 $\langle \text{proof} \rangle$

end

theory *Inequalities*

imports *Real-Vector-Spaces*

begin

lemma *Chebyshev-sum-upper:*

fixes $a b :: \text{nat} \Rightarrow 'a :: \text{linordered-idom}$
assumes $\bigwedge i j. i \leq j \implies j < n \implies a\ i \leq a\ j$
assumes $\bigwedge i j. i \leq j \implies j < n \implies b\ i \geq b\ j$
shows $\text{of-nat } n * (\sum k=0..<n. a\ k * b\ k) \leq (\sum k=0..<n. a\ k) * (\sum k=0..<n. b\ k)$
 $\langle \text{proof} \rangle$

lemma *Chebyshev-sum-upper-nat:*

fixes $a b :: \text{nat} \Rightarrow \text{nat}$
shows $(\bigwedge i j. \llbracket i \leq j; j < n \rrbracket \implies a\ i \leq a\ j) \implies$
 $(\bigwedge i j. \llbracket i \leq j; j < n \rrbracket \implies b\ i \geq b\ j) \implies$
 $n * (\sum i=0..<n. a\ i * b\ i) \leq (\sum i=0..<n. a\ i) * (\sum i=0..<n. b\ i)$
 $\langle \text{proof} \rangle$

end

109 Infinite Series

theory *Series*

imports *Limits Inequalities*

begin

109.1 Definition of infinite summability

definition $\text{sums} :: (\text{nat} \Rightarrow 'a :: \{\text{topological-space, comm-monoid-add}\}) \Rightarrow 'a \Rightarrow \text{bool}$

(**infixr** $\langle \text{sums} \rangle$ 80)
where $f \text{ sums } s \longleftrightarrow (\lambda n. \sum i < n. f i) \longrightarrow s$

definition $\text{summable} :: (\text{nat} \Rightarrow 'a :: \{\text{topological-space, comm-monoid-add}\}) \Rightarrow \text{bool}$
where $\text{summable } f \longleftrightarrow (\exists s. f \text{ sums } s)$

definition $\text{suminf} :: (\text{nat} \Rightarrow 'a :: \{\text{topological-space, comm-monoid-add}\}) \Rightarrow 'a$
 (**binder** $\langle \sum \rangle$ 10)
where $\text{suminf } f = (\text{THE } s. f \text{ sums } s)$

Variants of the definition

lemma $\text{sums-def}'$: $f \text{ sums } s \longleftrightarrow (\lambda n. \sum i = 0..n. f i) \longrightarrow s$
 $\langle \text{proof} \rangle$

lemma sums-def-le : $f \text{ sums } s \longleftrightarrow (\lambda n. \sum i \leq n. f i) \longrightarrow s$
 $\langle \text{proof} \rangle$

lemma $\text{bounded-imp-summable}$:
fixes $a :: \text{nat} \Rightarrow 'a :: \{\text{conditionally-complete-linorder, linorder-topology, linordered-comm-semiring-strict}\}$
assumes 0 : $\bigwedge n. a n \geq 0$ **and** bounded : $\bigwedge n. (\sum k \leq n. a k) \leq B$
shows $\text{summable } a$
 $\langle \text{proof} \rangle$

109.2 Infinite summability on topological monoids

lemma $\text{sums-subst}[\text{trans}]$: $f = g \Longrightarrow g \text{ sums } z \Longrightarrow f \text{ sums } z$
 $\langle \text{proof} \rangle$

lemma sums-cong : $(\bigwedge n. f n = g n) \Longrightarrow f \text{ sums } c \longleftrightarrow g \text{ sums } c$
 $\langle \text{proof} \rangle$

lemma sums-summable : $f \text{ sums } l \Longrightarrow \text{summable } f$
 $\langle \text{proof} \rangle$

lemma $\text{summable-iff-convergent}$: $\text{summable } f \longleftrightarrow \text{convergent } (\lambda n. \sum i < n. f i)$
 $\langle \text{proof} \rangle$

lemma $\text{summable-iff-convergent}'$: $\text{summable } f \longleftrightarrow \text{convergent } (\lambda n. \text{sum } f \{..n\})$
 $\langle \text{proof} \rangle$

lemma suminf-eq-lim : $\text{suminf } f = \text{lim } (\lambda n. \sum i < n. f i)$
 $\langle \text{proof} \rangle$

lemma $\text{sums-zero}[\text{simp, intro}]$: $(\lambda n. 0) \text{ sums } 0$
 $\langle \text{proof} \rangle$

lemma $\text{summable-zero}[\text{simp, intro}]$: $\text{summable } (\lambda n. 0)$
 $\langle \text{proof} \rangle$

lemma *sums-group*: $f \text{ sums } s \implies 0 < k \implies (\lambda n. \text{sum } f \{n * k ..< n * k + k\}) \text{ sums } s$
 <proof>

lemma *suminf-cong*: $(\bigwedge n. f \ n = g \ n) \implies \text{suminf } f = \text{suminf } g$
 <proof>

lemma *summable-cong*:
 fixes $f \ g :: \text{nat} \Rightarrow 'a :: \text{real-normed-vector}$
 assumes *eventually* $(\lambda x. f \ x = g \ x)$ *sequentially*
 shows *summable* $f = \text{summable } g$
 <proof>

lemma *sums-finite*:
 assumes [*simp*]: *finite* N
 and $f: \bigwedge n. n \notin N \implies f \ n = 0$
 shows *f sums* $(\sum_{n \in N}. f \ n)$
 <proof>

corollary *sums-0*: $(\bigwedge n. f \ n = 0) \implies (f \text{ sums } 0)$
 <proof>

lemma *summable-finite*: *finite* $N \implies (\bigwedge n. n \notin N \implies f \ n = 0) \implies \text{summable } f$
 <proof>

lemma *sums-If-finite-set*: *finite* $A \implies (\lambda r. \text{if } r \in A \text{ then } f \ r \text{ else } 0) \text{ sums } (\sum_{r \in A}. f \ r)$
 <proof>

lemma *summable-If-finite-set*[*simp*, *intro*]: *finite* $A \implies \text{summable } (\lambda r. \text{if } r \in A \text{ then } f \ r \text{ else } 0)$
 <proof>

lemma *sums-If-finite*: *finite* $\{r. P \ r\} \implies (\lambda r. \text{if } P \ r \text{ then } f \ r \text{ else } 0) \text{ sums } (\sum_{r \mid P \ r}. f \ r)$
 <proof>

lemma *summable-If-finite*[*simp*, *intro*]: *finite* $\{r. P \ r\} \implies \text{summable } (\lambda r. \text{if } P \ r \text{ then } f \ r \text{ else } 0)$
 <proof>

lemma *sums-single*: $(\lambda r. \text{if } r = i \text{ then } f \ r \text{ else } 0) \text{ sums } f \ i$
 <proof>

lemma *summable-single*[*simp*, *intro*]: *summable* $(\lambda r. \text{if } r = i \text{ then } f \ r \text{ else } 0)$
 <proof>

context
 fixes $f :: \text{nat} \Rightarrow 'a :: \{t2\text{-space}, \text{comm-monoid-add}\}$

begin

lemma *summable-sums[intro]*: $\text{summable } f \implies f \text{ sums } (\text{suminf } f)$
 ⟨proof⟩

lemma *summable-LIMSEQ*: $\text{summable } f \implies (\lambda n. \sum_{i < n}. f \ i) \longrightarrow \text{suminf } f$
 ⟨proof⟩

lemma *summable-LIMSEQ'*: $\text{summable } f \implies (\lambda n. \sum_{i \leq n}. f \ i) \longrightarrow \text{suminf } f$
 ⟨proof⟩

lemma *sums-unique*: $f \text{ sums } s \implies s = \text{suminf } f$
 ⟨proof⟩

lemma *sums-iff*: $f \text{ sums } x \longleftrightarrow \text{summable } f \wedge \text{suminf } f = x$
 ⟨proof⟩

lemma *summable-sums-iff*: $\text{summable } f \longleftrightarrow f \text{ sums } \text{suminf } f$
 ⟨proof⟩

lemma *sums-unique2*: $f \text{ sums } a \implies f \text{ sums } b \implies a = b$
for $a \ b :: 'a$
 ⟨proof⟩

lemma *sums-Uniq*: $\exists_{\leq 1} a. f \text{ sums } a$
for $a \ b :: 'a$
 ⟨proof⟩

lemma *suminf-finite*:
assumes N : *finite* N
and f : $\bigwedge n. n \notin N \implies f \ n = 0$
shows $\text{suminf } f = (\sum_{n \in N}. f \ n)$
 ⟨proof⟩

end

lemma *suminf-zero[simp]*: $\text{suminf } (\lambda n. 0 :: 'a :: \{t2\text{-space}, \text{comm-monoid-add}\}) = 0$
 ⟨proof⟩

109.3 Infinite summability on ordered, topological monoids

lemma *sums-le*: $(\bigwedge n. f \ n \leq g \ n) \implies f \text{ sums } s \implies g \text{ sums } t \implies s \leq t$
for $f \ g :: \text{nat} \Rightarrow 'a :: \{\text{ordered-comm-monoid-add}, \text{linorder-topology}\}$
 ⟨proof⟩

context

fixes $f :: \text{nat} \Rightarrow 'a :: \{\text{ordered-comm-monoid-add}, \text{linorder-topology}\}$
begin

lemma *suminf-le*: $(\bigwedge n. f\ n \leq g\ n) \implies \text{summable } f \implies \text{summable } g \implies \text{suminf } f \leq \text{suminf } g$
 ⟨proof⟩

lemma *sum-le-suminf*:
 shows $\text{summable } f \implies \text{finite } I \implies (\bigwedge n. n \in - I \implies 0 \leq f\ n) \implies \text{sum } f\ I \leq \text{suminf } f$
 ⟨proof⟩

lemma *suminf-nonneg*: $\text{summable } f \implies (\bigwedge n. 0 \leq f\ n) \implies 0 \leq \text{suminf } f$
 ⟨proof⟩

lemma *suminf-le-const*: $\text{summable } f \implies (\bigwedge n. \text{sum } f\ \{..
 ⟨proof⟩$

lemma *suminf-eq-zero-iff*:
 assumes $\text{summable } f$ and $\text{pos: } \bigwedge n. 0 \leq f\ n$
 shows $\text{suminf } f = 0 \longleftrightarrow (\forall n. f\ n = 0)$
 ⟨proof⟩

lemma *suminf-pos-iff*: $\text{summable } f \implies (\bigwedge n. 0 \leq f\ n) \implies 0 < \text{suminf } f \longleftrightarrow (\exists i. 0 < f\ i)$
 ⟨proof⟩

lemma *suminf-pos2*:
 assumes $\text{summable } f \wedge \bigwedge n. 0 \leq f\ n \wedge 0 < f\ i$
 shows $0 < \text{suminf } f$
 ⟨proof⟩

lemma *suminf-pos*: $\text{summable } f \implies (\bigwedge n. 0 < f\ n) \implies 0 < \text{suminf } f$
 ⟨proof⟩

end

context

fixes $f :: \text{nat} \Rightarrow 'a::\{\text{ordered-cancel-comm-monoid-add}, \text{linorder-topology}\}$
begin

lemma *sum-less-suminf2*:
 $\text{summable } f \implies (\bigwedge m. m \geq n \implies 0 \leq f\ m) \implies n \leq i \implies 0 < f\ i \implies \text{sum } f\ \{..
 ⟨proof⟩$

lemma *sum-less-suminf*: $\text{summable } f \implies (\bigwedge m. m \geq n \implies 0 < f\ m) \implies \text{sum } f\ \{..
 ⟨proof⟩$

end

lemma *summableI-nonneg-bounded*:

fixes $f :: \text{nat} \Rightarrow 'a::\{\text{ordered-comm-monoid-add}, \text{linorder-topology}, \text{conditionally-complete-linorder}\}$
assumes $\text{pos}[\text{simp}]: \bigwedge n. 0 \leq f\ n$
and $\text{le}: \bigwedge n. (\sum_{i < n}. f\ i) \leq x$
shows *summable* f
 $\langle \text{proof} \rangle$

lemma *summableI[intro, simp]: summable* f

for $f :: \text{nat} \Rightarrow 'a::\{\text{canonically-ordered-monoid-add}, \text{linorder-topology}, \text{complete-linorder}\}$
 $\langle \text{proof} \rangle$

lemma *suminf-eq-SUP-real*:

assumes $X: \text{summable } X \bigwedge i. 0 \leq X\ i$ **shows** $\text{suminf } X = (\text{SUP } i. \sum_{n < i}. X\ n::\text{real})$
 $\langle \text{proof} \rangle$

109.4 Infinite summability on topological monoids

context

fixes $f\ g :: \text{nat} \Rightarrow 'a::\{\text{t2-space}, \text{topological-comm-monoid-add}\}$

begin

lemma *sums-Suc*:

assumes $(\lambda n. f\ (Suc\ n)) \text{ sums } l$
shows $f \text{ sums } (l + f\ 0)$

$\langle \text{proof} \rangle$

lemma *sums-add*: $f \text{ sums } a \Longrightarrow g \text{ sums } b \Longrightarrow (\lambda n. f\ n + g\ n) \text{ sums } (a + b)$

$\langle \text{proof} \rangle$

lemma *summable-add*: $\text{summable } f \Longrightarrow \text{summable } g \Longrightarrow \text{summable } (\lambda n. f\ n + g\ n)$

$\langle \text{proof} \rangle$

lemma *suminf-add*: $\text{summable } f \Longrightarrow \text{summable } g \Longrightarrow \text{suminf } f + \text{suminf } g = (\sum n. f\ n + g\ n)$

$\langle \text{proof} \rangle$

end

context

fixes $f :: 'i \Rightarrow \text{nat} \Rightarrow 'a::\{\text{t2-space}, \text{topological-comm-monoid-add}\}$

and $I :: 'i \text{ set}$

begin

lemma *sums-sum*: $(\bigwedge i. i \in I \Longrightarrow (f\ i) \text{ sums } (x\ i)) \Longrightarrow (\lambda n. \sum_{i \in I}. f\ i\ n) \text{ sums } (\sum_{i \in I}. x\ i)$

$\langle \text{proof} \rangle$

lemma *suminf-sum*: $(\bigwedge i. i \in I \implies \text{summable } (f\ i)) \implies (\sum n. \sum i \in I. f\ i\ n) = (\sum i \in I. \sum n. f\ i\ n)$
 <proof>

lemma *summable-sum*: $(\bigwedge i. i \in I \implies \text{summable } (f\ i)) \implies \text{summable } (\lambda n. \sum i \in I. f\ i\ n)$
 <proof>

end

lemma *sums-If-finite-set'*:

fixes $f\ g :: \text{nat} \Rightarrow 'a::\{\text{t2-space, topological-ab-group-add}\}$

assumes $g \text{ sums } S \text{ and finite } A \text{ and } S' = S + (\sum n \in A. f\ n - g\ n)$

shows $(\lambda n. \text{if } n \in A \text{ then } f\ n \text{ else } g\ n) \text{ sums } S'$

<proof>

109.5 Infinite summability on real normed vector spaces

context

fixes $f :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$

begin

lemma *sums-Suc-iff*: $(\lambda n. f\ (Suc\ n)) \text{ sums } s \longleftrightarrow f \text{ sums } (s + f\ 0)$
 <proof>

lemma *summable-Suc-iff*: $\text{summable } (\lambda n. f\ (Suc\ n)) = \text{summable } f$
 <proof>

lemma *sums-Suc-imp*: $f\ 0 = 0 \implies (\lambda n. f\ (Suc\ n)) \text{ sums } s \implies (\lambda n. f\ n) \text{ sums } s$
 <proof>

end

context

fixes $f :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$

begin

lemma *sums-diff*: $f \text{ sums } a \implies g \text{ sums } b \implies (\lambda n. f\ n - g\ n) \text{ sums } (a - b)$
 <proof>

lemma *summable-diff*: $\text{summable } f \implies \text{summable } g \implies \text{summable } (\lambda n. f\ n - g\ n)$
 <proof>

lemma *suminf-diff*: $\text{summable } f \implies \text{summable } g \implies \text{suminf } f - \text{suminf } g = (\sum n. f\ n - g\ n)$
 <proof>

lemma *sums-minus*: $f \text{ sums } a \implies (\lambda n. - f n) \text{ sums } (- a)$
 ⟨proof⟩

lemma *summable-minus*: $\text{summable } f \implies \text{summable } (\lambda n. - f n)$
 ⟨proof⟩

lemma *suminf-minus*: $\text{summable } f \implies (\sum n. - f n) = - (\sum n. f n)$
 ⟨proof⟩

lemma *sums-iff-shift*: $(\lambda i. f (i + n)) \text{ sums } s \longleftrightarrow f \text{ sums } (s + (\sum i < n. f i))$
 ⟨proof⟩

corollary *sums-iff-shift'*: $(\lambda i. f (i + n)) \text{ sums } (s - (\sum i < n. f i)) \longleftrightarrow f \text{ sums } s$
 ⟨proof⟩

lemma *sums-zero-iff-shift*:
 assumes $\bigwedge i. i < n \implies f i = 0$
 shows $(\lambda i. f (i+n)) \text{ sums } s \longleftrightarrow (\lambda i. f i) \text{ sums } s$
 ⟨proof⟩

lemma *summable-iff-shift [simp]*: $\text{summable } (\lambda n. f (n + k)) \longleftrightarrow \text{summable } f$
 ⟨proof⟩

lemma *sums-split-initial-segment*: $f \text{ sums } s \implies (\lambda i. f (i + n)) \text{ sums } (s - (\sum i < n. f i))$
 ⟨proof⟩

lemma *summable-ignore-initial-segment*: $\text{summable } f \implies \text{summable } (\lambda n. f(n + k))$
 ⟨proof⟩

lemma *suminf-minus-initial-segment*: $\text{summable } f \implies (\sum n. f (n + k)) = (\sum n. f n) - (\sum i < k. f i)$
 ⟨proof⟩

lemma *suminf-split-initial-segment*: $\text{summable } f \implies \text{suminf } f = (\sum n. f(n + k)) + (\sum i < k. f i)$
 ⟨proof⟩

lemma *suminf-split-head*: $\text{summable } f \implies (\sum n. f (Suc n)) = \text{suminf } f - f 0$
 ⟨proof⟩

lemma *suminf-exist-split*:
 fixes $r :: \text{real}$
 assumes $0 < r$ and $\text{summable } f$
 shows $\exists N. \forall n \geq N. \text{norm } (\sum i. f (i + n)) < r$
 ⟨proof⟩

lemma *summable-LIMSEQ-zero*:

assumes *summable f* **shows** $f \longrightarrow 0$
 ⟨*proof*⟩

lemma *summable-imp-convergent*: *summable f* \implies *convergent f*
 ⟨*proof*⟩

lemma *summable-imp-Bseq*: *summable f* \implies *Bseq f*
 ⟨*proof*⟩

end

lemma *summable-minus-iff*: *summable* $(\lambda n. - f n) \longleftrightarrow$ *summable f*
for $f :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$
 ⟨*proof*⟩

lemma (**in** *bounded-linear*) *sums*: $(\lambda n. X n) \text{ sums } a \implies (\lambda n. f (X n)) \text{ sums } (f a)$
 ⟨*proof*⟩

lemma (**in** *bounded-linear*) *summable*: *summable* $(\lambda n. X n) \implies$ *summable* $(\lambda n. f (X n))$
 ⟨*proof*⟩

lemma (**in** *bounded-linear*) *suminf*: *summable* $(\lambda n. X n) \implies f (\sum n. X n) =$
 $(\sum n. f (X n))$
 ⟨*proof*⟩

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* $(\lambda -. c) \longleftrightarrow c = 0$
for $c :: 'a::\text{real-normed-vector}$
 ⟨*proof*⟩

109.6 Infinite summability on real normed algebras

context

fixes $f :: \text{nat} \Rightarrow 'a::\text{real-normed-algebra}$

begin

lemma *sums-mult*: $f \text{ sums } a \implies (\lambda n. c * f n) \text{ sums } (c * a)$

<proof>

lemma *summable-mult*: $\text{summable } f \implies \text{summable } (\lambda n. c * f n)$
<proof>

lemma *suminf-mult*: $\text{summable } f \implies \text{suminf } (\lambda n. c * f n) = c * \text{suminf } f$
<proof>

lemma *sums-mult2*: $f \text{ sums } a \implies (\lambda n. f n * c) \text{ sums } (a * c)$
<proof>

lemma *summable-mult2*: $\text{summable } f \implies \text{summable } (\lambda n. f n * c)$
<proof>

lemma *suminf-mult2*: $\text{summable } f \implies \text{suminf } f * c = (\sum n. f n * c)$
<proof>

end

lemma *sums-mult-iff*:
fixes $f :: \text{nat} \Rightarrow 'a :: \{\text{real-normed-algebra}, \text{field}\}$
assumes $c \neq 0$
shows $(\lambda n. c * f n) \text{ sums } (c * d) \longleftrightarrow f \text{ sums } d$
<proof>

lemma *sums-mult2-iff*:
fixes $f :: \text{nat} \Rightarrow 'a :: \{\text{real-normed-algebra}, \text{field}\}$
assumes $c \neq 0$
shows $(\lambda n. f n * c) \text{ sums } (d * c) \longleftrightarrow f \text{ sums } d$
<proof>

lemma *sums-of-real-iff*:
 $(\lambda n. \text{of-real } (f n) :: 'a :: \text{real-normed-div-algebra}) \text{ sums of-real } c \longleftrightarrow f \text{ sums } c$
<proof>

109.7 Infinite summability on real normed fields

context

fixes $c :: 'a :: \text{real-normed-field}$

begin

lemma *sums-divide*: $f \text{ sums } a \implies (\lambda n. f n / c) \text{ sums } (a / c)$
<proof>

lemma *summable-divide*: $\text{summable } f \implies \text{summable } (\lambda n. f n / c)$
<proof>

lemma *suminf-divide*: $\text{summable } f \implies \text{suminf } (\lambda n. f n / c) = \text{suminf } f / c$
<proof>

lemma *summable-inverse-divide*: $\text{summable } (\text{inverse} \circ f) \implies \text{summable } (\lambda n. c / f\ n)$
 ⟨proof⟩

lemma *sums-mult-D*: $(\lambda n. c * f\ n) \text{ sums } a \implies c \neq 0 \implies f \text{ sums } (a/c)$
 ⟨proof⟩

lemma *summable-mult-D*: $\text{summable } (\lambda n. c * f\ n) \implies c \neq 0 \implies \text{summable } f$
 ⟨proof⟩

Sum of a geometric progression.

lemma *geometric-sums*:
assumes $\text{norm } c < 1$
shows $(\lambda n. c^n) \text{ sums } (1 / (1 - c))$
 ⟨proof⟩

lemma *summable-geometric*: $\text{norm } c < 1 \implies \text{summable } (\lambda n. c^n)$
 ⟨proof⟩

lemma *suminf-geometric*: $\text{norm } c < 1 \implies \text{suminf } (\lambda n. c^n) = 1 / (1 - c)$
 ⟨proof⟩

lemma *summable-geometric-iff* [simp]: $\text{summable } (\lambda n. c^n) \longleftrightarrow \text{norm } c < 1$
 ⟨proof⟩

end

Biconditional versions for constant factors

context
fixes $c :: 'a::\text{real-normed-field}$
begin

lemma *summable-cmult-iff* [simp]: $\text{summable } (\lambda n. c * f\ n) \longleftrightarrow c=0 \vee \text{summable } f$
 ⟨proof⟩

lemma *summable-divide-iff* [simp]: $\text{summable } (\lambda n. f\ n / c) \longleftrightarrow c=0 \vee \text{summable } f$
 ⟨proof⟩

end

lemma *power-half-series*: $(\lambda n. (1/2::\text{real})^{\text{Suc } n}) \text{ sums } 1$
 ⟨proof⟩

109.8 Telescoping

lemma *telescope-sums*:

```

fixes  $c :: 'a::\text{real-normed-vector}$ 
assumes  $f \longrightarrow c$ 
shows  $(\lambda n. f (Suc\ n) - f\ n)\ \text{sums}\ (c - f\ 0)$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma telescope-sums':
fixes  $c :: 'a::\text{real-normed-vector}$ 
assumes  $f \longrightarrow c$ 
shows  $(\lambda n. f\ n - f (Suc\ n))\ \text{sums}\ (f\ 0 - c)$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma telescope-summable:
fixes  $c :: 'a::\text{real-normed-vector}$ 
assumes  $f \longrightarrow c$ 
shows summable  $(\lambda n. f (Suc\ n) - f\ n)$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma telescope-summable':
fixes  $c :: 'a::\text{real-normed-vector}$ 
assumes  $f \longrightarrow c$ 
shows summable  $(\lambda n. f\ n - f (Suc\ n))$ 
 $\langle \text{proof} \rangle$ 

```

109.9 Infinite summability on Banach spaces

Cauchy-type criterion for convergence of series (c.f. Harrison).

```

lemma summable-Cauchy: summable  $f \longleftrightarrow (\forall e>0. \exists N. \forall m\geq N. \forall n. \text{norm}\ (\text{sum}\ f\ \{m..<n\}) < e)$  (is - = ?rhs)
for  $f :: \text{nat} \Rightarrow 'a::\text{banach}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma summable-Cauchy':
fixes  $f :: \text{nat} \Rightarrow 'a :: \text{banach}$ 
assumes ev: eventually  $(\lambda m. \forall n\geq m. \text{norm}\ (\text{sum}\ f\ \{m..<n\}) \leq g\ m)$  sequentially
assumes  $g0$ :  $g \longrightarrow 0$ 
shows summable  $f$ 
 $\langle \text{proof} \rangle$ 

```

```

context
fixes  $f :: \text{nat} \Rightarrow 'a::\text{banach}$ 
begin

```

Absolute convergence implies normal convergence.

```

lemma summable-norm-cancel: summable  $(\lambda n. \text{norm}\ (f\ n)) \implies \text{summable}\ f$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma summable-norm: summable  $(\lambda n. \text{norm}\ (f\ n)) \implies \text{norm}\ (\text{suminf}\ f) \leq (\sum n. \text{norm}\ (f\ n))$ 
 $\langle \text{proof} \rangle$ 

```

Comparison tests.

lemma *summable-comparison-test*:

assumes $fg: \exists N. \forall n \geq N. \text{norm } (f\ n) \leq g\ n$ **and** $g: \text{summable } g$
shows $\text{summable } f$

<proof>

lemma *summable-comparison-test-ev*:

eventually $(\lambda n. \text{norm } (f\ n) \leq g\ n)$ *sequentially* $\implies \text{summable } g \implies \text{summable } f$

<proof>

A better argument order.

lemma *summable-comparison-test'*: $\text{summable } g \implies (\bigwedge n. n \geq N \implies \text{norm } (f\ n) \leq g\ n) \implies \text{summable } f$

<proof>

109.10 The Ratio Test

lemma *summable-ratio-test*:

assumes $c < 1 \wedge n. n \geq N \implies \text{norm } (f\ (\text{Suc } n)) \leq c * \text{norm } (f\ n)$
shows $\text{summable } f$

<proof>

end

Application to convergence of the log function

lemma *norm-summable-ln-series*:

fixes $z :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
assumes $\text{norm } z < 1$
shows $\text{summable } (\lambda n. \text{norm } (z \wedge^n / \text{of-nat } n))$

<proof>

Relations among convergence and absolute convergence for power series.

lemma *Abel-lemma*:

fixes $a :: \text{nat} \Rightarrow 'a :: \text{real-normed-vector}$
assumes $r: 0 \leq r$
and $r0: r < r0$
and $M: \bigwedge n. \text{norm } (a\ n) * r0 \wedge^n \leq M$
shows $\text{summable } (\lambda n. \text{norm } (a\ n) * r \wedge^n)$

<proof>

Summability of geometric series for real algebras.

lemma *complete-algebra-summable-geometric*:

fixes $x :: 'a :: \{\text{real-normed-algebra-1}, \text{banach}\}$
assumes $\text{norm } x < 1$
shows $\text{summable } (\lambda n. x \wedge^n)$

<proof>

109.11 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 $a\ b :: \text{nat} \Rightarrow 'a :: \{\text{real-normed-algebra}, \text{banach}\}$
assumes $a: \text{summable } (\lambda k. \text{norm } (a\ k))$
and $b: \text{summable } (\lambda k. \text{norm } (b\ k))$
shows $(\lambda k. \sum_{i \leq k}. a\ i * b\ (k - i)) \text{ sums } ((\sum k. a\ k) * (\sum k. b\ k))$
 $\langle \text{proof} \rangle$

lemma *Cauchy-product*:

fixes $a\ b :: \text{nat} \Rightarrow 'a :: \{\text{real-normed-algebra}, \text{banach}\}$
assumes $\text{summable } (\lambda k. \text{norm } (a\ k))$
and $\text{summable } (\lambda k. \text{norm } (b\ k))$
shows $(\sum k. a\ k) * (\sum k. b\ k) = (\sum k. \sum_{i \leq k}. a\ i * b\ (k - i))$
 $\langle \text{proof} \rangle$

lemma *summable-Cauchy-product*:

fixes $a\ b :: \text{nat} \Rightarrow 'a :: \{\text{real-normed-algebra}, \text{banach}\}$
assumes $\text{summable } (\lambda k. \text{norm } (a\ k))$
and $\text{summable } (\lambda k. \text{norm } (b\ k))$
shows $\text{summable } (\lambda k. \sum_{i \leq k}. a\ i * b\ (k - i))$
 $\langle \text{proof} \rangle$

109.12 Series on reals

lemma *summable-norm-comparison-test*:

$\exists N. \forall n \geq N. \text{norm } (f\ n) \leq g\ n \implies \text{summable } g \implies \text{summable } (\lambda n. \text{norm } (f\ n))$
 $\langle \text{proof} \rangle$

lemma *summable-rabs-comparison-test*: $\exists N. \forall n \geq N. |f\ n| \leq g\ n \implies \text{summable } g \implies \text{summable } (\lambda n. |f\ n|)$

for $f :: \text{nat} \Rightarrow \text{real}$
 $\langle \text{proof} \rangle$

lemma *summable-rabs-cancel*: $\text{summable } (\lambda n. |f\ n|) \implies \text{summable } f$

for $f :: \text{nat} \Rightarrow \text{real}$
 $\langle \text{proof} \rangle$

lemma *summable-rabs*: $\text{summable } (\lambda n. |f\ n|) \implies |\text{suminf } f| \leq (\sum n. |f\ n|)$

for $f :: \text{nat} \Rightarrow \text{real}$
 $\langle \text{proof} \rangle$

lemma *norm-suminf-le*:

assumes $\bigwedge n. \text{norm } (f\ n) \leq g\ n \text{ summable } g$
shows $\text{norm } (\text{suminf } f) \leq \text{suminf } g$

$\langle \text{proof} \rangle$

lemma *norm-sums-le*:

assumes $f \text{ sums } F \ g \text{ sums } G$
assumes $\bigwedge n. \text{norm } (f\ n :: 'a :: \text{banach}) \leq g\ n$
shows $\text{norm } F \leq G$
 $\langle \text{proof} \rangle$

lemma *summable-zero-power* [simp]: $\text{summable } (\lambda n. 0 \wedge n :: 'a :: \{\text{comm-ring-1}, \text{topological-space}\})$
 $\langle \text{proof} \rangle$

lemma *summable-zero-power'* [simp]: $\text{summable } (\lambda n. f\ n * 0 \wedge n :: 'a :: \{\text{ring-1}, \text{topological-space}\})$
 $\langle \text{proof} \rangle$

lemma *summable-power-series*:

fixes $z :: \text{real}$
assumes $\text{le-1}: \bigwedge i. f\ i \leq 1$
and $\text{nonneg}: \bigwedge i. 0 \leq f\ i$
and $z: 0 \leq z < 1$
shows $\text{summable } (\lambda i. f\ i * z \wedge i)$
 $\langle \text{proof} \rangle$

lemma *summable-0-powser*: $\text{summable } (\lambda n. f\ n * 0 \wedge n :: 'a :: \text{real-normed-div-algebra})$
 $\langle \text{proof} \rangle$

lemma *summable-powser-split-head*:

$\text{summable } (\lambda n. f\ (Suc\ n) * z \wedge n :: 'a :: \text{real-normed-div-algebra}) = \text{summable } (\lambda n. f\ n * z \wedge n)$
 $\langle \text{proof} \rangle$

lemma *summable-powser-ignore-initial-segment*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{real-normed-div-algebra}$
shows $\text{summable } (\lambda n. f\ (n + m) * z \wedge n) \longleftrightarrow \text{summable } (\lambda n. f\ n * z \wedge n)$
 $\langle \text{proof} \rangle$

lemma *powser-split-head*:

fixes $f :: \text{nat} \Rightarrow 'a :: \{\text{real-normed-div-algebra}, \text{banach}\}$
assumes $\text{summable } (\lambda n. f\ n * z \wedge n)$
shows $\text{suminf } (\lambda n. f\ n * z \wedge n) = f\ 0 + \text{suminf } (\lambda n. f\ (Suc\ n) * z \wedge n) * z$
and $\text{suminf } (\lambda n. f\ (Suc\ n) * z \wedge n) * z = \text{suminf } (\lambda n. f\ n * z \wedge n) - f\ 0$
and $\text{summable } (\lambda n. f\ (Suc\ n) * z \wedge n)$
 $\langle \text{proof} \rangle$

lemma *summable-partial-sum-bound*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{banach}$
and $e :: \text{real}$
assumes $\text{summable } f$
and $e: e > 0$
obtains N **where** $\bigwedge m\ n. m \geq N \implies \text{norm } (\sum_{k=m..n} f\ k) < e$
 $\langle \text{proof} \rangle$

lemma *powser-sums-if*:

($\lambda n. (if\ n = m\ then\ (1 :: 'a :: \{ring-1, topological-space\})\ else\ 0) * z^{\wedge} n) \text{ sums } z^{\wedge} m$)
 $\langle proof \rangle$

lemma

fixes $f :: nat \Rightarrow real$
assumes *summable f*
and *inj g*
and *pos: $\bigwedge x. 0 \leq f\ x$*
shows *summable-reindex: summable (f \circ g)*
and *suminf-reindex-mono: $suminf\ (f \circ g) \leq suminf\ f$*
and *suminf-reindex: $(\bigwedge x. x \notin range\ g \implies f\ x = 0) \implies suminf\ (f \circ g) = suminf\ f$*
 $\langle proof \rangle$

lemma *sums-mono-reindex*:

assumes *subseq: strict-mono g*
and *zero: $\bigwedge n. n \notin range\ g \implies f\ n = 0$*
shows $(\lambda n. f\ (g\ n)) \text{ sums } c \longleftrightarrow f \text{ sums } c$
 $\langle proof \rangle$

lemma *summable-mono-reindex*:

assumes *subseq: strict-mono g*
and *zero: $\bigwedge n. n \notin range\ g \implies f\ n = 0$*
shows *summable (λn. f (g n)) \longleftrightarrow summable f*
 $\langle proof \rangle$

lemma *suminf-mono-reindex*:

fixes $f :: nat \Rightarrow 'a :: \{t2-space, comm-monoid-add\}$
assumes *strict-mono g $\bigwedge n. n \notin range\ g \implies f\ n = 0$*
shows $suminf\ (\lambda n. f\ (g\ n)) = suminf\ f$
 $\langle proof \rangle$

lemma *summable-bounded-partials*:

fixes $f :: nat \Rightarrow 'a :: \{real-normed-vector, complete-space\}$
assumes *bound: eventually $(\lambda x 0. \forall a \geq x 0. \forall b > a. norm\ (sum\ f\ \{a <.. b\}) \leq g\ a)$*
sequentially
assumes $g: g \longrightarrow 0$
shows *summable f* $\langle proof \rangle$

end

110 Differentiation

theory *Deriv*

imports *Limits*

begin

110.1 Frechet derivative

definition *has-derivative* :: (*'a*::*real-normed-vector* \Rightarrow *'b*::*real-normed-vector*) \Rightarrow
 (*'a* \Rightarrow *'b*) \Rightarrow *'a filter* \Rightarrow *bool* (**infix** \langle (*has'-derivative*) \rangle 50)
where (*f has-derivative f'*) *F* \longleftrightarrow
bounded-linear f' \wedge
(($\lambda y. ((f y - f (Lim F (\lambda x. x))) - f' (y - Lim F (\lambda x. x))) /_R norm (y - Lim$
 $F (\lambda x. x))) \longrightarrow 0$) F

Usually the filter *F* is *at x within s*. (*f 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 *UNIV*.

These are the only cases we'll care about, probably.

lemma *has-derivative-within*: (*f has-derivative f'*) (*at x within s*) \longleftrightarrow
*bounded-linear f' \wedge (($\lambda y. (1 / norm(y - x)) *_R (f y - (f x + f' (y - x)))$*
 $\longrightarrow 0$) (*at x within s*)
 \langle proof \rangle

lemma *has-derivative-eq-rhs*: (*f has-derivative f'*) *F* \Longrightarrow *f' = g'* \Longrightarrow (*f has-derivative g'*) *F*
 \langle proof \rangle

definition *has-field-derivative* :: (*'a*::*real-normed-field* \Rightarrow *'a*) \Rightarrow *'a \Rightarrow 'a filter* \Rightarrow
bool
 (**infix** \langle (*has'-field'-derivative*) \rangle 50)
where (*f has-field-derivative D*) *F* \longleftrightarrow (*f has-derivative (*) D*) *F*

lemma *DERIV-cong*: (*f has-field-derivative X*) *F* \Longrightarrow *X = Y* \Longrightarrow (*f has-field-derivative Y*) *F*
 \langle proof \rangle

definition *has-vector-derivative* :: (*real* \Rightarrow *'b*::*real-normed-vector*) \Rightarrow *'b \Rightarrow real*
filter \Rightarrow *bool*
 (**infix** \langle (*has'-vector'-derivative*) \rangle 50)
where (*f has-vector-derivative f'*) *net* \longleftrightarrow (*f has-derivative ($\lambda x. x *_R f'$) net*

lemma *has-vector-derivative-eq-rhs*:
 (*f has-vector-derivative X*) *F* \Longrightarrow *X = Y* \Longrightarrow (*f has-vector-derivative Y*) *F*
 \langle proof \rangle

named-theorems *derivative-intros structural introduction rules for derivatives*
 \langle ML \rangle

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*

($\langle \langle \text{notation} = \langle \text{mixfix } FDERIV \rangle \rangle FDERIV \ (-) / \ (-) / \ :> \ (-) \rangle \ [1000, 1000, 60] \ 60$)
where $FDERIV \ f \ x \ :> \ f' \equiv (f \text{ has-derivative } f') \ (at \ x)$

lemma *has-derivative-bounded-linear*: $(f \text{ has-derivative } f') \ F \implies \text{bounded-linear } f'$
 $\langle \text{proof} \rangle$

lemma *has-derivative-linear*: $(f \text{ has-derivative } f') \ F \implies \text{linear } f'$
 $\langle \text{proof} \rangle$

lemma *has-derivative-ident*[*derivative-intros*, *simp*]: $((\lambda x. x) \text{ has-derivative } (\lambda x. x)) \ F$
 $\langle \text{proof} \rangle$

lemma *has-derivative-id* [*derivative-intros*, *simp*]: $(id \text{ has-derivative } id) \ F$
 $\langle \text{proof} \rangle$

lemma *shift-has-derivative-id*: $((+) \ d \text{ has-derivative } (\lambda x. x)) \ F$
 $\langle \text{proof} \rangle$

lemma *has-derivative-const*[*derivative-intros*, *simp*]: $((\lambda x. c) \text{ has-derivative } (\lambda x. 0)) \ F$
 $\langle \text{proof} \rangle$

lemma (**in** *bounded-linear*) *bounded-linear*: *bounded-linear* $f \ \langle \text{proof} \rangle$

lemma (**in** *bounded-linear*) *has-derivative*:
 $(g \text{ has-derivative } g') \ F \implies ((\lambda x. f \ (g \ x)) \text{ has-derivative } (\lambda x. f \ (g' \ x))) \ F$
 $\langle \text{proof} \rangle$

lemma *has-derivative-bot* [*intro*]: *bounded-linear* $f' \implies (f \text{ has-derivative } f') \ bot$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-bot* [*simp*, *intro*]: $(f \text{ has-field-derivative } f') \ bot$
 $\langle \text{proof} \rangle$

lemmas *has-derivative-scaleR-right* [*derivative-intros*] =
bounded-linear.has-derivative [*OF* *bounded-linear-scaleR-right*]

lemmas *has-derivative-scaleR-left* [*derivative-intros*] =
bounded-linear.has-derivative [*OF* *bounded-linear-scaleR-left*]

lemmas *has-derivative-mult-right* [*derivative-intros*] =
bounded-linear.has-derivative [*OF* *bounded-linear-mult-right*]

lemmas *has-derivative-mult-left* [*derivative-intros*] =
bounded-linear.has-derivative [*OF* *bounded-linear-mult-left*]

lemmas *has-derivative-of-real*[*derivative-intros*, *simp*] =
bounded-linear.has-derivative[*OF* *bounded-linear-of-real*]

lemma *has-derivative-add*[*simp*, *derivative-intros*]:

assumes $f: (f \text{ has-derivative } f') F$

and $g: (g \text{ has-derivative } g') F$

shows $((\lambda x. f x + g x) \text{ has-derivative } (\lambda x. f' x + g' x)) F$

$\langle \text{proof} \rangle$

lemma *has-derivative-sum*[*simp*, *derivative-intros*]:

$(\bigwedge i. i \in I \implies (f i \text{ has-derivative } f' i) F) \implies$

$((\lambda x. \sum_{i \in I} f i x) \text{ has-derivative } (\lambda x. \sum_{i \in I} f' i x)) F$

$\langle \text{proof} \rangle$

lemma *has-derivative-minus*[*simp*, *derivative-intros*]:

$(f \text{ has-derivative } f') F \implies ((\lambda x. - f x) \text{ has-derivative } (\lambda x. - f' x)) F$

$\langle \text{proof} \rangle$

lemma *has-derivative-diff*[*simp*, *derivative-intros*]:

$(f \text{ has-derivative } f') F \implies (g \text{ has-derivative } g') F \implies$

$((\lambda x. f x - g x) \text{ has-derivative } (\lambda x. f' x - g' x)) F$

$\langle \text{proof} \rangle$

lemma *has-derivative-at-within*:

$(f \text{ has-derivative } f') (at x \text{ within } s) \longleftrightarrow$

$(\text{bounded-linear } f' \wedge ((\lambda y. ((f y - f x) - f' (y - x)) /_R \text{norm } (y - x)) \longrightarrow$

$0) (at x \text{ within } s))$

$\langle \text{proof} \rangle$

lemma *has-derivative-iff-norm*:

$(f \text{ has-derivative } f') (at x \text{ within } s) \longleftrightarrow$

$\text{bounded-linear } f' \wedge ((\lambda y. \text{norm } ((f y - f x) - f' (y - x)) / \text{norm } (y - x))$

$\longrightarrow 0) (at x \text{ within } s)$

$\langle \text{proof} \rangle$

lemma *has-derivative-at*:

$(f \text{ has-derivative } D) (at x) \longleftrightarrow$

$(\text{bounded-linear } D \wedge ((\lambda h. \text{norm } (f (x + h) - f x - D h) / \text{norm } h) - 0 \rightarrow 0))$

$\langle \text{proof} \rangle$

lemma *field-has-derivative-at*:

fixes $x :: 'a :: \text{real-normed-field}$

shows $(f \text{ has-derivative } (*) D) (at x) \longleftrightarrow (\lambda h. (f (x + h) - f x) / h) - 0 \rightarrow D$

$(\text{is ?lhs} = \text{?rhs})$

$\langle \text{proof} \rangle$

lemma *has-derivative-iff-Ex*:

$(f \text{ has-derivative } f') (at x) \longleftrightarrow$

$\text{bounded-linear } f' \wedge (\exists e. (\forall h. f (x + h) = f x + f' h + e h) \wedge ((\lambda h. \text{norm } (e h)$

$/ \text{norm } h) \longrightarrow 0) (at 0))$

$\langle \text{proof} \rangle$

lemma *has-derivative-at-within-iff-Ex*:

assumes $x \in S$ *open* S
shows $(f \text{ has-derivative } f') \text{ (at } x \text{ within } S) \longleftrightarrow$
 $\text{bounded-linear } f' \wedge (\exists e. (\forall h. x+h \in S \longrightarrow f(x+h) = f x + f' h + e h) \wedge$
 $((\lambda h. \text{norm } (e h) / \text{norm } h) \longrightarrow 0) \text{ (at } 0))$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *has-derivativeI*:

$\text{bounded-linear } f' \implies$
 $((\lambda y. ((f y - f x) - f' (y - x)) /_R \text{norm } (y - x)) \longrightarrow 0) \text{ (at } x \text{ within } s) \implies$
 $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *has-derivativeI-sandwich*:

assumes $e: 0 < e$
and *bounded*: $\text{bounded-linear } f'$
and *sandwich*: $(\bigwedge y. y \in s \implies y \neq x \implies \text{dist } y x < e \implies$
 $\text{norm } ((f y - f x) - f' (y - x)) / \text{norm } (y - x) \leq H y)$
and $(H \longrightarrow 0) \text{ (at } x \text{ within } s)$
shows $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *has-derivative-subset*:

$(f \text{ has-derivative } f') \text{ (at } x \text{ within } s) \implies t \subseteq s \implies (f \text{ has-derivative } f') \text{ (at } x \text{ within } t)$
 $\langle \text{proof} \rangle$

lemma *has-derivative-within-singleton-iff*:

$(f \text{ has-derivative } g) \text{ (at } x \text{ within } \{x\}) \longleftrightarrow \text{bounded-linear } g$
 $\langle \text{proof} \rangle$

110.1.1 Limit transformation for derivatives

lemma *has-derivative-transform-within*:

assumes $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
and $0 < d$
and $x \in s$
and $\bigwedge x'. [x' \in s; \text{dist } x' x < d] \implies f x' = g x'$
shows $(g \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *has-derivative-transform-within-open*:

assumes $(f \text{ has-derivative } f') \text{ (at } x \text{ within } t)$
and *open* s
and $x \in s$
and $\bigwedge x. x \in s \implies f x = g x$
shows $(g \text{ has-derivative } f') \text{ (at } x \text{ within } t)$

$\langle \text{proof} \rangle$

lemma *has-derivative-transform*:

assumes $x \in s \wedge x. x \in s \implies g\ x = f\ x$
assumes $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
shows $(g \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *has-derivative-transform-eventually*:

assumes $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
 $(\forall_F x' \text{ in at } x \text{ within } s. f\ x' = g\ x')$
assumes $f\ x = g\ x \wedge x \in s$
shows $(g \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-transform-within*:

assumes $(f \text{ has-field-derivative } f') \text{ (at } a \text{ within } S)$
and $0 < d$
and $a \in S$
and $\wedge x. \llbracket x \in S; \text{dist } x\ a < d \rrbracket \implies f\ x = g\ x$
shows $(g \text{ has-field-derivative } f') \text{ (at } a \text{ within } S)$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-transform-within-open*:

assumes $(f \text{ has-field-derivative } f') \text{ (at } a)$
and $\text{open } S \wedge a \in S$
and $\wedge x. x \in S \implies f\ x = g\ x$
shows $(g \text{ has-field-derivative } f') \text{ (at } a)$
 $\langle \text{proof} \rangle$

110.2 Continuity

lemma *has-derivative-continuous*:

assumes $f: (f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
shows $\text{continuous (at } x \text{ within } s) f$
 $\langle \text{proof} \rangle$

110.3 Composition

lemma *tendsto-at-iff-tendsto-nhds-within*:

$f\ x = y \implies (f \longrightarrow y) \text{ (at } x \text{ within } s) \longleftrightarrow (f \longrightarrow y) \text{ (inf (nhds } x) \text{ (principal } s))}$
 $\langle \text{proof} \rangle$

lemma *has-derivative-in-compose*:

assumes $f: (f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
and $g: (g \text{ has-derivative } g') \text{ (at } (f\ x) \text{ within } (f's))$
shows $((\lambda x. g\ (f\ x)) \text{ has-derivative } (\lambda x. g'\ (f'\ x))) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *has-derivative-compose*:

$(f \text{ has-derivative } f') \text{ (at } x \text{ within } s) \implies (g \text{ has-derivative } g') \text{ (at } (f \ x)) \implies$
 $((\lambda x. g \ (f \ x)) \text{ has-derivative } (\lambda x. g' \ (f' \ x))) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *has-derivative-in-compose2*:

assumes $\bigwedge x. x \in t \implies (g \text{ has-derivative } g' \ x) \text{ (at } x \text{ within } t)$
assumes $f' \ s \subseteq t \ x \in s$
assumes $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
shows $((\lambda x. g \ (f \ x)) \text{ has-derivative } (\lambda y. g' \ (f' \ x) \ (f' \ y))) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma (*in bounded-bilinear*) *FDERIV*:

assumes $f: (f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$ **and** $g: (g \text{ has-derivative } g') \text{ (at } x \text{ within } s)$
shows $((\lambda x. f \ x \ ** \ g \ x) \text{ has-derivative } (\lambda h. f \ x \ ** \ g' \ h + f' \ h \ ** \ g \ x)) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

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 $f :: 'i \Rightarrow 'a :: \text{real-normed-vector} \Rightarrow 'b :: \text{real-normed-field}$
shows $(\bigwedge i. i \in I \implies (f \ i \text{ has-derivative } f' \ i) \text{ (at } x \text{ within } S)) \implies$
 $((\lambda x. \prod_{i \in I}. f \ i \ x) \text{ has-derivative } (\lambda y. \sum_{i \in I}. f' \ i \ y * (\prod_{j \in I - \{i\}} f \ j \ x)))$
 $\text{(at } x \text{ within } S)$
 $\langle \text{proof} \rangle$

lemma *has-derivative-power*[simp, derivative-intros]:

fixes $f :: 'a :: \text{real-normed-vector} \Rightarrow 'b :: \text{real-normed-field}$
assumes $f: (f \text{ has-derivative } f') \text{ (at } x \text{ within } S)$
shows $((\lambda x. f \ x^n) \text{ has-derivative } (\lambda y. \text{of-nat } n * f' \ y * f \ x^{(n-1)})) \text{ (at } x \text{ within } S)$
 $\langle \text{proof} \rangle$

lemma *has-derivative-inverse'*:

fixes $x :: 'a :: \text{real-normed-div-algebra}$
assumes $x: x \neq 0$
shows $(\text{inverse has-derivative } (\lambda h. - \ (\text{inverse } x * h * \text{inverse } x))) \text{ (at } x \text{ within } S)$
 $(\text{is } (- \text{ has-derivative } ?f) \ -)$
 $\langle \text{proof} \rangle$

lemma *has-derivative-inverse*[simp, derivative-intros]:

fixes $f :: - \Rightarrow 'a :: \text{real-normed-div-algebra}$
assumes $x: f \ x \neq 0$

and f : (f has-derivative f') (at x within S)
shows $((\lambda x. \text{inverse } (f x)) \text{ has-derivative } (\lambda h. - (\text{inverse } (f x) * f' h * \text{inverse } (f x))))$
 (at x within S)
 $\langle \text{proof} \rangle$

lemma *has-derivative-divide*[*simp*, *derivative-intros*]:
fixes $f :: - \Rightarrow 'a::\text{real-normed-div-algebra}$
assumes f : (f has-derivative f') (at x within S)
and g : (g has-derivative g') (at x within S)
assumes x : $g x \neq 0$
shows $((\lambda x. f x / g x) \text{ has-derivative } (\lambda h. - f x * (\text{inverse } (g x) * g' h * \text{inverse } (g x)) + f' h / g x))$ (at x within S)
 $\langle \text{proof} \rangle$

lemma *has-derivative-power-int'*:
fixes $x :: 'a::\text{real-normed-field}$
assumes x : $x \neq 0$
shows $((\lambda x. \text{power-int } x n) \text{ has-derivative } (\lambda y. y * (\text{of-int } n * \text{power-int } x (n - 1))))$ (at x within S)
 $\langle \text{proof} \rangle$

lemma *has-derivative-power-int*[*simp*, *derivative-intros*]:
fixes $f :: - \Rightarrow 'a::\text{real-normed-field}$
assumes x : $f x \neq 0$
and f : (f 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)
 $\langle \text{proof} \rangle$

Conventional form requires mult-AC laws. Types real and complex only.

lemma *has-derivative-divide'*[*derivative-intros*]:
fixes $f :: - \Rightarrow 'a::\text{real-normed-field}$
assumes f : (f has-derivative f') (at x within S)
and g : (g 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)
 $\langle \text{proof} \rangle$

110.4 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 has-derivative f') (at x) \longleftrightarrow
 $\text{bounded-linear } f' \wedge ((\lambda y. (1 / (\text{norm}(y - x))) *_R (f y - (f x + f' (y - x)))) \longrightarrow 0)$ (at x)

$\langle \text{proof} \rangle$

lemma *has-derivative-zero-unique*:

assumes $((\lambda x. 0) \text{ has-derivative } F) \text{ (at } x)$

shows $F = (\lambda h. 0)$

$\langle \text{proof} \rangle$

lemma *has-derivative-unique*:

assumes $(f \text{ has-derivative } F) \text{ (at } x)$

and $(f \text{ has-derivative } F') \text{ (at } x)$

shows $F = F'$

$\langle \text{proof} \rangle$

lemma *has-derivative-Uniq*: $\exists \leq_1 F. (f \text{ has-derivative } F) \text{ (at } x)$

$\langle \text{proof} \rangle$

110.5 Differentiability predicate

definition *differentiable* :: $('a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-vector}) \Rightarrow 'a$
 $\text{filter} \Rightarrow \text{bool}$

(infix $\langle \text{differentiable} \rangle$ 50)

where $f \text{ differentiable } F \longleftrightarrow (\exists D. (f \text{ has-derivative } D) F)$

lemma *differentiable-subset*:

$f \text{ differentiable (at } x \text{ within } s) \implies t \subseteq s \implies f \text{ differentiable (at } x \text{ within } t)$

$\langle \text{proof} \rangle$

lemmas *differentiable-within-subset* = *differentiable-subset*

lemma *differentiable-ident* [*simp*, *derivative-intros*]: $(\lambda x. x) \text{ differentiable } F$

$\langle \text{proof} \rangle$

lemma *differentiable-const* [*simp*, *derivative-intros*]: $(\lambda z. a) \text{ differentiable } F$

$\langle \text{proof} \rangle$

lemma *differentiable-in-compose*:

$f \text{ differentiable (at } (g \ x) \text{ within } (g's)) \implies g \text{ differentiable (at } x \text{ within } s) \implies$

$(\lambda x. f \ (g \ x)) \text{ differentiable (at } x \text{ within } s)$

$\langle \text{proof} \rangle$

lemma *differentiable-compose*:

$f \text{ differentiable (at } (g \ x)) \implies g \text{ differentiable (at } x \text{ within } s) \implies$

$(\lambda x. f \ (g \ x)) \text{ differentiable (at } x \text{ within } s)$

$\langle \text{proof} \rangle$

lemma *differentiable-add* [*simp*, *derivative-intros*]:

$f \text{ differentiable } F \implies g \text{ differentiable } F \implies (\lambda x. f \ x + g \ x) \text{ differentiable } F$

$\langle \text{proof} \rangle$

lemma *differentiable-sum* [*simp*, *derivative-intros*]:
assumes *finite s* $\forall a \in s. (f\ a)\ \text{differentiable}\ \text{net}$
shows $(\lambda x. \text{sum } (\lambda a. f\ a\ x)\ s)\ \text{differentiable}\ \text{net}$
<proof>

lemma *differentiable-minus* [*simp*, *derivative-intros*]:
f differentiable F $\implies (\lambda x. -\ f\ x)\ \text{differentiable}\ F$
<proof>

lemma *differentiable-diff* [*simp*, *derivative-intros*]:
f differentiable F $\implies g\ \text{differentiable}\ F \implies (\lambda x. f\ x - g\ x)\ \text{differentiable}\ F$
<proof>

lemma *differentiable-mult* [*simp*, *derivative-intros*]:
fixes *f g* :: '*a*::*real-normed-vector* \Rightarrow '*b*::*real-normed-algebra*
shows *f differentiable (at x within s)* $\implies g\ \text{differentiable}\ (at\ x\ \text{within}\ s) \implies$
 $(\lambda x. f\ x * g\ x)\ \text{differentiable}\ (at\ x\ \text{within}\ s)$
<proof>

lemma *differentiable-cmult-left-iff* [*simp*]:
fixes *c*::'*a*::*real-normed-field*
shows $(\lambda t. c * q\ t)\ \text{differentiable}\ \text{at}\ t \longleftrightarrow c = 0 \vee (\lambda t. q\ t)\ \text{differentiable}\ \text{at}\ t$
(is ?lhs = ?rhs)
<proof>

lemma *differentiable-cmult-right-iff* [*simp*]:
fixes *c*::'*a*::*real-normed-field*
shows $(\lambda t. q\ t * c)\ \text{differentiable}\ \text{at}\ t \longleftrightarrow c = 0 \vee (\lambda t. q\ t)\ \text{differentiable}\ \text{at}\ t$
(is ?lhs = ?rhs)
<proof>

lemma *differentiable-inverse* [*simp*, *derivative-intros*]:
fixes *f* :: '*a*::*real-normed-vector* \Rightarrow '*b*::*real-normed-field*
shows *f differentiable (at x within s)* $\implies f\ x \neq 0 \implies$
 $(\lambda x. \text{inverse}\ (f\ x))\ \text{differentiable}\ (at\ x\ \text{within}\ s)$
<proof>

lemma *differentiable-divide* [*simp*, *derivative-intros*]:
fixes *f g* :: '*a*::*real-normed-vector* \Rightarrow '*b*::*real-normed-field*
shows *f differentiable (at x within s)* $\implies g\ \text{differentiable}\ (at\ x\ \text{within}\ s) \implies$
 $g\ x \neq 0 \implies (\lambda x. f\ x / g\ x)\ \text{differentiable}\ (at\ x\ \text{within}\ s)$
<proof>

lemma *differentiable-power* [*simp*, *derivative-intros*]:
fixes *f g* :: '*a*::*real-normed-vector* \Rightarrow '*b*::*real-normed-field*
shows *f differentiable (at x within s)* $\implies (\lambda x. f\ x ^ n)\ \text{differentiable}\ (at\ x\ \text{within}\ s)$
<proof>

lemma *differentiable-power-int* [*simp*, *derivative-intros*]:
fixes $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-field}$
shows $f \text{ differentiable } (at\ x\ \text{within}\ s) \Longrightarrow f\ x \neq 0 \Longrightarrow$
 $(\lambda x. \text{power-int } (f\ x)\ n) \text{ differentiable } (at\ x\ \text{within}\ s)$
 $\langle \text{proof} \rangle$

lemma *differentiable-scaleR* [*simp*, *derivative-intros*]:
 $f \text{ differentiable } (at\ x\ \text{within}\ s) \Longrightarrow g \text{ differentiable } (at\ x\ \text{within}\ s) \Longrightarrow$
 $(\lambda x. f\ x *_{\mathbb{R}} g\ x) \text{ differentiable } (at\ x\ \text{within}\ s)$
 $\langle \text{proof} \rangle$

lemma *has-derivative-imp-has-field-derivative*:
 $(f \text{ has-derivative } D) \ F \Longrightarrow (\bigwedge x. x * D' = D\ x) \Longrightarrow (f \text{ has-field-derivative } D')\ F$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-imp-has-derivative*:
 $(f \text{ has-field-derivative } D) \ F \Longrightarrow (f \text{ has-derivative } (*)\ D) \ F$
 $\langle \text{proof} \rangle$

lemma *DERIV-subset*:
 $(f \text{ has-field-derivative } f') \ (at\ x\ \text{within}\ s) \Longrightarrow t \subseteq s \Longrightarrow$
 $(f \text{ has-field-derivative } f') \ (at\ x\ \text{within}\ t)$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-at-within*:
 $(f \text{ has-field-derivative } f') \ (at\ x) \Longrightarrow (f \text{ has-field-derivative } f') \ (at\ x\ \text{within}\ s)$
 $\langle \text{proof} \rangle$

abbreviation (*input*)
 $DERIV :: ('a::\text{real-normed-field} \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$
 $(\langle \langle \text{notation} = \langle \text{mixfix } DERIV \rangle \rangle DERIV\ (-) / (-) / :> (-) \rangle [1000, 1000, 60] 60)$
where $DERIV\ f\ x :> D \equiv (f \text{ has-field-derivative } D) \ (at\ x)$

abbreviation *has-real-derivative* :: $(\text{real} \Rightarrow \text{real}) \Rightarrow \text{real} \Rightarrow \text{real filter} \Rightarrow \text{bool}$
 $(\text{infix } \langle \langle \text{has'-real'-derivative} \rangle \rangle 50)$
where $(f \text{ has-real-derivative } D) \ F \equiv (f \text{ has-field-derivative } D) \ F$

lemma *real-differentiable-def*:
 $f \text{ differentiable at } x \text{ within } s \longleftrightarrow (\exists D. (f \text{ has-real-derivative } D) \ (at\ x\ \text{within}\ s))$
 $\langle \text{proof} \rangle$

lemma *real-differentiableE* [*elim?*]:
assumes $f: f \text{ differentiable } (at\ x\ \text{within}\ s)$
obtains df **where** $(f \text{ has-real-derivative } df) \ (at\ x\ \text{within}\ s)$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-iff*:
 $(f \text{ has-field-derivative } D) \ (at\ x\ \text{within}\ S) \longleftrightarrow$
 $((\lambda y. (f\ y - f\ x) / (y - x)) \longrightarrow D) \ (at\ x\ \text{within}\ S)$

<proof>

lemma *DERIV-def*: $DERIV\ f\ x\ :\>\ D \longleftrightarrow (\lambda h. (f\ (x + h) - f\ x) / h) -0\rightarrow D$
<proof>

lemma *has-field-derivative-unique*:
assumes $(f\ \text{has-field-derivative}\ f'1)\ (at\ x\ \text{within}\ A)$
assumes $(f\ \text{has-field-derivative}\ f'2)\ (at\ x\ \text{within}\ A)$
assumes $at\ x\ \text{within}\ A \neq bot$
shows $f'1 = f'2$
<proof>

due to Christian Pardillo Laursen, replacing a proper epsilon-delta horror

lemma *field-derivative-lim-unique*:
assumes $f: (f\ \text{has-field-derivative}\ df)\ (at\ z)$
and $s: s \longrightarrow 0 \ \bigwedge n. s\ n \neq 0$
and $a: (\lambda n. (f\ (z + s\ n) - f\ z) / s\ n) \longrightarrow a$
shows $df = a$
<proof>

lemma *mult-commute-abs*: $(\lambda x. x * c) = (*)\ c$
for $c :: 'a::ab-semigroup-mult$
<proof>

lemma *DERIV-compose-FDERIV*:
fixes $f::real\Rightarrow real$
assumes $DERIV\ f\ (g\ x) :\>\ f'$
assumes $(g\ \text{has-derivative}\ g')\ (at\ x\ \text{within}\ s)$
shows $((\lambda x. f\ (g\ x))\ \text{has-derivative}\ (\lambda x. g'\ x * f'))\ (at\ x\ \text{within}\ s)$
<proof>

110.6 Vector derivative

It's for real derivatives only, and not obviously generalisable to field derivatives

lemma *has-real-derivative-iff-has-vector-derivative*:
 $(f\ \text{has-real-derivative}\ y)\ F \longleftrightarrow (f\ \text{has-vector-derivative}\ y)\ F$
<proof>

lemma *has-field-derivative-subset*:
 $(f\ \text{has-field-derivative}\ y)\ (at\ x\ \text{within}\ s) \implies t \subseteq s \implies$
 $(f\ \text{has-field-derivative}\ y)\ (at\ x\ \text{within}\ t)$
<proof>

lemma *has-vector-derivative-const*[*simp, derivative-intros*]: $((\lambda x. c)\ \text{has-vector-derivative}\ 0)\ net$
<proof>

lemma *has-vector-derivative-id*[*simp, derivative-intros*]: $((\lambda x. x) \text{ has-vector-derivative } 1) \text{ net}$
 $\langle \text{proof} \rangle$

lemma *has-vector-derivative-minus*[*derivative-intros*]:
 $(f \text{ has-vector-derivative } f') \text{ net} \implies ((\lambda x. - f x) \text{ has-vector-derivative } (- f')) \text{ net}$
 $\langle \text{proof} \rangle$

lemma *has-vector-derivative-add*[*derivative-intros*]:
 $(f \text{ has-vector-derivative } f') \text{ net} \implies (g \text{ has-vector-derivative } g') \text{ net} \implies$
 $((\lambda x. f x + g x) \text{ has-vector-derivative } (f' + g')) \text{ net}$
 $\langle \text{proof} \rangle$

lemma *has-vector-derivative-sum*[*derivative-intros*]:
 $(\bigwedge i. i \in I \implies (f i \text{ has-vector-derivative } f' i) \text{ net}) \implies$
 $((\lambda x. \sum_{i \in I}. f i x) \text{ has-vector-derivative } (\sum_{i \in I}. f' i)) \text{ net}$
 $\langle \text{proof} \rangle$

lemma *has-vector-derivative-diff*[*derivative-intros*]:
 $(f \text{ has-vector-derivative } f') \text{ net} \implies (g \text{ has-vector-derivative } g') \text{ net} \implies$
 $((\lambda x. f x - g x) \text{ has-vector-derivative } (f' - g')) \text{ net}$
 $\langle \text{proof} \rangle$

lemma *has-vector-derivative-add-const*:
 $((\lambda t. g t + z) \text{ has-vector-derivative } f') \text{ net} = ((\lambda t. g t) \text{ has-vector-derivative } f') \text{ net}$
 $\langle \text{proof} \rangle$

lemma *has-vector-derivative-diff-const*:
 $((\lambda t. g t - z) \text{ has-vector-derivative } f') \text{ net} = ((\lambda t. g t) \text{ has-vector-derivative } f') \text{ net}$
 $\langle \text{proof} \rangle$

lemma (*in bounded-linear*) *has-vector-derivative*:
assumes $(g \text{ has-vector-derivative } g') F$
shows $((\lambda x. f (g x)) \text{ has-vector-derivative } f g') F$
 $\langle \text{proof} \rangle$

lemma (*in bounded-bilinear*) *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 $((\lambda x. f x ** g x) \text{ has-vector-derivative } (f x ** g' + f' ** g x)) (at x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *has-vector-derivative-scaleR*[*derivative-intros*]:
 $(f \text{ has-field-derivative } f') (at x \text{ within } s) \implies (g \text{ has-vector-derivative } g') (at x \text{ within } s) \implies$
 $((\lambda x. f x *_R g x) \text{ has-vector-derivative } (f x *_R g' + f' *_R g x)) (at x \text{ within } s)$

$\langle \text{proof} \rangle$

lemma *has-vector-derivative-mult*[*derivative-intros*]:

$(f \text{ has-vector-derivative } f') \text{ (at } x \text{ within } s) \implies (g \text{ has-vector-derivative } g') \text{ (at } x \text{ within } s) \implies$

$((\lambda x. f x * g x) \text{ has-vector-derivative } (f x * g' + f' * g x)) \text{ (at } x \text{ within } s)$

for $f g :: \text{real} \Rightarrow 'a :: \text{real-normed-algebra}$

$\langle \text{proof} \rangle$

lemma *has-vector-derivative-of-real*[*derivative-intros*]:

$(f \text{ has-field-derivative } D) F \implies ((\lambda x. \text{of-real } (f x)) \text{ has-vector-derivative (of-real } D)) F$

$\langle \text{proof} \rangle$

lemma *has-vector-derivative-real-field*:

$(f \text{ has-field-derivative } f') \text{ (at (of-real } a)) \implies ((\lambda x. f (\text{of-real } x)) \text{ has-vector-derivative } f') \text{ (at } a \text{ within } s)$

$\langle \text{proof} \rangle$

lemma *has-vector-derivative-continuous*:

$(f \text{ has-vector-derivative } D) \text{ (at } x \text{ within } s) \implies \text{continuous (at } x \text{ within } s) f$

$\langle \text{proof} \rangle$

lemma *continuous-on-vector-derivative*:

$(\bigwedge x. x \in S \implies (f \text{ has-vector-derivative } f' x) \text{ (at } x \text{ within } S)) \implies \text{continuous-on } S f$

$\langle \text{proof} \rangle$

lemma *has-vector-derivative-mult-right*[*derivative-intros*]:

fixes $a :: 'a :: \text{real-normed-algebra}$

shows $(f \text{ has-vector-derivative } x) F \implies ((\lambda x. a * f x) \text{ has-vector-derivative } (a * x)) F$

$\langle \text{proof} \rangle$

lemma *has-vector-derivative-mult-left*[*derivative-intros*]:

fixes $a :: 'a :: \text{real-normed-algebra}$

shows $(f \text{ has-vector-derivative } x) F \implies ((\lambda x. f x * a) \text{ has-vector-derivative } (x * a)) F$

$\langle \text{proof} \rangle$

lemma *has-vector-derivative-divide*[*derivative-intros*]:

fixes $a :: 'a :: \text{real-normed-field}$

shows $(f \text{ has-vector-derivative } x) F \implies ((\lambda x. f x / a) \text{ has-vector-derivative } (x / a)) F$

$\langle \text{proof} \rangle$

110.7 Derivatives

lemma *DERIV-D*: $\text{DERIV } f x :> D \implies (\lambda h. (f (x + h) - f x) / h) - 0 \rightarrow D$

$\langle \text{proof} \rangle$

lemma *has-field-derivativeD*:

$(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } S) \implies$
 $((\lambda y. (f y - f x) / (y - x)) \longrightarrow D) \text{ (at } x \text{ within } S)$
 $\langle \text{proof} \rangle$

lemma *DERIV-const* [*simp*, *derivative-intros*]: $((\lambda x. k) \text{ has-field-derivative } 0) F$
 $\langle \text{proof} \rangle$

lemma *DERIV-ident* [*simp*, *derivative-intros*]: $((\lambda x. x) \text{ has-field-derivative } 1) F$
 $\langle \text{proof} \rangle$

lemma *field-differentiable-add*[*derivative-intros*]:

$(f \text{ has-field-derivative } f') F \implies (g \text{ has-field-derivative } g') F \implies$
 $((\lambda z. f z + g z) \text{ has-field-derivative } f' + g') F$
 $\langle \text{proof} \rangle$

corollary *DERIV-add*:

$(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } s) \implies (g \text{ has-field-derivative } E) \text{ (at } x \text{ within } s) \implies$
 $((\lambda x. f x + g x) \text{ has-field-derivative } D + E) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *field-differentiable-minus*[*derivative-intros*]:

$(f \text{ has-field-derivative } f') F \implies ((\lambda z. - (f z)) \text{ has-field-derivative } -f') F$
 $\langle \text{proof} \rangle$

corollary *DERIV-minus*:

$(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } s) \implies$
 $((\lambda x. - f x) \text{ has-field-derivative } -D) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *field-differentiable-diff*[*derivative-intros*]:

$(f \text{ has-field-derivative } f') F \implies$
 $(g \text{ has-field-derivative } g') F \implies ((\lambda z. f z - g z) \text{ has-field-derivative } f' - g') F$
 $\langle \text{proof} \rangle$

corollary *DERIV-diff*:

$(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } s) \implies$
 $(g \text{ has-field-derivative } E) \text{ (at } x \text{ within } s) \implies$
 $((\lambda x. f x - g x) \text{ has-field-derivative } D - E) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *DERIV-continuous*: $(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } s) \implies \text{continuous (at } x \text{ within } s) f$
 $\langle \text{proof} \rangle$

corollary *DERIV-isCont*: $\text{DERIV } f x \text{ :> } D \implies \text{isCont } f x$

$\langle \text{proof} \rangle$

lemma *DERIV-atLeastAtMost-imp-continuous-on*:

assumes $\bigwedge x. \llbracket a \leq x; x \leq b \rrbracket \implies \exists y. \text{DERIV } f \ x \ :> \ y$

shows *continuous-on* $\{a..b\} \ f$

$\langle \text{proof} \rangle$

lemma *DERIV-continuous-on*:

$(\bigwedge x. x \in s \implies (f \text{ has-field-derivative } (D \ x)) \ (at \ x \ \text{within } s)) \implies \text{continuous-on } s \ f$

$\langle \text{proof} \rangle$

lemma *DERIV-mult'*:

$(f \text{ has-field-derivative } D) \ (at \ x \ \text{within } s) \implies (g \text{ has-field-derivative } E) \ (at \ x \ \text{within } s) \implies$

$((\lambda x. f \ x * g \ x) \text{ has-field-derivative } f \ x * E + D * g \ x) \ (at \ x \ \text{within } s)$

$\langle \text{proof} \rangle$

lemma *DERIV-mult[derivative-intros]*:

$(f \text{ has-field-derivative } Da) \ (at \ x \ \text{within } s) \implies (g \text{ has-field-derivative } Db) \ (at \ x \ \text{within } s) \implies$

$((\lambda x. f \ x * g \ x) \text{ has-field-derivative } Da * g \ x + Db * f \ x) \ (at \ x \ \text{within } s)$

$\langle \text{proof} \rangle$

Derivative of linear multiplication

lemma *DERIV-cmult*:

$(f \text{ has-field-derivative } D) \ (at \ x \ \text{within } s) \implies$

$((\lambda x. c * f \ x) \text{ has-field-derivative } c * D) \ (at \ x \ \text{within } s)$

$\langle \text{proof} \rangle$

lemma *DERIV-cmult-right*:

$(f \text{ has-field-derivative } D) \ (at \ x \ \text{within } s) \implies$

$((\lambda x. f \ x * c) \text{ has-field-derivative } D * c) \ (at \ x \ \text{within } s)$

$\langle \text{proof} \rangle$

lemma *DERIV-cmult-Id [simp]*: $((*) \ c \text{ has-field-derivative } c) \ (at \ x \ \text{within } s)$

$\langle \text{proof} \rangle$

lemma *DERIV-cdivide*:

$(f \text{ has-field-derivative } D) \ (at \ x \ \text{within } s) \implies$

$((\lambda x. f \ x / c) \text{ has-field-derivative } D / c) \ (at \ x \ \text{within } s)$

$\langle \text{proof} \rangle$

lemma *DERIV-unique*: $\text{DERIV } f \ x \ :> \ D \implies \text{DERIV } f \ x \ :> \ E \implies D = E$

$\langle \text{proof} \rangle$

lemma *DERIV-Uniq*: $\exists \leq_1 D. \text{DERIV } f \ x \ :> \ D$

$\langle \text{proof} \rangle$

lemma *DERIV-sum*[*derivative-intros*]:

$(\bigwedge n. n \in S \implies ((\lambda x. f\ x\ n)\ \text{has-field-derivative}\ (f'\ n))\ F) \implies$
 $((\lambda x. \text{sum}\ (f\ x)\ S)\ \text{has-field-derivative}\ \text{sum}\ f'\ S)\ F)$
 $\langle \text{proof} \rangle$

lemma *DERIV-inverse'*[*derivative-intros*]:

assumes $(f\ \text{has-field-derivative}\ D)\ (at\ x\ \text{within}\ s)$
and $f\ x \neq 0$
shows $((\lambda x. \text{inverse}\ (f\ x))\ \text{has-field-derivative}\ -\ (\text{inverse}\ (f\ x) * D * \text{inverse}\ (f\ x)))$
 $(at\ x\ \text{within}\ s)$
 $\langle \text{proof} \rangle$

Power of -1

lemma *DERIV-inverse*:

$x \neq 0 \implies ((\lambda x. \text{inverse}(x))\ \text{has-field-derivative}\ -\ (\text{inverse}\ x \wedge \text{Suc}\ (\text{Suc}\ 0)))\ (at\ x\ \text{within}\ s)$
 $\langle \text{proof} \rangle$

Derivative of inverse

lemma *DERIV-inverse-fun*:

$(f\ \text{has-field-derivative}\ d)\ (at\ x\ \text{within}\ s) \implies f\ x \neq 0 \implies$
 $((\lambda x. \text{inverse}\ (f\ x))\ \text{has-field-derivative}\ -\ (d * \text{inverse}(f\ x) \wedge \text{Suc}\ (\text{Suc}\ 0))))$
 $(at\ x\ \text{within}\ s)$
 $\langle \text{proof} \rangle$

Derivative of quotient

lemma *DERIV-divide*[*derivative-intros*]:

$(f\ \text{has-field-derivative}\ D)\ (at\ x\ \text{within}\ s) \implies$
 $(g\ \text{has-field-derivative}\ E)\ (at\ x\ \text{within}\ s) \implies g\ x \neq 0 \implies$
 $((\lambda x. f\ x / g\ x)\ \text{has-field-derivative}\ (D * g\ x - f\ x * E) / (g\ x * g\ x))\ (at\ x\ \text{within}\ s)$
 $\langle \text{proof} \rangle$

lemma *DERIV-quotient*:

$(f\ \text{has-field-derivative}\ d)\ (at\ x\ \text{within}\ s) \implies$
 $(g\ \text{has-field-derivative}\ e)\ (at\ x\ \text{within}\ s) \implies g\ x \neq 0 \implies$
 $((\lambda y. f\ y / g\ y)\ \text{has-field-derivative}\ (d * g\ x - (e * f\ x)) / (g\ x \wedge \text{Suc}\ (\text{Suc}\ 0))))$
 $(at\ x\ \text{within}\ s)$
 $\langle \text{proof} \rangle$

lemma *DERIV-power-Suc*:

$(f\ \text{has-field-derivative}\ D)\ (at\ x\ \text{within}\ s) \implies$
 $((\lambda x. f\ x \wedge \text{Suc}\ n)\ \text{has-field-derivative}\ (1 + \text{of-nat}\ n) * (D * f\ x \wedge n))\ (at\ x\ \text{within}\ s)$
 $\langle \text{proof} \rangle$

lemma *DERIV-power*[*derivative-intros*]:

$(f\ \text{has-field-derivative}\ D)\ (at\ x\ \text{within}\ s) \implies$

$((\lambda x. f\ x \wedge n) \text{ has-field-derivative of-nat } n * (D * f\ x \wedge (n - \text{Suc } 0))) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *DERIV-pow*: $((\lambda x. x \wedge n) \text{ has-field-derivative real } n * (x \wedge (n - \text{Suc } 0)))$
 $\text{(at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *DERIV-power-int* [*derivative-intros*]:
assumes [*derivative-intros*]: $(f \text{ has-field-derivative } d) \text{ (at } x \text{ within } s)$
and $n \geq 0 \vee f\ x \neq 0$
shows $((\lambda x. \text{power-int } (f\ x)\ n) \text{ has-field-derivative}$
 $\text{(of-int } n * \text{power-int } (f\ x)\ (n - 1) * d)) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *DERIV-chain'*: $(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } s) \implies \text{DERIV } g\ (f\ x) :> E \implies$
 $((\lambda x. g\ (f\ x)) \text{ has-field-derivative } E * D) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

corollary *DERIV-chain2*: $\text{DERIV } f\ (g\ x) :> Da \implies (g \text{ has-field-derivative } Db)$
 $\text{(at } x \text{ within } s) \implies$
 $((\lambda x. f\ (g\ x)) \text{ has-field-derivative } Da * Db) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

Derivative of a finite product

lemma *has-field-derivative-prod*:
assumes $\bigwedge x. x \in A \implies (f\ x \text{ has-field-derivative } f'\ x) \text{ (at } z)$
shows $((\lambda u. \prod_{x \in A}. f\ x\ u) \text{ has-field-derivative } (\sum_{x \in A}. f'\ x * (\prod_{y \in A - \{x\}}. f\ y\ z))) \text{ (at } z)$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-prod'*:
assumes $\bigwedge x. x \in A \implies f\ x\ z \neq 0$
assumes $\bigwedge x. x \in A \implies (f\ x \text{ has-field-derivative } f'\ x) \text{ (at } z)$
defines $P \equiv (\lambda A\ u. \prod_{x \in A}. f\ x\ u)$
shows $(P\ A \text{ has-field-derivative } (P\ A\ z * (\sum_{x \in A}. f'\ x / f\ x\ z))) \text{ (at } z)$
 $\langle \text{proof} \rangle$

Standard version

lemma *DERIV-chain*:
 $\text{DERIV } f\ (g\ x) :> Da \implies (g \text{ has-field-derivative } Db) \text{ (at } x \text{ within } s) \implies$
 $(f \circ g \text{ has-field-derivative } Da * Db) \text{ (at } x \text{ within } s)$
 $\langle \text{proof} \rangle$

lemma *DERIV-image-chain*:
 $(f \text{ has-field-derivative } Da) \text{ (at } (g\ x) \text{ within } (g\ 's)) \implies$
 $(g \text{ has-field-derivative } Db) \text{ (at } x \text{ within } s) \implies$
 $(f \circ g \text{ has-field-derivative } Da * Db) \text{ (at } x \text{ within } s)$

$\langle \text{proof} \rangle$

lemma *DERIV-chain-s*:

assumes $(\bigwedge x. x \in s \implies \text{DERIV } g \ x :> g'(x))$
and $\text{DERIV } f \ x :> f'$
and $f \ x \in s$
shows $\text{DERIV } (\lambda x. g(f \ x)) \ x :> f' * g'(f \ x)$
 $\langle \text{proof} \rangle$

lemma *DERIV-chain3*:

assumes $(\bigwedge x. \text{DERIV } g \ x :> g'(x))$
and $\text{DERIV } f \ x :> f'$
shows $\text{DERIV } (\lambda x. g(f \ x)) \ x :> f' * g'(f \ x)$
 $\langle \text{proof} \rangle$

Alternative definition for differentiability

lemma *DERIV-LIM-iff*:

fixes $f :: 'a :: \{\text{real-normed-vector}, \text{inverse}\} \Rightarrow 'a$
shows $((\lambda h. (f \ (a + h) - f \ a) / h) - 0 \rightarrow D) = ((\lambda x. (f \ x - f \ a) / (x - a)) - a \rightarrow D)$ **(is ?lhs = ?rhs)**
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-cong-ev*:

assumes $x = y$
and $*$: *eventually* $(\lambda x. x \in S \longrightarrow f \ x = g \ x)$ *(nhds x)*
and $u = v \ S = t \ x \in S$
shows $(f \ \text{has-field-derivative } u) \ (\text{at } x \ \text{within } S) = (g \ \text{has-field-derivative } v) \ (\text{at } y \ \text{within } t)$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-cong-eventually*:

assumes *eventually* $(\lambda x. f \ x = g \ x)$ *(at x within S)* $f \ x = g \ x$
shows $(f \ \text{has-field-derivative } u) \ (\text{at } x \ \text{within } S) = (g \ \text{has-field-derivative } u) \ (\text{at } x \ \text{within } S)$
 $\langle \text{proof} \rangle$

lemma *DERIV-cong-ev*:

$x = y \implies \text{eventually } (\lambda x. f \ x = g \ x)$ *(nhds x)* $\implies u = v \implies$
 $\text{DERIV } f \ x :> u \longleftrightarrow \text{DERIV } g \ y :> v$
 $\langle \text{proof} \rangle$

lemma *DERIV-mirror*: $(\text{DERIV } f \ (- \ x) :> y) \longleftrightarrow (\text{DERIV } (\lambda x. f \ (- \ x)) \ x :> - \ y)$

for $f :: \text{real} \Rightarrow \text{real}$ **and** $x \ y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *DERIV-shift*:

$(f \ \text{has-field-derivative } y) \ (\text{at } (x + z)) = ((\lambda x. f \ (x + z)) \ \text{has-field-derivative } y)$

(at x)
 ⟨proof⟩

lemma *DERIV-at-within-shift-lemma*:

assumes (f has-field-derivative y) (at (z+x) within (+) z ‘ S)
shows (f ∘ (+)z has-field-derivative y) (at x within S)
 ⟨proof⟩

lemma *DERIV-at-within-shift*:

(f has-field-derivative y) (at (z+x) within (+) z ‘ S) \longleftrightarrow
 ((λx. f (z+x)) has-field-derivative y) (at x within S) (is ?lhs = ?rhs)
 ⟨proof⟩

lemma *floor-has-real-derivative*:

fixes f :: real \Rightarrow 'a::{floor-ceiling,order-topology}
assumes isCont f x
and f x $\notin \mathbb{Z}$
shows ((λx. floor (f x)) has-real-derivative 0) (at x)
 ⟨proof⟩

lemmas has-derivative-floor[derivative-intros] =
 floor-has-real-derivative[THEN DERIV-compose-FDERIV]

lemma *continuous-floor*:

fixes x::real
shows x $\notin \mathbb{Z} \implies$ continuous (at x) (real-of-int ∘ floor)
 ⟨proof⟩

lemma *continuous-frac*:

fixes x::real
assumes x $\notin \mathbb{Z}$
shows continuous (at x) frac
 ⟨proof⟩

Caratheodory formulation of derivative at a point

lemma *CARAT-DERIV*:

(DERIV f x \Rightarrow l) \longleftrightarrow ($\exists g. (\forall z. f z - f x = g z * (z - x)) \wedge$ isCont g x \wedge g x = l)
 (is ?lhs = ?rhs)
 ⟨proof⟩

110.8 Local extrema

If $0 < f' x$ then x is Locally Strictly Increasing At The Right.

lemma *has-real-derivative-pos-inc-right*:

fixes f :: real \Rightarrow real
assumes der: (f has-real-derivative l) (at x within S)
and l: $0 < l$
shows $\exists d > 0. \forall h > 0. x + h \in S \longrightarrow h < d \longrightarrow f x < f (x + h)$

$\langle proof \rangle$

lemma *DERIV-pos-inc-right*:

fixes $f :: real \Rightarrow real$

assumes $der: DERIV f x :> l$

and $l: 0 < l$

shows $\exists d > 0. \forall h > 0. h < d \longrightarrow f x < f (x + h)$

$\langle proof \rangle$

lemma *has-real-derivative-neg-dec-left*:

fixes $f :: real \Rightarrow real$

assumes $der: (f \text{ has-real-derivative } l) \text{ (at } x \text{ within } S)$

and $l < 0$

shows $\exists d > 0. \forall h > 0. x - h \in S \longrightarrow h < d \longrightarrow f x < f (x - h)$

$\langle proof \rangle$

lemma *DERIV-neg-dec-left*:

fixes $f :: real \Rightarrow real$

assumes $der: DERIV f x :> l$

and $l < 0$

shows $\exists d > 0. \forall h > 0. h < d \longrightarrow f x < f (x - h)$

$\langle proof \rangle$

lemma *has-real-derivative-pos-inc-left*:

fixes $f :: real \Rightarrow real$

shows $(f \text{ has-real-derivative } l) \text{ (at } x \text{ within } S) \Longrightarrow 0 < l \Longrightarrow$

$\exists d > 0. \forall h > 0. x - h \in S \longrightarrow h < d \longrightarrow f (x - h) < f x$

$\langle proof \rangle$

lemma *DERIV-pos-inc-left*:

fixes $f :: real \Rightarrow real$

shows $DERIV f x :> l \Longrightarrow 0 < l \Longrightarrow \exists d > 0. \forall h > 0. h < d \longrightarrow f (x - h)$

$< f x$

$\langle proof \rangle$

lemma *has-real-derivative-neg-dec-right*:

fixes $f :: real \Rightarrow real$

shows $(f \text{ has-real-derivative } l) \text{ (at } x \text{ within } S) \Longrightarrow l < 0 \Longrightarrow$

$\exists d > 0. \forall h > 0. x + h \in S \longrightarrow h < d \longrightarrow f x > f (x + h)$

$\langle proof \rangle$

lemma *DERIV-neg-dec-right*:

fixes $f :: real \Rightarrow real$

shows $DERIV f x :> l \Longrightarrow l < 0 \Longrightarrow \exists d > 0. \forall h > 0. h < d \longrightarrow f x > f (x$

$+ h)$

$\langle proof \rangle$

lemma *DERIV-local-max*:

fixes $f :: real \Rightarrow real$

assumes $der: DERIV\ f\ x :> l$
and $d: 0 < d$
and $le: \forall y. |x - y| < d \longrightarrow f\ y \leq f\ x$
shows $l = 0$
 $\langle proof \rangle$

Similar theorem for a local minimum

lemma *DERIV-local-min*:
fixes $f :: real \Rightarrow real$
shows $DERIV\ f\ x :> l \Longrightarrow 0 < d \Longrightarrow \forall y. |x - y| < d \longrightarrow f\ x \leq f\ y \Longrightarrow l = 0$
 $\langle proof \rangle$

In particular, if a function is locally flat

lemma *DERIV-local-const*:
fixes $f :: real \Rightarrow real$
shows $DERIV\ f\ x :> l \Longrightarrow 0 < d \Longrightarrow \forall y. |x - y| < d \longrightarrow f\ x = f\ y \Longrightarrow l = 0$
 $\langle proof \rangle$

110.9 Rolle’s Theorem

Lemma about introducing open ball in open interval

lemma *lemma-interval-lt*:
fixes $a\ b\ x :: real$
assumes $a < x < b$
shows $\exists d. 0 < d \wedge (\forall y. |x - y| < d \longrightarrow a < y \wedge y < b)$
 $\langle proof \rangle$

lemma *lemma-interval*: $a < x \Longrightarrow x < b \Longrightarrow \exists d. 0 < d \wedge (\forall y. |x - y| < d \longrightarrow a \leq y \wedge y \leq b)$
for $a\ b\ x :: real$
 $\langle proof \rangle$

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 $x_0 \in (a, b)$ such that $f'\ x_0 = 0$

theorem *Rolle-deriv*:
fixes $f :: real \Rightarrow real$
assumes $a < b$
and $fab: f\ a = f\ b$
and $contf: continuous-on\ \{a..b\}\ f$
and $derf: \bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow (f\ \text{has-derivative}\ f'\ x)\ (at\ x)$
shows $\exists z. a < z \wedge z < b \wedge f'\ z = (\lambda v. 0)$
 $\langle proof \rangle$

corollary *Rolle*:
fixes $a\ b :: real$
assumes $ab: a < b\ f\ a = f\ b\ continuous-on\ \{a..b\}\ f$
and $dif\ [rule-format]: \bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow f\ \text{differentiable}\ (at\ x)$

shows $\exists z. a < z \wedge z < b \wedge \text{DERIV } f z :> 0$
 $\langle \text{proof} \rangle$

110.10 Mean Value Theorem

theorem *mvt*:

fixes $f :: \text{real} \Rightarrow \text{real}$
assumes $a < b$
and *contf*: *continuous-on* $\{a..b\}$ f
and *derf*: $\bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow (f \text{ has-derivative } f' x) (at x)$
obtains ξ **where** $a < \xi \wedge \xi < b \wedge f b - f a = (f' \xi) (b - a)$
 $\langle \text{proof} \rangle$

theorem *MVT*:

fixes $a b :: \text{real}$
assumes *lt*: $a < b$
and *contf*: *continuous-on* $\{a..b\}$ f
and *dif*: $\bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow f \text{ differentiable } (at x)$
shows $\exists l z. a < z \wedge z < b \wedge \text{DERIV } f z :> l \wedge f b - f a = (b - a) * l$
 $\langle \text{proof} \rangle$

corollary *MVT2*:

assumes $a < b$ **and** *der*: $\bigwedge x. \llbracket a \leq x; x \leq b \rrbracket \Longrightarrow \text{DERIV } f x :> f' x$
shows $\exists z :: \text{real}. a < z \wedge z < b \wedge (f b - f a = (b - a) * f' z)$
 $\langle \text{proof} \rangle$

110.10.1 A function is constant if its derivative is 0 over an interval.

lemma *DERIV-isconst-end*:

fixes $f :: \text{real} \Rightarrow \text{real}$
assumes $a < b$ **and** *contf*: *continuous-on* $\{a..b\}$ f
and *0*: $\bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow \text{DERIV } f x :> 0$
shows $f b = f a$
 $\langle \text{proof} \rangle$

lemma *DERIV-isconst2*:

fixes $f :: \text{real} \Rightarrow \text{real}$
assumes $a < b$ **and** *contf*: *continuous-on* $\{a..b\}$ f **and** *derf*: $\bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow \text{DERIV } f x :> 0$
and $a \leq x \wedge x \leq b$
shows $f x = f a$
 $\langle \text{proof} \rangle$

lemma *DERIV-isconst3*:

fixes $a b x y :: \text{real}$
assumes $a < b$
and $x \in \{a <..< b\}$
and $y \in \{a <..< b\}$
and *derivable*: $\bigwedge x. x \in \{a <..< b\} \Longrightarrow \text{DERIV } f x :> 0$

shows $f\ x = f\ y$
 $\langle proof \rangle$

lemma *DERIV-isconst-all*:
fixes $f :: real \Rightarrow real$
shows $\forall x. DERIV\ f\ x :> 0 \implies f\ x = f\ y$
 $\langle proof \rangle$

lemma *DERIV-const-ratio-const*:
fixes $f :: real \Rightarrow real$
assumes $a \neq b$ **and** $df: \bigwedge x. DERIV\ f\ x :> k$
shows $f\ b - f\ a = (b - a) * k$
 $\langle proof \rangle$

lemma *DERIV-const-ratio-const2*:
fixes $f :: real \Rightarrow real$
assumes $a \neq b$ **and** $df: \bigwedge x. DERIV\ f\ x :> k$
shows $(f\ b - f\ a) / (b - a) = k$
 $\langle proof \rangle$

lemma *real-average-minus-first [simp]*: $(a + b) / 2 - a = (b - a) / 2$
for $a\ b :: real$
 $\langle proof \rangle$

lemma *real-average-minus-second [simp]*: $(b + a) / 2 - a = (b - a) / 2$
for $a\ b :: real$
 $\langle proof \rangle$

Gallileo’s “trick”: average velocity = av. of end velocities.

lemma *DERIV-const-average*:
fixes $v :: real \Rightarrow real$
and $a\ b :: real$
assumes $neg: a \neq b$
and $der: \bigwedge x. DERIV\ v\ x :> k$
shows $v\ ((a + b) / 2) = (v\ a + v\ b) / 2$
 $\langle proof \rangle$

110.10.2 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 $f :: real \Rightarrow real$
assumes $a < b$
and $\bigwedge x. a < x \implies x < b \implies (\exists y. DERIV\ f\ x :> y \wedge y > 0)$
and $con: continuous-on\ \{a..b\}\ f$
shows $f\ a < f\ b$
 $\langle proof \rangle$

lemma *DERIV-pos-imp-increasing:*

fixes $a\ b :: \text{real}$ **and** $f :: \text{real} \Rightarrow \text{real}$

assumes $a < b$

and $\text{der}: \bigwedge x. \llbracket a \leq x; x \leq b \rrbracket \Longrightarrow \exists y. \text{DERIV } f\ x :> y \wedge y > 0$

shows $f\ a < f\ b$

$\langle \text{proof} \rangle$

lemma *DERIV-nonneg-imp-nondecreasing:*

fixes $a\ b :: \text{real}$

and $f :: \text{real} \Rightarrow \text{real}$

assumes $a \leq b$

and $\bigwedge x. \llbracket a \leq x; x \leq b \rrbracket \Longrightarrow \exists y. \text{DERIV } f\ x :> y \wedge y \geq 0$

shows $f\ a \leq f\ b$

$\langle \text{proof} \rangle$

lemma *DERIV-neg-imp-decreasing-open:*

fixes $a\ b :: \text{real}$

and $f :: \text{real} \Rightarrow \text{real}$

assumes $a < b$

and $\bigwedge x. a < x \Longrightarrow x < b \Longrightarrow \exists y. \text{DERIV } f\ x :> y \wedge y < 0$

and $\text{con}: \text{continuous-on } \{a..b\} f$

shows $f\ a > f\ b$

$\langle \text{proof} \rangle$

lemma *DERIV-neg-imp-decreasing:*

fixes $a\ b :: \text{real}$ **and** $f :: \text{real} \Rightarrow \text{real}$

assumes $a < b$

and $\text{der}: \bigwedge x. \llbracket a \leq x; x \leq b \rrbracket \Longrightarrow \exists y. \text{DERIV } f\ x :> y \wedge y < 0$

shows $f\ a > f\ b$

$\langle \text{proof} \rangle$

lemma *DERIV-nonpos-imp-nonincreasing:*

fixes $a\ b :: \text{real}$

and $f :: \text{real} \Rightarrow \text{real}$

assumes $a \leq b$

and $\bigwedge x. \llbracket a \leq x; x \leq b \rrbracket \Longrightarrow \exists y. \text{DERIV } f\ x :> y \wedge y \leq 0$

shows $f\ a \geq f\ b$

$\langle \text{proof} \rangle$

lemma *DERIV-pos-imp-increasing-at-bot:*

fixes $f :: \text{real} \Rightarrow \text{real}$

assumes $\bigwedge x. x \leq b \Longrightarrow (\exists y. \text{DERIV } f\ x :> y \wedge y > 0)$

and $\text{lim}: (f \longrightarrow \text{flim}) \text{ at-bot}$

shows $\text{flim} < f\ b$

$\langle \text{proof} \rangle$

lemma *DERIV-neg-imp-decreasing-at-top:*

fixes $f :: \text{real} \Rightarrow \text{real}$

assumes $\text{der}: \bigwedge x. x \geq b \Longrightarrow \exists y. \text{DERIV } f\ x :> y \wedge y < 0$

and *lim*: $(f \longrightarrow flim) \text{ at-top}$
shows $flim < f\ b$
 $\langle proof \rangle$

proposition *deriv-nonpos-imp-antimono*:

assumes *deriv*: $\bigwedge x. x \in \{a..b\} \implies (g \text{ has-real-derivative } g' \ x) \ (at \ x)$
assumes *nonneg*: $\bigwedge x. x \in \{a..b\} \implies g' \ x \leq 0$
assumes $a \leq b$
shows $g \ b \leq g \ a$
 $\langle proof \rangle$

lemma *DERIV-nonneg-imp-increasing-open*:

fixes $a \ b :: real$
and $f :: real \Rightarrow real$
assumes $a \leq b$
and $\bigwedge x. a < x \implies x < b \implies (\exists y. DERIV \ f \ x \ :> y \wedge y \geq 0)$
and *con*: *continuous-on* $\{a..b\} \ f$
shows $f \ a \leq f \ b$
 $\langle proof \rangle$

lemma *DERIV-nonpos-imp-decreasing-open*:

fixes $a \ b :: real$
and $f :: real \Rightarrow real$
assumes $a \leq b$
and $\bigwedge x. a < x \implies x < b \implies \exists y. DERIV \ f \ x \ :> y \wedge y \leq 0$
and *con*: *continuous-on* $\{a..b\} \ f$
shows $f \ a \geq f \ b$
 $\langle proof \rangle$

proposition *deriv-nonneg-imp-mono*:

assumes $\bigwedge x. x \in \{a..b\} \implies (g \text{ has-real-derivative } g' \ x) \ (at \ x)$
assumes $\bigwedge x. x \in \{a..b\} \implies g' \ x \geq 0$
assumes $a \leq b$
shows $g \ a \leq g \ b$
 $\langle proof \rangle$

Derivative of inverse function

lemma *DERIV-inverse-function*:

fixes $f \ g :: real \Rightarrow real$
assumes *der*: $DERIV \ f \ (g \ x) \ :> D$
and *neg*: $D \neq 0$
and $x: a < x \ x < b$
and *inj*: $\bigwedge y. [a < y; y < b] \implies f \ (g \ y) = y$
and *cont*: *isCont* $g \ x$
shows $DERIV \ g \ x \ :> inverse \ D$
 $\langle proof \rangle$

110.11 Generalized Mean Value Theorem

theorem *GMVT*:

fixes $a\ b :: \text{real}$
assumes $alb: a < b$
and $fc: \forall x. a \leq x \wedge x \leq b \longrightarrow \text{isCont } f\ x$
and $fd: \forall x. a < x \wedge x < b \longrightarrow f \text{ differentiable } (at\ x)$
and $gc: \forall x. a \leq x \wedge x \leq b \longrightarrow \text{isCont } g\ x$
and $gd: \forall x. a < x \wedge x < b \longrightarrow g \text{ differentiable } (at\ x)$
shows $\exists g'c\ f'c\ c.$
 $DERIV\ g\ c :> g'c \wedge DERIV\ f\ c :> f'c \wedge a < c \wedge c < b \wedge (f\ b - f\ a) * g'c =$
 $(g\ b - g\ a) * f'c$
 $\langle \text{proof} \rangle$

lemma *GMVT'*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$
assumes $a < b$
and $\text{isCont-}f: \bigwedge z. a \leq z \Longrightarrow z \leq b \Longrightarrow \text{isCont } f\ z$
and $\text{isCont-}g: \bigwedge z. a \leq z \Longrightarrow z \leq b \Longrightarrow \text{isCont } g\ z$
and $DERIV-g: \bigwedge z. a < z \Longrightarrow z < b \Longrightarrow DERIV\ g\ z :> (g'\ z)$
and $DERIV-f: \bigwedge z. a < z \Longrightarrow z < b \Longrightarrow DERIV\ f\ z :> (f'\ z)$
shows $\exists c. a < c \wedge c < b \wedge (f\ b - f\ a) * g'\ c = (g\ b - g\ a) * f'\ c$
 $\langle \text{proof} \rangle$

110.12 L'Hopitals rule

lemma *isCont-If-ge*:

fixes $a :: 'a :: \text{linorder-topology}$
assumes $\text{continuous } (at\text{-left } a)\ g$ **and** $f: (f \longrightarrow g\ a) (at\text{-right } a)$
shows $\text{isCont } (\lambda x. \text{if } x \leq a \text{ then } g\ x \text{ else } f\ x)\ a$ (**is** $\text{isCont } ?gf\ a$)
 $\langle \text{proof} \rangle$

lemma *lhospital-right-0*:

fixes $f0\ g0 :: \text{real} \Rightarrow \text{real}$
assumes $f-0: (f0 \longrightarrow 0) (at\text{-right } 0)$
and $g-0: (g0 \longrightarrow 0) (at\text{-right } 0)$
and $ev:$
 $\text{eventually } (\lambda x. g0\ x \neq 0) (at\text{-right } 0)$
 $\text{eventually } (\lambda x. g'\ x \neq 0) (at\text{-right } 0)$
 $\text{eventually } (\lambda x. DERIV\ f0\ x :> f'\ x) (at\text{-right } 0)$
 $\text{eventually } (\lambda x. DERIV\ g0\ x :> g'\ x) (at\text{-right } 0)$
and $\text{lim: filterlim } (\lambda x. (f'\ x / g'\ x))\ F (at\text{-right } 0)$
shows $\text{filterlim } (\lambda x. f0\ x / g0\ x)\ F (at\text{-right } 0)$
 $\langle \text{proof} \rangle$

lemma *lhospital-right*:

$(f \longrightarrow 0) (at\text{-right } x) \Longrightarrow (g \longrightarrow 0) (at\text{-right } x) \Longrightarrow$
 $\text{eventually } (\lambda x. g\ x \neq 0) (at\text{-right } x) \Longrightarrow$
 $\text{eventually } (\lambda x. g'\ x \neq 0) (at\text{-right } x) \Longrightarrow$
 $\text{eventually } (\lambda x. DERIV\ f\ x :> f'\ x) (at\text{-right } x) \Longrightarrow$

$\text{eventually } (\lambda x. \text{DERIV } g \ x :> g' \ x) \ (\text{at-right } x) \implies$
 $\text{filterlim } (\lambda x. (f' \ x / g' \ x)) \ F \ (\text{at-right } x) \implies$
 $\text{filterlim } (\lambda x. f \ x / g \ x) \ F \ (\text{at-right } x)$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *lhospital-left*:

$(f \longrightarrow 0) \ (\text{at-left } x) \implies (g \longrightarrow 0) \ (\text{at-left } x) \implies$
 $\text{eventually } (\lambda x. g \ x \neq 0) \ (\text{at-left } x) \implies$
 $\text{eventually } (\lambda x. g' \ x \neq 0) \ (\text{at-left } x) \implies$
 $\text{eventually } (\lambda x. \text{DERIV } f \ x :> f' \ x) \ (\text{at-left } x) \implies$
 $\text{eventually } (\lambda x. \text{DERIV } g \ x :> g' \ x) \ (\text{at-left } x) \implies$
 $\text{filterlim } (\lambda x. (f' \ x / g' \ x)) \ F \ (\text{at-left } x) \implies$
 $\text{filterlim } (\lambda x. f \ x / g \ x) \ F \ (\text{at-left } x)$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *lhospital*:

$(f \longrightarrow 0) \ (\text{at } x) \implies (g \longrightarrow 0) \ (\text{at } x) \implies$
 $\text{eventually } (\lambda x. g \ x \neq 0) \ (\text{at } x) \implies$
 $\text{eventually } (\lambda x. g' \ x \neq 0) \ (\text{at } x) \implies$
 $\text{eventually } (\lambda x. \text{DERIV } f \ x :> f' \ x) \ (\text{at } x) \implies$
 $\text{eventually } (\lambda x. \text{DERIV } g \ x :> g' \ x) \ (\text{at } x) \implies$
 $\text{filterlim } (\lambda x. (f' \ x / g' \ x)) \ F \ (\text{at } x) \implies$
 $\text{filterlim } (\lambda x. f \ x / g \ x) \ F \ (\text{at } x)$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *lhospital-right-0-at-top*:

fixes $f \ g :: \text{real} \Rightarrow \text{real}$
assumes $g\text{-}0$: $\text{LIM } x \ \text{at-right } 0. \ g \ x :> \text{at-top}$
and ev :
 $\text{eventually } (\lambda x. g' \ x \neq 0) \ (\text{at-right } 0)$
 $\text{eventually } (\lambda x. \text{DERIV } f \ x :> f' \ x) \ (\text{at-right } 0)$
 $\text{eventually } (\lambda x. \text{DERIV } g \ x :> g' \ x) \ (\text{at-right } 0)$
and lim : $((\lambda x. (f' \ x / g' \ x)) \longrightarrow x) \ (\text{at-right } 0)$
shows $((\lambda x. f \ x / g \ x) \longrightarrow x) \ (\text{at-right } 0)$
 $\langle \text{proof} \rangle$

lemma *lhospital-right-at-top*:

$\text{LIM } x \ \text{at-right } x. \ (g :: \text{real} \Rightarrow \text{real}) \ x :> \text{at-top} \implies$
 $\text{eventually } (\lambda x. g' \ x \neq 0) \ (\text{at-right } x) \implies$
 $\text{eventually } (\lambda x. \text{DERIV } f \ x :> f' \ x) \ (\text{at-right } x) \implies$
 $\text{eventually } (\lambda x. \text{DERIV } g \ x :> g' \ x) \ (\text{at-right } x) \implies$
 $((\lambda x. (f' \ x / g' \ x)) \longrightarrow y) \ (\text{at-right } x) \implies$
 $((\lambda x. f \ x / g \ x) \longrightarrow y) \ (\text{at-right } x)$
 $\langle \text{proof} \rangle$

lemma *lhospital-left-at-top*:

$LIM\ x\ at-left\ x.\ g\ x\ :>\ at-top \implies$
 $eventually\ (\lambda x.\ g'\ x \neq 0)\ (at-left\ x) \implies$
 $eventually\ (\lambda x.\ DERIV\ f\ x\ :>\ f'\ x)\ (at-left\ x) \implies$
 $eventually\ (\lambda x.\ DERIV\ g\ x\ :>\ g'\ x)\ (at-left\ x) \implies$
 $((\lambda x.\ (f'\ x / g'\ x)) \longrightarrow y)\ (at-left\ x) \implies$
 $((\lambda x.\ f\ x / g\ x) \longrightarrow y)\ (at-left\ x)$
for $x :: real$
 $\langle proof \rangle$

lemma *lhospital-at-top*:

$LIM\ x\ at\ x.\ (g :: real \Rightarrow real)\ x\ :>\ at-top \implies$
 $eventually\ (\lambda x.\ g'\ x \neq 0)\ (at\ x) \implies$
 $eventually\ (\lambda x.\ DERIV\ f\ x\ :>\ f'\ x)\ (at\ x) \implies$
 $eventually\ (\lambda x.\ DERIV\ g\ x\ :>\ g'\ x)\ (at\ x) \implies$
 $((\lambda x.\ (f'\ x / g'\ x)) \longrightarrow y)\ (at\ x) \implies$
 $((\lambda x.\ f\ x / g\ x) \longrightarrow y)\ (at\ x)$
 $\langle proof \rangle$

lemma *lhospital-at-top-at-top*:

fixes $f\ g :: real \Rightarrow real$
assumes $g-0$: $LIM\ x\ at-top.\ g\ x\ :>\ at-top$
and g' : $eventually\ (\lambda x.\ g'\ x \neq 0)\ at-top$
and Df : $eventually\ (\lambda x.\ DERIV\ f\ x\ :>\ f'\ x)\ at-top$
and Dg : $eventually\ (\lambda x.\ DERIV\ g\ x\ :>\ g'\ x)\ at-top$
and lim : $((\lambda x.\ (f'\ x / g'\ x)) \longrightarrow x)\ at-top$
shows $((\lambda x.\ f\ x / g\ x) \longrightarrow x)\ at-top$
 $\langle proof \rangle$

lemma *lhospital-right-at-top-at-top*:

fixes $f\ g :: real \Rightarrow real$
assumes $f-0$: $LIM\ x\ at-right\ a.\ f\ x\ :>\ at-top$
assumes $g-0$: $LIM\ x\ at-right\ a.\ g\ x\ :>\ at-top$
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 : $filterlim\ (\lambda x.\ (f'\ x / g'\ x))\ at-top\ (at-right\ a)$
shows $filterlim\ (\lambda x.\ f\ x / g\ x)\ at-top\ (at-right\ a)$
 $\langle proof \rangle$

lemma *lhospital-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>

lemma *lhospital-left-at-top-at-top*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

assumes $f\text{-}0$: $\text{LIM } x \text{ at-left } a. f\ x :> \text{at-top}$

assumes $g\text{-}0$: $\text{LIM } x \text{ at-left } a. g\ x :> \text{at-top}$

and *ev*:

eventually $(\lambda x. \text{DERIV } f\ x :> f'\ x) (\text{at-left } a)$

eventually $(\lambda x. \text{DERIV } g\ x :> g'\ x) (\text{at-left } a)$

and *lim*: $\text{filterlim } (\lambda x. (f'\ x / g'\ x)) \text{ at-top } (\text{at-left } a)$

shows $\text{filterlim } (\lambda x. f\ x / g\ x) \text{ at-top } (\text{at-left } a)$

<proof>

lemma *lhospital-left-at-top-at-bot*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

assumes $f\text{-}0$: $\text{LIM } x \text{ at-left } a. f\ x :> \text{at-top}$

assumes $g\text{-}0$: $\text{LIM } x \text{ at-left } a. g\ x :> \text{at-bot}$

and *ev*:

eventually $(\lambda x. \text{DERIV } f\ x :> f'\ x) (\text{at-left } a)$

eventually $(\lambda x. \text{DERIV } g\ x :> g'\ x) (\text{at-left } a)$

and *lim*: $\text{filterlim } (\lambda x. (f'\ x / g'\ x)) \text{ at-bot } (\text{at-left } a)$

shows $\text{filterlim } (\lambda x. f\ x / g\ x) \text{ at-bot } (\text{at-left } a)$

<proof>

lemma *lhospital-at-top-at-top*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

assumes $f\text{-}0$: $\text{LIM } x \text{ at } a. f\ x :> \text{at-top}$

assumes $g\text{-}0$: $\text{LIM } x \text{ at } a. g\ x :> \text{at-top}$

and *ev*:

eventually $(\lambda x. \text{DERIV } f\ x :> f'\ x) (\text{at } a)$

eventually $(\lambda x. \text{DERIV } g\ x :> g'\ x) (\text{at } a)$

and *lim*: $\text{filterlim } (\lambda x. (f'\ x / g'\ x)) \text{ at-top } (\text{at } a)$

shows $\text{filterlim } (\lambda x. f\ x / g\ x) \text{ at-top } (\text{at } a)$

<proof>

lemma *lhospital-at-top-at-bot*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

assumes $f\text{-}0$: $\text{LIM } x \text{ at } a. f\ x :> \text{at-top}$

assumes $g\text{-}0$: $\text{LIM } x \text{ at } a. g\ x :> \text{at-bot}$

and *ev*:

eventually $(\lambda x. \text{DERIV } f\ x :> f'\ x) (\text{at } a)$

eventually $(\lambda x. \text{DERIV } g\ x :> g'\ x) (\text{at } a)$

and *lim*: $\text{filterlim } (\lambda x. (f'\ x / g'\ x)) \text{ at-bot } (\text{at } a)$

shows $\text{filterlim } (\lambda x. f\ x / g\ x) \text{ at-bot } (\text{at } a)$

<proof>

end

111 Nth Roots of Real Numbers

```
theory NthRoot
  imports Deriv
begin
```

111.1 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  $\exists r > 0. r^{\wedge} n = a$ 
  <proof>
```

```
lemma realpow-pos-nth2: (0::real) < a  $\implies \exists r > 0. r^{\wedge} \text{Suc } n = a$ 
  <proof>
```

Uniqueness of nth positive root.

```
lemma realpow-pos-nth-unique: 0 < n  $\implies 0 < a \implies \exists! r. 0 < r \wedge r^{\wedge} n = a$  for
a :: real
  <proof>
```

111.2 Nth Root

We define roots of negative reals such that $\text{root } n \ (-x) = -\text{root } n \ x$. This allows us to omit side conditions from many theorems.

```
lemma inj-sgn-power:
  assumes 0 < n
  shows inj ( $\lambda y. \text{sgn } y * |y|^{\wedge} n$ ) :: real
    (is inj ?f)
  <proof>
```

```
lemma sgn-power-injE:
   $\text{sgn } a * |a|^{\wedge} n = x \implies x = \text{sgn } b * |b|^{\wedge} n \implies 0 < n \implies a = b$ 
  for a b :: real
  <proof>
```

```
definition root :: nat  $\Rightarrow$  real  $\Rightarrow$  real
  where root n x = (if n = 0 then 0 else the-inv ( $\lambda y. \text{sgn } y * |y|^{\wedge} n$ ) x)
```

```
lemma root-0 [simp]: root 0 x = 0
  <proof>
```

```
lemma root-sgn-power: 0 < n  $\implies \text{root } n \ (\text{sgn } y * |y|^{\wedge} n) = y$ 
  <proof>
```

lemma *sgn-power-root*:

assumes $0 < n$

shows $\text{sgn } (\text{root } n \ x) * |(\text{root } n \ x)|^{\wedge n} = x$

(is ?f $(\text{root } n \ x) = x$ **)**

$\langle \text{proof} \rangle$

lemma *split-root*: $P (\text{root } n \ x) \longleftrightarrow (n = 0 \longrightarrow P \ 0) \wedge (0 < n \longrightarrow (\forall y. \text{sgn } y * |y|^{\wedge n} = x \longrightarrow P \ y))$

$\langle \text{proof} \rangle$

lemma *real-root-zero* [simp]: $\text{root } n \ 0 = 0$

$\langle \text{proof} \rangle$

lemma *real-root-minus*: $\text{root } n \ (-x) = - \text{root } n \ x$

$\langle \text{proof} \rangle$

lemma *real-root-less-mono*: $0 < n \Longrightarrow x < y \Longrightarrow \text{root } n \ x < \text{root } n \ y$

$\langle \text{proof} \rangle$

lemma *real-root-gt-zero*: $0 < n \Longrightarrow 0 < x \Longrightarrow 0 < \text{root } n \ x$

$\langle \text{proof} \rangle$

lemma *real-root-ge-zero*: $0 \leq x \Longrightarrow 0 \leq \text{root } n \ x$

$\langle \text{proof} \rangle$

lemma *real-root-pow-pos*: $0 < n \Longrightarrow 0 < x \Longrightarrow \text{root } n \ x^{\wedge n} = x$

$\langle \text{proof} \rangle$

lemma *real-root-pow-pos2* [simp]: $0 < n \Longrightarrow 0 \leq x \Longrightarrow \text{root } n \ x^{\wedge n} = x$

$\langle \text{proof} \rangle$

lemma *sgn-root*: $0 < n \Longrightarrow \text{sgn } (\text{root } n \ x) = \text{sgn } x$

$\langle \text{proof} \rangle$

lemma *odd-real-root-pow*: $\text{odd } n \Longrightarrow \text{root } n \ x^{\wedge n} = x$

$\langle \text{proof} \rangle$

lemma *real-root-power-cancel*: $0 < n \Longrightarrow 0 \leq x \Longrightarrow \text{root } n \ (x^{\wedge n}) = x$

$\langle \text{proof} \rangle$

lemma *odd-real-root-power-cancel*: $\text{odd } n \Longrightarrow \text{root } n \ (x^{\wedge n}) = x$

$\langle \text{proof} \rangle$

lemma *real-root-pos-unique*: $0 < n \Longrightarrow 0 \leq y \Longrightarrow y^{\wedge n} = x \Longrightarrow \text{root } n \ x = y$

$\langle \text{proof} \rangle$

lemma *odd-real-root-unique*: $\text{odd } n \Longrightarrow y^{\wedge n} = x \Longrightarrow \text{root } n \ x = y$

$\langle \text{proof} \rangle$

lemma *real-root-one* [simp]: $0 < n \implies \text{root } n \ 1 = 1$
 ⟨proof⟩

Root function is strictly monotonic, hence injective.

lemma *real-root-le-mono*: $0 < n \implies x \leq y \implies \text{root } n \ x \leq \text{root } n \ y$
 ⟨proof⟩

lemma *real-root-less-iff* [simp]: $0 < n \implies \text{root } n \ x < \text{root } n \ y \longleftrightarrow x < y$
 ⟨proof⟩

lemma *real-root-le-iff* [simp]: $0 < n \implies \text{root } n \ x \leq \text{root } n \ y \longleftrightarrow x \leq y$
 ⟨proof⟩

lemma *real-root-eq-iff* [simp]: $0 < n \implies \text{root } n \ x = \text{root } n \ y \longleftrightarrow x = y$
 ⟨proof⟩

lemmas *real-root-gt-0-iff* [simp] = *real-root-less-iff* [where $x=0$, simplified]
lemmas *real-root-lt-0-iff* [simp] = *real-root-less-iff* [where $y=0$, simplified]
lemmas *real-root-ge-0-iff* [simp] = *real-root-le-iff* [where $x=0$, simplified]
lemmas *real-root-le-0-iff* [simp] = *real-root-le-iff* [where $y=0$, simplified]
lemmas *real-root-eq-0-iff* [simp] = *real-root-eq-iff* [where $y=0$, simplified]

lemma *real-root-gt-1-iff* [simp]: $0 < n \implies 1 < \text{root } n \ y \longleftrightarrow 1 < y$
 ⟨proof⟩

lemma *real-root-lt-1-iff* [simp]: $0 < n \implies \text{root } n \ x < 1 \longleftrightarrow x < 1$
 ⟨proof⟩

lemma *real-root-ge-1-iff* [simp]: $0 < n \implies 1 \leq \text{root } n \ y \longleftrightarrow 1 \leq y$
 ⟨proof⟩

lemma *real-root-le-1-iff* [simp]: $0 < n \implies \text{root } n \ x \leq 1 \longleftrightarrow x \leq 1$
 ⟨proof⟩

lemma *real-root-eq-1-iff* [simp]: $0 < n \implies \text{root } n \ x = 1 \longleftrightarrow x = 1$
 ⟨proof⟩

Roots of multiplication and division.

lemma *real-root-mult*: $\text{root } n \ (x * y) = \text{root } n \ x * \text{root } n \ y$
 ⟨proof⟩

lemma *real-root-inverse*: $\text{root } n \ (\text{inverse } x) = \text{inverse } (\text{root } n \ x)$
 ⟨proof⟩

lemma *real-root-divide*: $\text{root } n \ (x / y) = \text{root } n \ x / \text{root } n \ y$
 ⟨proof⟩

lemma *real-root-abs*: $0 < n \implies \text{root } n \ |x| = |\text{root } n \ x|$

$\langle \text{proof} \rangle$

lemma *root-abs-power*: $n > 0 \implies \text{abs } (\text{root } n \ (y \wedge n)) = \text{abs } y$
 $\langle \text{proof} \rangle$

lemma *real-root-power*: $0 < n \implies \text{root } n \ (x \wedge k) = \text{root } n \ x \wedge k$
 $\langle \text{proof} \rangle$

Roots of roots.

lemma *real-root-Suc-0* [simp]: $\text{root } (\text{Suc } 0) \ x = x$
 $\langle \text{proof} \rangle$

lemma *real-root-mult-exp*: $\text{root } (m * n) \ x = \text{root } m \ (\text{root } n \ x)$
 $\langle \text{proof} \rangle$

lemma *real-root-commute*: $\text{root } m \ (\text{root } n \ x) = \text{root } n \ (\text{root } m \ x)$
 $\langle \text{proof} \rangle$

Monotonicity in first argument.

lemma *real-root-strict-decreasing*:
 assumes $0 < n \ n < N \ 1 < x$
 shows $\text{root } N \ x < \text{root } n \ x$
 $\langle \text{proof} \rangle$

lemma *real-root-strict-increasing*:
 assumes $0 < n \ n < N \ 0 < x \ x < 1$
 shows $\text{root } n \ x < \text{root } N \ x$
 $\langle \text{proof} \rangle$

lemma *real-root-decreasing*: $0 < n \implies n \leq N \implies 1 \leq x \implies \text{root } N \ x \leq \text{root } n \ x$
 $\langle \text{proof} \rangle$

lemma *real-root-increasing*: $0 < n \implies n \leq N \implies 0 \leq x \implies x \leq 1 \implies \text{root } n \ x \leq \text{root } N \ x$
 $\langle \text{proof} \rangle$

Continuity and derivatives.

lemma *isCont-real-root*: $\text{isCont } (\text{root } n) \ x$
 $\langle \text{proof} \rangle$

lemma *tendsto-real-root* [tendsto-intros]:
 $(f \longrightarrow x) \ F \implies ((\lambda x. \text{root } n \ (f \ x)) \longrightarrow \text{root } n \ x) \ F$
 $\langle \text{proof} \rangle$

lemma *continuous-real-root* [continuous-intros]:
 $\text{continuous } F \ f \implies \text{continuous } F \ (\lambda x. \text{root } n \ (f \ x))$
 $\langle \text{proof} \rangle$

lemma *continuous-on-real-root* [continuous-intros]:

continuous-on s f \implies continuous-on s ($\lambda x. \text{root } n \ (f \ x)$)
 $\langle \text{proof} \rangle$

lemma *DERIV-real-root:*

assumes $n: 0 < n$
and $x: 0 < x$
shows *DERIV* ($\text{root } n$) $x :>$ *inverse* ($\text{real } n * \text{root } n \ x \wedge (n - \text{Suc } 0)$)
 $\langle \text{proof} \rangle$

lemma *DERIV-odd-real-root:*

assumes $n: \text{odd } n$
and $x: x \neq 0$
shows *DERIV* ($\text{root } n$) $x :>$ *inverse* ($\text{real } n * \text{root } n \ x \wedge (n - \text{Suc } 0)$)
 $\langle \text{proof} \rangle$

lemma *DERIV-even-real-root:*

assumes $n: 0 < n$
and *even* n
and $x: x < 0$
shows *DERIV* ($\text{root } n$) $x :>$ *inverse* ($- \text{real } n * \text{root } n \ x \wedge (n - \text{Suc } 0)$)
 $\langle \text{proof} \rangle$

lemma *DERIV-real-root-generic:*

assumes $0 < n$
and $x \neq 0$
and *even* $n \implies 0 < x \implies D = \text{inverse} (\text{real } n * \text{root } n \ x \wedge (n - \text{Suc } 0))$
and *even* $n \implies x < 0 \implies D = - \text{inverse} (\text{real } n * \text{root } n \ x \wedge (n - \text{Suc } 0))$
and *odd* $n \implies D = \text{inverse} (\text{real } n * \text{root } n \ x \wedge (n - \text{Suc } 0))$
shows *DERIV* ($\text{root } n$) $x :>$ D
 $\langle \text{proof} \rangle$

lemma *power-tendsto-0-iff [simp]:*

fixes $f :: 'a \Rightarrow \text{real}$
assumes $n > 0$
shows $((\lambda x. f \ x \wedge n) \longrightarrow 0) \ F \longleftrightarrow (f \longrightarrow 0) \ F$
 $\langle \text{proof} \rangle$

111.3 Square Root

definition *sqrt* :: $\text{real} \Rightarrow \text{real}$

where *sqrt* = *root 2*

lemma *pos2: 0 < (2::nat)*

$\langle \text{proof} \rangle$

lemma *real-sqrt-unique: $y^2 = x \implies 0 \leq y \implies \text{sqrt } x = y$*

$\langle \text{proof} \rangle$

lemma *real-sqrt-abs [simp]: $\text{sqrt } (x^2) = |x|$*

$\langle \text{proof} \rangle$

lemma *real-sqrt-pow2* [simp]: $0 \leq x \implies (\text{sqrt } x)^2 = x$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-pow2-iff* [simp]: $(\text{sqrt } x)^2 = x \longleftrightarrow 0 \leq x$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-zero* [simp]: $\text{sqrt } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-one* [simp]: $\text{sqrt } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-four* [simp]: $\text{sqrt } 4 = 2$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-minus*: $\text{sqrt } (-x) = -\text{sqrt } x$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-mult*: $\text{sqrt } (x * y) = \text{sqrt } x * \text{sqrt } y$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-mult-self* [simp]: $\text{sqrt } a * \text{sqrt } a = |a|$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-inverse*: $\text{sqrt } (\text{inverse } x) = \text{inverse } (\text{sqrt } x)$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-divide*: $\text{sqrt } (x / y) = \text{sqrt } x / \text{sqrt } y$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-power*: $\text{sqrt } (x ^ k) = \text{sqrt } x ^ k$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-gt-zero*: $0 < x \implies 0 < \text{sqrt } x$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-ge-zero*: $0 \leq x \implies 0 \leq \text{sqrt } x$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-less-mono*: $x < y \implies \text{sqrt } x < \text{sqrt } y$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-le-mono*: $x \leq y \implies \text{sqrt } x \leq \text{sqrt } y$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-less-iff* [simp]: $\text{sqrt } x < \text{sqrt } y \longleftrightarrow x < y$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-le-iff* [simp]: $\text{sqrt } x \leq \text{sqrt } y \longleftrightarrow x \leq y$
 ⟨proof⟩

lemma *real-sqrt-eq-iff* [simp]: $\text{sqrt } x = \text{sqrt } y \longleftrightarrow x = y$
 ⟨proof⟩

lemma *real-less-lsqrt*: $0 \leq y \implies x < y^2 \implies \text{sqrt } x < y$
 ⟨proof⟩

lemma *real-le-lsqrt*: $0 \leq y \implies x \leq y^2 \implies \text{sqrt } x \leq y$
 ⟨proof⟩

lemma *real-le-rsqrt*: $x^2 \leq y \implies x \leq \text{sqrt } y$
 ⟨proof⟩

lemma *real-less-rsqrt*: $x^2 < y \implies x < \text{sqrt } y$
 ⟨proof⟩

lemma *real-sqrt-power-even*:
 assumes *even* $n \geq 0$
 shows $\text{sqrt } x \wedge^n = x \wedge (n \text{ div } 2)$
 ⟨proof⟩

lemma *sqrt-le-D*: $\text{sqrt } x \leq y \implies x \leq y^2$
 ⟨proof⟩

lemma *sqrt-ge-absD*: $|x| \leq \text{sqrt } y \implies x^2 \leq y$
 ⟨proof⟩

lemma *sqrt-even-pow2*:
 assumes *n*: *even* n
 shows $\text{sqrt } (2 \wedge^n) = 2 \wedge (n \text{ div } 2)$
 ⟨proof⟩

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 $0 \leq x \ 0 \leq y$

shows $\text{sqrt } (x + y) \leq \text{sqrt } x + \text{sqrt } y$
 $\langle \text{proof} \rangle$

lemma *isCont-real-sqrt*: *isCont sqrt x*
 $\langle \text{proof} \rangle$

lemma *tendsto-real-sqrt* [*tendsto-intros*]:
 $(f \longrightarrow x) F \implies ((\lambda x. \text{sqrt } (f x)) \longrightarrow \text{sqrt } x) F$
 $\langle \text{proof} \rangle$

lemma *continuous-real-sqrt* [*continuous-intros*]:
 $\text{continuous } F f \implies \text{continuous } F (\lambda x. \text{sqrt } (f x))$
 $\langle \text{proof} \rangle$

lemma *continuous-on-real-sqrt* [*continuous-intros*]:
 $\text{continuous-on } s f \implies \text{continuous-on } s (\lambda x. \text{sqrt } (f x))$
 $\langle \text{proof} \rangle$

lemma *DERIV-real-sqrt-generic*:
assumes $x \neq 0$
and $x > 0 \implies D = \text{inverse } (\text{sqrt } x) / 2$
and $x < 0 \implies D = - \text{inverse } (\text{sqrt } x) / 2$
shows $\text{DERIV sqrt } x :> D$
 $\langle \text{proof} \rangle$

lemma *DERIV-real-sqrt*: $0 < x \implies \text{DERIV sqrt } x :> \text{inverse } (\text{sqrt } x) / 2$
 $\langle \text{proof} \rangle$

declare
 $\text{DERIV-real-sqrt-generic}[\text{THEN DERIV-chain2, derivative-intros}]$
 $\text{DERIV-real-root-generic}[\text{THEN DERIV-chain2, derivative-intros}]$

lemmas *has-derivative-real-sqrt*[*derivative-intros*] = *DERIV-real-sqrt*[*THEN DERIV-compose-FDERIV*]

lemma *not-real-square-gt-zero* [*simp*]: $\neg 0 < x * x \longleftrightarrow x = 0$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-abs2* [*simp*]: $\text{sqrt } (x * x) = |x|$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-abs'*: $\text{sqrt } |x| = |\text{sqrt } x|$
 $\langle \text{proof} \rangle$

lemma *real-inv-sqrt-pow2*: $0 < x \implies (\text{inverse } (\text{sqrt } x))^2 = \text{inverse } x$
 $\langle \text{proof} \rangle$

lemma *real-sqrt-eq-zero-cancel*: $0 \leq x \implies \text{sqrt } x = 0 \implies x = 0$

$\langle proof \rangle$

lemma *real-sqrt-ge-one*: $1 \leq x \implies 1 \leq \text{sqrt } x$
 $\langle proof \rangle$

lemma *sqrt-divide-self-eq*:
assumes *nneg*: $0 \leq x$
shows $\text{sqrt } x / x = \text{inverse } (\text{sqrt } x)$
 $\langle proof \rangle$

lemma *real-div-sqrt*: $0 \leq x \implies x / \text{sqrt } x = \text{sqrt } x$
 $\langle proof \rangle$

lemma *real-divide-square-eq* [*simp*]: $(r * a) / (r * r) = a / r$
for $a \ r :: \text{real}$
 $\langle proof \rangle$

lemma *lemma-real-divide-sqrt-less*: $0 < u \implies u / \text{sqrt } 2 < u$
 $\langle proof \rangle$

lemma *four-x-squared*: $4 * x^2 = (2 * x)^2$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *sqrt-at-top*: $\text{LIM } x \text{ at-top. } \text{sqrt } x :: \text{real} :> \text{at-top}$
 $\langle proof \rangle$

111.4 Square Root of Sum of Squares

lemma *sum-squares-bound*: $2 * x * y \leq x^2 + y^2$
for $x \ y :: 'a::\text{linordered-field}$
 $\langle proof \rangle$

lemma *arith-geo-mean*:
fixes $u :: 'a::\text{linordered-field}$
assumes $u^2 = x * y \ x \geq 0 \ y \geq 0$
shows $u \leq (x + y)/2$
 $\langle proof \rangle$

lemma *arith-geo-mean-sqrt*:
fixes $x :: \text{real}$
assumes $x \geq 0 \ y \geq 0$
shows $\text{sqrt } (x * y) \leq (x + y)/2$
 $\langle proof \rangle$

lemma *real-sqrt-sum-squares-mult-ge-zero* [*simp*]: $0 \leq \text{sqrt } ((x^2 + y^2) * (xa^2 + ya^2))$
 $\langle proof \rangle$

lemma *real-sqrt-sum-squares-mult-squared-eq* [simp]:
 $(\text{sqrt } ((x^2 + y^2) * (xa^2 + ya^2)))^2 = (x^2 + y^2) * (xa^2 + ya^2)$
 ⟨proof⟩

lemma *real-sqrt-sum-squares-eq-cancel*: $\text{sqrt } (x^2 + y^2) = x \implies y = 0$
 ⟨proof⟩

lemma *real-sqrt-sum-squares-eq-cancel2*: $\text{sqrt } (x^2 + y^2) = y \implies x = 0$
 ⟨proof⟩

lemma *real-sqrt-sum-squares-ge1* [simp]: $x \leq \text{sqrt } (x^2 + y^2)$
 ⟨proof⟩

lemma *real-sqrt-sum-squares-ge2* [simp]: $y \leq \text{sqrt } (x^2 + y^2)$
 ⟨proof⟩

lemma *real-sqrt-ge-abs1* [simp]: $|x| \leq \text{sqrt } (x^2 + y^2)$
 ⟨proof⟩

lemma *real-sqrt-ge-abs2* [simp]: $|y| \leq \text{sqrt } (x^2 + y^2)$
 ⟨proof⟩

lemma *le-real-sqrt-sumsq* [simp]: $x \leq \text{sqrt } (x * x + y * y)$
 ⟨proof⟩

lemma *sqrt-sum-squares-le-sum*:
 $\llbracket 0 \leq x; 0 \leq y \rrbracket \implies \text{sqrt } (x^2 + y^2) \leq x + y$
 ⟨proof⟩

lemma *L2-set-mult-ineq-lemma*:
fixes $a\ b\ c\ d :: \text{real}$
shows $2 * (a * c) * (b * d) \leq a^2 * d^2 + b^2 * c^2$
 ⟨proof⟩

lemma *sqrt-sum-squares-le-sum-abs*: $\text{sqrt } (x^2 + y^2) \leq |x| + |y|$
 ⟨proof⟩

lemma *real-sqrt-sum-squares-triangle-ineq*:
 $\text{sqrt } ((a + c)^2 + (b + d)^2) \leq \text{sqrt } (a^2 + b^2) + \text{sqrt } (c^2 + d^2)$
 ⟨proof⟩

lemma *real-sqrt-sum-squares-less*: $|x| < u / \text{sqrt } 2 \implies |y| < u / \text{sqrt } 2 \implies \text{sqrt } (x^2 + y^2) < u$
 ⟨proof⟩

lemma *sqrt2-less-2*: $\text{sqrt } 2 < (2 :: \text{real})$
 ⟨proof⟩

lemma *sqrt-sum-squares-half-less*:

$x < u/2 \implies y < u/2 \implies 0 \leq x \implies 0 \leq y \implies \text{sqrt } (x^2 + y^2) < u$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-root*: $(\lambda n. \text{root } n \ n) \longrightarrow 1$
 $\langle \text{proof} \rangle$

lemma *LIMSEQ-root-const*:
assumes $0 < c$
shows $(\lambda n. \text{root } n \ c) \longrightarrow 1$
 $\langle \text{proof} \rangle$

Legacy theorem names:

lemmas *real-root-pos2 = real-root-power-cancel*
lemmas *real-root-pos-pos = real-root-gt-zero [THEN order-less-imp-le]*
lemmas *real-root-pos-pos-le = real-root-ge-zero*
lemmas *real-sqrt-eq-zero-cancel-iff = real-sqrt-eq-0-iff*

end

112 Power Series, Transcendental Functions etc.

theory *Transcendental*
imports *Series Deriv NthRoot*
begin

A theorem about the factorial function on the reals.

lemma *square-fact-le-2-fact*: $\text{fact } n * \text{fact } n \leq (\text{fact } (2 * n) :: \text{real})$
 $\langle \text{proof} \rangle$

lemma *fact-in-Reals*: $\text{fact } n \in \mathbb{R}$
 $\langle \text{proof} \rangle$

lemma *of-real-fact [simp]*: $\text{of-real } (\text{fact } n) = \text{fact } n$
 $\langle \text{proof} \rangle$

lemma *pochhammer-of-real*: $\text{pochhammer } (\text{of-real } x) \ n = \text{of-real } (\text{pochhammer } x \ n)$
 $\langle \text{proof} \rangle$

lemma *norm-fact [simp]*: $\text{norm } (\text{fact } n :: 'a::\text{real-normed-algebra-1}) = \text{fact } n$
 $\langle \text{proof} \rangle$

lemma *root-test-convergence*:
fixes $f :: \text{nat} \Rightarrow 'a::\text{banach}$
assumes $f: (\lambda n. \text{root } n \ (\text{norm } (f \ n))) \longrightarrow x$ — could be weakened to \limsup
and $x < 1$
shows *summable f*
 $\langle \text{proof} \rangle$

112.1 Properties of Power Series

lemma *powser-zero* [simp]: $(\sum n. f\ n * 0^{\wedge} n) = f\ 0$
for $f :: \text{nat} \Rightarrow 'a::\text{real-normed-algebra-1}$
 ⟨proof⟩

lemma *powser-sums-zero*: $(\lambda n. a\ n * 0^{\wedge} n)$ sums $a\ 0$
for $a :: \text{nat} \Rightarrow 'a::\text{real-normed-div-algebra}$
 ⟨proof⟩

lemma *powser-sums-zero-iff* [simp]: $(\lambda n. a\ n * 0^{\wedge} n)$ sums $x \longleftrightarrow a\ 0 = x$
for $a :: \text{nat} \Rightarrow 'a::\text{real-normed-div-algebra}$
 ⟨proof⟩

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 $x\ z :: 'a::\text{real-normed-div-algebra}$
assumes 1: *summable* $(\lambda n. f\ n * x^{\wedge} n)$
and 2: $\text{norm}\ z < \text{norm}\ x$
shows *summable* $(\lambda n. \text{norm}\ (f\ n * z^{\wedge} n))$
 ⟨proof⟩

lemma *powser-inside*:
fixes $f :: \text{nat} \Rightarrow 'a::\{\text{real-normed-div-algebra}, \text{banach}\}$
shows
 $\text{summable}\ (\lambda n. f\ n * (x^{\wedge} n)) \implies \text{norm}\ z < \text{norm}\ x \implies$
 $\text{summable}\ (\lambda n. f\ n * (z^{\wedge} n))$
 ⟨proof⟩

lemma *powser-times-n-limit-0*:
fixes $x :: 'a::\{\text{real-normed-div-algebra}, \text{banach}\}$
assumes $\text{norm}\ x < 1$
shows $(\lambda n. \text{of-nat}\ n * x^{\wedge} n) \longrightarrow 0$
 ⟨proof⟩

corollary *lim-n-over-pown*:
fixes $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
shows $1 < \text{norm}\ x \implies ((\lambda n. \text{of-nat}\ n / x^{\wedge} n) \longrightarrow 0)$ *sequentially*
 ⟨proof⟩

lemma *sum-split-even-odd*:
fixes $f :: \text{nat} \Rightarrow \text{real}$
shows $(\sum i < 2 * n. \text{if even } i \text{ then } f\ i \text{ else } g\ i) = (\sum i < n. f\ (2 * i)) + (\sum i < n. g\ (2 * i + 1))$
 ⟨proof⟩

lemma *sums-if'*:
fixes $g :: \text{nat} \Rightarrow \text{real}$
assumes g sums x

shows $(\lambda n. \text{if even } n \text{ then } 0 \text{ else } g((n - 1) \text{ div } 2)) \text{ sums } x$
 $\langle \text{proof} \rangle$

lemma *sums-if*:

fixes $g :: \text{nat} \Rightarrow \text{real}$
assumes $g \text{ sums } x$ **and** $f \text{ sums } y$
shows $(\lambda n. \text{if even } n \text{ then } f(n \text{ div } 2) \text{ else } g((n - 1) \text{ div } 2)) \text{ sums } (x + y)$
 $\langle \text{proof} \rangle$

112.2 Alternating series test / Leibniz formula

lemma *sums-alternating-upper-lower*:

fixes $a :: \text{nat} \Rightarrow \text{real}$
assumes $\text{mono}: \bigwedge n. a(\text{Suc } n) \leq a n$
and $a\text{-pos}: \bigwedge n. 0 \leq a n$
and $a \longrightarrow 0$
shows $\exists l. ((\forall n. (\sum_{i < 2*n} (-1)^{i*a} i) \leq l) \wedge (\lambda n. \sum_{i < 2*n} (-1)^{i*a} i) \longrightarrow l) \wedge$
 $(\forall n. l \leq (\sum_{i < 2*n+1} (-1)^{i*a} i)) \wedge (\lambda n. \sum_{i < 2*n+1} (-1)^{i*a} i) \longrightarrow l)$
(is $\exists l. ((\forall n. ?f n \leq l) \wedge -) \wedge ((\forall n. l \leq ?g n) \wedge -))$
 $\langle \text{proof} \rangle$

lemma *summable-Leibniz'*:

fixes $a :: \text{nat} \Rightarrow \text{real}$
assumes $a\text{-zero}: a \longrightarrow 0$
and $a\text{-pos}: \bigwedge n. 0 \leq a n$
and $a\text{-monotone}: \bigwedge n. a(\text{Suc } n) \leq a n$
shows $\text{summable } (\lambda n. (-1)^n * a n)$
and $\bigwedge n. (\sum_{i < 2*n} (-1)^{i*a} i) \leq (\sum i. (-1)^{i*a} i)$
and $(\lambda n. \sum_{i < 2*n} (-1)^{i*a} i) \longrightarrow (\sum i. (-1)^{i*a} i)$
and $\bigwedge n. (\sum i. (-1)^{i*a} i) \leq (\sum_{i < 2*n+1} (-1)^{i*a} i)$
and $(\lambda n. \sum_{i < 2*n+1} (-1)^{i*a} i) \longrightarrow (\sum i. (-1)^{i*a} i)$
 $\langle \text{proof} \rangle$

theorem *summable-Leibniz*:

fixes $a :: \text{nat} \Rightarrow \text{real}$
assumes $a\text{-zero}: a \longrightarrow 0$
and $\text{monoseq } a$
shows $\text{summable } (\lambda n. (-1)^n * a n)$ **(is** $?summable)$
and $0 < a 0 \longrightarrow$
 $(\forall n. (\sum i. (-1)^{i*a} i) \in \{ \sum_{i < 2*n} (-1)^{i*a} i .. \sum_{i < 2*n+1} (-1)^{i*a} i \})$ **(is** $?pos)$
and $a 0 < 0 \longrightarrow$
 $(\forall n. (\sum i. (-1)^{i*a} i) \in \{ \sum_{i < 2*n+1} (-1)^{i*a} i .. \sum_{i < 2*n} (-1)^{i*a} i \})$ **(is** $?neg)$
and $(\lambda n. \sum_{i < 2*n} (-1)^{i*a} i) \longrightarrow (\sum i. (-1)^{i*a} i)$ **(is** $?f)$
and $(\lambda n. \sum_{i < 2*n+1} (-1)^{i*a} i) \longrightarrow (\sum i. (-1)^{i*a} i)$ **(is** $?g)$
 $\langle \text{proof} \rangle$

112.3 Term-by-Term Differentiability of Power Series

definition $\text{diffs} :: (\text{nat} \Rightarrow 'a::\text{ring-1}) \Rightarrow \text{nat} \Rightarrow 'a$
where $\text{diffs } c = (\lambda n. \text{of-nat } (\text{Suc } n) * c (\text{Suc } n))$

Lemma about distributing negation over it.

lemma diffs-minus : $\text{diffs } (\lambda n. - c n) = (\lambda n. - \text{diffs } c n)$
 $\langle \text{proof} \rangle$

lemma diffs-equiv :
fixes $x :: 'a::\{\text{real-normed-vector}, \text{ring-1}\}$
shows $\text{summable } (\lambda n. \text{diffs } c n * x^n) \implies$
 $(\lambda n. \text{of-nat } n * c n * x^{(n - \text{Suc } 0)}) \text{ sums } (\sum n. \text{diffs } c n * x^n)$
 $\langle \text{proof} \rangle$

lemma lemma-termdiff1 :
fixes $z :: 'a :: \{\text{monoid-mult}, \text{comm-ring}\}$
shows $(\sum p < m. (((z + h) ^ (m - p)) * (z ^ p)) - (z ^ m)) =$
 $(\sum p < m. (z ^ p) * (((z + h) ^ (m - p)) - (z ^ (m - p))))$
 $\langle \text{proof} \rangle$

lemma $\text{sumr-diff-mult-const2}$: $\text{sum } f \{.. $n\} - \text{of-nat } n * r = (\sum i < n. f i - r)$
for $r :: 'a::\text{ring-1}$
 $\langle \text{proof} \rangle$$

lemma lemma-termdiff2 :
fixes $h :: 'a::\text{field}$
assumes $h: h \neq 0$
shows $((z + h) ^ n - z ^ n) / h - \text{of-nat } n * z ^ (n - \text{Suc } 0) =$
 $h * (\sum p < n - \text{Suc } 0. \sum q < n - \text{Suc } 0 - p. (z + h) ^ q * z ^ (n - 2 -$
 $q))$
 $(\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma $\text{real-sum-nat-ivl-bounded2}$:
fixes $K :: 'a::\text{linordered-semidom}$
assumes $f: \bigwedge p::\text{nat}. p < n \implies f p \leq K$ **and** $K: 0 \leq K$
shows $\text{sum } f \{.. $n-k\} \leq \text{of-nat } n * K$
 $\langle \text{proof} \rangle$$

lemma lemma-termdiff3 :
fixes $h z :: 'a::\text{real-normed-field}$
assumes $1: h \neq 0$
and $2: \text{norm } z \leq K$
and $3: \text{norm } (z + h) \leq K$
shows $\text{norm } (((z + h) ^ n - z ^ n) / h - \text{of-nat } n * z ^ (n - \text{Suc } 0)) \leq$
 $\text{of-nat } n * \text{of-nat } (n - \text{Suc } 0) * K ^ (n - 2) * \text{norm } h$
 $\langle \text{proof} \rangle$

lemma *lemma-termdiff4*:
fixes $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-vector}$
and $k :: \text{real}$
assumes $k: 0 < k$
and $le: \bigwedge h. h \neq 0 \implies \text{norm } h < k \implies \text{norm } (f h) \leq K * \text{norm } h$
shows $f - 0 \rightarrow 0$
 $\langle \text{proof} \rangle$

lemma *lemma-termdiff5*:
fixes $g :: 'a::\text{real-normed-vector} \Rightarrow \text{nat} \Rightarrow 'b::\text{banach}$
and $k :: \text{real}$
assumes $k: 0 < k$
and $f: \text{summable } f$
and $le: \bigwedge h n. h \neq 0 \implies \text{norm } h < k \implies \text{norm } (g h n) \leq f n * \text{norm } h$
shows $(\lambda h. \text{suminf } (g h)) - 0 \rightarrow 0$
 $\langle \text{proof} \rangle$

lemma *termdiffs-aux*:
fixes $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes $1: \text{summable } (\lambda n. \text{diffs } (\text{diffs } c) n * K ^ n)$
and $2: \text{norm } x < \text{norm } K$
shows $(\lambda h. \sum n. c n * (((x + h) ^ n - x ^ n) / h - \text{of-nat } n * x ^ (n - \text{Suc } 0)))$
 $- 0 \rightarrow 0$
 $\langle \text{proof} \rangle$

lemma *termdiffs*:
fixes $K x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes $1: \text{summable } (\lambda n. c n * K ^ n)$
and $2: \text{summable } (\lambda n. (\text{diffs } c) n * K ^ n)$
and $3: \text{summable } (\lambda n. (\text{diffs } (\text{diffs } c)) n * K ^ n)$
and $4: \text{norm } x < \text{norm } K$
shows $\text{DERIV } (\lambda x. \sum n. c n * x ^ n) x :> (\sum n. (\text{diffs } c) n * x ^ n)$
 $\langle \text{proof} \rangle$

112.4 The Derivative of a Power Series Has the Same Radius of Convergence

lemma *termdiff-converges*:
fixes $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes $K: \text{norm } x < K$
and $sm: \bigwedge x. \text{norm } x < K \implies \text{summable } (\lambda n. c n * x ^ n)$
shows $\text{summable } (\lambda n. \text{diffs } c n * x ^ n)$
 $\langle \text{proof} \rangle$

lemma *termdiff-converges-all*:
fixes $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$

assumes $\bigwedge x. \text{summable } (\lambda n. c\ n * x^{\wedge} n)$
shows $\text{summable } (\lambda n. \text{diffs } c\ n * x^{\wedge} n)$
 $\langle \text{proof} \rangle$

lemma *termdiffs-strong*:
fixes $K\ x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
assumes $sm: \text{summable } (\lambda n. c\ n * K^{\wedge} n)$
and $K: \text{norm } x < \text{norm } K$
shows $\text{DERIV } (\lambda x. \sum n. c\ n * x^{\wedge} n)\ x :> (\sum n. \text{diffs } c\ n * x^{\wedge} n)$
 $\langle \text{proof} \rangle$

lemma *termdiffs-strong-converges-everywhere*:
fixes $K\ x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
assumes $\bigwedge y. \text{summable } (\lambda n. c\ n * y^{\wedge} n)$
shows $((\lambda x. \sum n. c\ n * x^{\wedge} n) \text{ has-field-derivative } (\sum n. \text{diffs } c\ n * x^{\wedge} n))\ (at\ x)$
 $\langle \text{proof} \rangle$

lemma *termdiffs-strong'*:
fixes $z :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
assumes $\bigwedge z. \text{norm } z < K \implies \text{summable } (\lambda n. c\ n * z^{\wedge} n)$
assumes $\text{norm } z < K$
shows $((\lambda z. \sum n. c\ n * z^{\wedge} n) \text{ has-field-derivative } (\sum n. \text{diffs } c\ n * z^{\wedge} n))\ (at\ z)$
 $\langle \text{proof} \rangle$

lemma *termdiffs-sums-strong*:
fixes $z :: 'a :: \{\text{banach}, \text{real-normed-field}\}$
assumes $\text{sums}: \bigwedge z. \text{norm } z < K \implies (\lambda n. c\ n * z^{\wedge} n) \text{ sums } f\ z$
assumes $\text{deriv}: (f \text{ has-field-derivative } f')\ (at\ z)$
assumes $\text{norm}: \text{norm } z < K$
shows $(\lambda n. \text{diffs } c\ n * z^{\wedge} n) \text{ sums } f'$
 $\langle \text{proof} \rangle$

lemma *isCont-powser*:
fixes $K\ x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
assumes $\text{summable } (\lambda n. c\ n * K^{\wedge} n)$
assumes $\text{norm } x < \text{norm } K$
shows $\text{isCont } (\lambda x. \sum n. c\ n * x^{\wedge} n)\ x$
 $\langle \text{proof} \rangle$

lemmas $\text{isCont-powser}' = \text{isCont-o2}[OF - \text{isCont-powser}]$

lemma *isCont-powser-converges-everywhere*:
fixes $K\ x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
assumes $\bigwedge y. \text{summable } (\lambda n. c\ n * y^{\wedge} n)$
shows $\text{isCont } (\lambda x. \sum n. c\ n * x^{\wedge} n)\ x$
 $\langle \text{proof} \rangle$

lemma *powser-limit-0*:
fixes $a :: \text{nat} \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$

assumes $s: 0 < s$
and $sm: \bigwedge x. \text{norm } x < s \implies (\lambda n. a \ n * x^{\wedge} n) \text{ sums } (f \ x)$
shows $(f \longrightarrow a \ 0) \text{ (at } 0)$
 $\langle \text{proof} \rangle$

lemma *powser-limit-0-strong*:
fixes $a :: \text{nat} \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
assumes $s: 0 < s$
and $sm: \bigwedge x. x \neq 0 \implies \text{norm } x < s \implies (\lambda n. a \ n * x^{\wedge} n) \text{ sums } (f \ x)$
shows $(f \longrightarrow a \ 0) \text{ (at } 0)$
 $\langle \text{proof} \rangle$

112.5 Derivability of power series

lemma *DERIV-series'*:
fixes $f :: \text{real} \Rightarrow \text{nat} \Rightarrow \text{real}$
assumes $\text{DERIV-}f: \bigwedge n. \text{DERIV } (\lambda x. f \ x \ n) \ x0 :> (f' \ x0 \ n)$
and $\text{allf-summable}: \bigwedge x. x \in \{a <..< b\} \implies \text{summable } (f \ x)$
and $\text{x0-in-I}: x0 \in \{a <..< b\}$
and $\text{summable } (f' \ x0)$
and $\text{summable } L$
and $L\text{-def}: \bigwedge n \ x \ y. x \in \{a <..< b\} \implies y \in \{a <..< b\} \implies |f \ x \ n - f \ y \ n| \leq$
 $L \ n * |x - y|$
shows $\text{DERIV } (\lambda x. \text{suminf } (f \ x)) \ x0 :> (\text{suminf } (f' \ x0))$
 $\langle \text{proof} \rangle$

lemma *DERIV-power-series'*:
fixes $f :: \text{nat} \Rightarrow \text{real}$
assumes $\text{converges}: \bigwedge x. x \in \{-R <..< R\} \implies \text{summable } (\lambda n. f \ n * \text{real } (\text{Suc } n) * x^{\wedge} n)$
and $\text{x0-in-I}: x0 \in \{-R <..< R\}$
and $0 < R$
shows $\text{DERIV } (\lambda x. (\sum n. f \ n * x^{\wedge} (\text{Suc } n))) \ x0 :> (\sum n. f \ n * \text{real } (\text{Suc } n) * x0^{\wedge} n)$
(is $\text{DERIV } (\lambda x. \text{suminf } (?f \ x)) \ x0 :> \text{suminf } (?f' \ x0))$
 $\langle \text{proof} \rangle$

lemma *geometric-deriv-sums*:
fixes $z :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
assumes $\text{norm } z < 1$
shows $(\lambda n. \text{of-nat } (\text{Suc } n) * z^{\wedge} n) \text{ sums } (1 / (1 - z)^{\wedge} 2)$
 $\langle \text{proof} \rangle$

lemma *isCont-pochhammer* [*continuous-intros*]: $\text{isCont } (\lambda z. \text{pochhammer } z \ n) \ z$
for $z :: 'a :: \text{real-normed-field}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-pochhammer* [*continuous-intros*]: $\text{continuous-on } A \ (\lambda z. \text{pochhammer } z \ n)$

for $A :: 'a::\text{real-normed-field set}$
 $\langle \text{proof} \rangle$

lemmas *continuous-on-pochhammer'* [*continuous-intros*] =
continuous-on-compose2[*OF continuous-on-pochhammer - subset-UNIV*]

112.6 Exponential Function

definition $\exp :: 'a \Rightarrow 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
where $\exp = (\lambda x. \sum n. x^n /_R \text{fact } n)$

lemma *summable-exp-generic*:
fixes $x :: 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
defines $S\text{-def}: S \equiv \lambda n. x^n /_R \text{fact } n$
shows *summable* S
 $\langle \text{proof} \rangle$

lemma *summable-norm-exp*: *summable* $(\lambda n. \text{norm } (x^n /_R \text{fact } n))$
for $x :: 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *summable-exp*: *summable* $(\lambda n. \text{inverse } (\text{fact } n) * x^n)$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *exp-converges*: $(\lambda n. x^n /_R \text{fact } n)$ *sums* $\exp x$
 $\langle \text{proof} \rangle$

lemma *exp-fdiffs*:
 $\text{diffs } (\lambda n. \text{inverse } (\text{fact } n)) = (\lambda n. \text{inverse } (\text{fact } n :: 'a::\{\text{real-normed-field}, \text{banach}\}))$
 $\langle \text{proof} \rangle$

lemma *diffs-of-real*: $\text{diffs } (\lambda n. \text{of-real } (f\ n)) = (\lambda n. \text{of-real } (\text{diffs } f\ n))$
 $\langle \text{proof} \rangle$

lemma *DERIV-exp* [*simp*]: *DERIV* $\exp x :> \exp x$
 $\langle \text{proof} \rangle$

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*: $\text{norm } (\exp x) \leq \exp (\text{norm } x)$
 $\langle \text{proof} \rangle$

lemma *isCont-exp*: *isCont* $\exp x$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$

$\langle \text{proof} \rangle$

lemma *isCont-exp'* [simp]: $\text{isCont } f \ a \implies \text{isCont } (\lambda x. \exp (f \ x)) \ a$
for $f :: - \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *tendsto-exp* [tendsto-intros]: $(f \longrightarrow a) \ F \implies ((\lambda x. \exp (f \ x)) \longrightarrow \exp a) \ F$
for $f :: - \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *continuous-exp* [continuous-intros]: $\text{continuous } F \ f \implies \text{continuous } F \ (\lambda x. \exp (f \ x))$
for $f :: - \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-exp* [continuous-intros]: $\text{continuous-on } s \ f \implies \text{continuous-on } s \ (\lambda x. \exp (f \ x))$
for $f :: - \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

112.6.1 Properties of the Exponential Function

lemma *exp-zero* [simp]: $\exp 0 = 1$
 $\langle \text{proof} \rangle$

lemma *exp-series-add-commuting*:
fixes $x \ y :: 'a :: \{\text{real-normed-algebra-1}, \text{banach}\}$
defines $S\text{-def}: S \equiv \lambda x \ n. \ x^n /_R \text{fact } n$
assumes *comm*: $x * y = y * x$
shows $S \ (x + y) \ n = (\sum_{i \leq n} S \ x \ i * S \ y \ (n - i))$
 $\langle \text{proof} \rangle$

lemma *exp-add-commuting*: $x * y = y * x \implies \exp (x + y) = \exp x * \exp y$
 $\langle \text{proof} \rangle$

lemma *exp-times-arg-commute*: $\exp A * A = A * \exp A$
 $\langle \text{proof} \rangle$

lemma *exp-add*: $\exp (x + y) = \exp x * \exp y$
for $x \ y :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *exp-double*: $\exp(2 * z) = \exp z ^ 2$
 $\langle \text{proof} \rangle$

lemmas *mult-exp-exp* = *exp-add* [symmetric]

lemma *exp-of-real*: $\exp (\text{of-real } x) = \text{of-real } (\exp x)$

$\langle \text{proof} \rangle$

lemmas *of-real-exp* = *exp-of-real*[*symmetric*]

corollary *exp-in-Reals* [*simp*]: $z \in \mathbb{R} \implies \exp z \in \mathbb{R}$
 $\langle \text{proof} \rangle$

lemma *exp-not-eq-zero* [*simp*]: $\exp x \neq 0$
 $\langle \text{proof} \rangle$

lemma *exp-minus-inverse*: $\exp x * \exp (-x) = 1$
 $\langle \text{proof} \rangle$

lemma *exp-minus*: $\exp (-x) = \text{inverse} (\exp x)$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *exp-diff*: $\exp (x - y) = \exp x / \exp y$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *exp-of-nat-mult*: $\exp (\text{of-nat } n * x) = \exp x ^ n$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

corollary *exp-of-nat2-mult*: $\exp (x * \text{of-nat } n) = \exp x ^ n$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *exp-sum*: $\text{finite } I \implies \exp (\text{sum } f I) = \text{prod } (\lambda x. \exp (f x)) I$
 $\langle \text{proof} \rangle$

lemma *exp-divide-power-eq*:
fixes $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes $n > 0$
shows $\exp (x / \text{of-nat } n) ^ n = \exp x$
 $\langle \text{proof} \rangle$

lemma *exp-power-int*:
fixes $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
shows $\exp x \text{ powi } n = \exp (\text{of-int } n * x)$
 $\langle \text{proof} \rangle$

112.6.2 Properties of the Exponential Function on Reals

Comparisons of $\exp x$ with zero.

Proof: because every exponential can be seen as a square.

lemma *exp-ge-zero* [*simp*]: $0 \leq \exp x$

for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *exp-gt-zero* [*simp*]: $0 < \exp x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *not-exp-less-zero* [*simp*]: $\neg \exp x < 0$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *not-exp-le-zero* [*simp*]: $\neg \exp x \leq 0$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *abs-exp-cancel* [*simp*]: $|\exp x| = \exp x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

Strict monotonicity of exponential.

lemma *exp-ge-add-one-self-aux*:
fixes $x :: \text{real}$
assumes $0 \leq x$
shows $1 + x \leq \exp x$
 $\langle \text{proof} \rangle$

lemma *exp-gt-one*: $0 < x \implies 1 < \exp x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *exp-less-mono*:
fixes $x y :: \text{real}$
assumes $x < y$
shows $\exp x < \exp y$
 $\langle \text{proof} \rangle$

lemma *exp-less-cancel*: $\exp x < \exp y \implies x < y$
for $x y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *exp-less-cancel-iff* [*iff*]: $\exp x < \exp y \longleftrightarrow x < y$
for $x y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *exp-le-cancel-iff* [*iff*]: $\exp x \leq \exp y \longleftrightarrow x \leq y$
for $x y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *exp-mono*:

```

fixes  $x\ y :: \text{real}$ 
assumes  $x \leq y$ 
shows  $\exp x \leq \exp y$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma exp-minus':  $\exp(-x) = 1 / (\exp x)$ 
for  $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma exp-inj-iff [iff]:  $\exp x = \exp y \longleftrightarrow x = y$ 
for  $x\ y :: \text{real}$ 
 $\langle \text{proof} \rangle$ 

```

Comparisons of $\exp x$ with one.

```

lemma one-less-exp-iff [simp]:  $1 < \exp x \longleftrightarrow 0 < x$ 
for  $x :: \text{real}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma exp-less-one-iff [simp]:  $\exp x < 1 \longleftrightarrow x < 0$ 
for  $x :: \text{real}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma one-le-exp-iff [simp]:  $1 \leq \exp x \longleftrightarrow 0 \leq x$ 
for  $x :: \text{real}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma exp-le-one-iff [simp]:  $\exp x \leq 1 \longleftrightarrow x \leq 0$ 
for  $x :: \text{real}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma exp-eq-one-iff [simp]:  $\exp x = 1 \longleftrightarrow x = 0$ 
for  $x :: \text{real}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma lemma-exp-total:  $1 \leq y \implies \exists x. 0 \leq x \wedge x \leq y - 1 \wedge \exp x = y$ 
for  $y :: \text{real}$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma exp-total:  $0 < y \implies \exists x. \exp x = y$ 
for  $y :: \text{real}$ 
 $\langle \text{proof} \rangle$ 

```

112.7 Natural Logarithm

```

class  $\text{ln} = \text{real-normed-algebra-1} + \text{banach} +$ 
fixes  $\text{ln} :: 'a \Rightarrow 'a$ 
assumes ln-one [simp]:  $\text{ln } 1 = 0$ 

```

```

definition powr ::  $'a \Rightarrow 'a \Rightarrow 'a :: \text{ln}$  (infixr  $\langle \text{powr} \rangle$  80)

```

— exponentation via \ln and \exp
where $x \text{ powr } a \equiv \text{if } x = 0 \text{ then } 0 \text{ else } \exp (a * \ln x)$

lemma *powr-0* [simp]: $0 \text{ powr } z = 0$
 $\langle \text{proof} \rangle$

We totalise \ln over all reals exactly as done in Mathlib

instantiation $\text{real} :: \ln$
begin

definition *raw-ln-real* :: $\text{real} \Rightarrow \text{real}$
where *raw-ln-real* $x \equiv (\text{THE } u. \exp u = x)$

definition *ln-real* :: $\text{real} \Rightarrow \text{real}$
where *ln-real* $\equiv \lambda x. \text{if } x=0 \text{ then } 0 \text{ else } \text{raw-ln-real } |x|$

instance
 $\langle \text{proof} \rangle$

end

lemma *powr-eq-0-iff* [simp]: $w \text{ powr } z = 0 \longleftrightarrow w = 0$
 $\langle \text{proof} \rangle$

lemma *raw-ln-exp* [simp]: $\text{raw-ln-real } (\exp x) = x$
 $\langle \text{proof} \rangle$

lemma *exp-raw-ln* [simp]: $0 < x \Longrightarrow \exp (\text{raw-ln-real } x) = x$
 $\langle \text{proof} \rangle$

lemma *raw-ln-unique*: $\exp y = x \Longrightarrow \text{raw-ln-real } x = y$
 $\langle \text{proof} \rangle$

lemma *abs-raw-ln*: $x \neq 0 \Longrightarrow \text{raw-ln-real } |x| = \ln x$
 $\langle \text{proof} \rangle$

lemma *ln-0* [simp]: $\ln (0::\text{real}) = 0$
 $\langle \text{proof} \rangle$

lemma *ln-minus*: $\ln (-x) = \ln x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-exp* [simp]: $\ln (\exp x) = x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *exp-ln-abs*:
fixes $x::\text{real}$

shows $x \neq 0 \implies \exp (\ln x) = |x|$
 $\langle \text{proof} \rangle$

lemma *exp-ln [simp]*: $0 < x \implies \exp (\ln x) = x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *exp-ln-iff [simp]*: $\exp (\ln x) = x \longleftrightarrow 0 < x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-unique*: $\exp y = x \implies \ln x = y$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-unique'*: $\exp y = |x| \implies \ln x = y$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *raw-ln-mult*: $x > 0 \implies y > 0 \implies \text{raw-ln-real } (x * y) = \text{raw-ln-real } x +$
 $\text{raw-ln-real } y$
 $\langle \text{proof} \rangle$

lemma *ln-mult*: $\ln (x * y) = (\text{if } x \neq 0 \wedge y \neq 0 \text{ then } \ln x + \ln y \text{ else } 0)$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-mult-pos*: $x > 0 \implies y > 0 \implies \ln (x * y) = \ln x + \ln y$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-prod*: $\text{finite } I \implies (\bigwedge i. i \in I \implies f i \neq 0) \implies \ln (\text{prod } f I) = \text{sum } (\lambda x.$
 $\ln(f x)) I$
for $f :: 'a \Rightarrow \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-inverse*: $\ln (\text{inverse } x) = - \ln x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-div*: $\ln (x/y) = (\text{if } x \neq 0 \wedge y \neq 0 \text{ then } \ln x - \ln y \text{ else } 0)$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-divide-pos*: $x > 0 \implies y > 0 \implies \ln (x/y) = \ln x - \ln y$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-realpow*: $\ln (x^{\wedge} n) = \text{real } n * \ln x$

$\langle proof \rangle$

lemma *ln-less-cancel-iff* [simp]: $0 < x \implies 0 < y \implies \ln x < \ln y \longleftrightarrow x < y$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *ln-le-cancel-iff* [simp]: $0 < x \implies 0 < y \implies \ln x \leq \ln y \longleftrightarrow x \leq y$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *ln-mono*: $\bigwedge x :: \text{real}. \llbracket x \leq y; 0 < x \rrbracket \implies \ln x \leq \ln y$
 $\langle proof \rangle$

lemma *ln-strict-mono*: $\bigwedge x :: \text{real}. \llbracket x < y; 0 < x \rrbracket \implies \ln x < \ln y$
 $\langle proof \rangle$

lemma *ln-inj-iff* [simp]: $0 < x \implies 0 < y \implies \ln x = \ln y \longleftrightarrow x = y$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *ln-add-one-self-le-self*: $0 \leq x \implies \ln (1 + x) \leq x$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *ln-less-self* [simp]: $0 < x \implies \ln x < x$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *ln-ge-iff*: $\bigwedge x :: \text{real}. 0 < x \implies y \leq \ln x \longleftrightarrow \exp y \leq x$
 $\langle proof \rangle$

lemma *ln-ge-zero* [simp]: $1 \leq x \implies 0 \leq \ln x$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *ln-ge-zero-imp-ge-one*: $0 \leq \ln x \implies 0 < x \implies 1 \leq x$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *ln-ge-zero-iff* [simp]: $0 < x \implies 0 \leq \ln x \longleftrightarrow 1 \leq x$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *ln-less-zero-iff* [simp]: $0 < x \implies \ln x < 0 \longleftrightarrow x < 1$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *ln-le-zero-iff* [simp]: $0 < x \implies \ln x \leq 0 \longleftrightarrow x \leq 1$
for $x :: \text{real}$

$\langle \text{proof} \rangle$

lemma *ln-gt-zero*: $1 < x \implies 0 < \ln x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-gt-zero-imp-gt-one*: $0 < \ln x \implies 0 < x \implies 1 < x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-gt-zero-iff [simp]*: $0 < x \implies 0 < \ln x \longleftrightarrow 1 < x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-eq-zero-iff [simp]*: $0 < x \implies \ln x = 0 \longleftrightarrow x = 1$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-less-zero*: $0 < x \implies x < 1 \implies \ln x < 0$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-eq-one-iff [simp]*:
 $a \text{ powr } x = 1 \longleftrightarrow x = 0 \text{ if } a > 1 \text{ for } a :: \text{real}$
 $\langle \text{proof} \rangle$

A consequence of our "totalising" of \ln

lemma *uminus-powr-eq*: $(-a) \text{ powr } x = a \text{ powr } x \text{ for } x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *isCont-ln-pos*:
fixes $x :: \text{real}$
assumes $x > 0$
shows *isCont* $\ln x$
 $\langle \text{proof} \rangle$

lemma *isCont-ln*:
fixes $x :: \text{real}$
assumes $x \neq 0$
shows *isCont* $\ln x$
 $\langle \text{proof} \rangle$

lemma *tendsto-ln [tendsto-intros]*: $(f \longrightarrow a) F \implies a \neq 0 \implies ((\lambda x. \ln (f x)) \longrightarrow \ln a) F$
for $a :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *continuous-ln*:
 $\text{continuous } F f \implies f (\text{Lim } F (\lambda x. x)) \neq 0 \implies \text{continuous } F (\lambda x. \ln (f x :: \text{real}))$

$\langle \text{proof} \rangle$

lemma *isCont-ln'* [*continuous-intros*]:

continuous (at x) $f \implies f\ x \neq 0 \implies \text{continuous}$ (at x) $(\lambda x. \ln (f\ x :: \text{real}))$

$\langle \text{proof} \rangle$

lemma *continuous-within-ln* [*continuous-intros*]:

continuous (at x within s) $f \implies f\ x \neq 0 \implies \text{continuous}$ (at x within s) $(\lambda x. \ln (f\ x :: \text{real}))$

$\langle \text{proof} \rangle$

lemma *continuous-on-ln* [*continuous-intros*]:

continuous-on $s\ f \implies (\forall x \in s. f\ x \neq 0) \implies \text{continuous-on}$ $s\ (\lambda x. \ln (f\ x :: \text{real}))$

$\langle \text{proof} \rangle$

lemma *DERIV-ln*: $0 < x \implies \text{DERIV } \ln\ x \text{ :> } \text{inverse } x$

for $x :: \text{real}$

$\langle \text{proof} \rangle$

lemma *DERIV-ln-divide*: $0 < x \implies \text{DERIV } \ln\ x \text{ :> } 1/x$

for $x :: \text{real}$

$\langle \text{proof} \rangle$

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 $\ln\ x = (\sum n. (-1)^n * (1 / \text{real } (n + 1)) * (x - 1)^n) \wedge (\text{Suc } n)$

(**is** $\ln\ x = \text{suminf } (?f\ (x - 1))$)

$\langle \text{proof} \rangle$

lemma *exp-first-terms*:

fixes $x :: 'a :: \{\text{real-normed-algebra-1}, \text{banach}\}$

shows $\exp\ x = (\sum n < k. \text{inverse}(\text{fact } n) *_R (x^{\wedge} n)) + (\sum n. \text{inverse}(\text{fact } (n + k)) *_R (x^{\wedge} (n + k)))$

$\langle \text{proof} \rangle$

lemma *exp-first-term*: $\exp\ x = 1 + (\sum n. \text{inverse } (\text{fact } (\text{Suc } n)) *_R (x^{\wedge} \text{Suc } n))$

for $x :: 'a :: \{\text{real-normed-algebra-1}, \text{banach}\}$

$\langle \text{proof} \rangle$

lemma *exp-first-two-terms*: $\exp\ x = 1 + x + (\sum n. \text{inverse } (\text{fact } (n + 2)) *_R (x^{\wedge} (n + 2)))$

for $x :: 'a :: \{\text{real-normed-algebra-1}, \text{banach}\}$

$\langle \text{proof} \rangle$

lemma *exp-bound*:

fixes $x :: \text{real}$

assumes $a: 0 \leq x$

and $b: x \leq 1$

shows $\exp x \leq 1 + x + x^2$

<proof>

corollary *exp-half-le2*: $\exp(1/2) \leq (2::\text{real})$

<proof>

corollary *exp-le*: $\exp 1 \leq (3::\text{real})$

<proof>

lemma *exp-bound-half*: $\text{norm } z \leq 1/2 \implies \text{norm } (\exp z) \leq 2$

<proof>

lemma *exp-bound-lemma*:

assumes $\text{norm } z \leq 1/2$

shows $\text{norm } (\exp z) \leq 1 + 2 * \text{norm } z$

<proof>

lemma *real-exp-bound-lemma*: $0 \leq x \implies x \leq 1/2 \implies \exp x \leq 1 + 2 * x$

for $x :: \text{real}$

<proof>

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>

lemma *exp-ge-add-one-self* [*simp*]: $1 + x \leq \exp x$

for $x :: \text{real}$

<proof>

lemma *exp-gt-self*: $x < \exp (x::\text{real})$

<proof>

lemma *ln-one-plus-pos-lower-bound*:

fixes $x :: \text{real}$

assumes $a: 0 \leq x$ **and** $b: x \leq 1$

shows $x - x^2 \leq \ln (1 + x)$

<proof>

lemma *ln-one-minus-pos-lower-bound*:

fixes $x :: \text{real}$

assumes $a: 0 \leq x$ **and** $b: x \leq 1/2$

shows $-x - 2 * x^2 \leq \ln (1 - x)$

$\langle proof \rangle$

lemma *ln-add-one-self-le-self2*:
fixes $x :: real$
shows $-1 < x \implies \ln (1 + x) \leq x$
 $\langle proof \rangle$

lemma *abs-ln-one-plus-x-minus-x-bound-nonneg*:
fixes $x :: real$
assumes $x: 0 \leq x$ **and** $x1: x \leq 1$
shows $|\ln (1 + x) - x| \leq x^2$
 $\langle proof \rangle$

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 $|\ln (1 + x) - x| \leq 2 * x^2$
 $\langle proof \rangle$

lemma *abs-ln-one-plus-x-minus-x-bound*:
fixes $x :: real$
assumes $|x| \leq 1/2$
shows $|\ln (1 + x) - x| \leq 2 * x^2$
 $\langle proof \rangle$

lemma *ln-x-over-x-mono*:
fixes $x :: real$
assumes $x: exp\ 1 \leq x$ $x \leq y$
shows $\ln y / y \leq \ln x / x$
 $\langle proof \rangle$

lemma *ln-le-minus-one*: $0 < x \implies \ln x \leq x - 1$
for $x :: real$
 $\langle proof \rangle$

corollary *ln-diff-le*: $0 < x \implies 0 < y \implies \ln x - \ln y \leq (x - y) / y$
for $x :: real$
 $\langle proof \rangle$

lemma *ln-add1-ge*:
fixes $t::real$
shows $t \geq 0 \implies \ln (t+1) \geq t / (1+t)$
 $\langle proof \rangle$

lemma *ln-eq-minus-one*:
fixes $x :: real$
assumes $0 < x$ $\ln x = x - 1$
shows $x = 1$
 $\langle proof \rangle$

corollary *ln-diff-less*: $0 < x \implies 0 < y \implies x \neq y \implies \ln x - \ln y < (x - y) / y$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-add1-gt*:
fixes $t :: \text{real}$
shows $t > 0 \implies \ln (t+1) > t / (1+t)$
 $\langle \text{proof} \rangle$

lemma *ln-add-one-self-less-self*:
fixes $x :: \text{real}$
assumes $x > 0$
shows $\ln (1 + x) < x$
 $\langle \text{proof} \rangle$

lemma *ln-x-over-x-tendsto-0*: $((\lambda x :: \text{real}. \ln x / x) \longrightarrow 0) \text{ at-top}$
 $\langle \text{proof} \rangle$

corollary *exp-1-gt-powr*:
assumes $x > (0 :: \text{real})$
shows $\exp 1 > (1 + 1/x) \text{ powr } x$
 $\langle \text{proof} \rangle$

lemma *exp-ge-one-plus-x-over-n-power-n*:
assumes $x \geq - \text{real } n \text{ } n > 0$
shows $(1 + x / \text{of-nat } n) ^ n \leq \exp x$
 $\langle \text{proof} \rangle$

lemma *exp-ge-one-minus-x-over-n-power-n*:
assumes $x \leq \text{real } n \text{ } n > 0$
shows $(1 - x / \text{of-nat } n) ^ n \leq \exp (-x)$
 $\langle \text{proof} \rangle$

lemma *exp-at-bot*: $(\exp \longrightarrow (0 :: \text{real})) \text{ at-bot}$
 $\langle \text{proof} \rangle$

lemma *exp-at-top*: $\text{LIM } x \text{ at-top. } \exp x :: \text{real} :> \text{at-top}$
 $\langle \text{proof} \rangle$

lemma *lim-exp-minus-1*: $((\lambda z :: 'a. (\exp(z) - 1) / z) \longrightarrow 1) (\text{at } 0)$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *ln-at-0*: $\text{LIM } x \text{ at-right } 0. \ln (x :: \text{real}) :> \text{at-bot}$
 $\langle \text{proof} \rangle$

lemma *ln-at-top*: $\text{LIM } x \text{ at-top. } \ln (x :: \text{real}) :> \text{at-top}$
 $\langle \text{proof} \rangle$

lemma *filtermap-ln-at-top*: *filtermap* (*ln*::*real* \Rightarrow *real*) *at-top* = *at-top*
 ⟨*proof*⟩

lemma *filtermap-exp-at-top*: *filtermap* (*exp*::*real* \Rightarrow *real*) *at-top* = *at-top*
 ⟨*proof*⟩

lemma *filtermap-ln-at-right*: *filtermap* *ln* (*at-right* (*0*::*real*)) = *at-bot*
 ⟨*proof*⟩

lemma *tendsto-power-div-exp-0*: $((\lambda x. x \wedge k / \exp x) \longrightarrow (0::\text{real})) \text{ at-top}$
 ⟨*proof*⟩

112.7.1 A couple of simple bounds

lemma *exp-plus-inverse-exp*:
fixes *x*::*real*
shows $2 \leq \exp x + \text{inverse}(\exp x)$
 ⟨*proof*⟩

lemma *real-le-x-sinh*:
fixes *x*::*real*
assumes $0 \leq x$
shows $x \leq (\exp x - \text{inverse}(\exp x)) / 2$
 ⟨*proof*⟩

lemma *real-le-abs-sinh*:
fixes *x*::*real*
shows $\text{abs } x \leq \text{abs}((\exp x - \text{inverse}(\exp x)) / 2)$
 ⟨*proof*⟩

112.8 The general logarithm

definition *log* :: *real* \Rightarrow *real* \Rightarrow *real*
 — logarithm of *x* to base *a*
where $\text{log } a \ x = \ln x / \ln a$

lemma *log-exp [simp]*: $\text{log } b (\exp x) = x / \ln b$
 ⟨*proof*⟩

lemma *tendsto-log [tendsto-intros]*:
 $(f \longrightarrow a) F \Longrightarrow (g \longrightarrow b) F \Longrightarrow 0 < a \Longrightarrow a \neq 1 \Longrightarrow b \neq 0 \Longrightarrow$
 $((\lambda x. \text{log } (f x) (g x)) \longrightarrow \text{log } a b) F$
 ⟨*proof*⟩

lemma *continuous-log*:
assumes *continuous F f*
and *continuous F g*
and $f (\text{Lim } F (\lambda x. x)) > 0$
and $f (\text{Lim } F (\lambda x. x)) \neq 1$

and $g \text{ (Lim } F \text{ (}\lambda x. x\text{))} \neq 0$
shows *continuous* $F \text{ (}\lambda x. \log (f x) (g x)\text{)}$
 $\langle \text{proof} \rangle$

lemma *continuous-at-within-log*[*continuous-intros*]:
assumes *continuous* $(\text{at } a \text{ within } s) f$
and *continuous* $(\text{at } a \text{ within } s) g$
and $0 < f a$
and $f a \neq 1$
and $g a \neq 0$
shows *continuous* $(\text{at } a \text{ within } s) (\lambda x. \log (f x) (g x))$
 $\langle \text{proof} \rangle$

lemma *continuous-on-log*[*continuous-intros*]:
assumes *continuous-on* $S f$ *continuous-on* $S g$
and $\forall x \in S. 0 < f x \ \forall x \in S. f x \neq 1 \ \forall x \in S. g x \neq 0$
shows *continuous-on* $S (\lambda x. \log (f x) (g x))$
 $\langle \text{proof} \rangle$

lemma *exp-powr-real*:
fixes $x :: \text{real}$ **shows** $\text{exp } x \text{ powr } y = \text{exp } (x * y)$
 $\langle \text{proof} \rangle$

lemma *powr-one-eq-one* [*simp*]: $1 \text{ powr } a = 1$
 $\langle \text{proof} \rangle$

lemma *powr-zero-eq-one* [*simp*]: $x \text{ powr } 0 = (\text{if } x = 0 \text{ then } 0 \text{ else } 1)$
 $\langle \text{proof} \rangle$

lemma *powr-eq-one-iff-gen*[*simp*]: $a \text{ powr } x = 1 \longleftrightarrow x = 0 \text{ if } a > 0 \ a \neq 1 \text{ for } a$
 $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-one-gt-zero-iff* [*simp*]: $x \text{ powr } 1 = x \longleftrightarrow 0 \leq x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

declare *powr-one-gt-zero-iff* [*THEN iffD2, simp*]

lemma *powr-diff*:
fixes $w :: 'a :: \{\text{ln, real-normed-field}\}$
shows $w \text{ powr } (z1 - z2) = w \text{ powr } z1 / w \text{ powr } z2$
 $\langle \text{proof} \rangle$

lemma *powr-mult*: $(x * y) \text{ powr } a = (x \text{ powr } a) * (y \text{ powr } a)$
for $a \ x \ y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *prod-powr-distrib*:
fixes $x :: 'a \Rightarrow \text{real}$

shows $(\text{prod } x \ I) \ \text{powr } r = (\prod_{i \in I}. x \ i \ \text{powr } r)$
 $\langle \text{proof} \rangle$

lemma *powr-ge-zero* [*simp*]: $0 \leq x \ \text{powr } y$
for $x \ y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-non-neg*[*simp*]: $\neg a \ \text{powr } x < 0$ **for** $a \ x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *inverse-powr*: $\bigwedge y :: \text{real}. \ \text{inverse } y \ \text{powr } a = \text{inverse } (y \ \text{powr } a)$
 $\langle \text{proof} \rangle$

lemma *powr-divide*: $(x / y) \ \text{powr } a = (x \ \text{powr } a) / (y \ \text{powr } a)$
for $a \ b \ x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-add*: $x \ \text{powr } (a + b) = (x \ \text{powr } a) * (x \ \text{powr } b)$
for $a \ b \ x :: 'a :: \{\text{ln}, \text{real-normed-field}\}$
 $\langle \text{proof} \rangle$

lemma *powr-mult-base*: $0 \leq x \implies x * x \ \text{powr } y = x \ \text{powr } (1 + y)$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-mult-base'*: $\text{abs } x * x \ \text{powr } y = x \ \text{powr } (1 + y)$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-powr*: $(x \ \text{powr } a) \ \text{powr } b = x \ \text{powr } (a * b)$
for $a \ b \ x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-power*:
fixes $z :: 'a :: \{\text{real-normed-field}, \text{ln}\}$
shows $z \neq 0 \implies (z \ \text{powr } u) ^ n = z \ \text{powr } (\text{of-nat } n * u)$
 $\langle \text{proof} \rangle$

lemma *powr-powr-swap*: $(x \ \text{powr } a) \ \text{powr } b = (x \ \text{powr } b) \ \text{powr } a$
for $a \ b \ x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-minus*: $x \ \text{powr } (- a) = \text{inverse } (x \ \text{powr } a)$
for $a \ x :: 'a :: \{\text{ln}, \text{real-normed-field}\}$
 $\langle \text{proof} \rangle$

lemma *powr-minus-divide*: $x \ \text{powr } (- a) = 1 / (x \ \text{powr } a)$
for $a \ x :: 'a :: \{\text{ln}, \text{real-normed-field}\}$
 $\langle \text{proof} \rangle$

lemma *powr-sum*:

assumes $x \neq 0$

shows $x \text{ powr sum } f \ A = (\prod_{y \in A} x \text{ powr } f \ y)$

$\langle \text{proof} \rangle$

lemma *divide-powr-uminus*: $a / b \text{ powr } c = a * b \text{ powr } (-c)$

for $a \ b \ c :: \text{real}$

$\langle \text{proof} \rangle$

lemma *powr-less-mono*: $a < b \implies 1 < x \implies x \text{ powr } a < x \text{ powr } b$

for $a \ b \ x :: \text{real}$

$\langle \text{proof} \rangle$

lemma *powr-less-cancel*: $x \text{ powr } a < x \text{ powr } b \implies 1 < x \implies a < b$

for $a \ b \ x :: \text{real}$

$\langle \text{proof} \rangle$

lemma *powr-less-cancel-iff* [simp]: $1 < x \implies x \text{ powr } a < x \text{ powr } b \longleftrightarrow a < b$

for $a \ b \ x :: \text{real}$

$\langle \text{proof} \rangle$

lemma *powr-le-cancel-iff* [simp]: $1 < x \implies x \text{ powr } a \leq x \text{ powr } b \longleftrightarrow a \leq b$

for $a \ b \ x :: \text{real}$

$\langle \text{proof} \rangle$

lemma *powr-realpow*: $0 < x \implies x \text{ powr } (\text{real } n) = x^{\wedge} n$

$\langle \text{proof} \rangle$

lemma *powr-realpow'*: $(z :: \text{real}) \geq 0 \implies n \neq 0 \implies z \text{ powr of-nat } n = z^{\wedge} n$

$\langle \text{proof} \rangle$

lemma *powr-real-of-int'*:

assumes $x \geq 0 \ x \neq 0 \vee n > 0$

shows $x \text{ powr real-of-int } n = \text{power-int } x \ n$

$\langle \text{proof} \rangle$

lemma *exp-minus-ge*:

fixes $x :: \text{real}$ **shows** $1 - x \leq \exp(-x)$

$\langle \text{proof} \rangle$

lemma *exp-minus-greater*:

fixes $x :: \text{real}$ **shows** $1 - x < \exp(-x) \longleftrightarrow x \neq 0$

$\langle \text{proof} \rangle$

lemma *log-ln*: $\ln x = \log(\exp 1) x$

$\langle \text{proof} \rangle$

lemma *DERIV-log*:

assumes $x > 0$
shows $DERIV (\lambda y. \log b y) x :> 1 / (\ln b * x)$
 $\langle proof \rangle$

lemmas $DERIV\text{-}\log[THEN\ DERIV\text{-}chain2, derivative\text{-}intros]$
and $DERIV\text{-}\log[THEN\ DERIV\text{-}chain2, unfolded\ has\text{-}field\text{-}derivative\text{-}def, derivative\text{-}intros]$

lemma $powr\text{-}\log\text{-}cancel [simp]: 0 < a \implies a \neq 1 \implies 0 < x \implies a\ powr (\log a x) = x$
 $\langle proof \rangle$

lemma $\log\text{-}powr\text{-}cancel [simp]: 0 < a \implies a \neq 1 \implies \log a (a\ powr x) = x$
 $\langle proof \rangle$

lemma $powr\text{-}eq\text{-}iff: \llbracket y > 0; a > 1 \rrbracket \implies a\ powr x = y \longleftrightarrow \log a y = x$
 $\langle proof \rangle$

lemma $\log\text{-}mult:$
 $\log a (x * y) = (\text{if } x \neq 0 \wedge y \neq 0 \text{ then } \log a x + \log a y \text{ else } 0)$
 $\langle proof \rangle$

lemma $\log\text{-}mult\text{-}pos:$
 $x > 0 \implies y > 0 \implies \log a (x * y) = \log a x + \log a y$
 $\langle proof \rangle$

lemma $\log\text{-}eq\text{-}div\text{-}\ln\text{-}mult\text{-}\log:$
 $0 < b \implies b \neq 1 \implies 0 < x \implies \log a x = (\ln b / \ln a) * \log b x$
 $\langle proof \rangle$

Base 10 logarithms

lemma $\log\text{-}base\text{-}10\text{-}eq1: 0 < x \implies \log 10 x = (\ln (\exp 1) / \ln 10) * \ln x$
 $\langle proof \rangle$

lemma $\log\text{-}base\text{-}10\text{-}eq2: 0 < x \implies \log 10 x = (\log 10 (\exp 1)) * \ln x$
 $\langle proof \rangle$

lemma $\log\text{-}one [simp]: \log a 1 = 0$
 $\langle proof \rangle$

lemma $\log\text{-}eq\text{-}one [simp]: 0 < a \implies a \neq 1 \implies \log a a = 1$
 $\langle proof \rangle$

lemma $\log\text{-}inverse: \log a (\text{inverse } x) = - \log a x$
 $\langle proof \rangle$

lemma $\log\text{-}recip: \log a (1/x) = - \log a x$
 $\langle proof \rangle$

lemma *log-divide*:

$\log a (x / y) = (\text{if } x \neq 0 \wedge y \neq 0 \text{ then } \log a x - \log a y \text{ else } 0)$

$\langle \text{proof} \rangle$

lemma *log-divide-pos*:

$x > 0 \implies y > 0 \implies \log a (x / y) = \log a x - \log a y$

$\langle \text{proof} \rangle$

lemma *powr-gt-zero* [simp]: $0 < x \text{ powr } a \iff x \neq 0$

for $a x :: \text{real}$

$\langle \text{proof} \rangle$

lemma *powr-nonneg-iff* [simp]: $a \text{ powr } x \leq 0 \iff a = 0$

for $a x :: \text{real}$

$\langle \text{proof} \rangle$

lemma *log-add-eq-powr*: $0 < b \implies b \neq 1 \implies x \neq 0 \implies \log b x + y = \log b (x * b \text{ powr } y)$

and *add-log-eq-powr*: $0 < b \implies b \neq 1 \implies x \neq 0 \implies y + \log b x = \log b (b \text{ powr } y * x)$

and *log-minus-eq-powr*: $0 < b \implies b \neq 1 \implies x \neq 0 \implies \log b x - y = \log b (x * b \text{ powr } -y)$

$\langle \text{proof} \rangle$

lemma *minus-log-eq-powr*: $0 < b \implies b \neq 1 \implies x \neq 0 \implies y - \log b x = \log b (b \text{ powr } y / x)$

$\langle \text{proof} \rangle$

lemma *log-less-cancel-iff* [simp]: $1 < a \implies 0 < x \implies 0 < y \implies \log a x < \log a y \iff x < y$

$\langle \text{proof} \rangle$

lemma *log-inj*:

assumes $1 < b$

shows *inj-on* $(\log b) \{0 < ..\}$

$\langle \text{proof} \rangle$

lemma *log-le-cancel-iff* [simp]: $1 < a \implies 0 < x \implies 0 < y \implies \log a x \leq \log a y \iff x \leq y$

$\langle \text{proof} \rangle$

lemma *log-mono*: $1 < a \implies 0 < x \implies x \leq y \implies \log a x \leq \log a y$

$\langle \text{proof} \rangle$

lemma *log-less*: $1 < a \implies 0 < x \implies x < y \implies \log a x < \log a y$

$\langle \text{proof} \rangle$

lemma *zero-less-log-cancel-iff* [simp]: $1 < a \implies 0 < x \implies 0 < \log a x \iff 1 < x$

$\langle \text{proof} \rangle$

lemma *zero-le-log-cancel-iff*[simp]: $1 < a \implies 0 < x \implies 0 \leq \log a\ x \longleftrightarrow 1 \leq x$
 $\langle \text{proof} \rangle$

lemma *log-less-zero-cancel-iff*[simp]: $1 < a \implies 0 < x \implies \log a\ x < 0 \longleftrightarrow x < 1$
 $\langle \text{proof} \rangle$

lemma *log-le-zero-cancel-iff*[simp]: $1 < a \implies 0 < x \implies \log a\ x \leq 0 \longleftrightarrow x \leq 1$
 $\langle \text{proof} \rangle$

lemma *one-less-log-cancel-iff*[simp]: $1 < a \implies 0 < x \implies 1 < \log a\ x \longleftrightarrow a < x$
 $\langle \text{proof} \rangle$

lemma *one-le-log-cancel-iff*[simp]: $1 < a \implies 0 < x \implies 1 \leq \log a\ x \longleftrightarrow a \leq x$
 $\langle \text{proof} \rangle$

lemma *log-less-one-cancel-iff*[simp]: $1 < a \implies 0 < x \implies \log a\ x < 1 \longleftrightarrow x < a$
 $\langle \text{proof} \rangle$

lemma *log-le-one-cancel-iff*[simp]: $1 < a \implies 0 < x \implies \log a\ x \leq 1 \longleftrightarrow x \leq a$
 $\langle \text{proof} \rangle$

lemma *le-log-iff*:
fixes $b\ x\ y :: \text{real}$
assumes $1 < b\ x > 0$
shows $y \leq \log b\ x \longleftrightarrow b\ \text{powr}\ y \leq x$
 $\langle \text{proof} \rangle$

lemma *less-log-iff*:
assumes $1 < b\ x > 0$
shows $y < \log b\ x \longleftrightarrow b\ \text{powr}\ y < x$
 $\langle \text{proof} \rangle$

lemma
assumes $1 < b\ x > 0$
shows *log-less-iff*: $\log b\ x < y \longleftrightarrow x < b\ \text{powr}\ y$
and *log-le-iff*: $\log b\ x \leq y \longleftrightarrow x \leq b\ \text{powr}\ y$
 $\langle \text{proof} \rangle$

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 $b^n \leq m\ 1 < b$
shows $n \leq \log b\ m$

<proof>

lemma *le-log2-of-power*: $2^n \leq m \implies n \leq \log 2 m$ **for** $m n :: \text{nat}$
<proof>

lemma *log-of-power-le*: $\llbracket m \leq b^n; b > 1; m > 0 \rrbracket \implies \log b (\text{real } m) \leq n$
<proof>

lemma *log2-of-power-le*: $\llbracket m \leq 2^n; m > 0 \rrbracket \implies \log 2 m \leq n$ **for** $m n :: \text{nat}$
<proof>

lemma *log-of-power-less*: $\llbracket m < b^n; b > 1; m > 0 \rrbracket \implies \log b (\text{real } m) < n$
<proof>

lemma *log2-of-power-less*: $\llbracket m < 2^n; m > 0 \rrbracket \implies \log 2 m < n$ **for** $m n :: \text{nat}$
<proof>

lemma *less-log-of-power*:
assumes $b^n < m$ $1 < b$
shows $n < \log b m$
<proof>

lemma *less-log2-of-power*: $2^n < m \implies n < \log 2 m$ **for** $m n :: \text{nat}$
<proof>

lemma *gr-one-powr*[*simp*]:
fixes $x y :: \text{real}$ **shows** $\llbracket x > 1; y > 0 \rrbracket \implies 1 < x^{\text{powr } y}$
<proof>

lemma *log-pow-cancel* [*simp*]:
 $a > 0 \implies a \neq 1 \implies \log a (a^b) = b$
<proof>

lemma *floor-log-eq-powr-iff*: $x > 0 \implies b > 1 \implies \lfloor \log b x \rfloor = k \iff b^{\text{powr } k} \leq x \wedge x < b^{\text{powr } (k+1)}$
<proof>

lemma *floor-log-nat-eq-powr-iff*:
fixes $b n k :: \text{nat}$
shows $\llbracket b \geq 2; k > 0 \rrbracket \implies \text{floor } (\log b (\text{real } k)) = n \iff b^n \leq k \wedge k < b^{n+1}$
<proof>

lemma *floor-log-nat-eq-if*:
fixes $b n k :: \text{nat}$
assumes $b^n \leq k$ $k < b^{n+1}$ $b \geq 2$
shows $\text{floor } (\log b (\text{real } k)) = n$
<proof>

lemma *ceiling-log-eq-powr-iff*:

$\llbracket x > 0; b > 1 \rrbracket \implies \lceil \log b \ x \rceil = \text{int } k + 1 \iff b \text{ powr } k < x \wedge x \leq b \text{ powr } (k + 1)$
 $\langle \text{proof} \rangle$

lemma *ceiling-log-nat-eq-powr-iff*:

fixes $b \ n \ k :: \text{nat}$
shows $\llbracket b \geq 2; k > 0 \rrbracket \implies \lceil \log b \ (\text{real } k) \rceil = \text{int } n + 1 \iff (b^{\wedge n} < k \wedge k \leq b^{\wedge (n+1)})$
 $\langle \text{proof} \rangle$

lemma *ceiling-log-nat-eq-if*:

fixes $b \ n \ k :: \text{nat}$
assumes $b^{\wedge n} < k \wedge k \leq b^{\wedge (n+1)} \wedge b \geq 2$
shows $\lceil \log (\text{real } b) (\text{real } k) \rceil = \text{int } n + 1$
 $\langle \text{proof} \rangle$

lemma *floor-log2-div2*:

fixes $n :: \text{nat}$
assumes $n \geq 2$
shows $\lfloor \log 2 \ (\text{real } n) \rfloor = \lfloor \log 2 \ (n \text{ div } 2) \rfloor + 1$
 $\langle \text{proof} \rangle$

lemma *ceiling-log2-div2*:

assumes $n \geq 2$
shows $\lceil \log 2 \ (\text{real } n) \rceil = \lceil \log 2 \ ((n-1) \text{ div } 2 + 1) \rceil + 1$
 $\langle \text{proof} \rangle$

lemma *powr-real-of-int*:

$x > 0 \implies x \text{ powr real-of-int } n = (\text{if } n \geq 0 \text{ then } x^{\wedge \text{nat } n} \text{ else inverse } (x^{\wedge \text{nat } (-n)}))$
 $\langle \text{proof} \rangle$

lemma *powr-numeral [simp]*: $0 \leq x \implies x \text{ powr } (\text{numeral } n :: \text{real}) = x^{\wedge (\text{numeral } n)}$

$\langle \text{proof} \rangle$

lemma *powr-int*:

assumes $x > 0$
shows $x \text{ powr } i = (\text{if } i \geq 0 \text{ then } x^{\wedge \text{nat } i} \text{ else } 1/x^{\wedge \text{nat } (-i)})$
 $\langle \text{proof} \rangle$

lemma *power-of-nat-log-ge*: $b > 1 \implies b^{\wedge \text{nat } \lceil \log b \ x \rceil} \geq x$

$\langle \text{proof} \rangle$

lemma *power-of-nat-log-le*:

assumes $b > 1 \wedge x \geq 1$
shows $b^{\wedge \text{nat } \lfloor \log b \ x \rfloor} \leq x$
 $\langle \text{proof} \rangle$

definition *powr-real* :: *real* \Rightarrow *real* \Rightarrow *real*

where [*code-abbrev*, *simp*]: *powr-real* = *Transcendental.powr*

lemma *compute-powr-real* [*code*]:

powr-real *b* *i* =
 (if $b \leq 0$ then *Code.abort* (*STR "powr-real with nonpositive base"*) (λ -. *powr-real* *b* *i*)
 else if $\lfloor i \rfloor = i$ then (if $0 \leq i$ then $b^{\text{nat } \lfloor i \rfloor}$ else $1 / b^{\text{nat } \lfloor -i \rfloor}$)
 else *Code.abort* (*STR "powr-real with non-integer exponent"*) (λ -. *powr-real* *b* *i*))
for *b* *i* :: *real*
<proof>

lemma *powr-one*: $0 \leq x \implies x \text{ powr } 1 = x$

for *x* :: *real*

<proof>

lemma *powr-one'* [*simp*]: $x \text{ powr } 1 = |x|$

for *x* :: *real*

<proof>

lemma *powr-neg-one*: $0 < x \implies x \text{ powr } -1 = 1/x$

for *x* :: *real*

<proof>

lemma *powr-neg-one'* [*simp*]: $x \text{ powr } -1 = 1/|x|$

for *x* :: *real*

<proof>

lemma *powr-neg-numeral*: $0 < x \implies x \text{ powr } - \text{numeral } n = 1/x^{\text{numeral } n}$

for *x* :: *real*

<proof>

lemma *root-powr-inverse*: $0 < n \implies 0 \leq x \implies \text{root } n \ x = x \text{ powr } (1/n)$

<proof>

lemma *powr-inverse-root*: $0 < n \implies x \text{ powr } (1/n) = |\text{root } n \ x|$

<proof>

lemma *ln-powr* [*simp*]: $\ln (x \text{ powr } y) = y * \ln x$

for *x* :: *real*

<proof>

lemma *ln-root*: $n > 0 \implies \ln (\text{root } n \ b) = \ln b / n$

<proof>

lemma *ln-sqrt*: $0 \leq x \implies \ln (\text{sqrt } x) = \ln x / 2$

<proof>

lemma *log-root*: $n > 0 \implies a \geq 0 \implies \log b (\text{root } n \ a) = \log b \ a / n$
 ⟨proof⟩

lemma *log-powr*: $\log b (x \text{ powr } y) = y * \log b \ x$
 ⟨proof⟩

lemma *log-nat-power*: $0 \leq x \implies \log b (x^n) = \text{real } n * \log b \ x$
 ⟨proof⟩

lemma *log-of-power-eq*:
 assumes $m = b^n \ b > 1$
 shows $n = \log b (\text{real } m)$
 ⟨proof⟩

lemma *log2-of-power-eq*: $m = 2^n \implies n = \log 2 \ m$ **for** $m \ n :: \text{nat}$
 ⟨proof⟩

lemma *log-base-change*: $0 < a \implies a \neq 1 \implies \log b \ x = \log a \ x / \log a \ b$
 ⟨proof⟩

lemma *log-base-pow*: $0 < a \implies \log (a^n) \ x = \log a \ x / n$
 ⟨proof⟩

lemma *log-base-powr*: $a \neq 0 \implies \log (a \text{ powr } b) \ x = \log a \ x / b$
 ⟨proof⟩

lemma *log-base-root*: $n > 0 \implies \log (\text{root } n \ b) \ x = n * (\log b \ x)$
 ⟨proof⟩

lemma *ln-bound*: $0 < x \implies \ln x \leq x$ **for** $x :: \text{real}$
 ⟨proof⟩

lemma *powr-less-one*:
 fixes $x :: \text{real}$
 assumes $1 < x \ y < 0$
 shows $x \text{ powr } y < 1$
 ⟨proof⟩

lemma *powr-le-one-le*: $\bigwedge x \ y :: \text{real}. 0 < x \implies x \leq 1 \implies 1 \leq y \implies x \text{ powr } y \leq x$
 ⟨proof⟩

lemma *powr-mono*:
 fixes $x :: \text{real}$
 assumes $a \leq b$ **and** $1 \leq x$ **shows** $x \text{ powr } a \leq x \text{ powr } b$
 ⟨proof⟩

lemma *ge-one-powr-ge-zero*: $1 \leq x \implies 0 \leq a \implies 1 \leq x \text{ powr } a$

for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-less-mono2*: $0 < a \implies 0 \leq x \implies x < y \implies x \text{ powr } a < y \text{ powr } a$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-less-mono2-neg*: $a < 0 \implies 0 < x \implies x < y \implies y \text{ powr } a < x \text{ powr } a$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-mono2*: $x \text{ powr } a \leq y \text{ powr } a$ **if** $0 \leq a$ $0 \leq x$ $x \leq y$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-less-cancel2*: $0 < a \implies 0 < x \implies 0 < y \implies x \text{ powr } a < y \text{ powr } a \implies x < y$
for $a \ x \ y :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr01-less-one*:
fixes $x :: \text{real}$
assumes $0 < x$ $x < 1$
shows $x \text{ powr } a < 1 \longleftrightarrow a > 0$
 $\langle \text{proof} \rangle$

lemma *powr-le1*: $0 \leq a \implies |x| \leq 1 \implies x \text{ powr } a \leq 1$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-mono2'*:
fixes $a \ x \ y :: \text{real}$
assumes $a \leq 0$ $x > 0$ $x \leq y$
shows $x \text{ powr } a \geq y \text{ powr } a$
 $\langle \text{proof} \rangle$

lemma *powr-mono'*: $a \leq (b :: \text{real}) \implies x \geq 0 \implies x \leq 1 \implies x \text{ powr } b \leq x \text{ powr } a$
 $\langle \text{proof} \rangle$

lemma *powr-mono-both*:
fixes $x :: \text{real}$
assumes $0 \leq a$ $a \leq b$ $1 \leq x$ $x \leq y$
shows $x \text{ powr } a \leq y \text{ powr } b$
 $\langle \text{proof} \rangle$

lemma *powr-mono-both'*:
fixes $x :: \text{real}$
assumes $a \geq b$ $b \geq 0$ $0 < x$ $x \leq y$ $y \leq 1$

shows $x \text{ powr } a \leq y \text{ powr } b$
 $\langle \text{proof} \rangle$

lemma *powr-less-mono'*:
assumes $(x::\text{real}) > 0 \ x < 1 \ a < b$
shows $x \text{ powr } b < x \text{ powr } a$
 $\langle \text{proof} \rangle$

lemma *powr-inj*: $0 < a \implies a \neq 1 \implies a \text{ powr } x = a \text{ powr } y \longleftrightarrow x = y$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *powr-half-sqrt*: $0 \leq x \implies x \text{ powr } (1/2) = \text{sqrt } x$
 $\langle \text{proof} \rangle$

lemma *powr-half-sqrt-powr*: $0 \leq x \implies x \text{ powr } (a/2) = \text{sqrt}(x \text{ powr } a)$
 $\langle \text{proof} \rangle$

lemma *square-powr-half* [simp]:
fixes $x::\text{real}$ **shows** $x^2 \text{ powr } (1/2) = |x|$
 $\langle \text{proof} \rangle$

lemma *ln-powr-bound*: $1 \leq x \implies 0 < a \implies \ln x \leq (x \text{ powr } a) / a$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *ln-powr-bound2*:
fixes $x :: \text{real}$
assumes $1 < x$ **and** $0 < a$
shows $(\ln x) \text{ powr } a \leq (a \text{ powr } a) * x$
 $\langle \text{proof} \rangle$

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 $((\lambda x. f \text{ powr } g \ x) \longrightarrow a \text{ powr } b) \ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-powr'*[*tendsto-intros*]:
fixes $a :: \text{real}$
assumes $f: (f \longrightarrow a) \ F$
and $g: (g \longrightarrow b) \ F$
and $a: a \neq 0 \vee (b > 0 \wedge \text{eventually } (\lambda x. f \ x \geq 0) \ F)$
shows $((\lambda x. f \text{ powr } g \ x) \longrightarrow a \text{ powr } b) \ F$
 $\langle \text{proof} \rangle$

lemma *continuous-powr*:

assumes *continuous* F f
and *continuous* F g
and f ($\text{Lim } F$ ($\lambda x. x$)) $\neq 0$
shows *continuous* F ($\lambda x. (f\ x) \text{ powr } (g\ x :: \text{real})$)
 $\langle \text{proof} \rangle$

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) ($\lambda x. (f\ x) \text{ powr } (g\ x)$)
 $\langle \text{proof} \rangle$

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 ($\lambda x. (f\ x) \text{ powr } (g\ x)$)
 $\langle \text{proof} \rangle$

lemma *tendsto-powr2*:
fixes $a :: \text{real}$
assumes $f: (f \longrightarrow a)\ F$
and $g: (g \longrightarrow b)\ F$
and $\forall_F x \text{ in } F. 0 \leq f\ x$
and $b: 0 < b$
shows $((\lambda x. f\ x \text{ powr } g\ x) \longrightarrow a \text{ powr } b)\ F$
 $\langle \text{proof} \rangle$

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 $\text{pos}: 0 < g\ x$ **and** $x \in X$
shows $((\lambda x. g\ x \text{ powr } f\ x :: \text{real}) \text{ has-derivative } (\lambda h. (g\ x \text{ powr } f\ x) * (f'\ h * \ln (g\ x) + g'\ h * f\ x / g\ x)))$ (*at* x *within* X)
 $\langle \text{proof} \rangle$

lemma *has-derivative-const-powr* [*derivative-intros*]:
fixes $a :: \text{real}$
assumes $\bigwedge x. (f \text{ has-derivative } f')\ (at\ x)$
shows $((\lambda x. a \text{ powr } (f\ x)) \text{ has-derivative } (\lambda y. f'\ y * \ln a * a \text{ powr } (f\ x)))$ (*at* x)
 $\langle \text{proof} \rangle$

lemma *has-real-derivative-const-powr* [*derivative-intros*]:
fixes $a :: \text{real}$
assumes $\bigwedge x. (f \text{ has-real-derivative } f'\ x)\ (at\ x)$
shows $((\lambda x. a \text{ powr } (f\ x)) \text{ has-real-derivative } (f'\ x * \ln a * a \text{ powr } (f\ x)))$ (*at* x)
 $\langle \text{proof} \rangle$

lemma *DERIV-powr*:

fixes $r :: \text{real}$
assumes $g: \text{DERIV } g \ x :> m$
and $\text{pos}: g \ x > 0$
and $f: \text{DERIV } f \ x :> r$
shows $\text{DERIV } (\lambda x. g \ x \text{ powr } f \ x) \ x :> (g \ x \text{ powr } f \ x) * (r * \ln (g \ x) + m * f \ x / g \ x)$
 $\langle \text{proof} \rangle$

lemma *DERIV-fun-powr*:

fixes $r :: \text{real}$
assumes $g: \text{DERIV } g \ x :> m$
and $\text{pos}: g \ x > 0$
shows $\text{DERIV } (\lambda x. (g \ x) \text{ powr } r) \ x :> r * (g \ x) \text{ powr } (r - \text{of-nat } 1) * m$
 $\langle \text{proof} \rangle$

lemma *has-real-derivative-powr*:

assumes $z > 0$
shows $((\lambda z. z \text{ powr } r) \text{ has-real-derivative } r * z \text{ powr } (r - 1)) \text{ (at } z)$
 $\langle \text{proof} \rangle$

declare *has-real-derivative-powr*[*THEN DERIV-chain2, derivative-intros*]

A more general version, by Johannes Hölzl

lemma *has-real-derivative-powr'*:

fixes $f \ g :: \text{real} \Rightarrow \text{real}$
assumes $(f \text{ has-real-derivative } f') \text{ (at } x)$
assumes $(g \text{ has-real-derivative } g') \text{ (at } x)$
assumes $f \ x > 0$
defines $h \equiv \lambda x. f \ x \text{ powr } g \ x * (g' * \ln (f \ x) + f' * g \ x / f \ x)$
shows $((\lambda x. f \ x \text{ powr } g \ x) \text{ has-real-derivative } h \ x) \text{ (at } x)$
 $\langle \text{proof} \rangle$

lemma *tendsto-zero-powrI*:

assumes $(f \longrightarrow (0::\text{real})) \ F \ (g \longrightarrow b) \ F \ \forall_F \ x \text{ in } F. 0 \leq f \ x < b$
shows $((\lambda x. f \ x \text{ powr } g \ x) \longrightarrow 0) \ F$
 $\langle \text{proof} \rangle$

lemma *continuous-on-powr'*:

fixes $f \ g :: - \Rightarrow \text{real}$
assumes *continuous-on* $s \ f$ *continuous-on* $s \ g$
and $\forall x \in s. f \ x \geq 0 \wedge (f \ x = 0 \longrightarrow g \ x > 0)$
shows *continuous-on* $s \ (\lambda x. (f \ x) \text{ powr } (g \ x))$
 $\langle \text{proof} \rangle$

lemma *tendsto-neg-powr*:

assumes $s < 0$
and $f: \text{LIM } x \ F. f \ x :> \text{at-top}$
shows $((\lambda x. f \ x \text{ powr } s) \longrightarrow (0::\text{real})) \ F$

$\langle \text{proof} \rangle$

lemma *tendsto-exp-limit-at-right*: $((\lambda y. (1 + x * y) \text{ powr } (1 / y)) \longrightarrow \exp x)$
(at-right 0)
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *tendsto-exp-limit-at-top*: $((\lambda y. (1 + x / y) \text{ powr } y) \longrightarrow \exp x)$ *at-top*
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *tendsto-exp-limit-sequentially*: $(\lambda n. (1 + x / n) ^ n) \longrightarrow \exp x$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

112.9 Sine and Cosine

definition *sin-coeff* $:: \text{nat} \Rightarrow \text{real}$
where $\text{sin-coeff} = (\lambda n. \text{if even } n \text{ then } 0 \text{ else } (-1) ^ (n - \text{Suc } 0) \text{ div } 2) / (\text{fact } n))$

definition *cos-coeff* $:: \text{nat} \Rightarrow \text{real}$
where $\text{cos-coeff} = (\lambda n. \text{if even } n \text{ then } ((-1) ^ (n \text{ div } 2)) / (\text{fact } n) \text{ else } 0)$

definition *sin* $:: 'a \Rightarrow 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
where $\text{sin} = (\lambda x. \sum n. \text{sin-coeff } n *_R x ^ n)$

definition *cos* $:: 'a \Rightarrow 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
where $\text{cos} = (\lambda x. \sum n. \text{cos-coeff } n *_R x ^ n)$

lemma *sin-coeff-0* [*simp*]: $\text{sin-coeff } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *cos-coeff-0* [*simp*]: $\text{cos-coeff } 0 = 1$
 $\langle \text{proof} \rangle$

lemma *sin-coeff-Suc*: $\text{sin-coeff } (\text{Suc } n) = \text{cos-coeff } n / \text{real } (\text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *cos-coeff-Suc*: $\text{cos-coeff } (\text{Suc } n) = - \text{sin-coeff } n / \text{real } (\text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *summable-norm-sin*: $\text{summable } (\lambda n. \text{norm } (\text{sin-coeff } n *_R x ^ n))$
for $x :: 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *summable-norm-cos*: $\text{summable } (\lambda n. \text{norm } (\text{cos-coeff } n *_R x ^ n))$
for $x :: 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *sin-converges*: $(\lambda n. \text{sin-coeff } n *_{\mathbb{R}} x^{\wedge} n) \text{ sums } \sin x$
 $\langle \text{proof} \rangle$

lemma *cos-converges*: $(\lambda n. \text{cos-coeff } n *_{\mathbb{R}} x^{\wedge} n) \text{ sums } \cos x$
 $\langle \text{proof} \rangle$

lemma *sin-of-real*: $\sin (\text{of-real } x) = \text{of-real } (\sin x)$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

corollary *sin-in-Reals* [simp]: $z \in \mathbb{R} \implies \sin z \in \mathbb{R}$
 $\langle \text{proof} \rangle$

lemma *cos-of-real*: $\cos (\text{of-real } x) = \text{of-real } (\cos x)$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

corollary *cos-in-Reals* [simp]: $z \in \mathbb{R} \implies \cos z \in \mathbb{R}$
 $\langle \text{proof} \rangle$

lemma *diffs-sin-coeff*: $\text{diffs sin-coeff} = \text{cos-coeff}$
 $\langle \text{proof} \rangle$

lemma *diffs-cos-coeff*: $\text{diffs cos-coeff} = (\lambda n. - \text{sin-coeff } n)$
 $\langle \text{proof} \rangle$

lemma *sin-int-times-real*: $\sin (\text{of-int } m * \text{of-real } x) = \text{of-real } (\sin (\text{of-int } m * x))$
 $\langle \text{proof} \rangle$

lemma *cos-int-times-real*: $\cos (\text{of-int } m * \text{of-real } x) = \text{of-real } (\cos (\text{of-int } m * x))$
 $\langle \text{proof} \rangle$

Now at last we can get the derivatives of exp, sin and cos.

lemma *DERIV-sin* [simp]: $\text{DERIV } \sin x :> \cos x$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

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]: $\text{DERIV } \cos x :> - \sin x$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

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 $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-sin-real*: *continuous-on* {*a..b*} *sin* **for** $a :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *isCont-cos*: *isCont cos x*
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-cos-real*: *continuous-on* {*a..b*} *cos* **for** $a :: \text{real}$
 $\langle \text{proof} \rangle$

context
fixes $f :: 'a :: t2\text{-space} \Rightarrow 'b :: \{\text{real-normed-field}, \text{banach}\}$
begin

lemma *isCont-sin'* [*simp*]: *isCont f a* \Longrightarrow *isCont* ($\lambda x. \sin (f x)$) *a*
 $\langle \text{proof} \rangle$

lemma *isCont-cos'* [*simp*]: *isCont f a* \Longrightarrow *isCont* ($\lambda x. \cos (f x)$) *a*
 $\langle \text{proof} \rangle$

lemma *tendsto-sin* [*tendsto-intros*]: ($f \longrightarrow a$) *F* \Longrightarrow ($(\lambda x. \sin (f x)) \longrightarrow \sin a$) *F*
 $\langle \text{proof} \rangle$

lemma *tendsto-cos* [*tendsto-intros*]: ($f \longrightarrow a$) *F* \Longrightarrow ($(\lambda x. \cos (f x)) \longrightarrow \cos a$) *F*
 $\langle \text{proof} \rangle$

lemma *continuous-sin* [*continuous-intros*]: *continuous F f* \Longrightarrow *continuous F* ($\lambda x. \sin (f x)$)
 $\langle \text{proof} \rangle$

lemma *continuous-on-sin* [*continuous-intros*]: *continuous-on s f* \Longrightarrow *continuous-on* $s (\lambda x. \sin (f x))$
 $\langle \text{proof} \rangle$

lemma *continuous-cos* [*continuous-intros*]: *continuous F f* \Longrightarrow *continuous F* ($\lambda x. \cos (f x)$)

$\langle \text{proof} \rangle$

lemma *continuous-on-cos* [*continuous-intros*]: *continuous-on* $s\ f \implies \text{continuous-on}$
 $s\ (\lambda x. \cos (f\ x))$
 $\langle \text{proof} \rangle$

end

lemma *continuous-within-sin*: *continuous* (at z within s) *sin*
for $z :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *continuous-within-cos*: *continuous* (at z within s) *cos*
for $z :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

112.10 Properties of Sine and Cosine

lemma *sin-zero* [*simp*]: *sin* $0 = 0$
 $\langle \text{proof} \rangle$

lemma *cos-zero* [*simp*]: *cos* $0 = 1$
 $\langle \text{proof} \rangle$

lemma *DERIV-fun-sin*: *DERIV* $g\ x :> m \implies \text{DERIV } (\lambda x. \sin (g\ x))\ x :> \cos (g\ x) * m$
 $\langle \text{proof} \rangle$

lemma *DERIV-fun-cos*: *DERIV* $g\ x :> m \implies \text{DERIV } (\lambda x. \cos (g\ x))\ x :> - \sin (g\ x) * m$
 $\langle \text{proof} \rangle$

112.11 Deriving the Addition Formulas

The product of two cosine series.

lemma *cos-x-cos-y*:
fixes $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
shows
 $(\lambda p. \sum_{n \leq p. \text{if even } p \wedge \text{even } n} \text{then } ((-1)^\wedge (p \text{ div } 2) * (p \text{ choose } n) / (\text{fact } p)) *_R (\widehat{x}^n) * \widehat{y}^{(p-n)} \text{ else } 0)$
 $\text{sums } (\cos x * \cos y)$
 $\langle \text{proof} \rangle$

The product of two sine series.

lemma *sin-x-sin-y*:
fixes $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
shows

$(\lambda p. \sum n \leq p.$
 $\quad \text{if even } p \wedge \text{odd } n$
 $\quad \text{then } -((-1)^{\wedge(p \text{ div } 2)} * (p \text{ choose } n) / (\text{fact } p)) *_R (x^{\wedge n}) * y^{\wedge(p-n)}$
 $\quad \text{else } 0)$
 $\quad \text{sums } (\sin x * \sin y)$
 $\langle \text{proof} \rangle$

lemma *sums-cos-x-plus-y*:

fixes $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
shows
 $(\lambda p. \sum n \leq p.$
 $\quad \text{if even } p$
 $\quad \text{then } ((-1)^{\wedge(p \text{ div } 2)} * (p \text{ choose } n) / (\text{fact } p)) *_R (x^{\wedge n}) * y^{\wedge(p-n)}$
 $\quad \text{else } 0)$
 $\quad \text{sums } \cos (x + y)$
 $\langle \text{proof} \rangle$

theorem *cos-add*:

fixes $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
shows $\cos (x + y) = \cos x * \cos y - \sin x * \sin y$
 $\langle \text{proof} \rangle$

lemma *sin-minus-converges*: $(\lambda n. -(\sin\text{-coeff } n *_R (-x)^{\wedge n})) \text{ sums } \sin x$
 $\langle \text{proof} \rangle$

lemma *sin-minus [simp]*: $\sin (-x) = -\sin x$
for $x :: 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *cos-minus-converges*: $(\lambda n. (\cos\text{-coeff } n *_R (-x)^{\wedge n})) \text{ sums } \cos x$
 $\langle \text{proof} \rangle$

lemma *cos-minus [simp]*: $\cos (-x) = \cos x$
for $x :: 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *cos-abs-real [simp]*: $\cos |x :: \text{real}| = \cos x$
 $\langle \text{proof} \rangle$

lemma *sin-cos-squared-add [simp]*: $(\sin x)^2 + (\cos x)^2 = 1$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *sin-cos-squared-add2 [simp]*: $(\cos x)^2 + (\sin x)^2 = 1$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *sin-cos-squared-add3 [simp]*: $\cos x * \cos x + \sin x * \sin x = 1$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$

$\langle proof \rangle$

lemma *sin-squared-eq*: $(\sin x)^2 = 1 - (\cos x)^2$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle proof \rangle$

lemma *cos-squared-eq*: $(\cos x)^2 = 1 - (\sin x)^2$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle proof \rangle$

lemma *abs-sin-le-one* [simp]: $|\sin x| \leq 1$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *sin-ge-minus-one* [simp]: $-1 \leq \sin x$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *sin-le-one* [simp]: $\sin x \leq 1$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *abs-cos-le-one* [simp]: $|\cos x| \leq 1$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *cos-ge-minus-one* [simp]: $-1 \leq \cos x$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *cos-le-one* [simp]: $\cos x \leq 1$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *cos-diff*: $\cos (x - y) = \cos x * \cos y + \sin x * \sin y$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle proof \rangle$

lemma *cos-double*: $\cos(2*x) = (\cos x)^2 - (\sin x)^2$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle proof \rangle$

lemma *sin-cos-le1*: $|\sin x * \sin y + \cos x * \cos y| \leq 1$
for $x :: \text{real}$
 $\langle proof \rangle$

lemma *DERIV-fun-pow*: $DERIV\ g\ x :> m \implies DERIV\ (\lambda x. (g\ x) \wedge^n) x :> \text{real}$
 $n * (g\ x) \wedge (n - 1) * m$
 $\langle proof \rangle$

lemma *DERIV-fun-exp*: $DERIV\ g\ x :> m \implies DERIV\ (\lambda x. exp\ (g\ x))\ x :> exp\ (g\ x) * m$
 ⟨proof⟩

112.12 The Constant Pi

definition *pi* :: *real*
 where $pi = 2 * (THE\ x. 0 \leq x \wedge x \leq 2 \wedge cos\ x = 0)$

Show that there’s a least positive x with $cos\ x = 0$; hence define *pi*.

lemma *sin-paired*: $(\lambda n. (-1)^n / (fact\ (2 * n + 1)) * x^{(2 * n + 1)})\ sums\ sin\ x$
 for $x :: real$
 ⟨proof⟩

lemma *sin-gt-zero-02*:
 fixes $x :: real$
 assumes $0 < x$ and $x < 2$
 shows $0 < sin\ x$
 ⟨proof⟩

lemma *cos-double-less-one*: $0 < x \implies x < 2 \implies cos\ (2 * x) < 1$
 for $x :: real$
 ⟨proof⟩

lemma *cos-paired*: $(\lambda n. (-1)^n / (fact\ (2 * n)) * x^{(2 * n)})\ sums\ cos\ x$
 for $x :: real$
 ⟨proof⟩

lemma *sum-pos-lt-pair*:
 fixes $f :: nat \Rightarrow real$
 assumes f : *summable* f and $fplus$: $\bigwedge d. 0 < f\ (k + (Suc(Suc\ 0) * d)) + f\ (k + ((Suc\ (Suc\ 0) * d) + 1))$
 shows $sum\ f\ \{..<k\} < suminf\ f$
 ⟨proof⟩

lemma *cos-two-less-zero* [*simp*]: $cos\ 2 < (0 :: real)$
 ⟨proof⟩

lemmas *cos-two-neq-zero* [*simp*] = *cos-two-less-zero* [*THEN less-imp-neq*]
lemmas *cos-two-le-zero* [*simp*] = *cos-two-less-zero* [*THEN order-less-imp-le*]

lemma *cos-is-zero*: $\exists! x :: real. 0 \leq x \wedge x \leq 2 \wedge cos\ x = 0$
 ⟨proof⟩

lemma *pi-half*: $pi/2 = (THE\ x. 0 \leq x \wedge x \leq 2 \wedge cos\ x = 0)$
 ⟨proof⟩

lemma *cos-pi-half* [simp]: $\cos (\pi/2) = 0$
 ⟨proof⟩

lemma *cos-of-real-pi-half* [simp]: $\cos ((\text{of-real } \pi/2) :: 'a) = 0$
 if *SORT-CONSTRAINT*('a::{real-field,banach,real-normed-algebra-1})
 ⟨proof⟩

lemma *pi-half-gt-zero* [simp]: $0 < \pi/2$
 ⟨proof⟩

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⟩

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]: $0 < \pi$
 ⟨proof⟩

lemma *pi-ge-zero* [simp]: $0 \leq \pi$
 ⟨proof⟩

lemma *pi-neq-zero* [simp]: $\pi \neq 0$
 ⟨proof⟩

lemma *pi-not-less-zero* [simp]: $\neg \pi < 0$
 ⟨proof⟩

lemma *minus-pi-half-less-zero*: $-(\pi/2) < 0$
 ⟨proof⟩

lemma *m2pi-less-pi*: $-(2*\pi) < \pi$
 ⟨proof⟩

lemma *sin-pi-half* [simp]: $\sin(\pi/2) = 1$
 ⟨proof⟩

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})
 ⟨proof⟩

lemma *sin-cos-eq*: $\sin x = \cos (\text{of-real } \pi/2 - x)$
 for $x :: 'a :: \{\text{real-normed-field,banach}\}$
 ⟨proof⟩

lemma *minus-sin-cos-eq*: $-\sin x = \cos (x + \text{of-real } \pi/2)$

for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *cos-sin-eq*: $\cos x = \sin (of\text{-real } \pi/2 - x)$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *sin-add*: $\sin (x + y) = \sin x * \cos y + \cos x * \sin y$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *sin-diff*: $\sin (x - y) = \sin x * \cos y - \cos x * \sin y$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *sin-double*: $\sin(2 * x) = 2 * \sin x * \cos x$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *cos-of-real-pi* [simp]: $\cos (of\text{-real } \pi) = -1$
 $\langle \text{proof} \rangle$

lemma *sin-of-real-pi* [simp]: $\sin (of\text{-real } \pi) = 0$
 $\langle \text{proof} \rangle$

lemma *cos-pi* [simp]: $\cos \pi = -1$
 $\langle \text{proof} \rangle$

lemma *sin-pi* [simp]: $\sin \pi = 0$
 $\langle \text{proof} \rangle$

lemma *sin-periodic-pi* [simp]: $\sin (x + \pi) = - \sin x$
 $\langle \text{proof} \rangle$

lemma *sin-periodic-pi2* [simp]: $\sin (\pi + x) = - \sin x$
 $\langle \text{proof} \rangle$

lemma *cos-periodic-pi* [simp]: $\cos (x + \pi) = - \cos x$
 $\langle \text{proof} \rangle$

lemma *cos-periodic-pi2* [simp]: $\cos (\pi + x) = - \cos x$
 $\langle \text{proof} \rangle$

lemma *sin-periodic* [simp]: $\sin (x + 2 * \pi) = \sin x$
 $\langle \text{proof} \rangle$

lemma *cos-periodic* [simp]: $\cos (x + 2 * \pi) = \cos x$
 $\langle \text{proof} \rangle$

lemma *cos-npi* [simp]: $\cos (\text{real } n * \pi) = (-1) ^ n$
 ⟨proof⟩

lemma *cos-npi2* [simp]: $\cos (\pi * \text{real } n) = (-1) ^ n$
 ⟨proof⟩

lemma *sin-npi* [simp]: $\sin (\text{real } n * \pi) = 0$
for $n :: \text{nat}$
 ⟨proof⟩

lemma *sin-npi2* [simp]: $\sin (\pi * \text{real } n) = 0$
for $n :: \text{nat}$
 ⟨proof⟩

lemma *sin-npi-numeral* [simp]: $\sin (\text{Num.numeral } n * \pi) = 0$
 ⟨proof⟩

lemma *sin-npi2-numeral* [simp]: $\sin (\pi * \text{Num.numeral } n) = 0$
 ⟨proof⟩

lemma *sin-npi-complex'* [simp]: $\sin (\text{of-nat } n * \text{of-real } \pi) = 0$
 ⟨proof⟩

lemma *cos-npi-numeral* [simp]: $\cos (\text{Num.numeral } n * \pi) = (-1) ^ \text{Num.numeral } n$
 ⟨proof⟩

lemma *cos-npi2-numeral* [simp]: $\cos (\pi * \text{Num.numeral } n) = (-1) ^ \text{Num.numeral } n$
 ⟨proof⟩

lemma *cos-npi-complex'* [simp]: $\cos (\text{of-nat } n * \text{of-real } \pi) = (-1) ^ n$ **for** n
 ⟨proof⟩

lemma *cos-two-pi* [simp]: $\cos (2 * \pi) = 1$
 ⟨proof⟩

lemma *sin-two-pi* [simp]: $\sin (2 * \pi) = 0$
 ⟨proof⟩

context
fixes $w :: 'a :: \{\text{real-normed-field}, \text{banach}\}$

begin

lemma *sin-times-sin*: $\sin w * \sin z = (\cos (w - z) - \cos (w + z)) / 2$
 ⟨proof⟩

lemma *sin-times-cos*: $\sin w * \cos z = (\sin (w + z) + \sin (w - z)) / 2$

$\langle proof \rangle$

lemma *cos-times-sin*: $\cos w * \sin z = (\sin (w + z) - \sin (w - z)) / 2$
 $\langle proof \rangle$

lemma *cos-times-cos*: $\cos w * \cos z = (\cos (w - z) + \cos (w + z)) / 2$
 $\langle proof \rangle$

lemma *cos-double-cos*: $\cos (2 * w) = 2 * \cos w ^ 2 - 1$
 $\langle proof \rangle$

lemma *cos-double-sin*: $\cos (2 * w) = 1 - 2 * \sin w ^ 2$
 $\langle proof \rangle$

end

lemma *sin-plus-sin*: $\sin w + \sin z = 2 * \sin ((w + z) / 2) * \cos ((w - z) / 2)$
for $w :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle proof \rangle$

lemma *sin-diff-sin*: $\sin w - \sin z = 2 * \sin ((w - z) / 2) * \cos ((w + z) / 2)$
for $w :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle proof \rangle$

lemma *cos-plus-cos*: $\cos w + \cos z = 2 * \cos ((w + z) / 2) * \cos ((w - z) / 2)$
for $w :: 'a::\{\text{real-normed-field}, \text{banach}, \text{field}\}$
 $\langle proof \rangle$

lemma *cos-diff-cos*: $\cos w - \cos z = 2 * \sin ((w + z) / 2) * \sin ((z - w) / 2)$
for $w :: 'a::\{\text{real-normed-field}, \text{banach}, \text{field}\}$
 $\langle proof \rangle$

lemma *sin-pi-minus [simp]*: $\sin (pi - x) = \sin x$
 $\langle proof \rangle$

lemma *cos-pi-minus [simp]*: $\cos (pi - x) = - (\cos x)$
 $\langle proof \rangle$

lemma *sin-minus-pi [simp]*: $\sin (x - pi) = - (\sin x)$
 $\langle proof \rangle$

lemma *cos-minus-pi [simp]*: $\cos (x - pi) = - (\cos x)$
 $\langle proof \rangle$

lemma *sin-2pi-minus [simp]*: $\sin (2 * pi - x) = - (\sin x)$
 $\langle proof \rangle$

lemma *cos-2pi-minus [simp]*: $\cos (2 * pi - x) = \cos x$
 $\langle proof \rangle$

lemma *sin-gt-zero2*: $0 < x \implies x < \pi/2 \implies 0 < \sin x$
 ⟨proof⟩

lemma *sin-less-zero*:
 assumes $-\pi/2 < x$ and $x < 0$
 shows $\sin x < 0$
 ⟨proof⟩

lemma *pi-less-4*: $\pi < 4$
 ⟨proof⟩

lemma *cos-gt-zero*: $0 < x \implies x < \pi/2 \implies 0 < \cos x$
 ⟨proof⟩

lemma *cos-gt-zero-pi*: $-(\pi/2) < x \implies x < \pi/2 \implies 0 < \cos x$
 ⟨proof⟩

lemma *cos-ge-zero*: $-(\pi/2) \leq x \implies x \leq \pi/2 \implies 0 \leq \cos x$
 ⟨proof⟩

lemma *sin-gt-zero*: $0 < x \implies x < \pi \implies 0 < \sin x$
 ⟨proof⟩

lemma *sin-lt-zero*: $\pi < x \implies x < 2 * \pi \implies \sin x < 0$
 ⟨proof⟩

lemma *pi-ge-two*: $2 \leq \pi$
 ⟨proof⟩

lemma *sin-ge-zero*: $0 \leq x \implies x \leq \pi \implies 0 \leq \sin x$
 ⟨proof⟩

lemma *sin-le-zero*: $\pi \leq x \implies x < 2 * \pi \implies \sin x \leq 0$
 ⟨proof⟩

lemma *sin-pi-divide-n-ge-0* [simp]:
 assumes $n \neq 0$
 shows $0 \leq \sin (\pi / \text{real } n)$
 ⟨proof⟩

lemma *sin-pi-divide-n-gt-0*:
 assumes $2 \leq n$
 shows $0 < \sin (\pi / \text{real } n)$
 ⟨proof⟩

Proof resembles that of *cos-is-zero* but with π for the upper bound

lemma *cos-total*:
 assumes $y: -1 \leq y \leq 1$

shows $\exists!x. 0 \leq x \wedge x \leq \pi \wedge \cos x = y$
 $\langle \text{proof} \rangle$

lemma *sin-total*:
assumes $y: -1 \leq y \leq 1$
shows $\exists!x. -(\pi/2) \leq x \wedge x \leq \pi/2 \wedge \sin x = y$
 $\langle \text{proof} \rangle$

lemma *cos-zero-lemma*:
assumes $0 \leq x \wedge \cos x = 0$
shows $\exists n. \text{odd } n \wedge x = \text{of-nat } n * (\pi/2)$
 $\langle \text{proof} \rangle$

lemma *sin-zero-lemma*:
assumes $0 \leq x \wedge \sin x = 0$
shows $\exists n::\text{nat}. \text{even } n \wedge x = \text{real } n * (\pi/2)$
 $\langle \text{proof} \rangle$

lemma *cos-zero-iff*:
 $\cos x = 0 \longleftrightarrow ((\exists n. \text{odd } n \wedge x = \text{real } n * (\pi/2)) \vee (\exists n. \text{odd } n \wedge x = -(\text{real } n * (\pi/2))))$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *sin-zero-iff*:
 $\sin x = 0 \longleftrightarrow ((\exists n. \text{even } n \wedge x = \text{real } n * (\pi/2)) \vee (\exists n. \text{even } n \wedge x = -(\text{real } n * (\pi/2))))$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *sin-zero-pi-iff*:
fixes $x::\text{real}$
assumes $|x| < \pi$
shows $\sin x = 0 \longleftrightarrow x = 0$
 $\langle \text{proof} \rangle$

lemma *cos-zero-iff-int*: $\cos x = 0 \longleftrightarrow (\exists i. \text{odd } i \wedge x = \text{of-int } i * (\pi/2))$
 $\langle \text{proof} \rangle$

lemma *sin-zero-iff-int*: $\sin x = 0 \longleftrightarrow (\exists i. \text{even } i \wedge x = \text{of-int } i * (\pi/2))$ **(is ?lhs = ?rhs)**
 $\langle \text{proof} \rangle$

lemma *sin-zero-iff-int2*: $\sin x = 0 \longleftrightarrow (\exists i::\text{int}. x = \text{of-int } i * \pi)$
 $\langle \text{proof} \rangle$

lemma *cos-zero-iff-int2*:
fixes $x::\text{real}$
shows $\cos x = 0 \longleftrightarrow (\exists n::\text{int}. x = n * \pi + \pi/2)$

$\langle \text{proof} \rangle$

lemma *sin-npi-int [simp]:* $\sin (\pi * \text{of-int } n) = 0$
 $\langle \text{proof} \rangle$

lemma *cos-monotone-0-pi:*
assumes $0 \leq y$ **and** $y < x$ **and** $x \leq \pi$
shows $\cos x < \cos y$
 $\langle \text{proof} \rangle$

lemma *cos-monotone-0-pi-le:*
assumes $0 \leq y$ **and** $y \leq x$ **and** $x \leq \pi$
shows $\cos x \leq \cos y$
 $\langle \text{proof} \rangle$

lemma *cos-monotone-minus-pi-0:*
assumes $-\pi \leq y$ **and** $y < x$ **and** $x \leq 0$
shows $\cos y < \cos x$
 $\langle \text{proof} \rangle$

lemma *cos-monotone-minus-pi-0':*
assumes $-\pi \leq y$ **and** $y \leq x$ **and** $x \leq 0$
shows $\cos y \leq \cos x$
 $\langle \text{proof} \rangle$

lemma *sin-monotone-2pi:*
assumes $-\pi/2 \leq y$ **and** $y < x$ **and** $x \leq \pi/2$
shows $\sin y < \sin x$
 $\langle \text{proof} \rangle$

lemma *sin-monotone-2pi-le:*
assumes $-\pi/2 \leq y$ **and** $y \leq x$ **and** $x \leq \pi/2$
shows $\sin y \leq \sin x$
 $\langle \text{proof} \rangle$

lemma *sin-x-le-x:*
fixes $x :: \text{real}$
assumes $x \geq 0$
shows $\sin x \leq x$
 $\langle \text{proof} \rangle$

lemma *sin-x-ge-neg-x:*
fixes $x :: \text{real}$
assumes $x \geq 0$
shows $\sin x \geq -x$
 $\langle \text{proof} \rangle$

lemma *abs-sin-x-le-abs-x:* $|\sin x| \leq |x|$
for $x :: \text{real}$

$\langle \text{proof} \rangle$

112.13 More Corollaries about Sine and Cosine

lemma *sin-cos-npi* [simp]: $\sin (\text{real } (\text{Suc } (2 * n)) * \pi / 2) = (-1) ^ n$
 $\langle \text{proof} \rangle$

lemma *cos-2npi* [simp]: $\cos (2 * \text{real } n * \pi) = 1$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *cos-3over2-pi* [simp]: $\cos (3/2 * \pi) = 0$
 $\langle \text{proof} \rangle$

lemma *sin-2npi* [simp]: $\sin (2 * \text{real } n * \pi) = 0$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *sin-3over2-pi* [simp]: $\sin (3/2 * \pi) = - 1$
 $\langle \text{proof} \rangle$

lemma *cos-pi-eq-zero* [simp]: $\cos (\pi * \text{real } (\text{Suc } (2 * m)) / 2) = 0$
 $\langle \text{proof} \rangle$

lemma *DERIV-cos-add* [simp]: $\text{DERIV } (\lambda x. \cos (x + k)) \text{ } xa :> - \sin (xa + k)$
 $\langle \text{proof} \rangle$

lemma *sin-zero-norm-cos-one*:
fixes $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
assumes $\sin x = 0$
shows $\text{norm } (\cos x) = 1$
 $\langle \text{proof} \rangle$

lemma *sin-zero-abs-cos-one*: $\sin x = 0 \implies |\cos x| = (1 :: \text{real})$
 $\langle \text{proof} \rangle$

lemma *cos-one-sin-zero*:
fixes $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
assumes $\cos x = 1$
shows $\sin x = 0$
 $\langle \text{proof} \rangle$

lemma *sin-times-pi-eq-0*: $\sin (x * \pi) = 0 \iff x \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *cos-one-2pi*: $\cos x = 1 \iff (\exists n :: \text{nat}. x = n * 2 * \pi) \vee (\exists n :: \text{nat}. x = - (n * 2 * \pi))$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *cos-one-2pi-int*: $\cos x = 1 \longleftrightarrow (\exists n::\text{int}. x = n * 2 * \pi)$ (**is** ?lhs = ?rhs)
 ⟨proof⟩

lemma *cos-npi-int* [simp]:
fixes $n::\text{int}$ **shows** $\cos (\pi * \text{of-int } n) = (\text{if even } n \text{ then } 1 \text{ else } -1)$
 ⟨proof⟩

lemma *sin-cos-sqrt*: $0 \leq \sin x \implies \sin x = \text{sqrt } (1 - (\cos(x) ^ 2))$
 ⟨proof⟩

lemma *sin-eq-0-pi*: $-\pi < x \implies x < \pi \implies \sin x = 0 \implies x = 0$
 ⟨proof⟩

lemma *cos-treble-cos*: $\cos (3 * x) = 4 * \cos x ^ 3 - 3 * \cos x$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 ⟨proof⟩

lemma *cos-45*: $\cos (\pi/4) = \text{sqrt } 2 / 2$
 ⟨proof⟩

lemma *cos-30*: $\cos (\pi/6) = \text{sqrt } 3 / 2$
 ⟨proof⟩

lemma *sin-45*: $\sin (\pi/4) = \text{sqrt } 2 / 2$
 ⟨proof⟩

lemma *sin-60*: $\sin (\pi/3) = \text{sqrt } 3 / 2$
 ⟨proof⟩

lemma *cos-60*: $\cos (\pi/3) = 1 / 2$
 ⟨proof⟩

lemma *sin-30*: $\sin (\pi/6) = 1 / 2$
 ⟨proof⟩

lemma *cos-120*: $\cos (2 * \pi/3) = -1 / 2$
and *sin-120*: $\sin (2 * \pi/3) = \text{sqrt } 3 / 2$
 ⟨proof⟩

lemma *cos-120'*: $\cos (\pi * 2 / 3) = -1 / 2$
 ⟨proof⟩

lemma *sin-120'*: $\sin (\pi * 2 / 3) = \text{sqrt } 3 / 2$
 ⟨proof⟩

lemma *cos-integer-2pi*: $n \in \mathbb{Z} \implies \cos(2 * \pi * n) = 1$
 ⟨proof⟩

lemma *sin-integer-2pi*: $n \in \mathbb{Z} \implies \sin(2 * \pi * n) = 0$
 $\langle \text{proof} \rangle$

lemma *cos-int-2pin* [simp]: $\cos((2 * \pi) * \text{of-int } n) = 1$
 $\langle \text{proof} \rangle$

lemma *sin-int-2pin* [simp]: $\sin((2 * \pi) * \text{of-int } n) = 0$
 $\langle \text{proof} \rangle$

lemma *sin-cos-eq-iff*: $\sin y = \sin x \wedge \cos y = \cos x \longleftrightarrow (\exists n::\text{int}. y = x + 2 * \pi * n) \text{ (is } ?L=?R)$
 $\langle \text{proof} \rangle$

lemma *sincos-principal-value*: $\exists y. (-\pi < y \wedge y \leq \pi) \wedge (\sin y = \sin x \wedge \cos y = \cos x)$
 $\langle \text{proof} \rangle$

112.14 Tangent

definition *tan* :: $'a \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
where $\text{tan} = (\lambda x. \sin x / \cos x)$

lemma *tan-of-real*: $\text{of-real}(\text{tan } x) = (\text{tan}(\text{of-real } x)) :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *tan-in-Reals* [simp]: $z \in \mathbb{R} \implies \text{tan } z \in \mathbb{R}$
for $z :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *tan-zero* [simp]: $\text{tan } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *tan-pi* [simp]: $\text{tan } \pi = 0$
 $\langle \text{proof} \rangle$

lemma *tan-npi* [simp]: $\text{tan}(\text{real } n * \pi) = 0$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *tan-pi-half* [simp]: $\text{tan}(\pi / 2) = 0$
 $\langle \text{proof} \rangle$

lemma *tan-minus* [simp]: $\text{tan}(-x) = -\text{tan } x$
 $\langle \text{proof} \rangle$

lemma *tan-periodic* [simp]: $\text{tan}(x + 2 * \pi) = \text{tan } x$
 $\langle \text{proof} \rangle$

lemma *lemma-tan-add1*: $\cos x \neq 0 \implies \cos y \neq 0 \implies 1 - \text{tan } x * \text{tan } y = \cos$

$(x + y) / (\cos x * \cos y)$
 $\langle \text{proof} \rangle$

lemma *add-tan-eq*: $\cos x \neq 0 \implies \cos y \neq 0 \implies \tan x + \tan y = \sin(x + y) / (\cos x * \cos y)$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *tan-eq-0-cos-sin*: $\tan x = 0 \longleftrightarrow \cos x = 0 \vee \sin x = 0$
 $\langle \text{proof} \rangle$

Note: half of these zeros would normally be regarded as undefined cases.

lemma *tan-eq-0-Ex*:
assumes $\tan x = 0$
obtains $k :: \text{int}$ **where** $x = (k/2) * \pi$
 $\langle \text{proof} \rangle$

lemma *tan-add*:
 $\cos x \neq 0 \implies \cos y \neq 0 \implies \cos(x + y) \neq 0 \implies \tan(x + y) = (\tan x + \tan y) / (1 - \tan x * \tan y)$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *tan-double*: $\cos x \neq 0 \implies \cos(2 * x) \neq 0 \implies \tan(2 * x) = (2 * \tan x) / (1 - (\tan x)^2)$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *tan-gt-zero*: $0 < x \implies x < \pi/2 \implies 0 < \tan x$
 $\langle \text{proof} \rangle$

lemma *tan-less-zero*:
assumes $-\pi/2 < x$ **and** $x < 0$
shows $\tan x < 0$
 $\langle \text{proof} \rangle$

lemma *tan-half*: $\tan x = \sin(2 * x) / (\cos(2 * x) + 1)$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}, \text{field}\}$
 $\langle \text{proof} \rangle$

lemma *tan-30*: $\tan(\pi/6) = 1 / \text{sqrt } 3$
 $\langle \text{proof} \rangle$

lemma *tan-45*: $\tan(\pi/4) = 1$
 $\langle \text{proof} \rangle$

lemma *tan-60*: $\tan(\pi/3) = \text{sqrt } 3$
 $\langle \text{proof} \rangle$

lemma *DERIV-tan* [*simp*]: $\cos x \neq 0 \implies \text{DERIV } \tan x :> \text{inverse } ((\cos x)^2)$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

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*: $\cos x \neq 0 \implies \text{isCont } \tan x$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *isCont-tan'* [*simp*, *continuous-intros*]:
fixes $a :: 'a :: \{\text{real-normed-field}, \text{banach}\}$ **and** $f :: 'a \Rightarrow 'a$
shows $\text{isCont } f \ a \implies \cos (f \ a) \neq 0 \implies \text{isCont } (\lambda x. \tan (f \ x)) \ a$
 $\langle \text{proof} \rangle$

lemma *tendsto-tan* [*tendsto-intros*]:
fixes $f :: 'a \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
shows $(f \longrightarrow a) \ F \implies \cos a \neq 0 \implies ((\lambda x. \tan (f \ x)) \longrightarrow \tan a) \ F$
 $\langle \text{proof} \rangle$

lemma *continuous-tan*:
fixes $f :: 'a \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
shows $\text{continuous } F \ f \implies \cos (f \ (\text{Lim } F \ (\lambda x. x))) \neq 0 \implies \text{continuous } F \ (\lambda x. \tan (f \ x))$
 $\langle \text{proof} \rangle$

lemma *continuous-on-tan* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
shows $\text{continuous-on } s \ f \implies (\forall x \in s. \cos (f \ x) \neq 0) \implies \text{continuous-on } s \ (\lambda x. \tan (f \ x))$
 $\langle \text{proof} \rangle$

lemma *continuous-within-tan* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
shows $\text{continuous (at } x \text{ within } s) \ f \implies \cos (f \ x) \neq 0 \implies \text{continuous (at } x \text{ within } s) \ (\lambda x. \tan (f \ x))$
 $\langle \text{proof} \rangle$

lemma *LIM-cos-div-sin*: $(\lambda x. \cos(x)/\sin(x)) \ -\pi/2 \rightarrow 0$
 $\langle \text{proof} \rangle$

lemma *lemma-tan-total*:
assumes $0 < y$ **shows** $\exists x. 0 < x \wedge x < \pi/2 \wedge y < \tan x$
 $\langle \text{proof} \rangle$

lemma *tan-total-pos*:

assumes $0 \leq y$ shows $\exists x. 0 \leq x \wedge x < \pi/2 \wedge \tan x = y$

<proof>

lemma *lemma-tan-total1*: $\exists x. -(\pi/2) < x \wedge x < (\pi/2) \wedge \tan x = y$

<proof>

proposition *tan-total*: $\exists! x. -(\pi/2) < x \wedge x < (\pi/2) \wedge \tan x = y$

<proof>

lemma *tan-monotone*:

assumes $-(\pi/2) < y$ and $y < x$ and $x < \pi/2$

shows $\tan y < \tan x$

<proof>

lemma *tan-monotone'*:

assumes $-(\pi/2) < y$

and $y < \pi/2$

and $-(\pi/2) < x$

and $x < \pi/2$

shows $y < x \longleftrightarrow \tan y < \tan x$

<proof>

lemma *tan-inverse*: $1 / (\tan y) = \tan (\pi/2 - y)$

<proof>

lemma *tan-periodic-pi[simp]*: $\tan (x + \pi) = \tan x$

<proof>

lemma *tan-periodic-nat[simp]*: $\tan (x + \text{real } n * \pi) = \tan x$

<proof>

lemma *tan-periodic-int[simp]*: $\tan (x + \text{of-int } i * \pi) = \tan x$

<proof>

lemma *tan-periodic-n[simp]*: $\tan (x + \text{numeral } n * \pi) = \tan x$

<proof>

lemma *tan-minus-45 [simp]*: $\tan (-(\pi/4)) = -1$

<proof>

lemma *tan-diff*:

$\cos x \neq 0 \implies \cos y \neq 0 \implies \cos (x - y) \neq 0 \implies \tan (x - y) = (\tan x - \tan y) / (1 + \tan x * \tan y)$

for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$

<proof>

lemma *tan-pos-pi2-le*: $0 \leq x \implies x < \pi/2 \implies 0 \leq \tan x$

<proof>

lemma *cos-tan*: $|x| < \pi/2 \implies \cos x = 1 / \sqrt{1 + \tan x^2}$
 ⟨proof⟩

lemma *cos-tan-half*: $\cos x \neq 0 \implies \cos (2*x) = (1 - (\tan x)^2) / (1 + (\tan x)^2)$
 ⟨proof⟩

lemma *sin-tan*: $|x| < \pi/2 \implies \sin x = \tan x / \sqrt{1 + \tan x^2}$
 ⟨proof⟩

lemma *sin-tan-half*: $\sin (2*x) = 2 * \tan x / (1 + (\tan x)^2)$
 ⟨proof⟩

lemma *tan-mono-le*: $-(\pi/2) < x \implies x \leq y \implies y < \pi/2 \implies \tan x \leq \tan y$
 ⟨proof⟩

lemma *tan-mono-lt-eq*:
 $-(\pi/2) < x \implies x < \pi/2 \implies -(\pi/2) < y \implies y < \pi/2 \implies \tan x < \tan y$
 $\longleftrightarrow x < y$
 ⟨proof⟩

lemma *tan-mono-le-eq*:
 $-(\pi/2) < x \implies x < \pi/2 \implies -(\pi/2) < y \implies y < \pi/2 \implies \tan x \leq \tan y$
 $\longleftrightarrow x \leq y$
 ⟨proof⟩

lemma *tan-bound-pi2*: $|x| < \pi/4 \implies |\tan x| < 1$
 ⟨proof⟩

lemma *tan-cot*: $\tan(\pi/2 - x) = \text{inverse}(\tan x)$
 ⟨proof⟩

112.15 Cotangent

definition *cot* :: 'a \Rightarrow 'a::{real-normed-field,banach}
 where $\text{cot} = (\lambda x. \cos x / \sin x)$

lemma *cot-of-real*: $\text{of-real}(\text{cot } x) = (\text{cot}(\text{of-real } x)) :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 ⟨proof⟩

lemma *cot-in-Reals* [simp]: $z \in \mathbb{R} \implies \text{cot } z \in \mathbb{R}$
 for $z :: 'a::\{\text{real-normed-field}, \text{banach}\}$
 ⟨proof⟩

lemma *cot-zero* [simp]: $\text{cot } 0 = 0$
 ⟨proof⟩

lemma *cot-pi* [simp]: $\text{cot } \pi = 0$

$\langle \text{proof} \rangle$

lemma *cot-npi* [*simp*]: $\cot (\text{real } n * \pi) = 0$
for $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *cot-minus* [*simp*]: $\cot (-x) = -\cot x$
 $\langle \text{proof} \rangle$

lemma *cot-periodic* [*simp*]: $\cot (x + 2 * \pi) = \cot x$
 $\langle \text{proof} \rangle$

lemma *cot-altdef*: $\cot x = \text{inverse} (\tan x)$
 $\langle \text{proof} \rangle$

lemma *tan-altdef*: $\tan x = \text{inverse} (\cot x)$
 $\langle \text{proof} \rangle$

lemma *tan-cot'*: $\tan (\pi/2 - x) = \cot x$
 $\langle \text{proof} \rangle$

lemma *cot-gt-zero*: $0 < x \implies x < \pi/2 \implies 0 < \cot x$
 $\langle \text{proof} \rangle$

lemma *cot-less-zero*:
assumes $lb: -\pi/2 < x$ **and** $x < 0$
shows $\cot x < 0$
 $\langle \text{proof} \rangle$

lemma *DERIV-cot* [*simp*]: $\sin x \neq 0 \implies \text{DERIV } \cot x :> -\text{inverse} ((\sin x)^2)$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *isCont-cot*: $\sin x \neq 0 \implies \text{isCont } \cot x$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *isCont-cot'* [*simp, continuous-intros*]:
 $\text{isCont } f \ a \implies \sin (f \ a) \neq 0 \implies \text{isCont } (\lambda x. \cot (f \ x)) \ a$
for $a :: 'a :: \{\text{real-normed-field}, \text{banach}\}$ **and** $f :: 'a \Rightarrow 'a$
 $\langle \text{proof} \rangle$

lemma *tendsto-cot* [*tendsto-intros*]: $(f \longrightarrow a) \ F \implies \sin a \neq 0 \implies ((\lambda x. \cot (f \ x)) \longrightarrow \cot a) \ F$
for $f :: 'a \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *continuous-cot*:
 $\text{continuous } F \ f \implies \sin (f \ (\text{Lim } F \ (\lambda x. x))) \neq 0 \implies \text{continuous } F \ (\lambda x. \cot (f \ x))$

for $f :: 'a \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-cot* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
shows $\text{continuous-on } s \ f \Longrightarrow (\forall x \in s. \sin (f x) \neq 0) \Longrightarrow \text{continuous-on } s \ (\lambda x. \cot (f x))$
 $\langle \text{proof} \rangle$

lemma *continuous-within-cot* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
shows $\text{continuous (at } x \text{ within } s) \ f \Longrightarrow \sin (f x) \neq 0 \Longrightarrow \text{continuous (at } x \text{ within } s) (\lambda x. \cot (f x))$
 $\langle \text{proof} \rangle$

112.16 Inverse Trigonometric Functions

definition *arcsin* :: $\text{real} \Rightarrow \text{real}$
where $\text{arcsin } y = (\text{THE } x. -(pi/2) \leq x \wedge x \leq pi/2 \wedge \sin x = y)$

definition *arccos* :: $\text{real} \Rightarrow \text{real}$
where $\text{arccos } y = (\text{THE } x. 0 \leq x \wedge x \leq pi \wedge \cos x = y)$

definition *arctan* :: $\text{real} \Rightarrow \text{real}$
where $\text{arctan } y = (\text{THE } x. -(pi/2) < x \wedge x < pi/2 \wedge \tan x = y)$

lemma *arcsin*: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin } y \wedge \text{arcsin } y \leq pi/2$
 $\wedge \sin (\text{arcsin } y) = y$
 $\langle \text{proof} \rangle$

lemma *arcsin-pi*: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin } y \wedge \text{arcsin } y \leq pi$
 $\wedge \sin (\text{arcsin } y) = y$
 $\langle \text{proof} \rangle$

lemma *sin-arcsin* [*simp*]: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow \sin (\text{arcsin } y) = y$
 $\langle \text{proof} \rangle$

lemma *arcsin-bounded*: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin } y \wedge \text{arcsin } y \leq pi/2$
 $\langle \text{proof} \rangle$

lemma *arcsin-lbound*: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin } y$
 $\langle \text{proof} \rangle$

lemma *arcsin-ubound*: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow \text{arcsin } y \leq pi/2$
 $\langle \text{proof} \rangle$

lemma *arcsin-lt-bounded*:
assumes $-1 < y \wedge y < 1$

shows $-(\pi/2) < \arcsin y \wedge \arcsin y < \pi/2$
 $\langle \text{proof} \rangle$

lemma *arcsin-sin*: $-(\pi/2) \leq x \implies x \leq \pi/2 \implies \arcsin(\sin x) = x$
 $\langle \text{proof} \rangle$

lemma *arcsin-unique*:
assumes $-\pi/2 \leq x$ **and** $x \leq \pi/2$ **and** $\sin x = y$ **shows** $\arcsin y = x$
 $\langle \text{proof} \rangle$

lemma *arcsin-0* [*simp*]: $\arcsin 0 = 0$
 $\langle \text{proof} \rangle$

lemma *arcsin-1* [*simp*]: $\arcsin 1 = \pi/2$
 $\langle \text{proof} \rangle$

lemma *arcsin-minus-1* [*simp*]: $\arcsin(-1) = -(\pi/2)$
 $\langle \text{proof} \rangle$

lemma *arcsin-minus*: $-1 \leq x \implies x \leq 1 \implies \arcsin(-x) = -\arcsin x$
 $\langle \text{proof} \rangle$

lemma *arcsin-one-half* [*simp*]: $\arcsin(1/2) = \pi/6$
and *arcsin-minus-one-half* [*simp*]: $\arcsin(-(1/2)) = -\pi/6$
 $\langle \text{proof} \rangle$

lemma *arcsin-one-over-sqrt-2*: $\arcsin(1/\sqrt{2}) = \pi/4$
 $\langle \text{proof} \rangle$

lemma *arcsin-eq-iff*: $|x| \leq 1 \implies |y| \leq 1 \implies \arcsin x = \arcsin y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *cos-arcsin-nonzero*: $-1 < x \implies x < 1 \implies \cos(\arcsin x) \neq 0$
 $\langle \text{proof} \rangle$

lemma *arccos*: $-1 \leq y \implies y \leq 1 \implies 0 \leq \arccos y \wedge \arccos y \leq \pi \wedge \cos(\arccos y) = y$
 $\langle \text{proof} \rangle$

lemma *cos-arccos* [*simp*]: $-1 \leq y \implies y \leq 1 \implies \cos(\arccos y) = y$
 $\langle \text{proof} \rangle$

lemma *arccos-bounded*: $-1 \leq y \implies y \leq 1 \implies 0 \leq \arccos y \wedge \arccos y \leq \pi$
 $\langle \text{proof} \rangle$

lemma *arccos-lbound*: $-1 \leq y \implies y \leq 1 \implies 0 \leq \arccos y$
 $\langle \text{proof} \rangle$

lemma *arccos-ubound*: $-1 \leq y \implies y \leq 1 \implies \arccos y \leq \pi$

$\langle proof \rangle$

lemma *arccos-lt-bounded*:

assumes $-1 < y \wedge y < 1$

shows $0 < \arccos y \wedge \arccos y < \pi$

$\langle proof \rangle$

lemma *arccos-cos*: $0 \leq x \implies x \leq \pi \implies \arccos (\cos x) = x$

$\langle proof \rangle$

lemma *arccos-cos2*: $x \leq 0 \implies -\pi \leq x \implies \arccos (\cos x) = -x$

$\langle proof \rangle$

lemma *arccos-unique*:

assumes $0 \leq x$ **and** $x \leq \pi$ **and** $\cos x = y$ **shows** $\arccos y = x$

$\langle proof \rangle$

lemma *cos-arcsin*:

assumes $-1 \leq x \wedge x \leq 1$

shows $\cos (\arcsin x) = \sqrt{1 - x^2}$

$\langle proof \rangle$

lemma *sin-arccos*:

assumes $-1 \leq x \wedge x \leq 1$

shows $\sin (\arccos x) = \sqrt{1 - x^2}$

$\langle proof \rangle$

lemma *arccos-0* [simp]: $\arccos 0 = \pi/2$

$\langle proof \rangle$

lemma *arccos-1* [simp]: $\arccos 1 = 0$

$\langle proof \rangle$

lemma *arccos-minus-1* [simp]: $\arccos (-1) = \pi$

$\langle proof \rangle$

lemma *arccos-minus*: $-1 \leq x \implies x \leq 1 \implies \arccos (-x) = \pi - \arccos x$

$\langle proof \rangle$

lemma *arccos-one-half* [simp]: $\arccos (1/2) = \pi / 3$

and *arccos-minus-one-half* [simp]: $\arccos (-(1/2)) = 2 * \pi / 3$

$\langle proof \rangle$

lemma *arccos-one-over-sqrt-2*: $\arccos (1 / \sqrt{2}) = \pi / 4$

$\langle proof \rangle$

corollary *arccos-minus-abs*:

assumes $|x| \leq 1$

shows $\arccos (-x) = \pi - \arccos x$

$\langle proof \rangle$

lemma *sin-arccos-nonzero*: $-1 < x \implies x < 1 \implies \sin (\arccos x) \neq 0$
 $\langle proof \rangle$

lemma *arctan*: $-(\pi/2) < \arctan y \wedge \arctan y < \pi/2 \wedge \tan (\arctan y) = y$
 $\langle proof \rangle$

lemma *tan-arctan*: $\tan (\arctan y) = y$
 $\langle proof \rangle$

lemma *arctan-bounded*: $-(\pi/2) < \arctan y \wedge \arctan y < \pi/2$
 $\langle proof \rangle$

lemma *arctan-lbound*: $-(\pi/2) < \arctan y$
 $\langle proof \rangle$

lemma *arctan-ubound*: $\arctan y < \pi/2$
 $\langle proof \rangle$

lemma *arctan-unique*:
assumes $-(\pi/2) < x$
and $x < \pi/2$
and $\tan x = y$
shows $\arctan y = x$
 $\langle proof \rangle$

lemma *arctan-tan*: $-(\pi/2) < x \implies x < \pi/2 \implies \arctan (\tan x) = x$
 $\langle proof \rangle$

lemma *arctan-zero-zero* [simp]: $\arctan 0 = 0$
 $\langle proof \rangle$

lemma *arctan-minus*: $\arctan (-x) = -\arctan x$
 $\langle proof \rangle$

lemma *cos-arctan-not-zero* [simp]: $\cos (\arctan x) \neq 0$
 $\langle proof \rangle$

lemma *tan-eq-arctan-Ex*:
shows $\tan x = y \longleftrightarrow (\exists k::int. x = \arctan y + k*\pi \vee (x = \pi/2 + k*\pi \wedge y=0))$
 $\langle proof \rangle$

lemma *arctan-tan-eq-abs-pi*:
assumes $\cos \vartheta \neq 0$
obtains k **where** $\arctan (\tan \vartheta) = \vartheta - of-int k * \pi$
 $\langle proof \rangle$

lemma *tan-eq*:

assumes $\tan x = \tan y$ $\tan x \neq 0$

obtains $k::\text{int}$ **where** $x = y + k * \pi$

$\langle \text{proof} \rangle$

lemma *cos-arctan*: $\cos (\arctan x) = 1 / \text{sqrt} (1 + x^2)$

$\langle \text{proof} \rangle$

lemma *sin-arctan*: $\sin (\arctan x) = x / \text{sqrt} (1 + x^2)$

$\langle \text{proof} \rangle$

lemma *tan-sec*: $\cos x \neq 0 \implies 1 + (\tan x)^2 = (\text{inverse} (\cos x))^2$

for $x :: 'a::\{\text{real-normed-field}, \text{banach}, \text{field}\}$

$\langle \text{proof} \rangle$

lemma *arctan-less-iff*: $\arctan x < \arctan y \longleftrightarrow x < y$

$\langle \text{proof} \rangle$

lemma *arctan-le-iff*: $\arctan x \leq \arctan y \longleftrightarrow x \leq y$

$\langle \text{proof} \rangle$

lemma *arctan-eq-iff*: $\arctan x = \arctan y \longleftrightarrow x = y$

$\langle \text{proof} \rangle$

lemma *zero-less-arctan-iff* [simp]: $0 < \arctan x \longleftrightarrow 0 < x$

$\langle \text{proof} \rangle$

lemma *arctan-less-zero-iff* [simp]: $\arctan x < 0 \longleftrightarrow x < 0$

$\langle \text{proof} \rangle$

lemma *zero-le-arctan-iff* [simp]: $0 \leq \arctan x \longleftrightarrow 0 \leq x$

$\langle \text{proof} \rangle$

lemma *arctan-le-zero-iff* [simp]: $\arctan x \leq 0 \longleftrightarrow x \leq 0$

$\langle \text{proof} \rangle$

lemma *arctan-eq-zero-iff* [simp]: $\arctan x = 0 \longleftrightarrow x = 0$

$\langle \text{proof} \rangle$

lemma *continuous-on-arcsin'*: $\text{continuous-on } \{-1 .. 1\} \text{ arcsin}$

$\langle \text{proof} \rangle$

lemma *continuous-on-arcsin* [continuous-intros]:

$\text{continuous-on } s \ f \implies (\forall x \in s. -1 \leq f x \wedge f x \leq 1) \implies \text{continuous-on } s \ (\lambda x.$

$\arcsin (f x))$

$\langle \text{proof} \rangle$

lemma *isCont-arcsin*: $-1 < x \implies x < 1 \implies \text{isCont } \arcsin x$

$\langle \text{proof} \rangle$

lemma *continuous-on-arccos'*: *continuous-on* $\{-1 \dots 1\}$ *arccos*
 ⟨proof⟩

lemma *continuous-on-arccos* [*continuous-intros*]:
continuous-on $s \ f \implies (\forall x \in s. -1 \leq f \ x \wedge f \ x \leq 1) \implies \text{continuous-on } s \ (\lambda x. \text{arccos } (f \ x))$
 ⟨proof⟩

lemma *isCont-arccos*: $-1 < x \implies x < 1 \implies \text{isCont } \text{arccos } x$
 ⟨proof⟩

lemma *isCont-arctan*: *isCont* *arctan* x
 ⟨proof⟩

lemma *tendsto-arctan* [*tendsto-intros*]: $(f \longrightarrow x) \ F \implies ((\lambda x. \text{arctan } (f \ x)) \longrightarrow \text{arctan } x) \ F$
 ⟨proof⟩

lemma *continuous-arctan* [*continuous-intros*]: *continuous* $F \ f \implies \text{continuous } F \ (\lambda x. \text{arctan } (f \ x))$
 ⟨proof⟩

lemma *continuous-on-arctan* [*continuous-intros*]:
continuous-on $s \ f \implies \text{continuous-on } s \ (\lambda x. \text{arctan } (f \ x))$
 ⟨proof⟩

lemma *DERIV-arcsin*:
 assumes $-1 < x \ x < 1$
 shows *DERIV* *arcsin* $x \text{ :> inverse } (\text{sqrt } (1 - x^2))$
 ⟨proof⟩

lemma *DERIV-arccos*:
 assumes $-1 < x \ x < 1$
 shows *DERIV* *arccos* $x \text{ :> inverse } (- \text{sqrt } (1 - x^2))$
 ⟨proof⟩

lemma *DERIV-arctan*: *DERIV* *arctan* $x \text{ :> inverse } (1 + x^2)$
 ⟨proof⟩

declare
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*]
DERIV-arctan [*THEN* *DERIV-chain2*, *derivative-intros*]
DERIV-arctan [*THEN* *DERIV-chain2*, *unfolded has-field-derivative-def*, *deriva-*

tive-intros]

lemmas *has-derivative-arctan*[*derivative-intros*] = *DERIV-arctan*[*THEN DERIV-compose-FDERIV*]
and *has-derivative-arccos*[*derivative-intros*] = *DERIV-arccos*[*THEN DERIV-compose-FDERIV*]
and *has-derivative-arcsin*[*derivative-intros*] = *DERIV-arcsin*[*THEN DERIV-compose-FDERIV*]

lemma *filterlim-tan-at-right*: *filterlim tan at-bot (at-right (− (pi/2)))*
 ⟨*proof*⟩

lemma *filterlim-tan-at-left*: *filterlim tan at-top (at-left (pi/2))*
 ⟨*proof*⟩

lemma *tendsto-arctan-at-top*: *(arctan ⟶ (pi/2)) at-top*
 ⟨*proof*⟩

lemma *tendsto-arctan-at-bot*: *(arctan ⟶ − (pi/2)) at-bot*
 ⟨*proof*⟩

lemma *sin-multiple-reduce*:
 $\sin (x * \text{numeral } n :: 'a :: \{\text{real-normed-field, banach}\}) =$
 $\sin x * \cos (x * \text{of-nat } (\text{pred-numeral } n)) + \cos x * \sin (x * \text{of-nat } (\text{pred-numeral } n))$
 ⟨*proof*⟩

lemma *cos-multiple-reduce*:
 $\cos (x * \text{numeral } n :: 'a :: \{\text{real-normed-field, banach}\}) =$
 $\cos (x * \text{of-nat } (\text{pred-numeral } n)) * \cos x - \sin (x * \text{of-nat } (\text{pred-numeral } n))$
 $* \sin x$
 ⟨*proof*⟩

lemma *arccos-eq-pi-iff*: $x \in \{-1..1\} \implies \arccos x = \text{pi} \longleftrightarrow x = -1$
 ⟨*proof*⟩

lemma *arccos-eq-0-iff*: $x \in \{-1..1\} \implies \arccos x = 0 \longleftrightarrow x = 1$
 ⟨*proof*⟩

112.17 Prove Totality of the Trigonometric Functions

lemma *cos-arccos-abs*: $|y| \leq 1 \implies \cos (\arccos y) = y$
 ⟨*proof*⟩

lemma *sin-arccos-abs*: $|y| \leq 1 \implies \sin (\arccos y) = \text{sqrt } (1 - y^2)$
 ⟨*proof*⟩

lemma *sin-mono-less-eq*:
 $-(\text{pi}/2) \leq x \implies x \leq \text{pi}/2 \implies -(\text{pi}/2) \leq y \implies y \leq \text{pi}/2 \implies \sin x < \sin y$
 $\longleftrightarrow x < y$
 ⟨*proof*⟩

lemma *sin-mono-le-eq*:

– $(\pi/2) \leq x \implies x \leq \pi/2 \implies -(\pi/2) \leq y \implies y \leq \pi/2 \implies \sin x \leq \sin y$
 $\longleftrightarrow x \leq y$
 ⟨proof⟩

lemma *sin-inj-pi*:

– $(\pi/2) \leq x \implies x \leq \pi/2 \implies -(\pi/2) \leq y \implies y \leq \pi/2 \implies \sin x = \sin y$
 $\implies x = y$
 ⟨proof⟩

lemma *arcsin-le-iff*:

assumes $x \geq -1 \ x \leq 1 \ y \geq -\pi/2 \ y \leq \pi/2$
shows $\arcsin x \leq y \longleftrightarrow x \leq \sin y$
 ⟨proof⟩

lemma *le-arcsin-iff*:

assumes $x \geq -1 \ x \leq 1 \ y \geq -\pi/2 \ y \leq \pi/2$
shows $\arcsin x \geq y \longleftrightarrow x \geq \sin y$
 ⟨proof⟩

lemma *cos-mono-less-eq*: $0 \leq x \implies x \leq \pi \implies 0 \leq y \implies y \leq \pi \implies \cos x < \cos y \longleftrightarrow y < x$
 ⟨proof⟩

lemma *cos-mono-le-eq*: $0 \leq x \implies x \leq \pi \implies 0 \leq y \implies y \leq \pi \implies \cos x \leq \cos y \longleftrightarrow y \leq x$
 ⟨proof⟩

lemma *cos-inj-pi*: $0 \leq x \implies x \leq \pi \implies 0 \leq y \implies y \leq \pi \implies \cos x = \cos y \implies x = y$
 ⟨proof⟩

lemma *arccos-le-pi2*: $\llbracket 0 \leq y; y \leq 1 \rrbracket \implies \arccos y \leq \pi/2$
 ⟨proof⟩

lemma *sincos-total-pi-half*:

assumes $0 \leq x \ 0 \leq y \ x^2 + y^2 = 1$
shows $\exists t. 0 \leq t \wedge t \leq \pi/2 \wedge x = \cos t \wedge y = \sin t$
 ⟨proof⟩

lemma *sincos-total-pi*:

assumes $0 \leq y \ x^2 + y^2 = 1$
shows $\exists t. 0 \leq t \wedge t \leq \pi \wedge x = \cos t \wedge y = \sin t$
 ⟨proof⟩

lemma *sincos-total-2pi-le*:

assumes $x^2 + y^2 = 1$
shows $\exists t. 0 \leq t \wedge t \leq 2 * \pi \wedge x = \cos t \wedge y = \sin t$
 ⟨proof⟩

lemma *sincos-total-2pi*:

assumes $x^2 + y^2 = 1$

obtains t **where** $0 \leq t < 2\pi$ $x = \cos t$ $y = \sin t$

<proof>

lemma *arcsin-less-mono*: $|x| \leq 1 \implies |y| \leq 1 \implies \arcsin x < \arcsin y \longleftrightarrow x < y$

<proof>

lemma *arcsin-le-mono*: $|x| \leq 1 \implies |y| \leq 1 \implies \arcsin x \leq \arcsin y \longleftrightarrow x \leq y$

<proof>

lemma *arcsin-less-arcsin*: $-1 \leq x \implies x < y \implies y \leq 1 \implies \arcsin x < \arcsin y$

<proof>

lemma *arcsin-le-arcsin*: $-1 \leq x \implies x \leq y \implies y \leq 1 \implies \arcsin x \leq \arcsin y$

<proof>

lemma *arcsin-nonneg*: $x \in \{0..1\} \implies \arcsin x \geq 0$

<proof>

lemma *arccos-less-mono*: $|x| \leq 1 \implies |y| \leq 1 \implies \arccos x < \arccos y \longleftrightarrow y < x$

<proof>

lemma *arccos-le-mono*: $|x| \leq 1 \implies |y| \leq 1 \implies \arccos x \leq \arccos y \longleftrightarrow y \leq x$

<proof>

lemma *arccos-less-arccos*: $-1 \leq x \implies x < y \implies y \leq 1 \implies \arccos y < \arccos x$

<proof>

lemma *arccos-le-arccos*: $-1 \leq x \implies x \leq y \implies y \leq 1 \implies \arccos y \leq \arccos x$

<proof>

lemma *arccos-eq-iff*: $|x| \leq 1 \wedge |y| \leq 1 \implies \arccos x = \arccos y \longleftrightarrow x = y$

<proof>

lemma *arccos-cos-eq-abs*:

assumes $|\vartheta| \leq \pi$

shows $\arccos (\cos \vartheta) = |\vartheta|$

<proof>

lemma *arccos-cos-eq-abs-2pi*:

obtains k **where** $\arccos (\cos \vartheta) = |\vartheta - \text{of-int } k * (2 * \pi)|$

<proof>

lemma *arccos-arctan*:

assumes $-1 < x < 1$

shows $\arccos x = \pi/2 - \arctan(x / \sqrt{1 - x^2})$

$\langle \text{proof} \rangle$

lemma *arcsin-plus-arccos*:

assumes $-1 \leq x \leq 1$

shows $\arcsin x + \arccos x = \pi/2$

$\langle \text{proof} \rangle$

lemma *arcsin-arccos-eq*: $-1 \leq x \implies x \leq 1 \implies \arcsin x = \pi/2 - \arccos x$

$\langle \text{proof} \rangle$

lemma *arccos-arcsin-eq*: $-1 \leq x \implies x \leq 1 \implies \arccos x = \pi/2 - \arcsin x$

$\langle \text{proof} \rangle$

lemma *arcsin-arctan*: $-1 < x \implies x < 1 \implies \arcsin x = \arctan(x / \sqrt{1 - x^2})$

$\langle \text{proof} \rangle$

lemma *arcsin-arccos-sqrt-pos*: $0 \leq x \implies x \leq 1 \implies \arcsin x = \arccos(\sqrt{1 - x^2})$

$\langle \text{proof} \rangle$

lemma *arcsin-arccos-sqrt-neg*: $-1 \leq x \implies x \leq 0 \implies \arcsin x = -\arccos(\sqrt{1 - x^2})$

$\langle \text{proof} \rangle$

lemma *arccos-arcsin-sqrt-pos*: $0 \leq x \implies x \leq 1 \implies \arccos x = \arcsin(\sqrt{1 - x^2})$

$\langle \text{proof} \rangle$

lemma *arccos-arcsin-sqrt-neg*: $-1 \leq x \implies x \leq 0 \implies \arccos x = \pi - \arcsin(\sqrt{1 - x^2})$

$\langle \text{proof} \rangle$

lemma *cos-limit-1*:

assumes $(\lambda j. \cos(\vartheta j)) \longrightarrow 1$

shows $\exists k. (\lambda j. \vartheta j - \text{of-int}(k j) * (2 * \pi)) \longrightarrow 0$

$\langle \text{proof} \rangle$

lemma *cos-diff-limit-1*:

assumes $(\lambda j. \cos(\vartheta j - \Theta)) \longrightarrow 1$

obtains k **where** $(\lambda j. \vartheta j - \text{of-int}(k j) * (2 * \pi)) \longrightarrow \Theta$

$\langle \text{proof} \rangle$

112.18 Machin’s formula

lemma *arctan-one*: $\arctan 1 = \pi/4$

$\langle \text{proof} \rangle$

lemma *tan-total-pi4*:

assumes $|x| < 1$

shows $\exists z. - (pi/4) < z \wedge z < pi/4 \wedge \tan z = x$
 $\langle proof \rangle$

lemma *arctan-add*:

assumes $|x| \leq 1 \ |y| < 1$
shows $\arctan x + \arctan y = \arctan ((x + y) / (1 - x * y))$
 $\langle proof \rangle$

lemma *arctan-double*: $|x| < 1 \implies 2 * \arctan x = \arctan ((2 * x) / (1 - x^2))$
 $\langle proof \rangle$

theorem *machin*: $pi/4 = 4 * \arctan (1 / 5) - \arctan (1/239)$
 $\langle proof \rangle$

lemma *machin-Euler*: $5 * \arctan (1 / 7) + 2 * \arctan (3 / 79) = pi/4$
 $\langle proof \rangle$

112.19 Introducing the inverse tangent power series

lemma *monoseq-arctan-series*:

fixes $x :: real$
assumes $|x| \leq 1$
shows $monoseq (\lambda n. 1 / real (n * 2 + 1) * x^{(n * 2 + 1)})$
 $(is\ monoseq\ ?a)$
 $\langle proof \rangle$

lemma *zeroseq-arctan-series*:

fixes $x :: real$
assumes $|x| \leq 1$
shows $(\lambda n. 1 / real (n * 2 + 1) * x^{(n * 2 + 1)}) \longrightarrow 0$
 $(is\ ?a \longrightarrow 0)$
 $\langle proof \rangle$

lemma *summable-arctan-series*:

fixes $n :: nat$
assumes $|x| \leq 1$
shows $summable (\lambda k. (-1)^k * (1 / real (k*2+1) * x^{(k*2+1)}))$
 $(is\ summable\ (?c\ x))$
 $\langle proof \rangle$

lemma *DERIV-arctan-series*:

assumes $|x| < 1$
shows $DERIV (\lambda x'. \sum k. (-1)^k * (1 / real (k * 2 + 1) * x'^{(k * 2 + 1)}))$
 $x :>$
 $(\sum k. (-1)^k * x^{(k * 2)})$
 $(is\ DERIV\ ?arctan\ - :>\ ?Int)$
 $\langle proof \rangle$

lemma *arctan-series*:

assumes $|x| \leq 1$
shows $\arctan x = (\sum k. (-1)^k * (1 / \text{real } (k * 2 + 1)) * x^{(k * 2 + 1)})$
 (is - = $\text{suminf } (\lambda n. ?c x n)$)
 $\langle \text{proof} \rangle$

lemma *arctan-half*: $\arctan x = 2 * \arctan (x / (1 + \text{sqrt}(1 + x^2)))$
for $x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *arctan-monotone*: $x < y \implies \arctan x < \arctan y$
 $\langle \text{proof} \rangle$

lemma *arctan-monotone'*: $x \leq y \implies \arctan x \leq \arctan y$
 $\langle \text{proof} \rangle$

lemma *arctan-inverse*:
assumes $x \neq 0$
shows $\arctan (1/x) = \text{sgn } x * \pi/2 - \arctan x$
 $\langle \text{proof} \rangle$

theorem *pi-series*: $\pi/4 = (\sum k. (-1)^k * 1 / \text{real } (k * 2 + 1))$
 (is - = $?SUM$)
 $\langle \text{proof} \rangle$

112.20 Existence of Polar Coordinates

lemma *cos-x-y-le-one*: $|x / \text{sqrt } (x^2 + y^2)| \leq 1$
 $\langle \text{proof} \rangle$

lemma *polar-Ex*: $\exists r :: \text{real}. \exists a. x = r * \cos a \wedge y = r * \sin a$
 $\langle \text{proof} \rangle$

112.21 Basics about polynomial functions: products, extremal behaviour and root counts

lemma *polynomial-product-nat*:
fixes $x :: \text{nat}$
assumes $m: \bigwedge i. i > m \implies \text{int } (a i) = 0$
and $n: \bigwedge j. j > n \implies \text{int } (b j) = 0$
shows $(\sum i \leq m. (a i) * x^i) * (\sum j \leq n. (b j) * x^j) =$
 $(\sum r \leq m + n. (\sum k \leq r. (a k) * (b (r - k))) * x^r)$
 $\langle \text{proof} \rangle$

lemma *polyfun-diff*:
fixes $x :: 'a :: \text{idom}$
assumes $1 \leq n$
shows $(\sum i \leq n. a i * x^i) - (\sum i \leq n. a i * y^i) =$
 $(x - y) * (\sum j < n. (\sum i = \text{Suc } j..n. a i * y^{(i - j - 1)}) * x^j)$
 $\langle \text{proof} \rangle$

lemma *polyfun-diff-alt*:

fixes $x :: 'a::idom$
assumes $1 \leq n$
shows $(\sum_{i \leq n}. a \ i * x \hat{i}) - (\sum_{i \leq n}. a \ i * y \hat{i}) =$
 $(x - y) * ((\sum_{j < n}. \sum_{k < n-j}. a(j + k + 1) * y \hat{k} * x \hat{j}))$
 $\langle proof \rangle$

lemma *polyfun-linear-factor*:

fixes $a :: 'a::idom$
shows $\exists b. \forall z. (\sum_{i \leq n}. c(i) * z \hat{i}) = (z - a) * (\sum_{i < n}. b(i) * z \hat{i}) + (\sum_{i \leq n}. c(i) * a \hat{i})$
 $\langle proof \rangle$

lemma *polyfun-linear-factor-root*:

fixes $a :: 'a::idom$
assumes $(\sum_{i \leq n}. c(i) * a \hat{i}) = 0$
obtains b **where** $\bigwedge z. (\sum_{i \leq n}. c \ i * z \hat{i}) = (z - a) * (\sum_{i < n}. b \ i * z \hat{i})$
 $\langle proof \rangle$

lemma *isCont-polynom*: *isCont* $(\lambda w. \sum_{i \leq n}. c \ i * w \hat{i}) \ a$

for $c :: nat \Rightarrow 'a::real-normed-div-algebra$
 $\langle proof \rangle$

lemma *zero-polynom-imp-zero-coeffs*:

fixes $c :: nat \Rightarrow 'a::\{ab-semigroup-mult, real-normed-div-algebra\}$
assumes $\bigwedge w. (\sum_{i \leq n}. c \ i * w \hat{i}) = 0 \ k \leq n$
shows $c \ k = 0$
 $\langle proof \rangle$

lemma *polyfun-rootbound*:

fixes $c :: nat \Rightarrow 'a::\{idom, real-normed-div-algebra\}$
assumes $c \ k \neq 0 \ k \leq n$
shows $finite \ \{z. (\sum_{i \leq n}. c(i) * z \hat{i}) = 0\} \wedge card \ \{z. (\sum_{i \leq n}. c(i) * z \hat{i}) = 0\}$
 $\leq n$
 $\langle proof \rangle$

lemma

fixes $c :: nat \Rightarrow 'a::\{idom, real-normed-div-algebra\}$
assumes $c \ k \neq 0 \ k \leq n$
shows *polyfun-roots-finite*: $finite \ \{z. (\sum_{i \leq n}. c(i) * z \hat{i}) = 0\}$
and *polyfun-roots-card*: $card \ \{z. (\sum_{i \leq n}. c(i) * z \hat{i}) = 0\} \leq n$
 $\langle proof \rangle$

lemma *polyfun-finite-roots*:

fixes $c :: nat \Rightarrow 'a::\{idom, real-normed-div-algebra\}$
shows $finite \ \{x. (\sum_{i \leq n}. c \ i * x \hat{i}) = 0\} \longleftrightarrow (\exists i \leq n. c \ i \neq 0)$

(is ?lhs = ?rhs)
 <proof>

lemma *polyfun-eq-0*: $(\forall x. (\sum_{i \leq n}. c \ i * x^i) = 0) \longleftrightarrow (\forall i \leq n. c \ i = 0)$
for $c :: \text{nat} \Rightarrow 'a :: \{\text{idom}, \text{real-normed-div-algebra}\}$

<proof>

lemma *polyfun-eq-coeffs*: $(\forall x. (\sum_{i \leq n}. c \ i * x^i) = (\sum_{i \leq n}. d \ i * x^i)) \longleftrightarrow$
 $(\forall i \leq n. c \ i = d \ i)$
for $c :: \text{nat} \Rightarrow 'a :: \{\text{idom}, \text{real-normed-div-algebra}\}$
 <proof>

lemma *polyfun-eq-const*:
fixes $c :: \text{nat} \Rightarrow 'a :: \{\text{idom}, \text{real-normed-div-algebra}\}$
shows $(\forall x. (\sum_{i \leq n}. c \ i * x^i) = k) \longleftrightarrow c \ 0 = k \wedge (\forall i \in \{1..n\}. c \ i = 0)$
 (is ?lhs = ?rhs)
 <proof>

lemma *root-polyfun*:
fixes $z :: 'a :: \text{idom}$
assumes $1 \leq n$
shows $z^n = a \longleftrightarrow (\sum_{i \leq n}. (\text{if } i = 0 \text{ then } -a \text{ else if } i=n \text{ then } 1 \text{ else } 0) * z^i) = 0$
 <proof>

lemma
assumes *SORT-CONSTRAINT* $('a :: \{\text{idom}, \text{real-normed-div-algebra}\})$
and $1 \leq n$
shows *finite-roots-unity*: $\text{finite } \{z :: 'a. z^n = 1\}$
and *card-roots-unity*: $\text{card } \{z :: 'a. z^n = 1\} \leq n$
 <proof>

112.22 Hyperbolic functions

definition *sinh* :: $'a :: \{\text{banach}, \text{real-normed-algebra-1}\} \Rightarrow 'a$ **where**
 $\sinh x = (\exp x - \exp (-x)) /_R 2$

definition *cosh* :: $'a :: \{\text{banach}, \text{real-normed-algebra-1}\} \Rightarrow 'a$ **where**
 $\cosh x = (\exp x + \exp (-x)) /_R 2$

definition *tanh* :: $'a :: \{\text{banach}, \text{real-normed-field}\} \Rightarrow 'a$ **where**
 $\tanh x = \sinh x / \cosh x$

definition *arsinh* :: $'a :: \{\text{banach}, \text{real-normed-algebra-1}, \ln\} \Rightarrow 'a$ **where**
 $\operatorname{arsinh} x = \ln (x + (x^2 + 1) \text{ powr of-real } (1/2))$

definition *arcosh* :: $'a :: \{\text{banach}, \text{real-normed-algebra-1}, \ln\} \Rightarrow 'a$ **where**
 $\operatorname{arcosh} x = \ln (x + (x^2 - 1) \text{ powr of-real } (1/2))$

definition $\text{artanh} :: 'a :: \{\text{banach, real-normed-field, ln}\} \Rightarrow 'a$ **where**
 $\text{artanh } x = \ln ((1 + x) / (1 - x)) / 2$

lemma arsinh-0 [simp]: $\text{arsinh } 0 = 0$
 $\langle \text{proof} \rangle$

lemma arcosh-1 [simp]: $\text{arcosh } 1 = 0$
 $\langle \text{proof} \rangle$

lemma artanh-0 [simp]: $\text{artanh } 0 = 0$
 $\langle \text{proof} \rangle$

lemma tanh-altdef :
 $\text{tanh } x = (\exp x - \exp (-x)) / (\exp x + \exp (-x))$
 $\langle \text{proof} \rangle$

lemma tanh-real-altdef : $\text{tanh } (x::\text{real}) = (1 - \exp (-2 * x)) / (1 + \exp (-2 * x))$
 $\langle \text{proof} \rangle$

lemma sinh-converges : $(\lambda n. \text{if even } n \text{ then } 0 \text{ else } x^n / \text{fact } n) \text{ sums sinh } x$
 $\langle \text{proof} \rangle$

lemma cosh-converges : $(\lambda n. \text{if even } n \text{ then } x^n / \text{fact } n \text{ else } 0) \text{ sums cosh } x$
 $\langle \text{proof} \rangle$

lemma sinh-0 [simp]: $\text{sinh } 0 = 0$
 $\langle \text{proof} \rangle$

lemma cosh-0 [simp]: $\text{cosh } 0 = 1$
 $\langle \text{proof} \rangle$

lemma tanh-0 [simp]: $\text{tanh } 0 = 0$
 $\langle \text{proof} \rangle$

lemma sinh-minus [simp]: $\text{sinh } (-x) = -\text{sinh } x$
 $\langle \text{proof} \rangle$

lemma cosh-minus [simp]: $\text{cosh } (-x) = \text{cosh } x$
 $\langle \text{proof} \rangle$

lemma tanh-minus [simp]: $\text{tanh } (-x) = -\text{tanh } x$
 $\langle \text{proof} \rangle$

lemma sinh-ln-real : $x > 0 \implies \text{sinh } (\ln x :: \text{real}) = (x - \text{inverse } x) / 2$
 $\langle \text{proof} \rangle$

lemma *cosh-ln-real*: $x > 0 \implies \cosh (\ln x :: \text{real}) = (x + \text{inverse } x) / 2$
 ⟨proof⟩

lemma *tanh-ln-real*:
 $\tanh (\ln x :: \text{real}) = (x^2 - 1) / (x^2 + 1)$ **if** $x > 0$
 ⟨proof⟩

lemma *has-field-derivative-scaleR-right* [derivative-intros]:
 $(f \text{ has-field-derivative } D) F \implies ((\lambda x. c *_{\text{R}} f x) \text{ has-field-derivative } (c *_{\text{R}} D)) F$
 ⟨proof⟩

lemma *has-field-derivative-sinh* [THEN DERIV-chain2, derivative-intros]:
 $(\sinh \text{ has-field-derivative } \cosh x) (\text{at } (x :: 'a :: \{\text{banach, real-normed-field}\}))$
 ⟨proof⟩

lemma *has-field-derivative-cosh* [THEN DERIV-chain2, derivative-intros]:
 $(\cosh \text{ has-field-derivative } \sinh x) (\text{at } (x :: 'a :: \{\text{banach, real-normed-field}\}))$
 ⟨proof⟩

lemma *has-field-derivative-tanh* [THEN DERIV-chain2, derivative-intros]:
 $\cosh x \neq 0 \implies (\tanh \text{ has-field-derivative } 1 - \tanh x^2)$
 $(\text{at } (x :: 'a :: \{\text{banach, real-normed-field}\}))$
 ⟨proof⟩

lemma *has-derivative-sinh* [derivative-intros]:
fixes $g :: 'a \Rightarrow ('a :: \{\text{banach, real-normed-field}\})$
assumes $(g \text{ has-derivative } (\lambda x. Db * x)) (\text{at } x \text{ within } s)$
shows $((\lambda x. \sinh (g x)) \text{ has-derivative } (\lambda y. (\cosh (g x) * Db) * y)) (\text{at } x \text{ within } s)$
 ⟨proof⟩

lemma *has-derivative-cosh* [derivative-intros]:
fixes $g :: 'a \Rightarrow ('a :: \{\text{banach, real-normed-field}\})$
assumes $(g \text{ has-derivative } (\lambda y. Db * y)) (\text{at } x \text{ within } s)$
shows $((\lambda x. \cosh (g x)) \text{ has-derivative } (\lambda y. (\sinh (g x) * Db) * y)) (\text{at } x \text{ within } s)$
 ⟨proof⟩

lemma *sinh-plus-cosh*: $\sinh x + \cosh x = \exp x$
 ⟨proof⟩

lemma *cosh-plus-sinh*: $\cosh x + \sinh x = \exp x$
 ⟨proof⟩

lemma *cosh-minus-sinh*: $\cosh x - \sinh x = \exp (-x)$
 ⟨proof⟩

lemma *sinh-minus-cosh*: $\sinh x - \cosh x = -\exp (-x)$
 ⟨proof⟩

context

fixes $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$

begin

lemma *sinh-zero-iff*: $\sinh x = 0 \longleftrightarrow \exp x \in \{1, -1\}$
 $\langle \text{proof} \rangle$

lemma *cosh-zero-iff*: $\cosh x = 0 \longleftrightarrow \exp x^2 = -1$
 $\langle \text{proof} \rangle$

lemma *cosh-square-eq*: $\cosh x^2 = \sinh x^2 + 1$
 $\langle \text{proof} \rangle$

lemma *sinh-square-eq*: $\sinh x^2 = \cosh x^2 - 1$
 $\langle \text{proof} \rangle$

lemma *hyperbolic-pythagoras*: $\cosh x^2 - \sinh x^2 = 1$
 $\langle \text{proof} \rangle$

lemma *sinh-add*: $\sinh (x + y) = \sinh x * \cosh y + \cosh x * \sinh y$
 $\langle \text{proof} \rangle$

lemma *sinh-diff*: $\sinh (x - y) = \sinh x * \cosh y - \cosh x * \sinh y$
 $\langle \text{proof} \rangle$

lemma *cosh-add*: $\cosh (x + y) = \cosh x * \cosh y + \sinh x * \sinh y$
 $\langle \text{proof} \rangle$

lemma *cosh-diff*: $\cosh (x - y) = \cosh x * \cosh y - \sinh x * \sinh y$
 $\langle \text{proof} \rangle$

lemma *tanh-add*:

$\tanh (x + y) = (\tanh x + \tanh y) / (1 + \tanh x * \tanh y)$

if $\cosh x \neq 0 \ \cosh y \neq 0$

$\langle \text{proof} \rangle$

lemma *sinh-double*: $\sinh (2 * x) = 2 * \sinh x * \cosh x$
 $\langle \text{proof} \rangle$

lemma *cosh-double*: $\cosh (2 * x) = \cosh x^2 + \sinh x^2$
 $\langle \text{proof} \rangle$

end

lemma *sinh-field-def*: $\sinh z = (\exp z - \exp (-z)) / (2 :: 'a :: \{\text{banach}, \text{real-normed-field}\})$
 $\langle \text{proof} \rangle$

lemma *cosh-field-def*: $\cosh z = (\exp z + \exp (-z)) / (2 :: 'a :: \{\text{banach}, \text{real-normed-field}\})$
 ⟨proof⟩

112.22.1 More specific properties of the real functions

lemma *plus-inverse-ge-2*:
 fixes $x :: \text{real}$
 assumes $x > 0$
 shows $x + \text{inverse } x \geq 2$
 ⟨proof⟩

lemma *sinh-real-nonneg-iff* [simp]: $\sinh (x :: \text{real}) \geq 0 \longleftrightarrow x \geq 0$
 ⟨proof⟩

lemma *sinh-real-pos-iff* [simp]: $\sinh (x :: \text{real}) > 0 \longleftrightarrow x > 0$
 ⟨proof⟩

lemma *sinh-real-nonpos-iff* [simp]: $\sinh (x :: \text{real}) \leq 0 \longleftrightarrow x \leq 0$
 ⟨proof⟩

lemma *sinh-real-neg-iff* [simp]: $\sinh (x :: \text{real}) < 0 \longleftrightarrow x < 0$
 ⟨proof⟩

lemma *cosh-real-ge-1*: $\cosh (x :: \text{real}) \geq 1$
 ⟨proof⟩

lemma *cosh-real-pos* [simp]: $\cosh (x :: \text{real}) > 0$
 ⟨proof⟩

lemma *cosh-real-nonneg* [simp]: $\cosh (x :: \text{real}) \geq 0$
 ⟨proof⟩

lemma *cosh-real-nonzero* [simp]: $\cosh (x :: \text{real}) \neq 0$
 ⟨proof⟩

lemma *arsinh-real-def*: $\text{arsinh } (x :: \text{real}) = \ln (x + \sqrt{x^2 + 1})$
 ⟨proof⟩

lemma *arcosh-real-def*: $x \geq 1 \implies \text{arcosh } (x :: \text{real}) = \ln (x + \sqrt{x^2 - 1})$
 ⟨proof⟩

lemma *arsinh-real-aux*: $0 < x + \sqrt{x^2 + 1} :: \text{real}$
 ⟨proof⟩

lemma *arsinh-minus-real* [simp]: $\text{arsinh } (-x :: \text{real}) = -\text{arsinh } x$
 ⟨proof⟩

lemma *artanh-minus-real* [simp]:
 assumes $\text{abs } x < 1$

shows $\operatorname{artanh} (-x :: \text{real}) = -\operatorname{artanh} x$
 $\langle \text{proof} \rangle$

lemma *sinh-less-cosh-real*: $\sinh (x :: \text{real}) < \cosh x$
 $\langle \text{proof} \rangle$

lemma *sinh-le-cosh-real*: $\sinh (x :: \text{real}) \leq \cosh x$
 $\langle \text{proof} \rangle$

lemma *tanh-real-lt-1*: $\tanh (x :: \text{real}) < 1$
 $\langle \text{proof} \rangle$

lemma *tanh-real-gt-neg1*: $\tanh (x :: \text{real}) > -1$
 $\langle \text{proof} \rangle$

lemma *tanh-real-bounds*: $\tanh (x :: \text{real}) \in \{-1 < .. < 1\}$
 $\langle \text{proof} \rangle$

context
fixes $x :: \text{real}$
begin

lemma *arsinh-sinh-real*: $\operatorname{arsinh} (\sinh x) = x$
 $\langle \text{proof} \rangle$

lemma *arcosh-cosh-real*: $x \geq 0 \implies \operatorname{arcosh} (\cosh x) = x$
 $\langle \text{proof} \rangle$

lemma *artanh-tanh-real*: $\operatorname{artanh} (\tanh x) = x$
 $\langle \text{proof} \rangle$

lemma *sinh-real-zero-iff* [simp]: $\sinh x = 0 \longleftrightarrow x = 0$
 $\langle \text{proof} \rangle$

lemma *cosh-real-one-iff* [simp]: $\cosh x = 1 \longleftrightarrow x = 0$
 $\langle \text{proof} \rangle$

lemma *tanh-real-nonneg-iff* [simp]: $\tanh x \geq 0 \longleftrightarrow x \geq 0$
 $\langle \text{proof} \rangle$

lemma *tanh-real-pos-iff* [simp]: $\tanh x > 0 \longleftrightarrow x > 0$
 $\langle \text{proof} \rangle$

lemma *tanh-real-nonpos-iff* [simp]: $\tanh x \leq 0 \longleftrightarrow x \leq 0$
 $\langle \text{proof} \rangle$

lemma *tanh-real-neg-iff* [simp]: $\tanh x < 0 \longleftrightarrow x < 0$
 $\langle \text{proof} \rangle$

lemma *tanh-real-zero-iff* [simp]: $\tanh x = 0 \longleftrightarrow x = 0$
 ⟨proof⟩

end

lemma *sinh-real-strict-mono*: *strict-mono* (*sinh* :: *real* \Rightarrow *real*)
 ⟨proof⟩

lemma *cosh-real-strict-mono*:
 assumes $0 \leq x$ and $x < (y::real)$
 shows $\cosh x < \cosh y$
 ⟨proof⟩

lemma *tanh-real-strict-mono*: *strict-mono* (*tanh* :: *real* \Rightarrow *real*)
 ⟨proof⟩

lemma *sinh-real-abs* [simp]: $\sinh (abs\ x :: real) = abs (\sinh x)$
 ⟨proof⟩

lemma *cosh-real-abs* [simp]: $\cosh (abs\ x :: real) = \cosh x$
 ⟨proof⟩

lemma *tanh-real-abs* [simp]: $\tanh (abs\ x :: real) = abs (\tanh x)$
 ⟨proof⟩

lemma *sinh-real-eq-iff* [simp]: $\sinh x = \sinh y \longleftrightarrow x = (y :: real)$
 ⟨proof⟩

lemma *tanh-real-eq-iff* [simp]: $\tanh x = \tanh y \longleftrightarrow x = (y :: real)$
 ⟨proof⟩

lemma *cosh-real-eq-iff* [simp]: $\cosh x = \cosh y \longleftrightarrow abs\ x = abs\ (y :: real)$
 ⟨proof⟩

lemma *sinh-real-le-iff* [simp]: $\sinh x \leq \sinh y \longleftrightarrow x \leq (y::real)$
 ⟨proof⟩

lemma *cosh-real-nonneg-le-iff*: $x \geq 0 \implies y \geq 0 \implies \cosh x \leq \cosh y \longleftrightarrow x \leq (y::real)$
 ⟨proof⟩

lemma *cosh-real-nonpos-le-iff*: $x \leq 0 \implies y \leq 0 \implies \cosh x \leq \cosh y \longleftrightarrow x \geq (y::real)$
 ⟨proof⟩

lemma *tanh-real-le-iff* [simp]: $\tanh x \leq \tanh y \longleftrightarrow x \leq (y::real)$
 ⟨proof⟩

lemma *sinh-real-less-iff* [simp]: $\sinh x < \sinh y \longleftrightarrow x < (y::\text{real})$
 ⟨proof⟩

lemma *cosh-real-nonneg-less-iff*: $x \geq 0 \implies y \geq 0 \implies \cosh x < \cosh y \longleftrightarrow x < (y::\text{real})$
 ⟨proof⟩

lemma *cosh-real-nonpos-less-iff*: $x \leq 0 \implies y \leq 0 \implies \cosh x < \cosh y \longleftrightarrow x > (y::\text{real})$
 ⟨proof⟩

lemma *tanh-real-less-iff* [simp]: $\tanh x < \tanh y \longleftrightarrow x < (y::\text{real})$
 ⟨proof⟩

112.22.2 Limits

lemma *sinh-real-at-top*: *filterlim* (*sinh* :: *real* \Rightarrow *real*) *at-top* *at-top*
 ⟨proof⟩

lemma *sinh-real-at-bot*: *filterlim* (*sinh* :: *real* \Rightarrow *real*) *at-bot* *at-bot*
 ⟨proof⟩

lemma *cosh-real-at-top*: *filterlim* (*cosh* :: *real* \Rightarrow *real*) *at-top* *at-top*
 ⟨proof⟩

lemma *cosh-real-at-bot*: *filterlim* (*cosh* :: *real* \Rightarrow *real*) *at-top* *at-bot*
 ⟨proof⟩

lemma *tanh-real-at-top*: (*tanh* \longrightarrow (*1*::*real*)) *at-top*
 ⟨proof⟩

lemma *tanh-real-at-bot*: (*tanh* \longrightarrow (*-1*::*real*)) *at-bot*
 ⟨proof⟩

112.22.3 Properties of the inverse hyperbolic functions

lemma *isCont-sinh*: *isCont* *sinh* (*x* :: '*a* :: {*real-normed-field*, *banach*})
 ⟨proof⟩

lemma *isCont-cosh*: *isCont* *cosh* (*x* :: '*a* :: {*real-normed-field*, *banach*})
 ⟨proof⟩

lemma *isCont-tanh*: $\cosh x \neq 0 \implies \text{isCont } \tanh$ (*x* :: '*a* :: {*real-normed-field*, *banach*})
 ⟨proof⟩

lemma *continuous-on-sinh* [*continuous-intros*]:
 fixes *f* :: - \Rightarrow '*a*::{*real-normed-field*,*banach*}
 assumes *continuous-on* *A* *f*
 shows *continuous-on* *A* ($\lambda x. \sinh (f x)$)

$\langle \text{proof} \rangle$

lemma *continuous-on-cosh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous-on* A f
shows *continuous-on* A $(\lambda x. \cosh (f x))$
 $\langle \text{proof} \rangle$

lemma *continuous-sinh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous* F f
shows *continuous* F $(\lambda x. \sinh (f x))$
 $\langle \text{proof} \rangle$

lemma *continuous-cosh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous* F f
shows *continuous* F $(\lambda x. \cosh (f x))$
 $\langle \text{proof} \rangle$

lemma *continuous-on-tanh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous-on* A $f \wedge x. x \in A \implies \cosh (f x) \neq 0$
shows *continuous-on* A $(\lambda x. \tanh (f x))$
 $\langle \text{proof} \rangle$

lemma *continuous-at-within-tanh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous* (at x within A) $f \cosh (f x) \neq 0$
shows *continuous* (at x within A) $(\lambda x. \tanh (f x))$
 $\langle \text{proof} \rangle$

lemma *continuous-tanh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous* F $f \cosh (f (\text{Lim } F (\lambda x. x))) \neq 0$
shows *continuous* F $(\lambda x. \tanh (f x))$
 $\langle \text{proof} \rangle$

lemma *tendsto-sinh* [*tendsto-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
shows $(f \longrightarrow a) F \implies ((\lambda x. \sinh (f x)) \longrightarrow \sinh a) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-cosh* [*tendsto-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
shows $(f \longrightarrow a) F \implies ((\lambda x. \cosh (f x)) \longrightarrow \cosh a) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-tanh* [*tendsto-intros*]:

fixes $f :: - \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
shows $(f \longrightarrow a) F \implies \cosh a \neq 0 \implies ((\lambda x. \tanh (f x)) \longrightarrow \tanh a) F$
 $\langle \text{proof} \rangle$

lemma *arsinh-real-has-field-derivative* [*derivative-intros*]:
fixes $x :: \text{real}$
shows $(\text{arsinh has-field-derivative } (1 / (\text{sqrt } (x^2 + 1)))) \text{ (at } x \text{ within } A)$
 $\langle \text{proof} \rangle$

lemma *arcosh-real-has-field-derivative* [*derivative-intros*]:
fixes $x :: \text{real}$
assumes $x > 1$
shows $(\text{arcosh has-field-derivative } (1 / (\text{sqrt } (x^2 - 1)))) \text{ (at } x \text{ within } A)$
 $\langle \text{proof} \rangle$

lemma *artanh-real-has-field-derivative* [*derivative-intros*]:
 $(\text{artanh has-field-derivative } (1 / (1 - x^2))) \text{ (at } x \text{ within } A) \text{ if }$
 $|x| < 1 \text{ for } x :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *cosh-double-cosh*: $\cosh (2 * x :: 'a :: \{\text{banach}, \text{real-normed-field}\}) = 2 * (\cosh x)^2 - 1$
 $\langle \text{proof} \rangle$

lemma *sinh-multiple-reduce*:
 $\sinh (x * \text{numeral } n :: 'a :: \{\text{real-normed-field}, \text{banach}\}) =$
 $\sinh x * \cosh (x * \text{of-nat } (\text{pred-numeral } n)) + \cosh x * \sinh (x * \text{of-nat } (\text{pred-numeral } n))$
 $\langle \text{proof} \rangle$

lemma *cosh-multiple-reduce*:
 $\cosh (x * \text{numeral } n :: 'a :: \{\text{real-normed-field}, \text{banach}\}) =$
 $\cosh (x * \text{of-nat } (\text{pred-numeral } n)) * \cosh x + \sinh (x * \text{of-nat } (\text{pred-numeral } n)) * \sinh x$
 $\langle \text{proof} \rangle$

lemma *cosh-arcosh-real* [*simp*]:
assumes $x \geq (1 :: \text{real})$
shows $\cosh (\text{arcosh } x) = x$
 $\langle \text{proof} \rangle$

lemma *arcosh-eq-0-iff-real* [*simp*]: $x \geq 1 \implies \text{arcosh } x = 0 \iff x = (1 :: \text{real})$
 $\langle \text{proof} \rangle$

lemma *arcosh-nonneg-real* [*simp*]:
assumes $x \geq 1$
shows $\text{arcosh } (x :: \text{real}) \geq 0$
 $\langle \text{proof} \rangle$

lemma *arcosh-real-strict-mono*:

fixes $x\ y :: \text{real}$
assumes $1 \leq x\ x < y$
shows $\text{arcosh } x < \text{arcosh } y$
 $\langle \text{proof} \rangle$

lemma *arcosh-less-iff-real [simp]*:

fixes $x\ y :: \text{real}$
assumes $1 \leq x\ 1 \leq y$
shows $\text{arcosh } x < \text{arcosh } y \longleftrightarrow x < y$
 $\langle \text{proof} \rangle$

lemma *arcosh-real-gt-1-iff [simp]*: $x \geq 1 \implies \text{arcosh } x > 0 \longleftrightarrow x \neq (1 :: \text{real})$
 $\langle \text{proof} \rangle$

lemma *sinh-arcosh-real*: $x \geq 1 \implies \sinh (\text{arcosh } x) = \text{sqrt } (x^2 - 1)$
 $\langle \text{proof} \rangle$

lemma *sinh-arsinh-real [simp]*: $\sinh (\text{arsinh } x :: \text{real}) = x$
 $\langle \text{proof} \rangle$

lemma *arsinh-real-strict-mono*:

fixes $x\ y :: \text{real}$
assumes $x < y$
shows $\text{arsinh } x < \text{arsinh } y$
 $\langle \text{proof} \rangle$

lemma *arsinh-less-iff-real [simp]*:

fixes $x\ y :: \text{real}$
shows $\text{arsinh } x < \text{arsinh } y \longleftrightarrow x < y$
 $\langle \text{proof} \rangle$

lemma *arsinh-real-eq-0-iff [simp]*: $\text{arsinh } x = 0 \longleftrightarrow x = (0 :: \text{real})$
 $\langle \text{proof} \rangle$

lemma *arsinh-real-pos-iff [simp]*: $\text{arsinh } x > 0 \longleftrightarrow x > (0 :: \text{real})$
 $\langle \text{proof} \rangle$

lemma *arsinh-real-neg-iff [simp]*: $\text{arsinh } x < 0 \longleftrightarrow x < (0 :: \text{real})$
 $\langle \text{proof} \rangle$

lemma *cosh-arsinh-real*: $\cosh (\text{arsinh } x) = \text{sqrt } (x^2 + 1)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-arsinh [continuous-intros]*: $\text{continuous-on } A\ (\text{arsinh} :: \text{real} \Rightarrow \text{real})$
 $\langle \text{proof} \rangle$

lemma *continuous-on-arcosh* [*continuous-intros*]:
assumes $A \subseteq \{1..\}$
shows *continuous-on* A (*arcosh* :: *real* \Rightarrow *real*)
 \langle *proof* \rangle

lemma *continuous-on-artanh* [*continuous-intros*]:
assumes $A \subseteq \{-1<..
shows *continuous-on* A (*artanh* :: *real* \Rightarrow *real*)
 \langle *proof* $\rangle$$

lemma *continuous-on-arsinh'* [*continuous-intros*]:
fixes $f :: \text{real} \Rightarrow \text{real}$
assumes *continuous-on* A f
shows *continuous-on* A ($\lambda x. \text{arsinh } (f\ x)$)
 \langle *proof* \rangle

lemma *continuous-on-arcosh'* [*continuous-intros*]:
fixes $f :: \text{real} \Rightarrow \text{real}$
assumes *continuous-on* A $f \wedge x. x \in A \implies f\ x \geq 1$
shows *continuous-on* A ($\lambda x. \text{arcosh } (f\ x)$)
 \langle *proof* \rangle

lemma *continuous-on-artanh'* [*continuous-intros*]:
fixes $f :: \text{real} \Rightarrow \text{real}$
assumes *continuous-on* A $f \wedge x. x \in A \implies f\ x \in \{-1<..
shows *continuous-on* A ($\lambda x. \text{artanh } (f\ x)$)
 \langle *proof* $\rangle$$

lemma *isCont-arsinh* [*continuous-intros*]: *isCont* *arsinh* ($x :: \text{real}$)
 \langle *proof* \rangle

lemma *isCont-arcosh* [*continuous-intros*]:
assumes $x > 1$
shows *isCont* *arcosh* ($x :: \text{real}$)
 \langle *proof* \rangle

lemma *isCont-artanh* [*continuous-intros*]:
assumes $x > -1$ $x < 1$
shows *isCont* *artanh* ($x :: \text{real}$)
 \langle *proof* \rangle

lemma *tendsto-arsinh* [*tendsto-intros*]: $(f \longrightarrow a) \ F \implies ((\lambda x. \text{arsinh } (f\ x)) \longrightarrow \text{arsinh } a) \ F$
for $f :: - \Rightarrow \text{real}$
 \langle *proof* \rangle

lemma *tendsto-arcosh-strong* [*tendsto-intros*]:
fixes $f :: - \Rightarrow \text{real}$

assumes $(f \longrightarrow a) \ F \ a \geq 1 \text{ eventually } (\lambda x. f \ x \geq 1) \ F$
shows $((\lambda x. \operatorname{arcosh} (f \ x)) \longrightarrow \operatorname{arcosh} a) \ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-arcosh*:

fixes $f :: - \Rightarrow \text{real}$
assumes $(f \longrightarrow a) \ F \ a > 1$
shows $((\lambda x. \operatorname{arcosh} (f \ x)) \longrightarrow \operatorname{arcosh} a) \ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-arcosh-at-left-1*: $(\operatorname{arcosh} \longrightarrow 0) \ (\text{at-right } (1 :: \text{real}))$
 $\langle \text{proof} \rangle$

lemma *tendsto-artanh* [*tendsto-intros*]:

fixes $f :: 'a \Rightarrow \text{real}$
assumes $(f \longrightarrow a) \ F \ a > -1 \ a < 1$
shows $((\lambda x. \operatorname{artanh} (f \ x)) \longrightarrow \operatorname{artanh} a) \ F$
 $\langle \text{proof} \rangle$

lemma *continuous-arsinh* [*continuous-intros*]:

continuous $F \ f \Longrightarrow \text{continuous } F \ (\lambda x. \operatorname{arsinh} (f \ x :: \text{real}))$
 $\langle \text{proof} \rangle$

lemma *continuous-arcosh-strong* [*continuous-intros*]:

assumes *continuous* $F \ f \text{ eventually } (\lambda x. f \ x \geq 1) \ F$
shows *continuous* $F \ (\lambda x. \operatorname{arcosh} (f \ x :: \text{real}))$
 $\langle \text{proof} \rangle$

lemma *continuous-arcosh*:

continuous $F \ f \Longrightarrow f \ (\operatorname{Lim} F \ (\lambda x. x)) > 1 \Longrightarrow \text{continuous } F \ (\lambda x. \operatorname{arcosh} (f \ x :: \text{real}))$
 $\langle \text{proof} \rangle$

lemma *continuous-artanh* [*continuous-intros*]:

continuous $F \ f \Longrightarrow f \ (\operatorname{Lim} F \ (\lambda x. x)) \in \{-1 < .. < 1\} \Longrightarrow \text{continuous } F \ (\lambda x. \operatorname{artanh} (f \ x :: \text{real}))$
 $\langle \text{proof} \rangle$

lemma *arsinh-real-at-top*:

filterlim $(\operatorname{arsinh} :: \text{real} \Rightarrow \text{real}) \ \text{at-top at-top}$
 $\langle \text{proof} \rangle$

lemma *arsinh-real-at-bot*:

filterlim $(\operatorname{arsinh} :: \text{real} \Rightarrow \text{real}) \ \text{at-bot at-bot}$
 $\langle \text{proof} \rangle$

lemma *arcosh-real-at-top*:

filterlim $(\operatorname{arcosh} :: \text{real} \Rightarrow \text{real}) \ \text{at-top at-top}$

⟨proof⟩

lemma *artanh-real-at-left-1*:

filterlim (artanh :: real \Rightarrow real) at-top (at-left 1)
 ⟨proof⟩

lemma *artanh-real-at-right-1*:

filterlim (artanh :: real \Rightarrow real) at-bot (at-right (-1))
 ⟨proof⟩

112.23 Simprocs for root and power literals

lemma *numeral-powr-numeral-real [simp]*:

numeral m powr numeral n = (numeral m \wedge numeral n :: real)
 ⟨proof⟩

context

begin

private lemma *sqrt-numeral-simproc-aux*:

assumes *m * m \equiv n*

shows *sqrt (numeral n :: real) \equiv numeral m*

⟨proof⟩ **lemma** *root-numeral-simproc-aux*:

assumes *Num.pow m n \equiv x*

shows *root (numeral n) (numeral x :: real) \equiv numeral m*

⟨proof⟩ **lemma** *powr-numeral-simproc-aux*:

assumes *Num.pow y n = x*

shows *numeral x powr (m / numeral n :: real) \equiv numeral y powr m*

⟨proof⟩ **lemma** *numeral-powr-inverse-eq*:

numeral x powr (inverse (numeral n)) = numeral x powr (1 / numeral n :: real)

⟨proof⟩

⟨ML⟩

end

⟨ML⟩

lemma *root 100 1267650600228229401496703205376 = 2*

⟨proof⟩

lemma *sqrt 196 = 14*

⟨proof⟩

lemma *256 powr (7 / 4 :: real) = 16384*

⟨proof⟩

```
lemma 27 powr (inverse 3) = (3::real)
  <proof>
```

```
end
```

113 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
  <proof>
```

```
lemma complex-eqI [intro?]: Re x = Re y  $\implies$  Im x = Im y  $\implies$  x = y
  <proof>
```

```
lemma complex-eq-iff: x = y  $\longleftrightarrow$  Re x = Re y  $\wedge$  Im x = Im y
  <proof>
```

113.1 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
```

$$\begin{array}{l} \text{Re } (x - y) = \text{Re } x - \text{Re } y \\ | \text{Im } (x - y) = \text{Im } x - \text{Im } y \end{array}$$

instance

$\langle \text{proof} \rangle$

end

113.2 Multiplication and Division

instantiation *complex* :: *field*

begin

primcorec *one-complex*

where

$$\begin{array}{l} \text{Re } 1 = 1 \\ | \text{Im } 1 = 0 \end{array}$$

primcorec *times-complex*

where

$$\begin{array}{l} \text{Re } (x * y) = \text{Re } x * \text{Re } y - \text{Im } x * \text{Im } y \\ | \text{Im } (x * y) = \text{Re } x * \text{Im } y + \text{Im } x * \text{Re } y \end{array}$$

primcorec *inverse-complex*

where

$$\begin{array}{l} \text{Re } (\text{inverse } x) = \text{Re } x / ((\text{Re } x)^2 + (\text{Im } x)^2) \\ | \text{Im } (\text{inverse } x) = - \text{Im } x / ((\text{Re } x)^2 + (\text{Im } x)^2) \end{array}$$

definition $x \text{ div } y = x * \text{inverse } y$ **for** $x \ y :: \text{complex}$

instance

$\langle \text{proof} \rangle$

end

lemma *Re-divide*: $\text{Re } (x / y) = (\text{Re } x * \text{Re } y + \text{Im } x * \text{Im } y) / ((\text{Re } y)^2 + (\text{Im } y)^2)$
 $\langle \text{proof} \rangle$

lemma *Im-divide*: $\text{Im } (x / y) = (\text{Im } x * \text{Re } y - \text{Re } x * \text{Im } y) / ((\text{Re } y)^2 + (\text{Im } y)^2)$
 $\langle \text{proof} \rangle$

lemma *Complex-divide*:

$$\begin{aligned} (x / y) = & \text{Complex } ((\text{Re } x * \text{Re } y + \text{Im } x * \text{Im } y) / ((\text{Re } y)^2 + (\text{Im } y)^2)) \\ & ((\text{Im } x * \text{Re } y - \text{Re } x * \text{Im } y) / ((\text{Re } y)^2 + (\text{Im } y)^2)) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *Re-power2*: $\text{Re } (x \wedge 2) = (\text{Re } x) \wedge 2 - (\text{Im } x) \wedge 2$

$\langle \text{proof} \rangle$

lemma *Im-power2*: $\text{Im } (x \wedge 2) = 2 * \text{Re } x * \text{Im } x$
 $\langle \text{proof} \rangle$

lemma *Re-power-real [simp]*: $\text{Im } x = 0 \implies \text{Re } (x \wedge n) = \text{Re } x \wedge n$
 $\langle \text{proof} \rangle$

lemma *Im-power-real [simp]*: $\text{Im } x = 0 \implies \text{Im } (x \wedge n) = 0$
 $\langle \text{proof} \rangle$

113.3 Scalar Multiplication

instantiation *complex* :: *real-field*
begin

primcorec *scaleR-complex*
where
 $\text{Re } (\text{scaleR } r \ x) = r * \text{Re } x$
 $| \text{Im } (\text{scaleR } r \ x) = r * \text{Im } x$

instance
 $\langle \text{proof} \rangle$

end

113.4 Numerals, Arithmetic, and Embedding from R

declare $[[\text{coercion of-real} :: \text{real} \Rightarrow \text{complex}]]$
declare $[[\text{coercion of-rat} :: \text{rat} \Rightarrow \text{complex}]]$
declare $[[\text{coercion of-int} :: \text{int} \Rightarrow \text{complex}]]$
declare $[[\text{coercion of-nat} :: \text{nat} \Rightarrow \text{complex}]]$

abbreviation *complex-of-nat* :: $\text{nat} \Rightarrow \text{complex}$
where *complex-of-nat* \equiv *of-nat*

abbreviation *complex-of-int* :: $\text{int} \Rightarrow \text{complex}$
where *complex-of-int* \equiv *of-int*

abbreviation *complex-of-rat* :: $\text{rat} \Rightarrow \text{complex}$
where *complex-of-rat* \equiv *of-rat*

abbreviation *complex-of-real* :: $\text{real} \Rightarrow \text{complex}$
where *complex-of-real* \equiv *of-real*

lemma *complex-Re-of-nat [simp]*: $\text{Re } (\text{of-nat } n) = \text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma *complex-Im-of-nat [simp]*: $\text{Im } (\text{of-nat } n) = 0$
 $\langle \text{proof} \rangle$

lemma *complex-Re-of-int* [simp]: $\text{Re } (\text{of-int } z) = \text{of-int } z$
 ⟨proof⟩

lemma *complex-Im-of-int* [simp]: $\text{Im } (\text{of-int } z) = 0$
 ⟨proof⟩

lemma *complex-Re-numeral* [simp]: $\text{Re } (\text{numeral } v) = \text{numeral } v$
 ⟨proof⟩

lemma *complex-Im-numeral* [simp]: $\text{Im } (\text{numeral } v) = 0$
 ⟨proof⟩

lemma *Re-complex-of-real* [simp]: $\text{Re } (\text{complex-of-real } z) = z$
 ⟨proof⟩

lemma *Im-complex-of-real* [simp]: $\text{Im } (\text{complex-of-real } z) = 0$
 ⟨proof⟩

lemma *Re-divide-numeral* [simp]: $\text{Re } (z / \text{numeral } w) = \text{Re } z / \text{numeral } w$
 ⟨proof⟩

lemma *Im-divide-numeral* [simp]: $\text{Im } (z / \text{numeral } w) = \text{Im } z / \text{numeral } w$
 ⟨proof⟩

lemma *Re-divide-of-nat* [simp]: $\text{Re } (z / \text{of-nat } n) = \text{Re } z / \text{of-nat } n$
 ⟨proof⟩

lemma *Im-divide-of-nat* [simp]: $\text{Im } (z / \text{of-nat } n) = \text{Im } z / \text{of-nat } n$
 ⟨proof⟩

lemma *Re-inverse* [simp]: $r \in \mathbb{R} \implies \text{Re } (\text{inverse } r) = \text{inverse } (\text{Re } r)$
 ⟨proof⟩

lemma *Im-inverse* [simp]: $r \in \mathbb{R} \implies \text{Im } (\text{inverse } r) = 0$
 ⟨proof⟩

lemma *of-real-Re* [simp]: $z \in \mathbb{R} \implies \text{of-real } (\text{Re } z) = z$
 ⟨proof⟩

lemma *complex-Re-fact* [simp]: $\text{Re } (\text{fact } n) = \text{fact } n$
 ⟨proof⟩

lemma *surj-Re*: *surj Re*
 ⟨proof⟩

lemma *surj-Im*: *surj Im*
 ⟨proof⟩

lemma *complex-Im-fact* [simp]: $\text{Im } (\text{fact } n) = 0$
 ⟨proof⟩

lemma *Re-prod-Reals*: $(\bigwedge x. x \in A \implies f x \in \mathbb{R}) \implies \text{Re } (\text{prod } f A) = \text{prod } (\lambda x. \text{Re } (f x)) A$
 ⟨proof⟩

113.5 The Complex Number i

primcorec *imaginary-unit* :: *complex* (i)
 where
 $\text{Re } i = 0$
 $\text{Im } i = 1$

lemma *Complex-eq*: $\text{Complex } a \ b = a + i * b$
 ⟨proof⟩

lemma *complex-eq*: $a = \text{Re } a + i * \text{Im } a$
 ⟨proof⟩

lemma *fun-complex-eq*: $f = (\lambda x. \text{Re } (f x) + i * \text{Im } (f x))$
 ⟨proof⟩

lemma *i-squared* [simp]: $i * i = -1$
 ⟨proof⟩

lemma *power2-i* [simp]: $i^2 = -1$
 ⟨proof⟩

lemma *inverse-i* [simp]: $\text{inverse } i = -i$
 ⟨proof⟩

lemma *divide-i* [simp]: $x / i = -i * x$
 ⟨proof⟩

lemma *complex-i-mult-minus* [simp]: $i * (i * x) = -x$
 ⟨proof⟩

lemma *complex-i-not-zero* [simp]: $i \neq 0$
 ⟨proof⟩

lemma *complex-i-not-one* [simp]: $i \neq 1$
 ⟨proof⟩

lemma *complex-i-not-numeral* [simp]: $i \neq \text{numeral } w$
 ⟨proof⟩

lemma *complex-i-not-neg-numeral* [simp]: $i \neq -\text{numeral } w$
 ⟨proof⟩

lemma *complex-split-polar*: $\exists r\ a.\ z = \text{complex-of-real } r * (\cos a + i * \sin a)$
 $\langle \text{proof} \rangle$

lemma *i-even-power* [simp]: $i^{\wedge} (n * 2) = (-1)^{\wedge} n$
 $\langle \text{proof} \rangle$

lemma *i-even-power'* [simp]: $\text{even } n \implies i^{\wedge} n = (-1)^{\wedge} (n \text{ div } 2)$
 $\langle \text{proof} \rangle$

lemma *Re-i-times* [simp]: $\text{Re } (i * z) = - \text{Im } z$
 $\langle \text{proof} \rangle$

lemma *Im-i-times* [simp]: $\text{Im } (i * z) = \text{Re } z$
 $\langle \text{proof} \rangle$

lemma *i-times-eq-iff*: $i * w = z \longleftrightarrow w = - (i * z)$
 $\langle \text{proof} \rangle$

lemma *divide-numeral-i* [simp]: $z / (\text{numeral } n * i) = - (i * z) / \text{numeral } n$
 $\langle \text{proof} \rangle$

lemma *imaginary-eq-real-iff* [simp]:
 assumes $y \in \text{Reals } x \in \text{Reals}$
 shows $i * y = x \longleftrightarrow x=0 \wedge y=0$
 $\langle \text{proof} \rangle$

lemma *real-eq-imaginary-iff* [simp]:
 assumes $y \in \text{Reals } x \in \text{Reals}$
 shows $x = i * y \longleftrightarrow x=0 \wedge y=0$
 $\langle \text{proof} \rangle$

113.6 Vector Norm

instantiation *complex* :: *real-normed-field*
begin

definition *norm* $z = \text{sqrt } ((\text{Re } z)^2 + (\text{Im } z)^2)$

abbreviation *cmod* :: *complex* \Rightarrow *real*
where $cmod \equiv \text{norm}$

definition *complex-sgn-def*: $\text{sgn } x = x /_R cmod\ x$

definition *dist-complex-def*: $\text{dist } x\ y = cmod\ (x - y)$

definition *uniformity-complex-def* [code del]:
 $(\text{uniformity} :: (\text{complex} \times \text{complex}) \text{ filter}) = (\text{INF } e \in \{0 <.. \}. \text{principal } \{(x, y). \text{dist } x\ y < e\})$

definition *open-complex-def* [*code del*]:

open ($U :: \text{complex set}$) $\longleftrightarrow (\forall x \in U. \text{eventually } (\lambda(x', y). x' = x \longrightarrow y \in U)$
uniformity)

instance

$\langle \text{proof} \rangle$

end

declare *uniformity-Abort*[**where** $'a = \text{complex}$, *code*]

lemma *Re-divide'*: $\text{Re } (x / y) = (\text{Re } x * \text{Re } y + \text{Im } x * \text{Im } y) / (\text{norm } y)^2$
 $\langle \text{proof} \rangle$

lemma *Im-divide'*: $\text{Im } (x / y) = (\text{Im } x * \text{Re } y - \text{Re } x * \text{Im } y) / (\text{norm } y)^2$
 $\langle \text{proof} \rangle$

lemma *norm-ii* [*simp*]: $\text{norm } i = 1$
 $\langle \text{proof} \rangle$

lemma *cmod-unit-one*: $\text{cmod } (\cos a + i * \sin a) = 1$
 $\langle \text{proof} \rangle$

lemma *cmod-complex-polar*: $\text{cmod } (r * (\cos a + i * \sin a)) = |r|$
 $\langle \text{proof} \rangle$

lemma *complex-Re-le-cmod*: $\text{Re } x \leq \text{cmod } x$
 $\langle \text{proof} \rangle$

lemma *complex-mod-minus-le-complex-mod*: $-\text{cmod } x \leq \text{cmod } x$
 $\langle \text{proof} \rangle$

lemma *complex-mod-triangle-ineq2*: $\text{cmod } (b + a) - \text{cmod } b \leq \text{cmod } a$
 $\langle \text{proof} \rangle$

lemma *abs-Re-le-cmod*: $|\text{Re } x| \leq \text{cmod } x$
 $\langle \text{proof} \rangle$

lemma *abs-Im-le-cmod*: $|\text{Im } x| \leq \text{cmod } x$
 $\langle \text{proof} \rangle$

lemma *cmod-le*: $\text{cmod } z \leq |\text{Re } z| + |\text{Im } z|$
 $\langle \text{proof} \rangle$

lemma *cmod-eq-Re*: $\text{Im } z = 0 \implies \text{cmod } z = |\text{Re } z|$
 $\langle \text{proof} \rangle$

lemma *cmod-eq-Im*: $\text{Re } z = 0 \implies \text{cmod } z = |\text{Im } z|$

<proof>

lemma *cmod-power2*: $(\text{cmod } z)^2 = (\text{Re } z)^2 + (\text{Im } z)^2$
<proof>

lemma *cmod-plus-Re-le-0-iff*: $\text{cmod } z + \text{Re } z \leq 0 \longleftrightarrow \text{Re } z = - \text{cmod } z$
<proof>

lemma *cmod-Re-le-iff*: $\text{Im } x = \text{Im } y \implies \text{cmod } x \leq \text{cmod } y \longleftrightarrow |\text{Re } x| \leq |\text{Re } y|$
<proof>

lemma *cmod-Im-le-iff*: $\text{Re } x = \text{Re } y \implies \text{cmod } x \leq \text{cmod } y \longleftrightarrow |\text{Im } x| \leq |\text{Im } y|$
<proof>

lemma *Im-eq-0*: $|\text{Re } z| = \text{cmod } z \implies \text{Im } z = 0$
<proof>

lemma *abs-sqrt-wlog*: $(\bigwedge x. x \geq 0 \implies P \ x \ (x^2)) \implies P \ |x| \ (x^2)$
for *x::'a::linordered-idom*
<proof>

lemma *complex-abs-le-norm*: $|\text{Re } z| + |\text{Im } z| \leq \text{sqrt } 2 * \text{norm } z$
<proof>

lemma *complex-unit-circle*: $z \neq 0 \implies (\text{Re } z / \text{cmod } z)^2 + (\text{Im } z / \text{cmod } z)^2 = 1$
<proof>

Properties of complex signum.

lemma *sgn-eq*: $\text{sgn } z = z / \text{complex-of-real } (\text{cmod } z)$
<proof>

lemma *Re-sgn [simp]*: $\text{Re}(\text{sgn } z) = \text{Re}(z) / \text{cmod } z$
<proof>

lemma *Im-sgn [simp]*: $\text{Im}(\text{sgn } z) = \text{Im}(z) / \text{cmod } z$
<proof>

113.7 Absolute value

instantiation *complex* :: *field-abs-sgn*
begin

definition *abs-complex* :: *complex* \Rightarrow *complex*
where *abs-complex* = *of-real* \circ *norm*

instance
<proof>
end

113.8 Completeness of the Complexes

lemma *bounded-linear-Re: bounded-linear Re*
 ⟨proof⟩

lemma *bounded-linear-Im: bounded-linear Im*
 ⟨proof⟩

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]:*
 $\text{continuous-on } A \ f \implies \text{continuous-on } A \ g \implies \text{continuous-on } A \ (\lambda x. \text{Complex } (f \ x) \ (g \ x))$
 ⟨proof⟩

lemma *tendsto-Complex [tendsto-intros]:*
 $(f \longrightarrow a) \ F \implies (g \longrightarrow b) \ F \implies ((\lambda x. \text{Complex } (f \ x) \ (g \ x)) \longrightarrow \text{Complex } a \ b) \ F$
 ⟨proof⟩

lemma *tendsto-complex-iff:*
 $(f \longrightarrow x) \ F \longleftrightarrow (((\lambda x. \text{Re } (f \ x)) \longrightarrow \text{Re } x) \ F \wedge ((\lambda x. \text{Im } (f \ x)) \longrightarrow \text{Im } x) \ F)$
 ⟨proof⟩

lemma *continuous-complex-iff:*
 $\text{continuous } F \ f \longleftrightarrow \text{continuous } F \ (\lambda x. \text{Re } (f \ x)) \wedge \text{continuous } F \ (\lambda x. \text{Im } (f \ x))$
 ⟨proof⟩

lemma *continuous-on-of-real-o-iff [simp]:*
 $\text{continuous-on } S \ (\lambda x. \text{complex-of-real } (g \ x)) = \text{continuous-on } S \ g$

$\langle \text{proof} \rangle$

lemma *continuous-on-of-real-id* [simp]:
 $\text{continuous-on } S \text{ (of-real :: real} \Rightarrow 'a::\text{real-normed-algebra-1})$
 $\langle \text{proof} \rangle$

lemma *has-vector-derivative-complex-iff*: $(f \text{ has-vector-derivative } x) \ F \longleftrightarrow$
 $((\lambda x. \text{Re } (f \ x)) \text{ has-field-derivative } (\text{Re } x)) \ F \wedge$
 $((\lambda x. \text{Im } (f \ x)) \text{ has-field-derivative } (\text{Im } x)) \ F$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-Re*[*derivative-intros*]:
 $(f \text{ has-vector-derivative } D) \ F \Longrightarrow ((\lambda x. \text{Re } (f \ x)) \text{ has-field-derivative } (\text{Re } D)) \ F$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-Im*[*derivative-intros*]:
 $(f \text{ has-vector-derivative } D) \ F \Longrightarrow ((\lambda x. \text{Im } (f \ x)) \text{ has-field-derivative } (\text{Im } D)) \ F$
 $\langle \text{proof} \rangle$

instance *complex* :: *banach*
 $\langle \text{proof} \rangle$

declare *DERIV-power*[**where** *'a=complex, unfolded of-nat-def[symmetric], derivative-intros*]

113.9 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]: $\text{cnj } x = \text{cnj } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *complex-cnj-cnj* [simp]: $\text{cnj } (\text{cnj } z) = z$
 $\langle \text{proof} \rangle$

lemma *in-image-cnj-iff*: $z \in \text{cnj } 'A \longleftrightarrow \text{cnj } z \in A$
 $\langle \text{proof} \rangle$

lemma *image-cnj-conv-vimage-cnj*: $\text{cnj } 'A = \text{cnj } - 'A$
 $\langle \text{proof} \rangle$

lemma *complex-cnj-zero* [simp]: $\text{cnj } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *complex-cnj-zero-iff* [iff]: $\text{cnj } z = 0 \longleftrightarrow z = 0$
 $\langle \text{proof} \rangle$

lemma *complex-cnj-one-iff* [simp]: $\text{cnj } z = 1 \longleftrightarrow z = 1$
 ⟨proof⟩

lemma *complex-cnj-add* [simp]: $\text{cnj } (x + y) = \text{cnj } x + \text{cnj } y$
 ⟨proof⟩

lemma *cnj-sum* [simp]: $\text{cnj } (\text{sum } f \ s) = (\sum x \in s. \text{cnj } (f \ x))$
 ⟨proof⟩

lemma *complex-cnj-diff* [simp]: $\text{cnj } (x - y) = \text{cnj } x - \text{cnj } y$
 ⟨proof⟩

lemma *complex-cnj-minus* [simp]: $\text{cnj } (- \ x) = - \ \text{cnj } x$
 ⟨proof⟩

lemma *complex-cnj-one* [simp]: $\text{cnj } 1 = 1$
 ⟨proof⟩

lemma *complex-cnj-mult* [simp]: $\text{cnj } (x * y) = \text{cnj } x * \text{cnj } y$
 ⟨proof⟩

lemma *cnj-prod* [simp]: $\text{cnj } (\text{prod } f \ s) = (\prod x \in s. \text{cnj } (f \ x))$
 ⟨proof⟩

lemma *complex-cnj-inverse* [simp]: $\text{cnj } (\text{inverse } x) = \text{inverse } (\text{cnj } x)$
 ⟨proof⟩

lemma *complex-cnj-divide* [simp]: $\text{cnj } (x / y) = \text{cnj } x / \text{cnj } y$
 ⟨proof⟩

lemma *complex-cnj-power* [simp]: $\text{cnj } (x ^ n) = \text{cnj } x ^ n$
 ⟨proof⟩

lemma *complex-cnj-of-nat* [simp]: $\text{cnj } (\text{of-nat } n) = \text{of-nat } n$
 ⟨proof⟩

lemma *complex-cnj-of-int* [simp]: $\text{cnj } (\text{of-int } z) = \text{of-int } z$
 ⟨proof⟩

lemma *complex-cnj-numeral* [simp]: $\text{cnj } (\text{numeral } w) = \text{numeral } w$
 ⟨proof⟩

lemma *complex-cnj-neg-numeral* [simp]: $\text{cnj } (- \ \text{numeral } w) = - \ \text{numeral } w$
 ⟨proof⟩

lemma *complex-cnj-scaleR* [simp]: $\text{cnj } (\text{scaleR } r \ x) = \text{scaleR } r \ (\text{cnj } x)$
 ⟨proof⟩

lemma *complex-mod-cnj* [simp]: $\text{cmod } (\text{cnj } z) = \text{cmod } z$
 ⟨proof⟩

lemma *complex-cnj-complex-of-real* [simp]: $\text{cnj } (\text{of-real } x) = \text{of-real } x$
 ⟨proof⟩

lemma *complex-cnj-i* [simp]: $\text{cnj } i = -i$
 ⟨proof⟩

lemma *complex-add-cnj*: $z + \text{cnj } z = \text{complex-of-real } (2 * \text{Re } z)$
 ⟨proof⟩

lemma *complex-diff-cnj*: $z - \text{cnj } z = \text{complex-of-real } (2 * \text{Im } z) * i$
 ⟨proof⟩

lemma *Ints-cnj* [intro]: $x \in \mathbb{Z} \implies \text{cnj } x \in \mathbb{Z}$
 ⟨proof⟩

lemma *cnj-in-Ints-iff* [simp]: $\text{cnj } x \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$
 ⟨proof⟩

lemma *complex-mult-cnj*: $z * \text{cnj } z = \text{complex-of-real } ((\text{Re } z)^2 + (\text{Im } z)^2)$
 ⟨proof⟩

lemma *cnj-add-mult-eq-Re*: $z * \text{cnj } w + \text{cnj } z * w = 2 * \text{Re } (z * \text{cnj } w)$
 ⟨proof⟩

lemma *complex-mod-mult-cnj*: $\text{cmod } (z * \text{cnj } z) = (\text{cmod } z)^2$
 ⟨proof⟩

lemma *complex-mod-sqrt-Re-mult-cnj*: $\text{cmod } z = \text{sqrt } (\text{Re } (z * \text{cnj } z))$
 ⟨proof⟩

lemma *complex-In-mult-cnj-zero* [simp]: $\text{Im } (z * \text{cnj } z) = 0$
 ⟨proof⟩

lemma *complex-cnj-fact* [simp]: $\text{cnj } (\text{fact } n) = \text{fact } n$
 ⟨proof⟩

lemma *complex-cnj-pochhammer* [simp]: $\text{cnj } (\text{pochhammer } z \ n) = \text{pochhammer } (\text{cnj } z) \ n$
 ⟨proof⟩

lemma *bounded-linear-cnj*: *bounded-linear* *cnj*
 ⟨proof⟩

lemma *linear-cnj*: *linear* *cnj*
 ⟨proof⟩

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*: $((\lambda x. \text{cnj}(f\ x)) \longrightarrow \text{cnj } l) \ F \longleftrightarrow (f \longrightarrow l) \ F$
 $\langle \text{proof} \rangle$

lemma *sums-cnj*: $((\lambda x. \text{cnj}(f\ x)) \text{ sums } \text{cnj } l) \longleftrightarrow (f \text{ sums } l)$
 $\langle \text{proof} \rangle$

lemma *differentiable-cnj-iff*:
 $(\lambda z. \text{cnj } (f\ z)) \text{ differentiable at } x \text{ within } A \longleftrightarrow f \text{ differentiable at } x \text{ within } A$
 $\langle \text{proof} \rangle$

lemma *has-vector-derivative-cnj* [*derivative-intros*]:
assumes $(f \text{ has-vector-derivative } f') \text{ (at } z \text{ within } A)$
shows $((\lambda z. \text{cnj } (f\ z)) \text{ has-vector-derivative } \text{cnj } f') \text{ (at } z \text{ within } A)$
 $\langle \text{proof} \rangle$

lemma *has-field-derivative-cnj-cnj*:
assumes $(f \text{ has-field-derivative } F) \text{ (at } (\text{cnj } z))$
shows $((\text{cnj} \circ f \circ \text{cnj}) \text{ has-field-derivative } \text{cnj } F) \text{ (at } z)$
 $\langle \text{proof} \rangle$

113.10 Basic Lemmas

lemma *complex-of-real-code*[*code-unfold*]: $\text{of-real} = (\lambda x. \text{Complex } x\ 0)$
 $\langle \text{proof} \rangle$

lemma *complex-eq-0*: $z=0 \longleftrightarrow (\text{Re } z)^2 + (\text{Im } z)^2 = 0$
 $\langle \text{proof} \rangle$

lemma *complex-neq-0*: $z \neq 0 \longleftrightarrow (\text{Re } z)^2 + (\text{Im } z)^2 > 0$
 $\langle \text{proof} \rangle$

lemma *complex-norm-square*: $\text{of-real } ((\text{norm } z)^2) = z * \text{cnj } z$
 $\langle \text{proof} \rangle$

lemma *complex-div-cnj*: $a / b = (a * \text{cnj } b) / (\text{norm } b)^2$
 $\langle \text{proof} \rangle$

lemma *Re-complex-div-eq-0*: $\text{Re } (a / b) = 0 \longleftrightarrow \text{Re } (a * \text{cnj } b) = 0$
 $\langle \text{proof} \rangle$

lemma *Im-complex-div-eq-0*: $\text{Im } (a / b) = 0 \longleftrightarrow \text{Im } (a * \text{cnj } b) = 0$
 ⟨proof⟩

lemma *complex-div-gt-0*: $(\text{Re } (a / b) > 0 \longleftrightarrow \text{Re } (a * \text{cnj } b) > 0) \wedge (\text{Im } (a / b) > 0 \longleftrightarrow \text{Im } (a * \text{cnj } b) > 0)$
 ⟨proof⟩

lemma *Re-complex-div-gt-0*: $\text{Re } (a / b) > 0 \longleftrightarrow \text{Re } (a * \text{cnj } b) > 0$
and *Im-complex-div-gt-0*: $\text{Im } (a / b) > 0 \longleftrightarrow \text{Im } (a * \text{cnj } b) > 0$
 ⟨proof⟩

lemma *Re-complex-div-ge-0*: $\text{Re } (a / b) \geq 0 \longleftrightarrow \text{Re } (a * \text{cnj } b) \geq 0$
 ⟨proof⟩

lemma *Im-complex-div-ge-0*: $\text{Im } (a / b) \geq 0 \longleftrightarrow \text{Im } (a * \text{cnj } b) \geq 0$
 ⟨proof⟩

lemma *Re-complex-div-lt-0*: $\text{Re } (a / b) < 0 \longleftrightarrow \text{Re } (a * \text{cnj } b) < 0$
 ⟨proof⟩

lemma *Im-complex-div-lt-0*: $\text{Im } (a / b) < 0 \longleftrightarrow \text{Im } (a * \text{cnj } b) < 0$
 ⟨proof⟩

lemma *Re-complex-div-le-0*: $\text{Re } (a / b) \leq 0 \longleftrightarrow \text{Re } (a * \text{cnj } b) \leq 0$
 ⟨proof⟩

lemma *Im-complex-div-le-0*: $\text{Im } (a / b) \leq 0 \longleftrightarrow \text{Im } (a * \text{cnj } b) \leq 0$
 ⟨proof⟩

lemma *Re-divide-of-real [simp]*: $\text{Re } (z / \text{of-real } r) = \text{Re } z / r$
 ⟨proof⟩

lemma *Im-divide-of-real [simp]*: $\text{Im } (z / \text{of-real } r) = \text{Im } z / r$
 ⟨proof⟩

lemma *Re-divide-Reals [simp]*: $r \in \mathbb{R} \implies \text{Re } (z / r) = \text{Re } z / \text{Re } r$
 ⟨proof⟩

lemma *Im-divide-Reals [simp]*: $r \in \mathbb{R} \implies \text{Im } (z / r) = \text{Im } z / \text{Re } r$
 ⟨proof⟩

lemma *Re-sum[simp]*: $\text{Re } (\text{sum } f \text{ } s) = (\sum x \in s. \text{Re } (f \text{ } x))$
 ⟨proof⟩

lemma *Im-sum[simp]*: $\text{Im } (\text{sum } f \text{ } s) = (\sum x \in s. \text{Im } (f \text{ } x))$
 ⟨proof⟩

lemma *Rats-complex-of-real-iff [iff]*: $\text{complex-of-real } x \in \mathbb{Q} \longleftrightarrow x \in \mathbb{Q}$
 ⟨proof⟩

lemma *sum-Re-le-cmod*: $(\sum i \in I. \text{Re } (z \ i)) \leq \text{cmod } (\sum i \in I. z \ i)$
 $\langle \text{proof} \rangle$

lemma *sum-Im-le-cmod*: $(\sum i \in I. \text{Im } (z \ i)) \leq \text{cmod } (\sum i \in I. z \ i)$
 $\langle \text{proof} \rangle$

lemma *sums-complex-iff*: $f \text{ sums } x \longleftrightarrow ((\lambda x. \text{Re } (f \ x)) \text{ sums } \text{Re } x) \wedge ((\lambda x. \text{Im } (f \ x)) \text{ sums } \text{Im } x)$
 $\langle \text{proof} \rangle$

lemma *summable-complex-iff*: $\text{summable } f \longleftrightarrow \text{summable } (\lambda x. \text{Re } (f \ x)) \wedge \text{summable } (\lambda x. \text{Im } (f \ x))$
 $\langle \text{proof} \rangle$

lemma *summable-complex-of-real [simp]*: $\text{summable } (\lambda n. \text{complex-of-real } (f \ n)) \longleftrightarrow \text{summable } f$
 $\langle \text{proof} \rangle$

lemma *summable-Re*: $\text{summable } f \implies \text{summable } (\lambda x. \text{Re } (f \ x))$
 $\langle \text{proof} \rangle$

lemma *summable-Im*: $\text{summable } f \implies \text{summable } (\lambda x. \text{Im } (f \ x))$
 $\langle \text{proof} \rangle$

lemma *complex-is-Nat-iff*: $z \in \mathbb{N} \longleftrightarrow \text{Im } z = 0 \wedge (\exists i. \text{Re } z = \text{of-nat } i)$
 $\langle \text{proof} \rangle$

lemma *complex-is-Int-iff*: $z \in \mathbb{Z} \longleftrightarrow \text{Im } z = 0 \wedge (\exists i. \text{Re } z = \text{of-int } i)$
 $\langle \text{proof} \rangle$

lemma *complex-is-Real-iff*: $z \in \mathbb{R} \longleftrightarrow \text{Im } z = 0$
 $\langle \text{proof} \rangle$

lemma *sgn-complex-iff*: $\text{sgn } x = \text{sgn } (\text{Re } x) \longleftrightarrow x \in \mathbb{R}$
 $\langle \text{proof} \rangle$

lemma *Reals-cnj-iff*: $z \in \mathbb{R} \longleftrightarrow \text{cnj } z = z$
 $\langle \text{proof} \rangle$

lemma *in-Reals-norm*: $z \in \mathbb{R} \implies \text{norm } z = |\text{Re } z|$
 $\langle \text{proof} \rangle$

lemma *Re-Reals-divide*: $r \in \mathbb{R} \implies \text{Re } (r / z) = \text{Re } r * \text{Re } z / (\text{norm } z)^2$
 $\langle \text{proof} \rangle$

lemma *Im-Reals-divide*: $r \in \mathbb{R} \implies \text{Im } (r / z) = -\text{Re } r * \text{Im } z / (\text{norm } z)^2$
 $\langle \text{proof} \rangle$

lemma *series-comparison-complex*:
fixes $f :: \text{nat} \Rightarrow 'a :: \text{banach}$
assumes $sg: \text{summable } g$
and $\bigwedge n. g\ n \in \mathbb{R} \bigwedge n. \text{Re } (g\ n) \geq 0$
and $fg: \bigwedge n. n \geq N \implies \text{norm}(f\ n) \leq \text{norm}(g\ n)$
shows $\text{summable } f$
 $\langle \text{proof} \rangle$

113.11 Polar Form for Complex Numbers

lemma *complex-unimodular-polar*:
assumes $\text{norm } z = 1$
obtains t **where** $0 \leq t < 2 * \pi$ $z = \text{Complex } (\cos t) (\sin t)$
 $\langle \text{proof} \rangle$

113.11.1 $\cos \theta + i \sin \theta$

primcorec $\text{cis} :: \text{real} \Rightarrow \text{complex}$
where
 $\text{Re } (\text{cis } a) = \cos a$
 $| \text{Im } (\text{cis } a) = \sin a$

lemma *cis-zero [simp]*: $\text{cis } 0 = 1$
 $\langle \text{proof} \rangle$

lemma *norm-cis [simp]*: $\text{norm } (\text{cis } a) = 1$
 $\langle \text{proof} \rangle$

lemma *sgn-cis [simp]*: $\text{sgn } (\text{cis } a) = \text{cis } a$
 $\langle \text{proof} \rangle$

lemma *cis-2pi [simp]*: $\text{cis } (2 * \pi) = 1$
 $\langle \text{proof} \rangle$

lemma *cis-neq-zero [simp]*: $\text{cis } a \neq 0$
 $\langle \text{proof} \rangle$

lemma *cis-cnj*: $\text{cnj } (\text{cis } t) = \text{cis } (-t)$
 $\langle \text{proof} \rangle$

lemma *cis-mult*: $\text{cis } a * \text{cis } b = \text{cis } (a + b)$
 $\langle \text{proof} \rangle$

lemma *DeMoivre*: $(\text{cis } a) ^ n = \text{cis } (\text{real } n * a)$
 $\langle \text{proof} \rangle$

lemma *cis-inverse [simp]*: $\text{inverse } (\text{cis } a) = \text{cis } (-a)$
 $\langle \text{proof} \rangle$

lemma *cis-divide*: $\text{cis } a / \text{cis } b = \text{cis } (a - b)$

$\langle proof \rangle$

lemma *cis-power-int*: $cis\ x\ powi\ n = cis\ (of-int\ n * x)$
 $\langle proof \rangle$

lemma *complex-cnj-power-int* [simp]: $cnj\ (x\ powi\ n) = cnj\ x\ powi\ n$
 $\langle proof \rangle$

lemma *divide-conv-cnj*: $norm\ z = 1 \implies x / z = x * cnj\ z$
 $\langle proof \rangle$

lemma *i-not-in-Reals* [simp, intro]: $i \notin \mathbb{R}$
 $\langle proof \rangle$

lemma *cos-n-Re-cis-pow-n*: $cos\ (real\ n * a) = Re\ (cis\ a ^ n)$
 $\langle proof \rangle$

lemma *sin-n-Im-cis-pow-n*: $sin\ (real\ n * a) = Im\ (cis\ a ^ n)$
 $\langle proof \rangle$

lemma *cis-pi* [simp]: $cis\ pi = -1$
 $\langle proof \rangle$

lemma *cis-pi-half* [simp]: $cis\ (pi / 2) = i$
 $\langle proof \rangle$

lemma *cis-minus-pi-half* [simp]: $cis\ (-(pi / 2)) = -i$
 $\langle proof \rangle$

lemma *cis-multiple-2pi* [simp]: $n \in \mathbb{Z} \implies cis\ (2 * pi * n) = 1$
 $\langle proof \rangle$

lemma *minus-cis*: $-cis\ x = cis\ (x + pi)$
 $\langle proof \rangle$

lemma *minus-cis'*: $-cis\ x = cis\ (x - pi)$
 $\langle proof \rangle$

113.11.2 $r(\cos \theta + i \sin \theta)$

definition *rcis* :: $real \Rightarrow real \Rightarrow complex$
where *rcis* $r\ a = complex-of-real\ r * cis\ a$

lemma *Re-rcis* [simp]: $Re(rcis\ r\ a) = r * cos\ a$
 $\langle proof \rangle$

lemma *Im-rcis* [simp]: $Im(rcis\ r\ a) = r * sin\ a$
 $\langle proof \rangle$

lemma *rcis-Ex*: $\exists r\ a.\ z = \text{rcis } r\ a$
 $\langle \text{proof} \rangle$

lemma *complex-mod-rcis* [simp]: $\text{cmod } (\text{rcis } r\ a) = |r|$
 $\langle \text{proof} \rangle$

lemma *cis-rcis-eq*: $\text{cis } a = \text{rcis } 1\ a$
 $\langle \text{proof} \rangle$

lemma *rcis-mult*: $\text{rcis } r1\ a * \text{rcis } r2\ b = \text{rcis } (r1 * r2)\ (a + b)$
 $\langle \text{proof} \rangle$

lemma *rcis-zero-mod* [simp]: $\text{rcis } 0\ a = 0$
 $\langle \text{proof} \rangle$

lemma *rcis-zero-arg* [simp]: $\text{rcis } r\ 0 = \text{complex-of-real } r$
 $\langle \text{proof} \rangle$

lemma *rcis-eq-zero-iff* [simp]: $\text{rcis } r\ a = 0 \longleftrightarrow r = 0$
 $\langle \text{proof} \rangle$

lemma *DeMoivre2*: $(\text{rcis } r\ a) ^ n = \text{rcis } (r ^ n)\ (\text{real } n * a)$
 $\langle \text{proof} \rangle$

lemma *rcis-inverse*: $\text{inverse}(\text{rcis } r\ a) = \text{rcis } (1 / r)\ (- a)$
 $\langle \text{proof} \rangle$

lemma *rcis-divide*: $\text{rcis } r1\ a / \text{rcis } r2\ b = \text{rcis } (r1 / r2)\ (a - b)$
 $\langle \text{proof} \rangle$

113.11.3 Complex exponential

lemma *exp-Reals-eq*:
assumes $z \in \mathbb{R}$
shows $\exp z = \text{of-real } (\exp (\text{Re } z))$
 $\langle \text{proof} \rangle$

lemma *cis-conv-exp*: $\text{cis } b = \exp (\text{i} * b)$
 $\langle \text{proof} \rangle$

lemma *exp-eq-polar*: $\exp z = \exp (\text{Re } z) * \text{cis } (\text{Im } z)$
 $\langle \text{proof} \rangle$

lemma *Re-exp*: $\text{Re } (\exp z) = \exp (\text{Re } z) * \cos (\text{Im } z)$
 $\langle \text{proof} \rangle$

lemma *Im-exp*: $\text{Im } (\exp z) = \exp (\text{Re } z) * \sin (\text{Im } z)$
 $\langle \text{proof} \rangle$

lemma *norm-cos-sin* [simp]: $\text{norm } (\text{Complex } (\cos t) (\sin t)) = 1$
 ⟨proof⟩

lemma *norm-exp-eq-Re* [simp]: $\text{norm } (\exp z) = \exp (\text{Re } z)$
 ⟨proof⟩

lemma *complex-exp-exists*: $\exists a \ r. z = \text{complex-of-real } r * \exp a$
 ⟨proof⟩

lemma *exp-pi-i* [simp]: $\exp (\text{of-real } \pi * i) = -1$
 ⟨proof⟩

lemma *exp-pi-i'* [simp]: $\exp (i * \text{of-real } \pi) = -1$
 ⟨proof⟩

lemma *exp-two-pi-i* [simp]: $\exp (2 * \text{of-real } \pi * i) = 1$
 ⟨proof⟩

lemma *exp-two-pi-i'* [simp]: $\exp (i * (\text{of-real } \pi * 2)) = 1$
 ⟨proof⟩

lemma *continuous-on-cis* [continuous-intros]:
 $\text{continuous-on } A \ f \implies \text{continuous-on } A \ (\lambda x. \text{cis } (f x))$
 ⟨proof⟩

lemma *tendsto-exp-0-Re-at-bot*: $(\exp \longrightarrow 0) (\text{filtercomap } \text{Re } \text{at-bot})$
 ⟨proof⟩

lemma *filterlim-exp-at-infinity-Re-at-top*: $\text{filterlim } \exp \text{ at-infinity } (\text{filtercomap } \text{Re } \text{at-top})$
 ⟨proof⟩

lemma *tendsto-cis* [tendsto-intros]:
assumes $(f \longrightarrow x) \ F$
shows $((\lambda u. \text{cis } (f u)) \longrightarrow \text{cis } x) \ F$
 ⟨proof⟩

lemma *tendsto-rcis* [tendsto-intros]:
assumes $(f \longrightarrow r) \ F \ (g \longrightarrow x) \ F$
shows $((\lambda u. \text{rcis } (f u) (g u)) \longrightarrow \text{rcis } r x) \ F$
 ⟨proof⟩

lemma *continuous-on-rcis* [continuous-intros]:
 $\text{continuous-on } A \ f \implies \text{continuous-on } A \ g \implies \text{continuous-on } A \ (\lambda x. \text{rcis } (f x) (g x))$
 ⟨proof⟩

lemma *has-derivative-cis* [derivative-intros]:
assumes $(f \text{ has-derivative } d) \ (\text{at } x \text{ within } A)$

shows $((\lambda x. \text{cis } (f x)) \text{ has-derivative } (\lambda t. d \ t *_R (\text{i} * \text{cis } (f x)))) \text{ (at } x \text{ within } A)$
 $\langle \text{proof} \rangle$

113.11.4 Complex argument

definition $\text{Arg} :: \text{complex} \Rightarrow \text{real}$

where $\text{Arg } z = (\text{if } z = 0 \text{ then } 0 \text{ else } (\text{SOME } a. \text{sgn } z = \text{cis } a \wedge -\pi < a \wedge a \leq \pi))$

lemma Arg-zero : $\text{Arg } 0 = 0$
 $\langle \text{proof} \rangle$

lemma cis-Arg-unique :

assumes $\text{sgn } z = \text{cis } x$ **and** $-\pi < x$ **and** $x \leq \pi$

shows $\text{Arg } z = x$

$\langle \text{proof} \rangle$

lemma Arg-correct :

assumes $z \neq 0$

shows $\text{sgn } z = \text{cis } (\text{Arg } z) \wedge -\pi < \text{Arg } z \wedge \text{Arg } z \leq \pi$

$\langle \text{proof} \rangle$

lemma Arg-bounded : $-\pi < \text{Arg } z \wedge \text{Arg } z \leq \pi$
 $\langle \text{proof} \rangle$

lemma cis-Arg : $z \neq 0 \implies \text{cis } (\text{Arg } z) = \text{sgn } z$
 $\langle \text{proof} \rangle$

lemma rcis-cmod-Arg : $\text{rcis } (\text{cmod } z) (\text{Arg } z) = z$
 $\langle \text{proof} \rangle$

lemma rcis-cn :

shows $\text{cnj } a = \text{rcis } (\text{cmod } a) (-\text{Arg } a)$

$\langle \text{proof} \rangle$

lemma $\text{cos-Arg-i-mult-zero}$ [simp]: $y \neq 0 \implies \text{Re } y = 0 \implies \cos (\text{Arg } y) = 0$
 $\langle \text{proof} \rangle$

lemma Arg-ii [simp]: $\text{Arg } \text{i} = \pi/2$
 $\langle \text{proof} \rangle$

lemma Arg-minus-ii [simp]: $\text{Arg } (-\text{i}) = -\pi/2$
 $\langle \text{proof} \rangle$

lemma cos-Arg : $z \neq 0 \implies \cos (\text{Arg } z) = \text{Re } z / \text{norm } z$
 $\langle \text{proof} \rangle$

lemma sin-Arg : $z \neq 0 \implies \sin (\text{Arg } z) = \text{Im } z / \text{norm } z$
 $\langle \text{proof} \rangle$

113.12 Complex n-th roots

lemma *bij-betw-roots-unity*:

assumes $n > 0$

shows $\text{bij-betw } (\lambda k. \text{cis } (2 * \pi * \text{real } k / \text{real } n)) \{..<n\} \{z. z^n = 1\}$
 (is *bij-betw* ?f - -)

<proof>

lemma *card-roots-unity-eq*:

assumes $n > 0$

shows $\text{card } \{z::\text{complex}. z^n = 1\} = n$

<proof>

lemma *bij-betw-nth-root-unity*:

fixes $c :: \text{complex}$ **and** $n :: \text{nat}$

assumes $c \neq 0$ **and** $n: n > 0$

defines $c' \equiv \text{root } n (\text{norm } c) * \text{cis } (\text{Arg } c / n)$

shows $\text{bij-betw } (\lambda z. c' * z) \{z. z^n = 1\} \{z. z^n = c\}$

<proof>

lemma *finite-nth-roots* [intro]:

assumes $n > 0$

shows $\text{finite } \{z::\text{complex}. z^n = c\}$

<proof>

lemma *card-nth-roots*:

assumes $c \neq 0$ $n > 0$

shows $\text{card } \{z::\text{complex}. z^n = c\} = n$

<proof>

lemma *sum-roots-unity*:

assumes $n > 1$

shows $\sum \{z::\text{complex}. z^n = 1\} = 0$

<proof>

lemma *sum-nth-roots*:

assumes $n > 1$

shows $\sum \{z::\text{complex}. z^n = c\} = 0$

<proof>

113.13 Square root of complex numbers

primcorec *csqrt* :: *complex* \Rightarrow *complex*

where

$\text{Re } (\text{csqrt } z) = \text{sqrt } ((\text{cmod } z + \text{Re } z) / 2)$

$|\text{Im } (\text{csqrt } z) = (\text{if } \text{Im } z = 0 \text{ then } 1 \text{ else } \text{sgn } (\text{Im } z)) * \text{sqrt } ((\text{cmod } z - \text{Re } z) / 2)$

lemma *csqrt-of-real-nonneg* [simp]: $\text{Im } x = 0 \implies \text{Re } x \geq 0 \implies \text{csqrt } x = \text{sqrt } (\text{Re } x)$

$\langle \text{proof} \rangle$

lemma *csqrt-of-real-nonpos* [simp]: $\text{Im } x = 0 \implies \text{Re } x \leq 0 \implies \text{csqrt } x = i * \text{sqrt } |\text{Re } x|$
 $\langle \text{proof} \rangle$

lemma *of-real-sqrt*: $x \geq 0 \implies \text{of-real } (\text{sqrt } x) = \text{csqrt } (\text{of-real } x)$
 $\langle \text{proof} \rangle$

lemma *csqrt-0* [simp]: $\text{csqrt } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *csqrt-1* [simp]: $\text{csqrt } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *csqrt-ii* [simp]: $\text{csqrt } i = (1 + i) / \text{sqrt } 2$
 $\langle \text{proof} \rangle$

lemma *power2-csqrt*[simp, algebra]: $(\text{csqrt } z)^2 = z$
 $\langle \text{proof} \rangle$

lemma *csqrt-power-even*:
assumes *even n*
shows $\text{csqrt } z ^ n = z ^ (n \text{ div } 2)$
 $\langle \text{proof} \rangle$

lemma *norm-csqrt* [simp]: $\text{norm } (\text{csqrt } z) = \text{sqrt } (\text{norm } z)$
 $\langle \text{proof} \rangle$

lemma *csqrt-eq-0* [simp]: $\text{csqrt } z = 0 \longleftrightarrow z = 0$
 $\langle \text{proof} \rangle$

lemma *csqrt-eq-1* [simp]: $\text{csqrt } z = 1 \longleftrightarrow z = 1$
 $\langle \text{proof} \rangle$

lemma *csqrt-principal*: $0 < \text{Re } (\text{csqrt } z) \vee \text{Re } (\text{csqrt } z) = 0 \wedge 0 \leq \text{Im } (\text{csqrt } z)$
 $\langle \text{proof} \rangle$

lemma *Re-csqrt*: $0 \leq \text{Re } (\text{csqrt } z)$
 $\langle \text{proof} \rangle$

lemma *csqrt-square*:
assumes $0 < \text{Re } b \vee (\text{Re } b = 0 \wedge 0 \leq \text{Im } b)$
shows $\text{csqrt } (b^2) = b$
 $\langle \text{proof} \rangle$

lemma *csqrt-unique*: $w^2 = z \implies 0 < \text{Re } w \vee \text{Re } w = 0 \wedge 0 \leq \text{Im } w \implies \text{csqrt } z = w$
 $\langle \text{proof} \rangle$

lemma *csqrt-minus* [simp]:

assumes $\text{Im } x < 0 \vee (\text{Im } x = 0 \wedge 0 \leq \text{Re } x)$

shows $\text{csqrt } (-x) = i * \text{csqrt } x$

<proof>

lemma *csqrt-neg-neg-real*:

assumes $\text{Im } x = 0 \text{ Re } x < 0$

shows $\text{csqrt } z \neq x$

<proof>

lemma *csqrt-of-real*: $x \geq 0 \implies \text{csqrt } (\text{of-real } x) = \text{of-real } (\text{sqrt } x)$

<proof>

lemma *csqrt-of-real'*: $\text{csqrt } (\text{of-real } x) = \text{of-real } (\text{sqrt } |x|) * (\text{if } x \geq 0 \text{ then } 1 \text{ else } i)$

<proof>

lemma *csqrt-minus-Reals*:

assumes $x \in \mathbb{R}$

shows $\text{csqrt } (-x) = \text{sgn } (\text{Re } x) * i * \text{csqrt } x$

<proof>

lemmas *cmod-def* = *norm-complex-def*

lemma *Complex-simps*:

shows *Complex-eq-0*: $\text{Complex } a \ b = 0 \longleftrightarrow a = 0 \wedge b = 0$

and *complex-add*: $\text{Complex } a \ b + \text{Complex } c \ d = \text{Complex } (a + c) \ (b + d)$

and *complex-minus*: $-(\text{Complex } a \ b) = \text{Complex } (-a) \ (-b)$

and *complex-diff*: $\text{Complex } a \ b - \text{Complex } c \ d = \text{Complex } (a - c) \ (b - d)$

and *Complex-eq-1*: $\text{Complex } a \ b = 1 \longleftrightarrow a = 1 \wedge b = 0$

and *Complex-eq-neg-1*: $\text{Complex } a \ b = -1 \longleftrightarrow a = -1 \wedge b = 0$

and *complex-mult*: $\text{Complex } a \ b * \text{Complex } c \ d = \text{Complex } (a * c - b * d) \ (a * d + b * c)$

and *complex-inverse*: $\text{inverse } (\text{Complex } a \ b) = \text{Complex } (a / (a^2 + b^2)) \ (-b / (a^2 + b^2))$

and *Complex-eq-numeral*: $\text{Complex } a \ b = \text{numeral } w \longleftrightarrow a = \text{numeral } w \wedge b = 0$

and *Complex-eq-neg-numeral*: $\text{Complex } a \ b = -\text{numeral } w \longleftrightarrow a = -\text{numeral } w \wedge b = 0$

and *complex-scaleR*: $\text{scaleR } r \ (\text{Complex } a \ b) = \text{Complex } (r * a) \ (r * b)$

and *Complex-eq-i*: $\text{Complex } x \ y = i \longleftrightarrow x = 0 \wedge y = 1$

and *i-mult-Complex*: $i * \text{Complex } a \ b = \text{Complex } (-b) \ a$

and *Complex-mult-i*: $\text{Complex } a \ b * i = \text{Complex } (-b) \ a$

and *i-complex-of-real*: $i * \text{complex-of-real } r = \text{Complex } 0 \ r$

and *complex-of-real-i*: $\text{complex-of-real } r * i = \text{Complex } 0 \ r$

and *Complex-add-complex-of-real*: $\text{Complex } x \ y + \text{complex-of-real } r = \text{Complex } (x+r) \ y$

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*: $\text{Complex } x \ y * \text{complex-of-real } r = \text{Complex } (x*r) \ (y*r)$
and *complex-of-real-mult-Complex*: $\text{complex-of-real } r * \text{Complex } x \ y = \text{Complex } (r*x) \ (r*y)$
and *complex-eq-cancel-iff2*: $(\text{Complex } x \ y = \text{complex-of-real } xa) = (x = xa \wedge y = 0)$
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*: $\text{Complex } (\text{sum } f \ s) \ 0 = \text{sum } (\lambda x. \text{Complex } (f \ x) \ 0) \ s$
and *complex-of-real-def*: $\text{complex-of-real } r = \text{Complex } r \ 0$
and *complex-norm*: $\text{cmod } (\text{Complex } x \ y) = \text{sqrt } (x^2 + y^2)$
 ⟨proof⟩

lemma *Complex-in-Reals*: $\text{Complex } x \ 0 \in \mathbb{R}$
 ⟨proof⟩

lemma *Complex-divide-complex-of-real*: $\text{Complex } x \ y / \text{of-real } r = \text{Complex } (x/r) \ (y/r)$
 ⟨proof⟩

lemma *cmod-neg-real*: $\text{cmod } (\text{Complex } (-x) \ y) = \text{cmod } (\text{Complex } x \ y)$
 ⟨proof⟩

Express a complex number as a linear combination of two others, not collinear with the origin

lemma *complex-axes*:
assumes $\text{Im } (y/x) \neq 0$
obtains $a \ b$ **where** $z = \text{of-real } a * x + \text{of-real } b * y$
 ⟨proof⟩

end

114 MacLaurin and Taylor Series

theory *MacLaurin*
imports *Transcendental*
begin

114.1 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*:
 $0 < h \implies$
 $\exists B::\text{real}. f \ h = (\sum m < n. (j \ m / (\text{fact } m)) * (h^{\wedge} m)) + (B * ((h^{\wedge} n) / (\text{fact } n)))$
 ⟨proof⟩

lemma *eq-diff-eq'*: $x = y - z \longleftrightarrow y = x + z$
for $x\ y\ z :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *fact-diff-Suc*: $n < \text{Suc } m \implies \text{fact } (\text{Suc } m - n) = (\text{Suc } m - n) * \text{fact } (m - n)$
 $\langle \text{proof} \rangle$

lemma *Maclaurin-lemma2*:
fixes B
assumes *DERIV*: $\forall m\ t. m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow \text{DERIV } (\text{diff } m)\ t :> \text{diff } (\text{Suc } m)\ t$
and *INIT*: $n = \text{Suc } k$
defines *difg* \equiv
 $(\lambda m\ t :: \text{real}. \text{diff } m\ t -$
 $((\sum p < n - m. \text{diff } (m + p)\ 0 / \text{fact } p * t \wedge p) + B * (t \wedge (n - m) / \text{fact } (n - m))))$
(is *difg* $\equiv (\lambda m\ t. \text{diff } m\ t - ?\text{difg } m\ t))$
shows $\forall m\ t. m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow \text{DERIV } (\text{difg } m)\ t :> \text{difg } (\text{Suc } m)\ t$
 $\langle \text{proof} \rangle$

lemma *Maclaurin*:
assumes *h*: $0 < h$
and *n*: $0 < n$
and *diff-0*: $\text{diff } 0 = f$
and *diff-Suc*: $\forall m\ t. m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow \text{DERIV } (\text{diff } m)\ t :> \text{diff } (\text{Suc } m)\ t$
shows
 $\exists t :: \text{real}. 0 < t \wedge t < h \wedge$
 $f\ h = \text{sum } (\lambda m. (\text{diff } m\ 0 / \text{fact } m) * h \wedge m) \{..<n\} + (\text{diff } n\ t / \text{fact } n) * h$
 $\wedge n$
 $\langle \text{proof} \rangle$

lemma *Maclaurin2*:
fixes $n :: \text{nat}$
and $h :: \text{real}$
assumes *INIT1*: $0 < h$
and *INIT2*: $\text{diff } 0 = f$
and *DERIV*: $\forall m\ t. m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow \text{DERIV } (\text{diff } m)\ t :> \text{diff } (\text{Suc } m)\ t$
shows $\exists t. 0 < t \wedge t \leq h \wedge f\ h = (\sum m < n. \text{diff } m\ 0 / (\text{fact } m) * h \wedge m) + \text{diff } n\ t / \text{fact } n * h \wedge n$
 $\langle \text{proof} \rangle$

lemma *Maclaurin-minus*:
fixes $n :: \text{nat}$ **and** $h :: \text{real}$
assumes $h < 0$ $0 < n$ *diff 0 = f*
and *DERIV*: $\forall m\ t. m < n \wedge h \leq t \wedge t \leq 0 \longrightarrow \text{DERIV } (\text{diff } m)\ t :> \text{diff } (\text{Suc } m)\ t$

shows $\exists t. h < t \wedge t < 0 \wedge f h = (\sum m < n. \text{diff } m \ 0 / \text{fact } m * h \wedge^m) + \text{diff } n \ t / \text{fact } n * h \wedge^n$
 $\langle \text{proof} \rangle$

114.2 More Convenient "Bidirectional" Version.

lemma *Maclaurin-bi-le*:

fixes $n :: \text{nat}$ **and** $x :: \text{real}$
assumes $\text{diff } 0 = f$
and $\text{DERIV} : \forall m \ t. m < n \wedge |t| \leq |x| \longrightarrow \text{DERIV } (\text{diff } m) \ t :> \text{diff } (\text{Suc } m)$
 t
shows $\exists t. |t| \leq |x| \wedge f x = (\sum m < n. \text{diff } m \ 0 / (\text{fact } m) * x \wedge^m) + \text{diff } n \ t / (\text{fact } n) * x \wedge^n$
(is $\exists t. - \wedge f x = ?f \ x \ t)$
 $\langle \text{proof} \rangle$

lemma *Maclaurin-all-lt*:

fixes $x :: \text{real}$
assumes $\text{INIT1} : \text{diff } 0 = f$
and $\text{INIT2} : 0 < n$
and $\text{INIT3} : x \neq 0$
and $\text{DERIV} : \forall m \ x. \text{DERIV } (\text{diff } m) \ x :> \text{diff } (\text{Suc } m) \ x$
shows $\exists t. 0 < |t| \wedge |t| < |x| \wedge f x =$
 $(\sum m < n. (\text{diff } m \ 0 / \text{fact } m) * x \wedge^m) + (\text{diff } n \ t / \text{fact } n) * x \wedge^n$
(is $\exists t. - \wedge - \wedge f x = ?f \ x \ t)$
 $\langle \text{proof} \rangle$

lemma *Maclaurin-zero*: $x = 0 \implies n \neq 0 \implies (\sum m < n. (\text{diff } m \ 0 / \text{fact } m) * x \wedge^m) = \text{diff } 0 \ 0$
for $x :: \text{real}$ **and** $n :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *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 $\exists t. |t| \leq |x| \wedge f x = (\sum m < n. (\text{diff } m \ 0 / \text{fact } m) * x \wedge^m) + (\text{diff } n \ t / \text{fact } n) * x \wedge^n$
(is $\exists t. - \wedge f x = ?f \ x \ t)$
 $\langle \text{proof} \rangle$

lemma *Maclaurin-all-le-objl*:

$\text{diff } 0 = f \wedge (\forall m \ x. \text{DERIV } (\text{diff } m) \ x :> \text{diff } (\text{Suc } m) \ x) \longrightarrow$
 $(\exists t :: \text{real}. |t| \leq |x| \wedge f x = (\sum m < n. (\text{diff } m \ 0 / \text{fact } m) * x \wedge^m) + (\text{diff } n \ t / \text{fact } n) * x \wedge^n)$
for $x :: \text{real}$ **and** $n :: \text{nat}$
 $\langle \text{proof} \rangle$

114.3 Version for Exponential Function

lemma *Maclaurin-exp-lt*:

fixes $x :: \text{real}$ **and** $n :: \text{nat}$

shows

$x \neq 0 \implies n > 0 \implies$
 $(\exists t. 0 < |t| \wedge |t| < |x| \wedge \exp x = (\sum m < n. (x \wedge m) / \text{fact } m) + (\exp t / \text{fact } n) * x \wedge n)$
 $\langle \text{proof} \rangle$

lemma *Maclaurin-exp-le*:

fixes $x :: \text{real}$ **and** $n :: \text{nat}$

shows $\exists t. |t| \leq |x| \wedge \exp x = (\sum m < n. (x \wedge m) / \text{fact } m) + (\exp t / \text{fact } n) * x \wedge n$
 $\langle \text{proof} \rangle$

corollary *exp-lower-Taylor-quadratic*: $0 \leq x \implies 1 + x + x^2 / 2 \leq \exp x$

for $x :: \text{real}$

$\langle \text{proof} \rangle$

corollary *ln-2-less-1*: $\ln 2 < (1 :: \text{real})$

$\langle \text{proof} \rangle$

114.4 Version for Sine Function

lemma *mod-exhaust-less-4*: $m \bmod 4 = 0 \vee m \bmod 4 = 1 \vee m \bmod 4 = 2 \vee m \bmod 4 = 3$

for $m :: \text{nat}$

$\langle \text{proof} \rangle$

It is unclear why so many variant results are needed.

lemma *sin-expansion-lemma*: $\sin (x + \text{real } (\text{Suc } m) * \pi / 2) = \cos (x + \text{real } m * \pi / 2)$

$\langle \text{proof} \rangle$

lemma *Maclaurin-sin-expansion2*:

$\exists t. |t| \leq |x| \wedge$

$\sin x = (\sum m < n. \text{sin-coeff } m * x \wedge m) + (\sin (t + 1/2 * \text{real } n * \pi) / \text{fact } n) * x \wedge n$
 $\langle \text{proof} \rangle$

lemma *Maclaurin-sin-expansion*:

$\exists t. \sin x = (\sum m < n. \text{sin-coeff } m * x \wedge m) + (\sin (t + 1/2 * \text{real } n * \pi) / \text{fact } n) * x \wedge n$
 $\langle \text{proof} \rangle$

lemma *Maclaurin-sin-expansion3*:

assumes $n > 0$ $x > 0$

shows $\exists t. 0 < t \wedge t < x \wedge$

$\sin x = (\sum m < n. \sin\text{-coeff } m * x^{\wedge} m) + (\sin (t + 1/2 * \text{real } n * \pi) / \text{fact } n) * x^{\wedge} n$
 <proof>

lemma *Maclaurin-sin-expansion4*:

assumes $0 < x$

shows $\exists t. 0 < t \wedge t \leq x \wedge \sin x = (\sum m < n. \sin\text{-coeff } m * x^{\wedge} m) + (\sin (t + 1/2 * \text{real } n * \pi) / \text{fact } n) * x^{\wedge} n$
 <proof>

114.5 Maclaurin Expansion for Cosine Function

lemma *sumr-cos-zero-one [simp]*: $(\sum m < \text{Suc } n. \cos\text{-coeff } m * 0^{\wedge} m) = 1$
 <proof>

lemma *cos-expansion-lemma*: $\cos (x + \text{real } (\text{Suc } m) * \pi / 2) = - \sin (x + \text{real } m * \pi / 2)$
 <proof>

lemma *Maclaurin-cos-expansion*:

$\exists t :: \text{real}. |t| \leq |x| \wedge$

$\cos x = (\sum m < n. \cos\text{-coeff } m * x^{\wedge} m) + (\cos(t + 1/2 * \text{real } n * \pi) / \text{fact } n) * x^{\wedge} n$
 <proof>

lemma *Maclaurin-cos-expansion2*:

assumes $x > 0 \wedge n > 0$

shows $\exists t. 0 < t \wedge t < x \wedge$

$\cos x = (\sum m < n. \cos\text{-coeff } m * x^{\wedge} m) + (\cos (t + 1/2 * \text{real } n * \pi) / \text{fact } n) * x^{\wedge} n$
 <proof>

lemma *Maclaurin-minus-cos-expansion*:

assumes $n > 0 \wedge x < 0$

shows $\exists t. x < t \wedge t < 0 \wedge$

$\cos x = (\sum m < n. \cos\text{-coeff } m * x^{\wedge} m) + ((\cos (t + 1/2 * \text{real } n * \pi) / \text{fact } n) * x^{\wedge} n)$
 <proof>

lemma *sin-bound-lemma*: $x = y \implies |u| \leq v \implies |(x + u) - y| \leq v$

for $x \ y \ u \ v :: \text{real}$

<proof>

lemma *Maclaurin-sin-bound*: $|\sin x - (\sum m < n. \sin\text{-coeff } m * x^{\wedge} m)| \leq \text{inverse } (\text{fact } n) * |x|^{\wedge} n$

$\langle proof \rangle$

115 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*: $n > 0$ *diff* $0 = f$
and *DERIV*: $\forall m\ t.\ m < n \wedge a \leq t \wedge t \leq b \longrightarrow \text{DERIV } (\text{diff } m)\ t :> (\text{diff } (Suc\ m)\ t)$
and *INTERV*: $a \leq c < b$
shows $\exists t :: \text{real}.\ c < t \wedge t < b \wedge$
 $f\ b = (\sum_{m < n} (\text{diff } m\ c / \text{fact } m) * (b - c)^m) + (\text{diff } n\ t / \text{fact } n) * (b - c)^n$
 $\langle proof \rangle$

lemma *Taylor-down:*

fixes $a :: \text{real}$ **and** $n :: \text{nat}$
assumes *INIT*: $n > 0$ *diff* $0 = f$
and *DERIV*: $(\forall m\ t.\ m < n \wedge a \leq t \wedge t \leq b \longrightarrow \text{DERIV } (\text{diff } m)\ t :> \text{diff } (Suc\ m)\ t)$
and *INTERV*: $a < c < b$
shows $\exists t.\ a < t \wedge t < c \wedge$
 $f\ a = (\sum_{m < n} (\text{diff } m\ c / \text{fact } m) * (a - c)^m) + (\text{diff } n\ t / \text{fact } n) * (a - c)^n$
 $\langle proof \rangle$

theorem *Taylor:*

fixes $a :: \text{real}$ **and** $n :: \text{nat}$
assumes *INIT*: $n > 0$ *diff* $0 = f$
and *DERIV*: $\forall m\ t.\ m < n \wedge a \leq t \wedge t \leq b \longrightarrow \text{DERIV } (\text{diff } m)\ t :> \text{diff } (Suc\ m)\ t$
and *INTERV*: $a \leq c < b$ $a \leq x \leq b$ $x \neq c$
shows $\exists t.$
 $(\text{if } x < c \text{ then } x < t \wedge t < c \text{ else } c < t \wedge t < x) \wedge$
 $f\ x = (\sum_{m < n} (\text{diff } m\ c / \text{fact } m) * (x - c)^m) + (\text{diff } n\ t / \text{fact } n) * (x - c)^n$
 $\langle proof \rangle$

end

116 More facts about binomial coefficients

These facts could have been proven before, but having real numbers makes the proofs a lot easier. Thanks to Alexander Maletzky among others.

theory *Binomial-Plus*

imports *Real*
begin

116.1 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 $n \implies n \text{ choose } (\text{Suc } (n \text{ div } 2)) = n \text{ choose } (n \text{ div } 2)$
 ⟨proof⟩

lemma *binomial-less-binomial-Suc*:

assumes $k: k < n \text{ div } 2$
shows $n \text{ choose } k < n \text{ choose } (\text{Suc } k)$
 ⟨proof⟩

lemma *binomial-strict-mono*:

assumes $k < k' \ 2 * k' \leq n$
shows $n \text{ choose } k < n \text{ choose } k'$
 ⟨proof⟩

lemma *binomial-mono*:

assumes $k \leq k' \ 2 * k' \leq n$
shows $n \text{ choose } k \leq n \text{ choose } k'$
 ⟨proof⟩

lemma *binomial-strict-antimono*:

assumes $k < k' \ 2 * k \geq n \ k' \leq n$
shows $n \text{ choose } k > n \text{ choose } k'$
 ⟨proof⟩

lemma *binomial-antimono*:

assumes $k \leq k' \ k \geq n \text{ div } 2 \ k' \leq n$
shows $n \text{ choose } k \geq n \text{ choose } k'$
 ⟨proof⟩

lemma *binomial-maximum*: $n \text{ choose } k \leq n \text{ choose } (n \text{ div } 2)$

⟨proof⟩

lemma *binomial-maximum'*: $(2 * n) \text{ choose } k \leq (2 * n) \text{ choose } n$

⟨proof⟩

lemma *central-binomial-lower-bound*:

assumes $n > 0$
shows $4^n / (2 * \text{real } n) \leq \text{real } ((2 * n) \text{ choose } n)$
 ⟨proof⟩

lemma *upper-le-binomial*:

assumes $0 < k$ and $k < n$

shows $n \leq n \text{ choose } k$
 $\langle \text{proof} \rangle$

116.2 Results about binomials and integers, thanks to Alexander Maletzky

Restore original sort constraints: semidom rather than field of char 0

$\langle ML \rangle$

lemma *gbinomial-eq-0-int*:
assumes $n < k$
shows $(\text{int } n) \text{ gchoose } k = 0$
 $\langle \text{proof} \rangle$

corollary *gbinomial-eq-0*: $0 \leq a \implies a < \text{int } k \implies a \text{ gchoose } k = 0$
 $\langle \text{proof} \rangle$

lemma *gbinomial-mono*:
fixes $k::\text{nat}$ **and** $a::\text{real}$
assumes $\text{of-nat } k \leq a \leq b$ **shows** $a \text{ gchoose } k \leq b \text{ gchoose } k$
 $\langle \text{proof} \rangle$

lemma *int-binomial*: $\text{int } (n \text{ choose } k) = (\text{int } n) \text{ gchoose } k$
 $\langle \text{proof} \rangle$

lemma *falling-fact-pochhammer*: $\text{prod } (\lambda i. a - \text{int } i) \{0..<k\} = (-1) ^ k * \text{pochhammer } (-a) k$
 $\langle \text{proof} \rangle$

lemma *falling-fact-pochhammer'*: $\text{prod } (\lambda i. a - \text{int } i) \{0..<k\} = \text{pochhammer } (a - \text{int } k + 1) k$
 $\langle \text{proof} \rangle$

lemma *gbinomial-int-pochhammer*: $(a::\text{int}) \text{ gchoose } k = (-1) ^ k * \text{pochhammer } (-a) k \text{ div fact } k$
 $\langle \text{proof} \rangle$

lemma *gbinomial-int-pochhammer'*: $a \text{ gchoose } k = \text{pochhammer } (a - \text{int } k + 1) k \text{ div fact } k$
 $\langle \text{proof} \rangle$

lemma *fact-dvd-pochhammer*: $\text{fact } k \text{ dvd } \text{pochhammer } (a::\text{int}) k$
 $\langle \text{proof} \rangle$

lemma *gbinomial-int-negated-upper*: $(a \text{ gchoose } k) = (-1) ^ k * ((\text{int } k - a - 1) \text{ gchoose } k)$
 $\langle \text{proof} \rangle$

lemma *gbinomial-int-mult-fact*: $\text{fact } k * (a \text{ gchoose } k) = (\prod i = 0..<k. a - \text{int } i)$

$\langle \text{proof} \rangle$

corollary *gbinomial-int-mult-fact'*: $(a \text{ gchoose } k) * \text{fact } k = (\prod_{i=0..<k} a - \text{int } i)$
 $\langle \text{proof} \rangle$

lemma *gbinomial-int-binomial*:

$a \text{ gchoose } k = (\text{if } 0 \leq a \text{ then } \text{int } ((\text{nat } a) \text{ choose } k) \text{ else } (-1::\text{int})^{\wedge} k * \text{int } ((k + (\text{nat } (-a)) - 1) \text{ choose } k))$
 $\langle \text{proof} \rangle$

corollary *gbinomial-nneg*: $0 \leq a \implies a \text{ gchoose } k = \text{int } ((\text{nat } a) \text{ choose } k)$
 $\langle \text{proof} \rangle$

corollary *gbinomial-neg*: $a < 0 \implies a \text{ gchoose } k = (-1::\text{int})^{\wedge} k * \text{int } ((k + (\text{nat } (-a)) - 1) \text{ choose } k)$
 $\langle \text{proof} \rangle$

lemma *of-int-gbinomial*: $\text{of-int } (a \text{ gchoose } k) = (\text{of-int } a :: 'a::\text{field-char-0}) \text{ gchoose } k$
 $\langle \text{proof} \rangle$

lemma *uminus-one-gbinomial* [simp]: $(-1::\text{int}) \text{ gchoose } k = (-1)^{\wedge} k$
 $\langle \text{proof} \rangle$

lemma *gbinomial-int-Suc-Suc*: $(x + 1::\text{int}) \text{ gchoose } (\text{Suc } k) = (x \text{ gchoose } k) + (x \text{ gchoose } (\text{Suc } k))$
 $\langle \text{proof} \rangle$

corollary *plus-Suc-gbinomial*:

$(x + (1 + \text{int } k)) \text{ gchoose } (\text{Suc } k) = ((x + \text{int } k) \text{ gchoose } k) + ((x + \text{int } k) \text{ gchoose } (\text{Suc } k))$
 (is ?l = ?r)
 $\langle \text{proof} \rangle$

lemma *gbinomial-int-n-n* [simp]: $(\text{int } n) \text{ gchoose } n = 1$
 $\langle \text{proof} \rangle$

lemma *gbinomial-int-Suc-n* [simp]: $(1 + \text{int } n) \text{ gchoose } n = 1 + \text{int } n$
 $\langle \text{proof} \rangle$

lemma *zbinomial-eq-0-iff* [simp]: $a \text{ gchoose } k = 0 \longleftrightarrow (0 \leq a \wedge a < \text{int } k)$
 $\langle \text{proof} \rangle$

116.3 Sums

lemma *gchoose-rising-sum-nat*: $(\sum_{j \leq n} \text{int } j + \text{int } k \text{ gchoose } k) = (\text{int } n + \text{int } k + 1) \text{ gchoose } (\text{Suc } k)$
 $\langle \text{proof} \rangle$

```

lemma gchoose-rising-sum:
  assumes  $0 \leq n$  — Necessary condition.
  shows  $(\sum_{j=0..n}. j + \text{int } k \text{ gchoose } k) = (n + \text{int } k + 1) \text{ gchoose } (\text{Suc } k)$ 
  <proof>

end

```

117 Comprehensive Complex Theory

```

theory Complex-Main
imports
  Complex
  MacLaurin
  Binomial-Plus
begin

end

```

References

- [1] H. Davenport. *The Higher Arithmetic*. Cambridge University Press, 1992.
- [2] M. Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.
- [3] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison–Wesley, Boston, MA, USA, 2nd edition, 1994.
- [4] M. Holz, K. Steffens, and E. Weitz. *Introduction to Cardinal Arithmetic*. Birkhäuser, 1999.
- [5] D. Leijen. Division and modulus for computer scientists. 2001.
- [6] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, LNCS 664, pages 328–345. Springer, 1993.